Optimization problems arising in software architecture

Leo Liberti

LIX, École Polytechnique, F-91128 Palaiseau, France
Email:liberti@lix.polytechnique.fr

November 15, 2006

Abstract

Software architecture is the process of planning and designing a large-scale software, and a fundamental industrial discipline within the field of software engineering. The current body of knowledge in software architecture is a mixture of personal experience and precise methods. In this paper we move a step towards the formalization of this discipline by describing some optimization problems that arise in the field.

1 Introduction

Software architecture involves the detailed planning of several aspects of a large-scale software, such as the flowsheet, the data model, the data exchange process, the modularization, the interfaces between the modules, and the resource utilization. The architecture of a software is intertwined with its engineering cycle, which also includes requirement analysis, commercial offers, legal contracts, documentation, social interactions, project planning, integration, verification, validation, deployment, maintenance and decommissioning [Som07]. Of these processes, verification and validation rely on a significant proportion of mathematics-based methods for their dispatch **cite**, whereas the rest are more in a grey area between experience and science. Project planning is often carried out by using scheduling tools **cite**. Some Computer-Aided Software Engineering (CASE) tools also exist for software architecture, such as UML **cite**. Producing a sound software architecture must necessarily rely on a creative process which may well be impossible to formalize; however, every effort should be made to ensure that mathematics-based tools and precise methods are used whenever possible. The object of this paper is to move a step in this direction by formalizing some problems of combinatorial nature which may arise during the construction phase of the architecture of a software.

CASE tools based on UML (or UML-like) offer a variety of diagrams, such as the activity diagram, the class diagram and so on **cite**, as a modelling device. These diagrams are supposed to clarify thought in the eye of the beholder (the software architect), to communicate ideas effectively, and sometimes even to produce code **cite**. We can convey the information in UML-type diagrams by means of graph theory: diagrams may also be represented as either directed or undirected graphs where the diagram objects are the vertices and the links are either (undirected) edges or (directed) arcs. Different object or link types may be formalized as colours, and quantitative indices as weights on vertices and edges/arcs. This representation allows the system architect to perform operations on the diagrams by means of mathematical programming techniques and combinatorial optimization algorithms. Typically, the system architecture design process starts from the analysis of requirements, and attempts to add details and synthesize relations in an iterative manner; this stops when the system architect is satisfied with the level of details present in the architecture. Whilst a part of this process is creative and thus appears impossible to formalize precisely at the current state of affairs, another part involves a diagram reformulation (such as clustering certain entities into subsystems or interfaces, or simplifying the flow path) which can clearly be cast as a series of precisely formulated combinatorial optimization problems.

1.1 The optimization techniques used in this paper

Most (if not all) optimization problems can be precisely cast as *mathematical programs*, i.e. mathematical problem formulations consisting of a set of *numerical parameters* (whose value is known and defines the "problem data"), a set of *decision variables*, whose value will be the output of the solution algorithm, one or more *objective functions* to be minimized or maximized by the solution(s) and a set of *constraints* which the solution(s) should satisfy. Objective functions and constraints are expressed in terms of mathematical functions / relations among parameters and variables. The set of solutions which satisfy the problem constraints is called the *feasible region*. The *size* of a formulation is given by the number of decision variables summed to the number of constraints.

1.1 Example
As an example, we cast a typical combinatorial optimization problem as a mathematical program.

> Minimum Vertex Cover. *Given an undirected graph $G = (V, E)$, find a vertex subset $C \subset V$ of minimum cardinality such that for each edge is incident to at least one vertex in C.*

The parameters of this problem are V and E. As decision variables, we select $x : V \to \{0, 1\}$ to express membership in C ($x(v) = 1$ if $v \in C$ and 0 otherwise). We seek to minimize the cardinality of C: $\min_x \sum_{v \in V} x(v)$. *The edge adjacency condition can be expressed as follows:* $\forall \{u, v\} \in E\ x(u) + x(v) \geq 1$.

The reason for casting all (or most) combinatorial problems described in this paper as mathematical programs is that there exists several very efficient computer programs for solving mathematical programming formulations exhibiting special properties (such as linearity or convexity in the objective function and feasible region, with continuous or integer-valued variables) [FG02, ILO02]. Furthermore, it is sometimes possible to apply symbolic manipulation techniques to mathematical programs to alter their linearity/convexity properties, e.g. formulations with quadratic terms consisting of binary variables may be recast to linear formulations of higher size [For60, Lib04].

2 The network view

In this section we shall present a unified diagram (which we call the *network view*) to help create a software architecture. A network view $\mathcal{N}$ consists in a sequence of directed graphs $G_i = (V_i, A_i)$ (with $i \leq |\mathcal{N}|$) and procedures $P_i : G_{i-1} \to G_i$. The procedures P_i are composed of two parts P'_i, P''_i where P'_i (the *reformulation*) can be formalized precisely whilst P''_i (the *analysis*) relies on a creative process on the part of the software architect. Depending on how P_i is implemented, many different network views may arise. The set of all the network views should be a significant help to the software break-down and architecture design processes. Note that the construction of G_0 is also a creative process.

2.1 Vertex types

For each network view $\mathcal{N}$ and each $i \leq |\mathcal{N}|$, the vertex set V_i represents entities of different types (formalized as colours in the graph), such as (but not limited to):

- program functionality
- program module
- test
- data source

- server
- workstation
- hardware unit
- person
- team.

Vertices may be weighted by several quantitative indices, such as the security/safety/reliability/importance of a program functionality or module and so on.

2.2 Arc types

For each network view $\mathcal{N}$ and each $i \leq |\mathcal{N}|$, the arc set A_i represents relations of different types (formalized as colours in the graph), such as (but not limited to):

- program flow
- data exchange
- assignment
- membership
- inheritance.

Arcs may be weighted by several quantitative indices, such as the amount of data to be exchanged, the speed of the communication line and so on.

2.3 Iterative procedure

Each procedure $P_i : G_{i-1} \to G_i$ follows the general schema below:

1. Apply P_i' (reformulation) as described in Section 3 below.
2. Apply P_i'' (analysis): for each $v \in V_{i-1}$ consider a graph $H = (\bar{V}, \bar{A})$ which describes the properties/functions of v in more details. Let $V_i \to V_i \cup \bar{V}$ and $A_i \to A_i \cup \bar{A}$.
3. If a termination condition is verified, terminate.
4. Iterate from 1.

The termination condition in Step 3 is usually defined by personal experience. When the software architecture is sufficiently detailed, we stop. P_i is not unique at each step. Implementing different P_i's gives rise to a family of network views.

3 Reformulation

The reformulation procedure at the i-th step (denoted by P'_i above) consists of a series of graph transformations on G_{i-1} which can be carried out algorithmically. Its overall effect is to facilitate the following analysis, which will eventually give rise to G_i. Such transformations include adding, deleting, replacing (sets of) vertices and/or edges, modifying vertex and/or edge colours and weights; their number is infinite, and the decision to deploy them will depend on the problem at hand. In this section we shall list some of what we think are the most useful and generally applicable such transformations: interface creation and clustering.

3.1 Interface creation

Interface creation has a tremendous impact in the reduction of flow or data arcs in the graph. An *interface* ι is a software, or hardware, interconnection component. Given an edge colouring $\mu : A_{i-1}\mathbb{N}$ of G_{i-1} and subgraph $H = (U, F)$ of G_{i-1} such that $\mu(e) = \mu(f)$ for all $e, f \in F$, G_i is defined as G_{i-1} with the subgraph H replaced by a subgraph $H' = (U', F')$ where $U' = U \cup \{\iota\}$; $(u, \iota), (\iota, v) \in F'$ if and only if $(u, v) \in F$. The aim of this reformulation is to simplify a set of interconnections of the same type. In the extreme case where H is a complete subgraph, the reformulation replaces it with a complete star around ι, which reduces the number of interconnections by a factor $|U|$. Naturally, in order for this reformulation to be worthwhile, we require $|F'| < |F|$. As $|F'|$ is bounded above by $2|U|$, it is interesting to study the problem of finding a (not necessarily induced) subgraph $H = (U, F)$ of G_{i-1} whose arcs have the same colour and such that $|F| - |U|$ is maximum.

Let $\{1, \ldots, K\}$ be the set of arc colours in G. For all $v \in V_{i-1}$ consider binary variables $x_v = 1$ if $v \in U$ and 0 otherwise. For any colour $k \leq K$, the problem of finding the "densest" proper uniformly coloured subgraph $H_k = (U, F)$ of G can be formulated as follows.

$$\max_{x,y} \sum_{(u,v)\in A_{i-1}} x_u x_v - \sum_{v\in V_{i-1}} x_v \tag{1}$$

$$\forall\, (u, v) \in A_{i-1} \quad x_u x_v \leq \min(\max(0, \mu_{uv} - k + 1), \max(0, k - \mu_{uv} + 1)) \tag{2}$$

$$x \in \{0, 1\}^{|V_{i-1}|}. \tag{3}$$

3.2 Clustering

Modules in a software architecture need to be clustered for at least two good reasons: (a) to give an idea of the different independent (or nearly independent) "streams" in the architecture; (b) to be able to assign separate sets of modules to separate teams.

One of the most common ways to bootstrap a software architecture design process is to construct the initial graph G_0 by means of a brainstorming session: this will almost always give rise to a very "tangled" architecture. Modules will roughly correspond to the requirements list, and will be heavily interconnected. Further, each (creative) analytical procedure P''_i is likely to produce a similar situation. Clustering these modules in an arbitrary way according to their perceived semantics may give rise to clusters whose degree of inter-dependency is not minimal, which will greatly complicate team interactions and possibly impair the whole development process.

In its most basic form, the clustering procedure acts on a weighted, undirected graph $G = (V, E, c)$ (where $G = G_i$ for some i, $E = \{\{u, v\} \mid (u, v) \in A_i \vee (v, u) \in A_i\}$ and $w : E \to \mathbb{R}$), ignoring vertex and edge colours, and outputs an assignment of vertices in V to a set of clusters such that the weights of inter-cluster edges is minimized. Such a problem is known in the combinatorial optimization literature as the Graph Partitioning Problem (GPP) [BB99, JAMS89, FMdS+96, BEP06].

Its formulation in terms of mathematical programming is as as follows: given the weighted undirected graph G and an integer $K \leq |V|$, the problem consists of finding a partition of k subsets (clusters) of V minimizing the total weight of edges $\{u, v\}$ where u, v belong to different clusters. To each vertex $v \in V$ and for each cluster $k \leq K$, we associate a binary variable x_{vk} which is 1 if vertex v is in cluster h and 0 otherwise. We formulate the problem as follows:

$$\min_x \frac{1}{2} \sum_{k \neq l \leq K} \sum_{\{u,v\} \in E} c_{uv} x_{uk} x_{vl} \tag{4}$$

$$\forall\, v \in V \quad \sum_{k \leq K} x_{vk} = 1 \tag{5}$$

$$\forall\, k \leq K \quad \sum_{v \in V} x_{vk} \geq 1 \tag{6}$$

$$\forall\, v \in V, k \leq K \quad x_{vk} \in \{0, 1\}. \tag{7}$$

As can be easily ascertained, it relies on binary variables and includes many (nonconvex) quadratic terms. Various ways to linearize the formulation have been suggested [Bou04, Lib05]. The objective function (4) tends to minimize the total weight of edges with adjacent vertices in different clusters. Constraints (5) make sure that each vertex is assigned to exactly one cluster. Constraints (6) excludes the trivial solution (all the vertices in one cluster) and ensures each cluster exists. Further conditions, such as the clusters not exceeding a "balanced" cardinality, may also be imposed:

$$\forall\, k \leq K \quad \sum_{v \in V} x_{vk} \leq \left\lceil \frac{|V|}{2} \right\rceil. \tag{8}$$

A useful variant of the problem asks for all adjacent vertices with like colours to be clustered together. The vertex colours are defined by an integer-valued function $\lambda : V \to \mathbb{N}$ (we denote $\lambda(u)$ by λ_u). For all $u, v \in V$ (with $u \neq v$) we introduce binary parameters $\gamma_{uv} = 1$ if u, v have different colours. We must then add the following constraints:

$$\forall\, u \neq v \in V, k \neq l \leq K \; x_{uk} x_{vl} \leq \gamma_{uv}. \tag{9}$$

Should these constraints be too restrictive, we may want to relax them to a degree. We can do this by removing (9) and adding the term $\sum_{u \neq v \in V} \sum_{k \neq l \leq K} |x_{uk} x_{vl} - \gamma_{uv}|$ to the objective function (4).

Another useful variant allows the optimization process to determine the number of clusters actually used. For all $k \leq K$ we introduce binary variables $z_k = 1$ if cluster k is non-empty and 0 otherwise. We change constraints (6) to

$$\forall\, k \leq K \quad \sum_{v \in V} x_{vk} \geq z_k, \tag{10}$$

to ensure that a cluster that does not exist need not have any vertices assigned to it, and then we add the term $\sum_{k \leq K} z_k$ to the objective function to be minimized, thus ensuring that the maximum number of clusters should be empty.

Once the set of clusters $\mathcal{K}$ have been identified, the current graph G_{i-1} may be replaced in the network view by a graph $G_i = (V_i, A_i)$ where V_i is the set of clusters $\mathcal{K}$ and $(u, v) \in A_i$ if there are $w \in u, z \in v$ (recall u, v are subsets of V_{i-1}) such that $(w, u) \in A_{i-1}$.

4 Bootstrapping the network view

The construction of the initial graph G_0 must necessarily be led by human insight and personal experience. From the analysis of requirements, the software architect must construct a directed graph of the type

described above with as many details as possible. All the same, we shall describe a set of procedures which usually facilitate this task. Each step in the procedures is marked *[a]* if it can be performed by a computer, or *[h]* otherwise. Note that *[h]* steps contain some vague notions that can only be interpreted by a person, or require some creative thought.

4.1 Computation modules

Our first procedure synthesizes many of the necessary computation modules starting with the software requirement list (as well as the available input data).

1. *[h]* List the input parameter vector $\iota = (\iota_1, \ldots, \iota_n)$ and output parameter vector $\omega = (\omega_1, \ldots, \omega_m)$ using the software requirement list.
2. *[h]* Draw a *data computation diagram* formalized as a graph $G = (V, A)$ where each $v \in V$ represents a piece of data, and $(u, v) \in A$ if the value of u is necessary to the computation of the value of v. Let $\tau : V \to \mathbb{N}$ be the size (in bits) of the data (thus, for example, if v represents a boolean parameter, $\tau(v) = 1$). This graph must be such that $\iota_i \in V$ for all $i \leq n$, $\omega_j \in V$ for all $j \leq m$, and
$$\forall\, v \in V \qquad \sum_{u \in \delta^-(v)} \tau(u) \geq \tau(v). \tag{11}$$
Let $\alpha : A \to \mathbb{N}$ be a colouring on the arcs such that "similar" data computations have the same colour.
3. *[a]* Perform the interface creation process (see Section 3.1) on G using the arc colouring α. Each subgraph H_k (where k is a colour), as well as all arcs (u, v) not belonging to any H_k, correspond to a computation module in the software.

Some practical considerations on the data computation diagram construction. First, we believe it is easier to build this diagram working backwards from the outputs. Secondly, instead of associating the τ vertex colouring with data sizes, it may be easier to consider data types. Although it becomes somewhat more difficult to enforce Eq. (11), there are at least two advantages: (a) treating with qualitative instead of quantitative data is easier for a person; (b) the output of the procedure will also provide the exact specifications (i.e. input/output argument types) of each computation module.

References

[BB99] R. Battiti and A. Bertossi. Greedy, prohibition and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, 1999.

[BEP06] A. Billionnet, S. Elloumi, and M.-C. Plateau. Quadratic convex reformulation: a computational study of the graph bisection problem. Technical Report RC1003, Conservatoire National des Arts et Métiers, Paris, 2006.

[Bou04] M. Boulle. Compact mathematical formulation for graph partitioning. *Optimization and Engineering*, 5:315–333, 2004.

[FG02] R. Fourer and D. Gay. *The AMPL Book*. Duxbury Press, Pacific Grove, 2002.

[FMdS+96] C.E. Ferreira, A. Martin, C. Carvalho de Souza, R. Weismantel, and L.A. Wolsey. Formulations and valid inequalities for the node capacitated graph partitioning problem. *Mathematical Programming*, 74:247–266, 1996.

[For60] R. Fortet. Applications de l'algèbre de boole en recherche opérationelle. *Revue Française de Recherche Opérationelle*, 4:17–26, 1960.

[ILO02] ILOG. *ILOG CPLEX 8.0 User's Manual*. ILOG S.A., Gentilly, France, 2002.

[JAMS89] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations Research*, 37:865–892, 1989.

[Lib04] L. Liberti. Reformulation and convex relaxation techniques for global optimization. *4OR*, 2:255–258, 2004.

[Lib05] L. Liberti. Compact linearization for binary quadratic problems. *4OR*, pages DOI 10.1007/s10288–006–0015–3, 2005.

[Som07] I. Sommerville. *Software Engineering*. Addison-Wesley, London, 2007.