Automated Module Composition

Stavros Tripakis*

Abstract

We define an abstract problem of module composition, where modules are seen as black boxes with input and output ports and the objective is to instantiate a number of modules and connect their ports, in order to obtain a target module. A compatibility relation defines which ports can be connected to each other. Constraints are also 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.

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

Keywords: Modules, Components, Object composition, Integer Programming, NP-complete.

1 Introduction

In this paper, we are interested in automating the composition of modules. We view modules as "black boxes" with input and output ports. No semantics are associated with the modules or ports. Composition consists in instantiating a number of modules and connecting their ports. A binary *compatibility* relation models which port can be connected to which other. Constraints are also imposed on the number of instances of each module and the number of copies of each port in an instance. A linear objective function can be given to minimize the total cost of module instances and port copies.

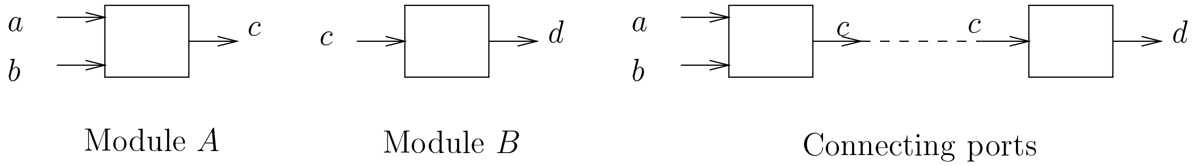
To motivate the reader, let us give an example. Figure 1(a) shows two modules A and B. Module A has two input ports named a and b and one output port named c. Module B has one output port named d, and one input port named c. Ports with the same name are compatible. The result of connecting two ports is shown at the top-right of the figure. Once two ports are connected, they can be "hidden", as indicated in part (b) of Figure 1. The connected modules A and B can be seen as an *implementation* of a module C.

The question then is: given a set of *available* modules and a *target* module, how can we connect the available modules in order to obtain an implementation of the target module? In fact, as part (c) of Figure 1 shows, this problem can be formulated in a more general way: given a set of modules (which includes the available and target modules), find a composition which includes exactly one instance of the target module, and implements a module with no ports (that is, a composition where all ports are connected).

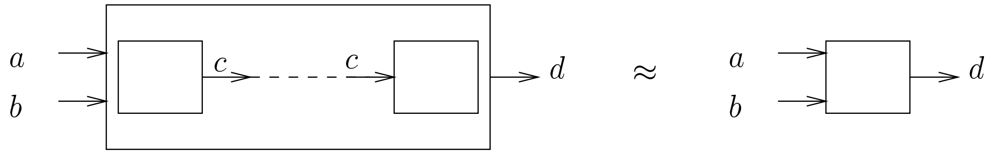
This abstract problem has a number of practical applications. For example, in a software engineering context, modules and ports might represent classes and interfaces. In a circuit layout context, they might represent chips and wires.

A particular application (which has motivated this work) comes from the domain of automotive embedded system design and implementation. 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 [6]). The interest behind the problem arises from the large number of legacy components developed by different software groups. These components are used initially for simulation and later for code generation. To facilitate re-use, they are stored in a common repository. The problem is to choose the right components and compose them. This is a tedious work to be done manually (this is currently the practice, though), due to the large number of components (hundreds or thousands). See [6] for details. It is obvious that an automatic composition would greatly reduce the software production cost.

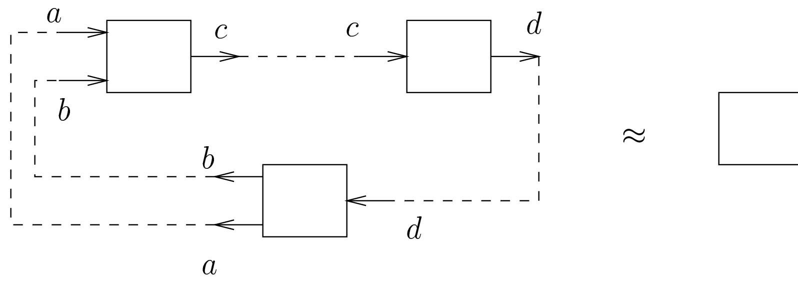
*University of California, Berkeley, and VERIMAG, France. E-mail: stavros@eecs.berkeley.edu.



(a)



(b)



(c)

Figure 1: Module Composition

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

2 Module Composition Problem Definition

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

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

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

A *module type* is a tuple $(In, Out, f_{in}, f_{out})$, where $In \subseteq P_{in}$, $Out \subseteq P_{out}$, $f_{in} : In \rightarrow 2^N$ and $f_{out} : Out \rightarrow 2^N$. 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 can there be in an instance of the module. Each copy of p will be connected to a single copy of another port. The meaning of f_{in} is symmetric. For simplicity, we require that $f_{in}(p)$ and $f_{out}(p)$ are non-negative integer intervals, e.g., $[0, 5]$, $[1, 1]$, and so on.

Given a module type $A = (In, Out, f_{in}, f_{out})$, 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 $q \in Out$, $g_{out}(q) \in f_{out}(q)$. That is, g_{in} and g_{out} fix the fan-in and fan-out factors of each port to specific numbers, provided these numbers are allowed by f_{in} and f_{out}, respectively. We say that the module instance has $g_{out}(q)$ copies of port q, $g_{in}(p)$ copies of port p, and so on.

We will use the following notation. 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.

From now on, we assume that for each module type, its set of input and output ports are disjoint. We also 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:

$$\forall A \in \mathcal{I}, \forall p \in in(A), |\{(A, p, _, _) \in X\}| = g_{in}(A)(p) \quad (1)$$
$$\forall A \in \mathcal{I}, \forall q \in out(A), |\{(_, _, A, q) \in X\}| = g_{out}(A)(q) \quad (2)$$

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

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

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 $NumInst : \mathcal{M} \rightarrow 2^N$, where $\mathcal{M}$ is the set of module types and for each module type A in $\mathcal{M}$, $NumInst(A)$ is an interval. For example, if $NumInst(A) = [1, 10]$, this means that at most 10 instances of A are available and at least 1 instance should be used, whereas if $NumInst(A) = N$, then an arbitrary number of instances of A can be used, possibly none. We say that a set of module instances $\mathcal{I}$ *respects* $NumInst$ 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 $n_A \in NumInst(A)$.

We are now ready to state the module composition problem.

Module Composition Problem (MCP). Given

(a) disjoint sets of input and output ports P_{in}, P_{out},

(b) a compatibility relation $C \subseteq P_{in} \times P_{out}$,

(c) a set of module types $\mathcal{M}$, and

(d) constraints on the number of module instances $NumInst : \mathcal{M} \to 2^N$,

find

(e) a set of module instances $\mathcal{I}$ that respects $NumInst$, and

(f) a composition X on $\mathcal{I}$ that respects C and is closed.

Example. Consider the set of module types shown at the top of Figure 2. We have named the ports so that ports are compatible iff they have the same name.[1] Exactly one instance of module A_0 should be present in the composition, while there are no constraints on the number of instances of A_1, A_2, A_3. In other words, $NumInst(A_0) = \{1\}$ and $NumInst(A_1) = NumInst(A_2) = NumInst(A_3) = N$. 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. Two solutions of this MCP are shown at the bottom of the figure. In both solutions, there is one instance of A_0, A_2, A_3 and two instances of A_1. Also, notice that 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.

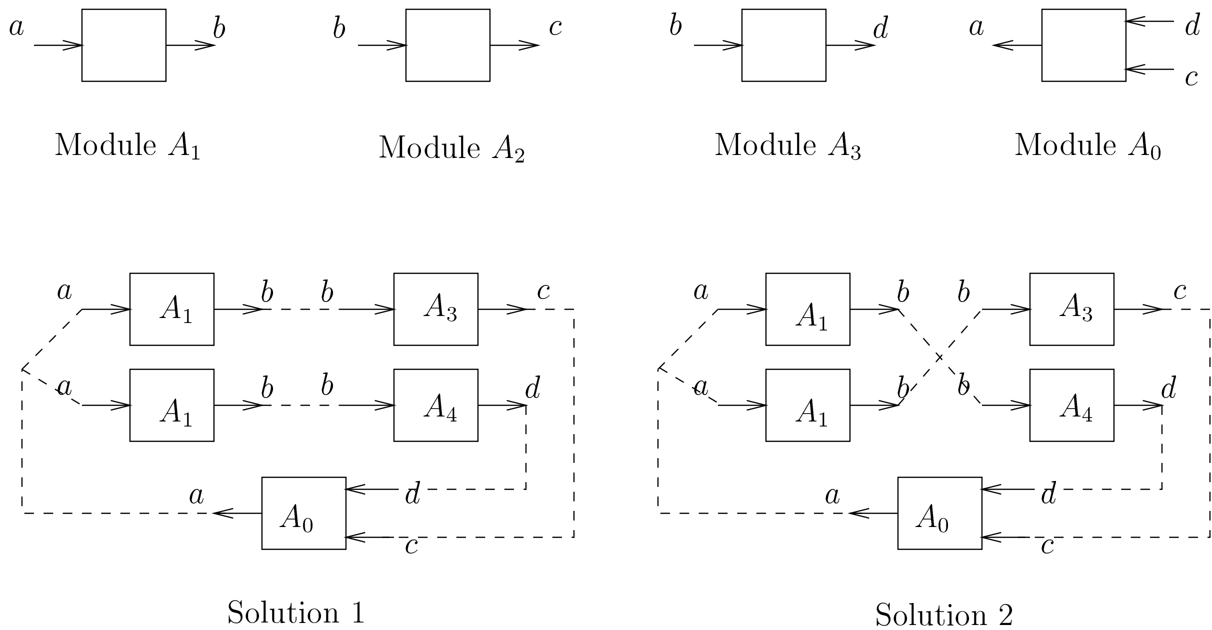


Figure 2: A module composition problem and two possible solutions

Non-uniqueness of solutions, Minimality. We observe that an MCP 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.

[1]We note that there are compatibility relations which cannot be reduced to naming (see section 3.2.1). The algorithm we present in section 3 can handle all compatibility relations. This example is simple for the purpose of illustration.

- Even if we fix the number of instances, there might be multiple ways to instantiate the fan factors of some ports. For example, if $(p, q) \in C$ 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.
- 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. For example, this is the case of the two solutions shown in Figure 2.

The algorithm that we give in the following section finds a *minimal* solution to an MCP, provided a solution exists. The solution is minimal in the sense that the sum of the number of module instances and the number of port copies is minimal. The algorithm can be modified in a straightforward way to account for more sophisticated notions of minimality, for example, where each module type A_i is associated with a *cost* c_i, each port p is associated with a cost c_p, and the objective is to minimize the total cost of the instance set $\mathcal{I}$. Details are given in section 3.2.

3 Module Composition Problem Solution

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

- The general MCP is NP-complete.
- In a number of special cases, MCP can be solved in polynomial time.

The basic idea for proving decidability is to formulate an integer program, such that the MCP 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 a minimal set of instances $\mathcal{I}$ for the MCP. Given $\mathcal{I}$, finding a composition X is straightforward. NP-hardness is proved by reducing the Knapsack problem to the MCP.

We start by illustrating our approach with a simple example.

3.1 A simple example

Consider again the MCP 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 begin by creating integer variables x_0, x_1, x_2, x_3, where Variable x_i represents the number of instances of module A_i in the solution. From *NumInst*, we obtain the constraints $x_0 = 1$, $x_i \geq 0$, $i = 1, 2, 3, 4$.

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:

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

or equivalently:

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

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:

$$x_0 - x_1 = 0 \tag{6}$$

$$x_1 - x_2 - x_3 = 0 \tag{7}$$

$$-x_0 + x_2 = 0 \tag{8}$$

$$-x_0 + x_3 = 0 \tag{9}$$

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 MCP.

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:

$$x_0 \cdot 0 \leq y_0^a \leq x_0 \cdot 3 \tag{10}$$

Equation 4 now becomes:

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

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

$$y_0^a - 3x_0 + s = 0 \tag{12}$$

$$y_0^a - x_1 = 0 \tag{13}$$

$$x_1 - x_2 - x_3 = 0 \tag{14}$$

$$-x_0 + x_2 = 0 \tag{15}$$

$$-x_0 + x_3 = 0 \tag{16}$$

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 minimal 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. This is the case in the example of Figure 2.

3.2 Integer Program Formulation

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 $NumInst(A_i) = [l_i, u_i]$. 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, $p \in P_{in}$, 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, $q \in P_{out}$, 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 $x_i \cdot a_q \leq z_q$.

- $w_{p,q}$, $(p,q) \in C$, 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 $p \in P_{in}$, $q \in P_{out}$:

$$y_p = \sum_{(p,q) \in C} w_{p,q} \tag{20}$$

$$z_q = \sum_{(p,q) \in C} w_{p,q} \tag{21}$$

  Constraints 20 and 21 will ensure that there exists a closed connection.

Proposition 3.1 *There is a solution to the MCP iff there exist non-negative integers x_i, for $i = 0, ..., n$, y_p, for $p \in P_{in}$, z_q, for $q \in P_{out}$, $w_{p,q}$, for $(p,q) \in C$, such that constraints 17-21 are satisfied.*

It is worth making the following observations:

- If for all $i = 0, ..., n$, $l_i = 0$, then the trivial solution $x_i = y_p = z_q = w_{p,q} = 0$ for all i, p, q, is both feasible and minimal.
- 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 $x_i = l_i$, $y_p = z_q = w_{p,q} = 0$ for all i, p, q, is both feasible and minimal.

Therefore, the interesting cases are 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 $q \in out(A_i)$, $a_q \geq 1$.

Regarding minimality, it can be easily incorporated in the above formulation, by introducing an objective function to be minimized, thus turning the integer feasibility problem into an integer optimization problem:

minimize $(\sum_{i=0,...,n} c_i \cdot x_i) + (\sum_{p \in P_{in}} c_p \cdot y_p) + (\sum_{q \in P_{out}} c_q \cdot z_q)$
s.t. constraints 17-21 are satisfied, and $x_i \geq 0, y_p \geq 0, z_q \geq 0$ and integer,

where c_i is the cost of an instance of module A_i, c_p is the cost of a copy of port p, and c_q is the cost of a copy of port q.

3.2.1 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.[2]

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.

[2]This section may be skipped.

does not have the Z property   has the Z property

Figure 3: Illustration of the Z property

Formally, we say that a compatibility relation $C \subseteq P_{in} \times P_{out}$ *has the Z property* if

$$\forall p_1, p_2 \in P_{in}, q_1, q_2 \in P_{out}, (p_1, q_1) \in C \wedge (p_1, q_2) \in C \wedge (p_2, q_2) \in C \Rightarrow (p_2, q_1) \in C. \tag{22}$$

Then, we can show the following:

Lemma 3.1 *We can find a function name* : $P_{in} \cup P_{out} \rightarrow N$ *such that* $\forall p \in P_{in}, q \in P_{out}, name(p) = name(q) \Leftrightarrow (p,q)C$*, iff* C *has the Z property.*

Thus, if C has the Z property, we can partition P_{in} and P_{out} into disjoint subsets $P_{in} = P_{in}^0 \cup P_{in}^1 \cup \cdots P_{in}^k$, $P_{out} = P_{out}^0 \cup P_{out}^1 \cup \cdots P_{out}^k$, such that $p \in P_{in}^i$ iff $name(p) = i$ and $q \in P_{out}^i$ iff $name(q) = i$. Then, we can eliminate variables $w_{p,q}$, and replace the set of constraints 20-21 by one constraint as follows, for each $i = 0, ..., k$:

$$\sum_{p \in P_{in}^i} y_p = \sum_{q \in P_{out}^i} z_q. \tag{23}$$

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.

3.3 NP-Completeness

Theorem 3.1 *The Module Composition Problem is NP-Complete.*

Proof: First we show that MCP is in NP. We observe that 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 [7]). 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, MCP is in NP.

To prove that MCP is NP-hard, we reduce a variant of the *Knapsack problem* (KP) to the MCP. 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 [3].

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

- $NumInst(A_0) = \{1\}$ and $NumInst(A_i) = \{0, 1\}$, 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 MCP has a solution. ■

3.4 Special Cases of MCP with Polynomial Complexity

Although the worst-case complexity of the general MCP is exponential, there are many interesting special cases where the MCP 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 this cases.

3.4.1 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 $(p, q) \in C$, 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 $(\sum_{p \in P_{in}} c_p \cdot y_p) + (\sum_{q \in P_{out}} c_q \cdot z_q)$.

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

3.4.2 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 $(p, q) \in C$, 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 $\sum_{i=0,\ldots,n} c_i \cdot x_i$.

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

3.5 General Algorithm for the Module Composition Problem

Based on the results of the previous sections, the general algorithm to solve the MCP has two 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, the MCP has no solution (by proposition 3.1). 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.[3] Similarly, we assign a total of z_q copies of output port q.

  Finally, we build the composition as follows. Initially all copies of all ports are unconnected. 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.[4]

[3] By proposition 3.1, it is guaranteed that we can assign copies such that the above constraints are met.
[4] Again, by proposition 3.1, this procedure is guaranteed to terminate with all ports connected.

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, 7] 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:

$$(\sum_i \lambda_i(u_i - x_i) + \mu_i(x_i - l_i)) + (\sum_p \lambda_p(b_p x_i - y_p) + \mu_p(y_p - a_p x_i)) + (\sum_q \lambda_q(b_q x_i - z_q) + \mu_q(z_q - a_q x_i)),$$

   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 by luck 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 Conclusions

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 software design of large 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.

Our work is related to *Architecture Description Languages* (e.g., see [2]) and other component models (e.g., see [8]). However, our work differs from the above in that we are interested in obtaining an efficient automatic composition procedure, versus offering a design methodology. Another component-modeling language is Alloy [4], for which it is possible to perform automatic analysis in a related 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 within that framework.

References

[1] R.K. Ahuja, T.L. Magnanti and J.B. Orlin. *Network Flows – Theory, Algorithms and Applications*. Prentice-Hall, 1993.

[2] Links to Architecture Description Languages, at: http://www.sei.cmu.edu/architecture/adl.html.

[3] M.R. Garey and D.S. Johnson, *Computers and Intractability – A Guide to the theory of NP-Completeness*, Freeman, 1979.

[4] D. Jackson, *Alloy: A Lightweight Object Modelling Notation*, 2000.

[5] D. Jackson, *Automating First-Order Relational Logic*. FSE'00.

[6] W. Milam and A. Chutinan, Ford Motor Company, *Model Composition and Analysis Challenge Problems* white paper. Available at: http://vehicle.me.berkeley.edu/mobies.

[7] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*. Wiley, 1988.

[8] R. van Ommering, F. van der Linden, J. Kramer and J. Magee, *The Koala Component Model for Consumer Electronics Software*. IEEE Computer, March 2000.