Automated Module Composition[⋆]

Stavros Tripakis

VERIMAG, Centre Equation, 2, rue de Vignate, 38610 Gières, France.
`www-verimag.imag.fr`, `tripakis@imag.fr`.

Abstract. We define an abstract problem of module composition (MC). In MC, modules are seen as black boxes with input and output ports. The objective is, given a set of available modules, to instantiate some of them (one or more times) and connect their ports, in order to obtain a target module. A general compatibility relation defines which ports can be connected to each other. Constraints are imposed on the number of instances of each module and the number of copies of each port. A linear objective function can be given to minimize the total cost of module instances and port copies.

The MC problem is motivated by the need to automate the composition of legacy modules used in the development of software embedded in cars. Due to the large number of modules, composition "by hand" is tedious and error-prone, and its automation would lead to significant cost reduction.

We show that the MC problem is NP-complete, by formulating an equivalent integer optimization problem. We also identify a number of special cases where the MC problem can be solved in polynomial time. Finally, we suggest techniques that can be used for the general cases.

1 Introduction

In this paper we introduce a problem of *module composition* and provide algorithms and fundamental worst-case complexity results for it.

The setting is as follows. We are given a set of available module types. Each module type is characterized simply by a set of input ports and a set of output ports. A binary compatibility relation between input and output ports models which input port can be connected to which output port. The problem is to (1) generate zero or more instances of each module type, (2) generate zero or more copies of each port in each module instance, and (3) connect all ports in a compatible way so that no port remains disconnected. As we show below, this problem is general enough to capture the problem of finding a composition of available modules that "implements" a given target module.

Constraints can be imposed on the number of instances of each module type (e.g., "at least 2 copies of A, at most 1 copy of B", etc) and the number of copies of each port in an instance (e.g., "can fan-out port p at most 10 times,

[⋆] This work has been supported in part by the European project "Next TTA" under project No IST-2001-32111.

port q needs at least 2 inputs", etc). A linear objective function can be given to minimize the total cost of module instances, port copies and connections.

To motivate the reader, we give an example, shown in Figure 1. The figure shows two available module types A and B and a target module T. The goal is to instantiate and connect A and B to obtain T. The only compatible ports are output port q_1 of A with input port p_3 of B (we use an arrow from p_3 to q_1 to denote that). For simplicity, we assume there are no bounds on the number of instances of modules or the number of copies of ports. By instantiating each of A and B and their ports once and connecting q_1 to p_3, we achieve our goal.

Now, it is easy to see that the above problem is equivalent to the following slightly modified problem. Given A, B and T^{-1} (the "inverted" version of T, that is, where input ports are turned into outputs and outputs into inputs), find a composition which includes exactly one instance of T^{-1} and results in the "closed" module (that is, where all ports are connected). Therefore, in the rest of this paper, we consider this modified version of the problem where there are only available modules and the goal is to reach a closed composition. We call this the module composition problem (MC).

The results we obtain for MC are the following. First, we give an equivalent formulation as an integer programming optimization problem. Second, we show that MC is NP-complete. Third, we identify a number of special cases where MC can be solved in polynomial time, by solving an equivalent network flow problem. Finally, we suggest methods to be used in the general case.

Applications. We believe that the abstract problem introduced and studied in this paper has a number of practical applications. For example, in a circuit layout context, modules might represent chips and ports might represent wires. In a software engineering context, modules and ports might represent software components and their interfaces. Automatic composition can be especially useful in a dynamic and distributed environment, such as programs communicating over the Internet using middleware like CORBA or JINI: in such an environment, it is important for fault tolerance and reconfiguration to dispose of fast automatic composition techniques.

The application that has initially motivated our work comes from the domain of automotive embedded software development [12,2]. The problem there is to "design a software tool that automatically composes a fully executable model from a list of components and an architectural description of the final model" (quote from [12]). The components are used for simulation and eventually code generation. Due to the large number of legacy components (hundreds or more, often developed by different groups), the current practice of composition "by hand" is extremely tedious, as well as error prone. It is obvious that an automatic composition technique would greatly reduce the software production cost.

Related work. Automating software composition is not a new idea. Tools such as `make` are widely used to automate compilation or any other software transformation, taking into account dependencies, and so on. The major difference

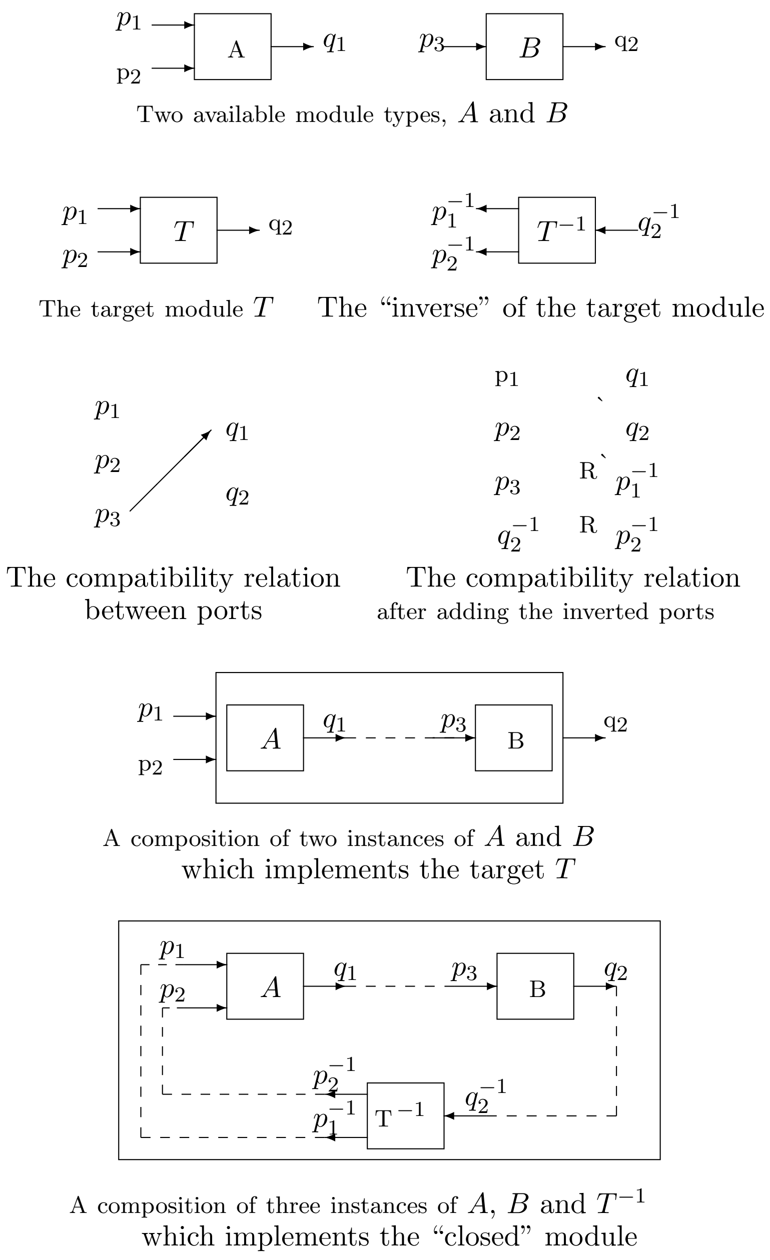
Two available module types, A and B
The target module T
The "inverse" of the target module
The compatibility relation between ports
The compatibility relation after adding the inverted ports
A composition of two instances of A and B which implements the target T
A composition of three instances of A, B and T^{-1} which implements the "closed" module

Fig. 1. Module composition example.

of these tools with our approach is, however, that in `make`, all alternative rules (i.e., possible solution chains) have to be *a-priori* known and hardcoded in the `makefile`. Another drawback is that hardcoding preferences among alternative solutions is not always easy in tools like `make`.

Our work is obviously related to Architecture Description Languages[1] as well as to Software Architectures (e.g., see [15]) and other component models (e.g., see [18]). Our approach is independent of methodological and language-specific aspects. We have focused in defining a basic formal problem (with obvious practical applications) and obtaining fundamental complexity results. We have also been interested in a "light-weight" approach. Our framework is intentionally simple: for example, it cannot impose global constraints on a solution and modules do not have operational semantics. This simplicity makes automation in the large possible.

[16,11] propose the tool *Metaframe*, based on *linear temporal logic* and using a variant of the synthesis algorithm of [10] (of exponential worst-case complexity). The major difference with our framework is that they build only *linear* compositions (i.e., chains) rather than general graphs, as we do. Other differences are that in Metaframe, all possible solutions can be obtained, whereas in our case, we obtain only one solution. In Metaframe, constraints on the order of modules in a chain can be expressed (e.g., A must appear before B), however, it is not clear how to specify preference constraints or bounds on the number of modules.

The problem in [7,8] is to automatically generate all possible configurations of middleware architectures. Here, too, all possible solutions are found. Their framework is not restricted to linear architectures, however, the number of components to be used in a solution is fixed in advance, and preference constraints cannot be expressed. Like Steffen et al, Kloukinas et al. also use algorithms inspired by *model-checking* and *formal verification* techniques.

Also related is the dataflow composition based framework implemented in the tool *Sword* [6,14]. The problem is quite similar to ours: the synthesis of a composite service based on available services. As with the above works, it is not clear whether multiple copies of each service can be used and whether preferences or bounds can be expressed. The algorithm used in Sword is also quite different from ours: it uses rule-based techniques from the domain of artificial intelligence.

Finally, related is the component-modeling language *Alloy*, for which it is possible to perform automatic analysis in a first-order logic [5]. The analysis is sound but not complete, since the logic is undecidable. We do not know yet whether it is possible to express some type of module composition and automate it within the Alloy framework.

This work extends the results of [17] which deal with the problem of automatically composing modules in a chain.

2 Module Composition Problem

Let N denote the set of natural numbers $\{0, 1, 2, ...\}$.

[1] E.g., see www.sei.cmu.edu/architecture/adl.html for a list of ADLs.

We consider a finite set of *ports*, $P = \{p_1, ..., p_k\}$. We assume that $P = \mathsf{P_{in}} \cup \mathsf{P_{out}}$, where $\mathsf{P_{in}} \cap \mathsf{P_{out}} = \emptyset$. $\mathsf{P_{in}}$ is the set of *input* ports, $\mathsf{P_{out}}$ is the set of *output* ports.

We also consider a *compatibility relation* $\mathsf{C} \subseteq \mathsf{P_{in}} \times \mathsf{P_{out}}$, between input and output ports. The understanding is that input port p can be connected to output port q iff $(p, q) \in \mathsf{C}$.

A *module type* is a tuple $A = (In, Out, f_{in}, f_{out})$, where $In \subseteq \mathsf{P_{in}}$, $Out \subseteq \mathsf{P_{out}}$, $f_{in} : In \rightarrow 2^N$ and $f_{out} : Out \rightarrow 2^N$. An *instance* of A is a tuple $(In, Out, g_{in}, g_{out})$, where $g_{in} : In \rightarrow N$ and $g_{out} : Out \rightarrow N$, such that for all $p \in In$, $g_{in}(p) \in f_{in}(p)$, and for all $q \in Out$, $g_{out}(q) \in f_{out}(q)$. The meaning is that $f_{out}(p)$ defines the possible *fan-out* factors of output port p, namely, how many (compatible) input ports can be connected to p. For example, if $f_{out}(p) = \{1, 2\}$ then at least one and at most two ports should be connected to p. We can view $f_{out}(p)$ as specifying how many *copies* of p an instance of module A can have. Each copy of p will be connected to a single copy of another port. Then, in a given instance of A, $g_{out}(p)$ fixes exactly how many copies of p there are, within the range specified by $f_{out}(p)$. The meanings of f_{in} and g_{in} are symmetric.

We will use the following notation and assumptions. For a module type $A = (In, Out, f_{in}, f_{out})$, $in(A)$, $out(A)$, $f_{in}(A)$ and $f_{out}(A)$ denote In, Out, f_{in} and f_{out}, respectively. Similarly for module instances. We will assume that $f_{in}(p)$ and $f_{out}(q)$ are non-negative integer intervals, e.g., $[0, 5]$, $[1, 1]$, and so on. We will also assume that the set of input and output ports of a module type are disjoint. Finally, we will assume that given a set of module types, they all have disjoint sets of input ports and disjoint sets of output ports.

Given a multiset of module instances $\mathcal{I}$ (not necessarily all of the same module type), a *composition* on $\mathcal{I}$ is defined as a multiset X of tuples of the form (A, p, B, q), where $A, B \in \mathcal{I}$, $p \in in(A)$ and $q \in out(B)$. The meaning of (A, p, B, q) is that input port p of A is connected to output port q of B.

We say that a composition X on $\mathcal{I}$ is *closed* if the following conditions hold:

$$\forall A \in \mathcal{I}, \forall p \in in(A), |\{(A, p, \_, \_) \in \mathsf{X}\}| = g_{in}(A)(p) \tag{1}$$

$$\forall A \in \mathcal{I}, \forall q \in out(A), |\{(\_, \_, A, q) \in \mathsf{X}\}| = g_{out}(A)(q) \tag{2}$$

Conditions 1 and 2 say that X connects a port to exactly as many ports as specified by its fan factor.

We say that a composition X on $\mathcal{I}$ *respects a compatibility relation* C if

$$\forall p \in \mathsf{P_{in}}, q \in \mathsf{P_{out}}, (\_, p, \_, q) \in \mathsf{X} \Rightarrow (p, q) \in \mathsf{C} \tag{3}$$

The last element we need is a set of constraints on how many instances of a module type we can have. We formalize that as a function $\mathsf{N_{inst}} : \mathcal{M} \rightarrow 2^N$, where $\mathcal{M}$ is the set of module types and for each module type A in $\mathcal{M}$, $\mathsf{N_{inst}}(A)$ is an interval. For example, if $\mathsf{N_{inst}}(A) = [1, 10]$, this means that at most 10 instances of A are available and at least 1 instance should be used, whereas if $\mathsf{N_{inst}}(A) = N$, then an arbitrary number of instances of A can be used, possibly none. We say that a set of module instances $\mathcal{I}$ *respects* $\mathsf{N_{inst}}$ if for each module type A, if n_A is the number of instances of A in $\mathcal{I}$ (notice that $\mathcal{I}$ is a multiset), then $n_A \in \mathsf{N_{inst}}(A)$.

We are now ready to state the module composition problem.

Definition 1 (Module Composition Problem (MC)). *Given*

(a) disjoint sets of input and output ports $\mathsf{P_{in}}, \mathsf{P_{out}}$*,*
(b) a compatibility relation $\mathsf{C} \subseteq \mathsf{P_{in}} \times \mathsf{P_{out}}$*,*
(c) a set of module types $\mathcal{M}$*, and*
(d) constraints on the number of module instances

$$\mathsf{N_{inst}} : \mathcal{M} \to 2^N,$$

find

(e) a set of module instances $\mathcal{I}$ *that respects* $\mathsf{N_{inst}}$*, and*
(f) a composition X *on* $\mathcal{I}$ *that respects* C *and is closed.*

Example 2. Consider the set of module types shown at the top of Figure 2. For the sake of simplicity, we assume that two ports are compatible iff they have the same name.[2] We require that exactly one instance of module A_0 be present in the composition, while there are no constraints on the number of instances of A_1, A_2, A_3. In our framework this can be modeled by setting $\mathsf{N_{inst}}(A_0) = \{1\}$ and $\mathsf{N_{inst}}(A_1) = \mathsf{N_{inst}}(A_2) = \mathsf{N_{inst}}(A_3) = N$. The fan factors of all ports are constrained to be exactly 1, except for output port a of module A_0, which can have from 0 up to 3 copies. That is, $f_{out}(A_0)(a) = [0, 3]$, and $f_{in}(p) = f_{out}(q) = [1, 1]$, for all other ports p, q in the system. A solution of this MC is shown at the bottom of the figure. In the solution, there is one instance of A_0, A_2, A_3 and two instances of A_1. Also, there are two copies of output port a of A_0, each connected to a copy of input port a of a different instance of A_1.

Remark 3 (The compatibility relation). We assume that the compatibility relation is given. Indeed, how compatibility of ports is derived is application specific, and out of the scope of this paper. Still, we give some examples of the range of expressiveness of the compatibility relation below. Note that the compatibility relation can be derived (possibly automatically) independently of the composition problem: this can be done once and used in many composition instances.

As in Example 2, compatibility could be simply based on names: two ports are compatible iff they have the same name (note that our framework is strictly more expressive than naming, see footnote 2). Another possibility is type matching: each port represents a type, and two ports are compatible iff their types match (for example, p_{int} matches q_{int} and q_{real}). Note that *polymorphic* types can be handled as well: for example, a module computing the length of a list of objects of any type can have an input port p which is compatible with all output ports

[2] There exist compatibility relations which can be expressed in our framework, i.e., as a binary relation $\mathsf{C} \subseteq \mathsf{P_{in}} \times \mathsf{P_{out}}$, yet cannot be reduced to naming. Section 3.2 contains an example of such a relation (Figure 3). The algorithm we present in Section 3 can handle any relation $\mathsf{C} \subseteq \mathsf{P_{in}} \times \mathsf{P_{out}}$.

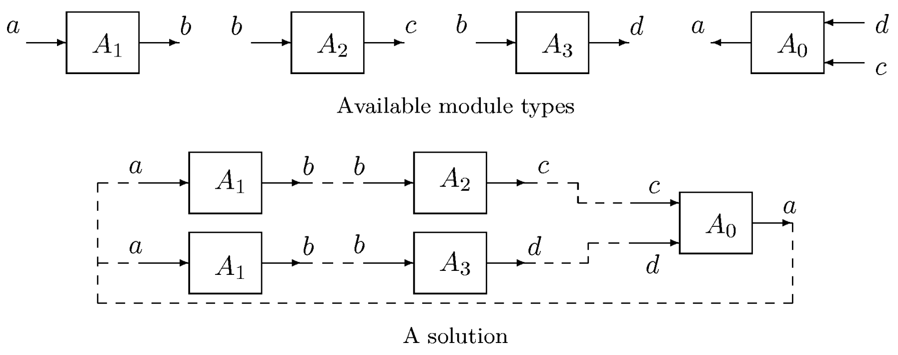

Fig. 2. A module composition problem and a solution.

representing lists.[3] Ports could also represent interfaces (i.e., sets of functions defined by their signatures), in which case port p is compatible with port q if the interface defined by p is a subset of the one defined by q.[4]

Ports could even be given semantics in terms of automata (as in [9] or [3]) which can express, for instance, the acceptable order of function calls in an interface[5] (e.g., "before you call *get()* you must call *init()*"). Here, compatibility can be defined as a *simulation* relation between automata and can be derived automatically by testing offline, for every pair of input/output ports (p, q), whether the automaton of p simulates the one of q.

Non-uniqueness of solutions, optimality. We observe that an MC does not necessarily have a unique solution. This is because of the following reasons:

– If a set of instances works, then doubling the instances will work too (provided the constraints on number of instances are met).
– Even if we require the number of elements in $\mathcal{I}$ to be minimal, there might be more than one minimal solutions.
– Even if we fix the number of instances, there might be multiple ways to instantiate the fan factors of some ports. For example, if $(p, q) \in \mathsf{C}$ and $f_{in}(p) \cap f_{out}(q) = [1, 2]$, then we could let $g_{in}(p) = g_{out}(q) = 1$ or $g_{in}(p) = g_{out}(q) = 2$, which amounts to creating one or two copies of each port and connecting them.

[3] When polymorphism is allowed, it is sometimes desirable, in case type p is compatible with many types $q_1, ..., q_k$, to select the type q_i that is "closest" to p. This can be done in our framework through the use of an objective function to be minimized, see below.

[4] We see the module having input p as the *caller* component and the module having output q as the *called-upon* component.

[5] For example, see the *RosettaNet* project (www.rosettanet.org), where every "Partner Interface Process" (i.e. every interaction between two entities) is standardized, its syntax given in XML and its semantics described by an automaton.

- Even if we fix both the number of module instances and port copies, there may be different compositions (i.e., different ways to "wire" the modules) possible.

The algorithm that we give in the following section finds an *optimal* solution to an MC, provided a solution exists. The solution is optimal with respect to a given (linear) *objective function* which represents the sum of the costs of creating instances of a module (an instance of module type A_i will add a cost c_i), creating copies of ports (a copy of port p will add a cost c_p), and connecting ports (connecting port p to port q will add a cost $c_{p,q}$). This is useful, since we can express the fact that some modules might be more costly than others, as well as express preferences/priorities in connecting pairs of ports.

3 Automated Module Composition

In this section we present the main results of our study of MC:

- The general MC is decidable and NP-complete.
- In a number of special cases (e.g., when the number of instances for each module type is known), MC can be solved in polynomial time.

The basic idea for proving decidability is to formulate an integer program, such that MC has a solution iff the integer program has a solution. Moreover, an optimal solution of the integer program with respect to a linear objective function corresponds to an optimal set of instances $\mathcal{I}$ for MC. Given $\mathcal{I}$, finding a composition X is straightforward. NP-hardness is proved by reducing the Knapsack problem to MC. We start by illustrating our approach with a simple example.

3.1 A Simple Example

Consider again the MC problem of Figure 2. Initially, let us assume that the fan factors of all ports are equal to 1 (we relax this assumption later). That is, $f_{in}(p) = f_{out}(q) = [1, 1]$, for all ports p, q in the system. We will also assume throughout this example that all costs are equal to 1, that is, a solution is optimal if it minimizes the sum of module instances, port copies and connections.

We begin by creating integer variables x_0, x_1, x_2, x_3, where x_i represents the number of instances of module A_i in the solution. From $\mathsf{N}_{\mathsf{inst}}$, we obtain the constraints $x_0 = 1$, $x_i \geq 0$, $i = 1, 2, 3$.

Viewing the port names as independent vectors, we can represent each module type by a simple linear expression on these vectors. Doing so, we get the expressions $-c - d + a$ for A_0, $-a + b$ for A_1, $-b + c$ for A_2, and $-b + d$ for A_3.

Now, it is easy to see that the requirement that the composition be closed is equivalent to the following constraint:

$$x_0 \cdot (-c - d + a) + x_1 \cdot (-a + b) + x_2 \cdot (-b + c) + x_3 \cdot (-b + d) = 0 \qquad (4)$$

or equivalently:

$$a \cdot (x_0 - x_1) + b \cdot (x_1 - x_2 - x_3) + c \cdot (-x_0 + x_2) + d \cdot (-x_0 + x_3) = 0 \quad (5)$$

Indeed, module A_0 will “contribute” x_0 copies of its output port named a (since the fan factors are set to 1), and each of these has to be connected to a copy of the input port of A_1 named a, therefore, x_0 must be equal to x_1 in order to have a closed composition.

Since a, b, c, d are independent vectors, equation (5) is equivalent to the set of equations:

$$x_0 - x_1 = 0 \quad (6)$$

$$x_1 - x_2 - x_3 = 0 \quad (7)$$

$$-x_0 + x_2 = 0 \quad (8)$$

$$-x_0 + x_3 = 0 \quad (9)$$

Thus, we ended up with a simple system of linear equations on integer variables, namely, equations (6)-(9), along with the initial constraint $x_0 = 1$. Solving the system (e.g., by Gaussian elimination), we find that it is infeasible. This means that there exists no solution to the above instance of MC.

We now relax the constraint that the fan factors of all ports should be 1. For example, suppose that $f_{out}(A_0)(a) = [0, 3]$. To model this, we create an additional variable y_0^a, which represents the fan-out factor of port a of A_0. Since each instance of A_0 can contribute from 0 up to 3 copies of a, we have the constraint:

$$x_0 \cdot 0 \leq y_0^a \leq x_0 \cdot 3 \quad (10)$$

Equation (4) now becomes:

$$y_0^a \cdot a + x_0 \cdot (-c - d) + x_1 \cdot (-a + b) + x_2 \cdot (-b + c) + x_3 \cdot (-b + d) = 0 \quad (11)$$

We can transform constraints (10) and (11) to an equivalent set of equations:

$$y_0^a - 3x_0 + s = 0 \quad (12)$$

$$y_0^a - x_1 = 0 \quad (13)$$

$$x_1 - x_2 - x_3 = 0 \quad (14)$$

$$-x_0 + x_2 = 0 \quad (15)$$

$$-x_0 + x_3 = 0 \quad (16)$$

where $s \geq 0$ is a *slack* variable that transforms the inequality constraint into an equality. Solving equations $x_0 = 1$ and (12)-(16) by Gaussian elimination, we obtain the optimal solution $x_0 = x_2 = x_3 = 1$, $y_0^a = x_1 = 2$, $s = 1$. This corresponds to the set of module instances shown in the bottom of Figure 2.

Once the set of module instances is determined, connecting the ports (i.e., finding a composition) is trivial: pick any unconnected copy of an input port and connect it to an unconnected copy of a compatible output port; repeat until all ports are connected. Notice that this procedure is guaranteed to terminate with all ports connected, since the above constraints ensure that a closed composition exists. Also notice that since connections are made at random, there might be more than one compositions possible.

3.2 Formulation as an Integer Programming Problem

Let $\mathcal{M} = \{A_0, A_1, ..., A_n\}$ be the set of module types. The integer program contains the following non-negative integer variables and constraints.

- x_i, $i = 0, ..., n$, represents the number of instances of module A_i. Let $\mathsf{N_{inst}}(A_i) = [l_i, u_i]$. For each x_i, we have the constraint
$$l_i \leq x_i \leq u_i. \tag{17}$$
If $u_i = \infty$, then the constraint becomes $l_i \leq x_i$.
- y_p, $p \in \mathsf{P_{in}}$, represents the total number of copies of input port p of a module A_i, in all instances of A_i. Letting $f_{in}(A_i)(p) = [a_p, b_p]$, each instance of A_i can "contribute" from a_p up to b_p copies of p. Therefore, for each p, we have the constraint
$$x_i \cdot a_p \leq y_p \leq x_i \cdot b_p. \tag{18}$$
If $b_p = \infty$, then the constraint becomes $x_i \cdot a_p \leq y_p$.
- z_q, $q \in \mathsf{P_{out}}$, represents the total number of copies of output port q of a module A_i, in all instances of A_i. Letting $f_{out}(A_i)(q) = [a_q, b_q]$, each instance of A_i can "contribute" from a_q up to b_q copies of q. Therefore, for each q, we have the constraint
$$x_i \cdot a_q \leq z_q \leq x_i \cdot b_q. \tag{19}$$
If $b_q = \infty$, then the constraint becomes $x_i \cdot a_q \leq z_q$.
- $w_{p,q}$, $(p,q) \in \mathsf{C}$, represents the number of connections between a copy of p and a copy of q. We relate the $w_{p,q}$ with the y_p and z_q variables with the following sets of constraints, for each $p \in \mathsf{P_{in}}$, $q \in \mathsf{P_{out}}$:
$$y_p = \sum_{(p,q)\in\mathsf{C}} w_{p,q} \quad \text{and} \quad z_q = \sum_{(p,q)\in\mathsf{C}} w_{p,q} \tag{20}$$
Constraints (20) ensure that a closed connection exists.

Proposition 4. *There is a solution to MC iff there exist non-negative integers x_i, for $i = 0, ..., n$, y_p, for $p \in \mathsf{P_{in}}$, z_q, for $q \in \mathsf{P_{out}}$, $w_{p,q}$, for $(p,q) \in \mathsf{C}$, such that constraints (17)-(20) are satisfied.*

Regarding optimality, it can be easily incorporated in the above formulation, by introducing an objective function to be minimized. Let $c_i \geq 0$ be the cost of an instance of module A_i, $c_p \geq 0$ be the cost of a copy of port p, $c_q \geq 0$ be the cost of a copy of port q, and $c_{p,q} \geq 0$ be the cost of connecting a copy of p to a copy of q. We define the objective function to be:

$$f(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{w}) = \textstyle\sum_{i=0,...,n} c_i x_i + \sum_{p\in\mathsf{P_{in}}} c_p y_p + \sum_{q\in\mathsf{P_{out}}} c_q z_q + \sum_{q\in\mathsf{P_{out}}} c_{p,q} w_{p,q},$$

where $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{w}$ are the vectors of variables $x_i, y_p, z_q, w_{p,q}$, respectively.

Then, we get an integer optimization problem:

minimize $f(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{w})$, subject to constraints (17)-(20),
and $x_i, y_p, z_q, w_{p,q}$ are non-negative integers.

It is worth observing that, first, if $l_i = 0$ for all $i = 0, ..., n$, then the trivial solution $x_i = y_p = z_q = w_{p,q} = 0$ is both feasible and optimal. Second, if for all i such that $l_i \geq 1$, for all $p \in in(A_i)$, $a_p = 0$, and for all $q \in out(A_i)$, $a_q = 0$, then the solution $x_i = l_i$, $y_p = z_q = w_{p,q} = 0$ is both feasible and optimal. Therefore, the interesting cases arise when there exists some i such that $l_i \geq 1$ and for some $p \in in(A_i)$, $a_p \geq 1$, or for some $q \in out(A_i)$, $a_q \geq 1$.

Eliminating redundancy in the integer program. In many cases, there may be a lot of redundancy in the above integer program, in the sense that the number of variables and constraints can be reduced. We have seen such a case in the example of section 3.1, where we used not all of the y_p or z_q variables and none of the $w_{p,q}$ variables. We now explain which variables and constraints can be eliminated and when. Our discussion on eliminating $w_{p,q}$ variables also shows that not all compatibility relations can be reduced to naming.

Eliminating y_p or z_q variables. In the case where $f_{in}(p) = [1, 1]$ for some input port p (i.e., the fan factor of p is 1), constraint (18) above becomes $x_i \cdot 1 \leq y_p \leq x_i \cdot 1$, or $y_p = x_i$. In that case, we can eliminate variable y_p and use x_i in its place. Similarly, for some output port q, we can eliminate variable z_q if $f_{out}(q) = [1, 1]$.

Eliminating $w_{p,q}$ variables. The reason why we did not have to use any $w_{p,q}$ variables in the example of section 3.1 is that we assumed that the compatibility relation of that example can be reduced to port naming, such that two ports are compatible iff they have the same name. This cannot always be done. For example, look at Figure 3. For the compatibility relation on the left, we cannot find a naming that works. Indeed, assume we give p_1 the name a. Then we have to name q_1 and q_2 by a too. Since q_2 is named a and is compatible with p_2, we have to name p_2 by a as well. But now p_2 and q_1 have the same name, even though they are not compatible. On the other hand, for the compatibility relation on the right of the figure, we can find a naming (just name all ports a) that works.

Formally, we say that a compatibility relation $\mathsf{C} \subseteq P_{in} \times P_{out}$ *has the Z property* if for any $p_1, p_2 \in P_{in}$ and any $q_1, q_2 \in P_{out}$

$$(p_1, q_1) \in \mathsf{C} \wedge (p_1, q_2) \in \mathsf{C} \wedge (p_2, q_2) \in \mathsf{C} \;\Rightarrow\; (p_2, q_1) \in \mathsf{C}. \Rightarrow (p_2, q_1) \in \mathsf{C}.$$

Then, we can show the following:

Lemma 5. *We can find a function* $\mathsf{name} : P_{in} \cup P_{out} \to N$ *such that* $\forall p \in P_{in}, q \in P_{out}, \mathsf{name}(p) = \mathsf{name}(q) \Leftrightarrow (p, q) \in \mathsf{C}$*, iff* C *has the Z property.*

Thus, if C has the Z property, we can partition $\mathsf{P_{in}}$ and $\mathsf{P_{out}}$ into disjoint subsets $\mathsf{P_{in}} = \mathsf{P_{in}}^0 \cup \mathsf{P_{in}}^1 \cup \cdots \mathsf{P_{in}}^k$, $\mathsf{P_{out}} = \mathsf{P_{out}}^0 \cup \mathsf{P_{out}}^1 \cup \cdots \mathsf{P_{out}}^k$, such that $p \in \mathsf{P_{in}}^i$ iff $\mathsf{name}(p) = i$ and $q \in \mathsf{P_{out}}^i$ iff $\mathsf{name}(q) = i$. Then, we can eliminate variables

$w_{p,q}$, and replace the set of constraints (20)-(20) by one constraint as follows, for each $i = 0, ..., k$:

$$\sum_{p \in \mathsf{P_{in}}^i} y_p = \sum_{q \in \mathsf{P_{out}}^i} z_q. \tag{21}$$

In fact, even if C does not have the Z property, "parts" of it might possess it. In such a case, we can perform the above optimization for these parts, eliminating the corresponding $w_{p,q}$ variables.

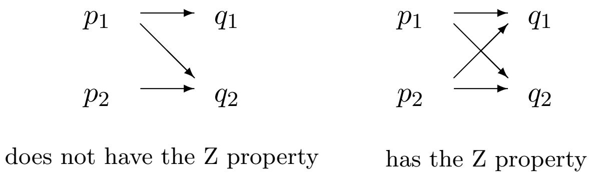

Fig. 3. Illustration of the Z property.

3.3 NP-Completeness

Theorem 6. *The Module Composition Problem is NP-Complete.*

Proof: First we show that MC is in NP. It has been shown that any integer programming problem can be transformed in polynomial time into a 0-1 integer programming problem, that is, where all variables take the values 0 or 1 (e.g., see chapter I.5 of [13]). Since there is a polynomial number of 0-1 variables, say k, there are 2^k possible solutions. For each solution, it can be checked in polynomial time whether it is feasible. If a feasible solution is found, the composition can be computed in linear time in the number of ports, as discussed in section 3.5. Therefore, MC is in NP.

To prove that MC is NP-hard, we reduce a variant of the *Knapsack problem* (KP) to MC. KP is defined as follows. We are given a set of numbers $S = \{c_1, ..., c_n\} \subset N$, and some $M \in N$. We are asked whether there exists a subset $S' \subseteq S$, such that $\sum_{x \in S'} x = M$. The problem is known to be NP-complete [4].

We reduce KP to MC as follows. Let $A_0, A_1, ..., A_n$ be module types, where:

- $\mathsf{N_{inst}}(A_0) = \{1\}$ and $\mathsf{N_{inst}}(A_i) = \{0, 1\}$, for each $i = 1, ..., n$.
- A_0 has no output ports, M input ports, and $f_{in}(A_0)(p) = [1, 1]$ for each input port p of A_0.
- For each $i = 1, ..., n$, A_i has no input ports, c_i output ports, and $f_{out}(A_i)(q) = [1, 1]$ for each output port q of A_i.
- Every input port is compatible with every output port.

It is easy to see that KP has a solution iff the above MC has a solution. ■

3.4 Special Cases of Polynomial Complexity

Although the worst-case complexity of the general MC is exponential, there are many interesting special cases where MC can be solved in polynomial time. This is when the integer program of section 3.2 can be transformed to an equivalent *network flow problem*, which can be solved in polynomial time using a variety of algorithms (e.g., see [1]). We now identify some of these cases.

Special case I: known number of module instances. Suppose that the number of module instances are known, that is, $l_i = u_i$, for all $i = 0, ..., n$. In this case the problem is equivalent to a *min-cost flow problem*, in a network defined as follows. There is one node for each input port p, one node for each output port q, a source node s and a sink node t. There is a directed link from s to each node p, and the flow along this link corresponds to y_p. There is a directed link from each node q to t, and the flow along this link corresponds to z_q. There is a directed link from p to q, for each $(p, q) \in \mathsf{C}$, and the flow along this link corresponds to $w_{p,q}$. There is a directed link from t to s. Finally, there are lower and upper *capacity bounds* on the links (s, p) and (q, t), corresponding to constraints (18) and (19). The objective function is similar to the one given in section 3.2, namely, minimize $(\sum_{p \in \mathsf{P_{in}}} c_p \cdot y_p) + (\sum_{q \in \mathsf{P_{out}}} c_q \cdot z_q)$.

A number of min-cost flow algorithms can be used in this case. For example, the *enhanced capacity scaling algorithm* has complexity $O((e \log v) \cdot (e + v \log v))$, where v is the number of nodes and e the number of edges. Notice that in the network construction above, we have $v = |\mathsf{P_{in}}| + |\mathsf{P_{out}}| + 2$ and $e = |\mathsf{P_{in}}| + |\mathsf{P_{out}}| + |\mathsf{C}|$.

Special case II: modules with single input/output ports. Suppose that each module type has at most one input port and at most one output port, that is, $|in(A_i)| \leq 1$ and $|out(A_i)| \leq 1$, for all $i = 0, ..., n$, and $f_{in}(p) = f_{out}(q) = [1, 1]$, for all ports p, q. Then, the problem can be again transformed into a min-cost network flow problem, where the network is defined as follows. There are two nodes, s_i and t_i, for each $i = 0, ..., n$. There is a directed link from s_i to t_i with capacity $x_i \in [l_i, u_i]$. If A_i has an input port p and A_j has an output port q, such that $(p, q) \in \mathsf{C}$, then there is a directed link from t_i to s_j. The objective function is similar to the one given in section 3.2, namely, minimize $\sum_{i=0,...,n} c_i \cdot x_i$.

Again, any min-cost flow algorithm can be used. In this case, the number of nodes is $2n$ and the number of edges is $n + |\mathsf{C}|$.

3.5 Overall Algorithm for the Module Composition Problem

Based on the results of the previous sections, the overall algorithm to solve MC has three stages.

In Stage 1, we try to find an optimal solution to the integer optimization program given in section 3.2. If no feasible solution exists, then MC has no solution (by proposition 4). If an optimal solution is found, we proceed to stage 2.

In Stage 2, we know the x_i's, y_p's, z_q's and $w_{p,q}$'s. From these, we can build a set of module instances $\mathcal{I}$ as follows. There will be x_i instances of module A_i. For each input port p of A_i, there will be a total of y_p copies of p in all instances of A_i. How many copies are assigned to each instance does not matter, as long as there are between a_p and b_p copies of p in each instance. Therefore, we can assign copies to instances at random, making sure the above constraints are met.[6] Similarly, we assign a total of z_q copies of port q.

Finally, in Stage 3, we build the composition as follows. Initially all copies of all ports are disconnected. We repeat the following, until all copies of all ports are connected: pick any unconnected copy of an input port and connect it to an unconnected copy of a compatible output port.[7]

Obviously, the hard part is Stage 1: solving the integer program. For this, the following strategy can be employed. First, check whether the problem instance belongs to one of the special cases given in section 3.4. These tests can be done automatically. If the problem belongs to a special case, apply a min-cost flow algorithm of choice.

If the problem does not belong to any special case, apply a heuristic of choice. There is a large number of heuristics for integer optimization problems developed in the literature. We just mention a few here, referring the reader to [1,13] for details.

1. Apply *Lagrangian relaxation*, by removing the constraints (17)-(19) from the feasible region and adding the following term to the objective function:

$$\begin{aligned} g(\boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{w}) = {} & (\textstyle\sum_i \lambda_i(u_i - x_i) + \mu_i(x_i - l_i)) \\ & + (\textstyle\sum_p \lambda_p(b_p x_i - y_p) + \mu_p(y_p - a_p x_i)) \\ & + (\textstyle\sum_q \lambda_q(b_q x_i - z_q) + \mu_q(z_q - a_q x_i)), \end{aligned}$$

   where the λ's and μ's are the Lagrange multipliers. By doing this, we end-up with a min-cost network flow problem, which can be iteratively solved to obtain a strict bound on the optimal solution and perhaps also find one.
2. Use *branch-and-bound* to iteratively solve *linear relaxations* of the integer program (i.e., where the variables are not restricted to be integers), and converge to an integer solution.
3. Use a *cutting-plane* algorithm to do the above.

4 Summary, Discussion, and Perspectives

We have introduced an abstract problem of module composition and showed how its solution can be computed effectively. We believe that the problem arises in many instances in practice, in particular in the development of large software

[6] By proposition 4, it is guaranteed that we can assign copies such that the above constraints are met.

[7] Again, by proposition 4, this procedure is guaranteed to terminate with all ports connected.

systems or the deployment of dynamic systems. We also believe that an automated procedure can result to significant cost savings, both by speeding-up the assembly process and by facilitating component re-use.

One question about our work might be, what is the behavior of the composite module and how is it ensured that this behavior is correct? It has been our conscious decision not to associate detailed semantics with our model. We could, for instance, specify the behavior of a module by an automaton and define the composition of modules in terms of automata composition. This would permit us to express desirable properties of the global system in a formal specification language (e.g., temporal logic) and use formal verification techniques (e.g., model checking) to check whether the property holds or not. Instead of doing this, we opted for a more "light-weight" approach, where modules are "black boxes" with no internal semantics and composition only guarantees local port compatibility. The motivations behind our choice have been the following:

1. We care about scalability of our method. State-of-the-art formal verification techniques cannot cope with more than in the order of tens of modules, thus are not sufficient for large systems.
2. We argue that (local) port compatibility can be made as expressive as necessary, thereby capturing the intended semantics and ensuring that systems that "compose well" are also "correct by construction" to a certain degree. (See Remark 3 for examples of compatibility relations.)
3. Even in the case where the semantics of ports are not strong enough to capture all necessary global properties, we believe that our algorithms are still useful as a "first pass" of module composition, which is automated, therefore fast. After this first pass, the designer can examine the result using more sophisticated techniques in order to verify all properties of interest. The first pass is still useful since it has relieved the designer from the tedious process of composing a large number of components "by hand".

Regarding future work, we intend to apply our methods in the particular context of embedded software development for cars. In order to make the approach more usable, we plan to develop techniques to automate the resolution of the compatibility relation: deciding whether two ports are compatible can be done automatically based on a set of rules such name matching, type matching, matching of sampling rates, etc. Finally, we intend to investigate the power of local port compatibility with respect to global properties of the composite system.

References

1. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows – Theory, Algorithms and Applications.* Prentice-Hall, 1993.
2. K. Butts, D. Bostic, A. Chutinan, J. Cook, B. Milam, and Y. Wang. Usage scenarios for a model compiler. In *EMSOFT'01.* Springer, LNCS 2211, 2001.
3. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *EMSOFT'01.* Springer, LNCS 2211, 2001.

4. M. Garey and D. Johnson. *Computers and Intractability: a guide to the theory of NP-completeness.* Freeman, 1979.
5. D. Jackson. Automating first-order relational logic. In *Foundations of Software Engineering*, 2000.
6. E. Kiciman, L. Melloul, and A. Fox. Towards zero-code service composition. In *8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, 2001.
7. C. Kloukinas and V. Issarny. Automating the composition of middleware configurations. In *Automated Software Engineering*, 2000.
8. C. Kloukinas and V. Issarny. SPIN-ning Software Architectures: A Method for Exploring Complex Systems. In *Working IEEE/IFIP Conference on Software Architecture (WICSA2001)*, 2001.
9. E.A. Lee and Y. Xiong. System-level types for component-based design. Technical report, Technical Memorandum UCB/ERL M00/8, University of California, Berkeley, 2000.
10. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM TOPLAS*, 6(1), January 1984.
11. T. Margaria and B. Steffen. Backtracking-free design planning by automatic synthesis in metaframe. In *Fundamental Aspects of Software Engineering*, 1998.
12. W. Milam and A. Chutinan. Model composition and analysis challenge problems. Technical report of Mobies project. Available at: http://vehicle.me.berkeley.edu/mobies, 2001.
13. G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, 1988.
14. S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *11th World Wide Web Conference (Web Engineering Track)*, 2002.
15. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
16. B. Steffen, T. Margaria, and M. von der Beeck. Automatic synthesis of linear process models from temporal constraints: An incremental approach. In *ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97)*, 1997.
17. S. Tripakis. Automated composition of module chains. In *ETAPS'02 Workshop on Software Composition.* Volume 65, issue 4 of ENTCS, Elsevier, 2002.
18. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, March 2000.