Automating Component-Based System Assembly

Panagiotis Manolios[†], Gayatri Subramanian[‡], and Daron Vroon[†]

[†]College of Computing
Georgia Institute of Technology
266 Ferst Drive
Atlanta, Georgia 30332-0765 USA
{manolios, vroon}@cc.gatech.edu

[‡]Oracle Corporation
600 Oracle Parkway
Redwood Shores, California 94065 USA
gayatri@gatech.edu

ABSTRACT

One of the major challenges in the development of large component-based software systems is the system assembly problem: from a sea of available components, which should be selected and how should they be connected, integrated, and assembled so that the overall system requirements are satisfied? We present a powerful framework for automatically solving the system assembly problem directly from system requirements. Our framework includes an expressive language for declaratively describing system-level requirements, including component interfaces and dependencies, resource requirements, safety properties, objective functions, and various types of constraints. We show how to automatically solve system assembly problems using verification technology that takes advantage of current advances in Boolean satisfiability methods. We have implemented our techniques in the CoBaSA tool (Component-Based System Assembly), and we have successfully applied it to several large-scale industrial examples.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design

Keywords

System Assembly Problem, Component-Based Software Development, Integrated Modular Avionics, Pseudo-Boolean Satisfiability

1. INTRODUCTION

One of the main goals of component-based software development (CBSD) is to enable the construction of large industrial systems by integrating components, especially commercial off-the-shelf (COTS) components. The potential benefits include greater reliability, lower development costs, shorter development cycles, less testing and validation, more flexibility and reuse, etc. [16] Component-based systems can consist of thousands of components, and ensuring that they are connected, integrated, and assembled in a way that satisfies the system requirements is a formidable undertaking that currently requires significant human effort and is the responsibility of the system architect. We refer to this problem as the *system assembly* problem.

In this paper, we present a new, automated approach to the system assembly problem. The idea, simply stated, is to algorithmically find an optimal solution to the system assembly problem directly from the system requirements. A key insight is that the system assembly problem can be seen as the following satisfiability problem: does there exist a way of selecting and assembling the available components that satisfies all of the system and component requirements? These requirements can include component interface dependencies, resource requirements, safety properties, separation and safety requirements, as well as replication, timing, and scheduling constraints, etc.

We have developed a powerful framework that includes an expressive object-oriented language for declaratively describing system-level requirements and constraints such as those mentioned above, and more. If the resulting satisfiability problem has a solution, then we often want to find the *best* solution, *i.e.*, a solution that optimizes a given objective function. We show how to algorithmically handle such satisfiability/optimization problems using novel constraint solving methods that take advantage of current SAT-based methods.

We have implemented our approach in the CoBaSA (Component-Based System Assembly) tool. CoBaSA provides an expressive declarative language for expressing system constraints. After applying constraint solving methods to the system constraints, the result is compiled to a pseudo-Boolean satisfiability and optimization (PBSAT) problem, which can be handled by any of the existing PBSAT solvers [2]. We can also generate a Boolean Satisfiability (SAT) problem. An advantage of generating satisfiability problems is that we can take advantage of the current, rapid advances in SAT solving technology [35]. If the SAT/PBSAT solver finds a solution to the satisfiabilty problem we generate, CoBaSA transforms the solution into an optimal, detailed architecture satisfying the original constraints, as shown in Figure 1.

We describe in detail an industrial case study involving Boeing. The case study allows us to evaluate our techniques and the CoBaSA tool in an industrial setting, by supporting the development of the Boeing 787 Dreamliner.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

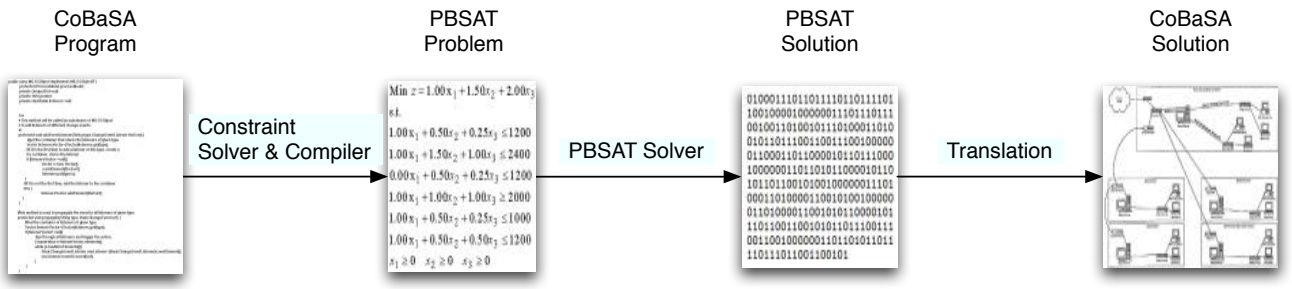

Figure 1: Solving system assembly problems with CoBaSA. The system requirements are written using the CoBaSA language. These requirements are processed by a constraint solver and are transformed into pseudo-Boolean SAT problems that are given to a pseudo-Boolean solver. The solution is translated back into a solution of the component assembly problem.

Component-based software development is very important and relevant to the avionics domain because software has become one of the major costs of developing aircraft. To curtail costs, the industry has moved away from federated systems—where subsystems use dedicated processing and I/O components—to integrated modular avionic (IMA) systems. IMA systems allow multiple subsystems to share the same resources and can lead to drastic savings in power, weight, development and maintenance costs, as well as overall efficiency [1, 25, 34]. [1]

As one can imagine, IMA systems tend to be rather large and complex. They contain thousands of components, many with similar interfaces. In addition, Federal Aviation Administration (FAA) regulations add hundreds of restrictions on how components may be assembled. For example, safety-critical processes have redundancy constraints requiring that the processes be replicated among resources in such a way that a small number of isolated failures do not lead to catastrophic consequences. In systems of this size and complexity, component integration and assembly becomes a major challenge.

To get a sense of the complexity of the Boeing problems, we make a few brief comments about one of the examples we considered. It turns out that this example was actually one of the simplest Boeing examples we considered, but it is the only one for which we have Boeing provided statistics. It took Boeing engineers about half a person-month to generate a CoBaSA model for this example. This number does not account for the time required to determine what information to gather and the time required to gather and check this information from the various component suppliers. The model is quite detailed and accounts for I/O time, latency, network jitter, context switching time, cache flushing time, memory latencies, etc. Since solutions to the system assembly problem have to satisfy hard real-time guarantees, the numbers and analyses are based on the worst-case execution times. Once we generated a solution to the above mentioned system assembly problem, it took Boeing engineers over a person-week to check the validity of the solution. Clearly, the human effort required to actually solve these problems is *much* larger, especially when it comes to the other, more complex Boeing examples we considered.

CoBaSA was able to solve the Boeing problems in a matter of seconds. In addition to the substantial savings in terms of human effort, CoBaSA empowers system architects to work in completely new ways. Architects can easily explore the design space and can consider various "what-if" scenarios in a matter of minutes. This exploration will help identify and clarify the limits, issues, strengths, weaknesses, etc. of various architectural-level decisions at a very early stage in the design cycle, which will in turn lead to more flexibility and reduced costs. Another advantage of our method is that it allows system architects to quickly respond to unplanned and often significant changes in requirements. Such changes are inevitable when designing large, complex systems and often require drastic changes in the architecture. Unfortunately, such problems are often compounded by severe time-to-market constraints. Therefore, without tool support, it is almost impossible to properly explore the space of possible architectural solutions.

It is worth pointing out that CoBaSA is designed to be a flexible and generally applicable system, not a domain-specific tool. In addition, CoBaSA was designed before we had access to any of the Boeing models, and as will be apparent in later sections, there is nothing in CoBaSA that is specialized to the avionics domain. Therefore, we expect CoBaSA to be widely applicable, *e.g.*, during our work with Boeing, CoBaSA was able to easily handle what Boeing described as "serious architecture changes." Also, in Section 6, we outline a few of the many possible applications we envision for CoBaSA.

The main contributions of our work are:

- We introduce a novel, general approach to automating the system assembly problem. This is the first such approach we know of. In fact, in chapter 26 (The Prophecies) of "Building Systems from Commercial Components" [32], the use of reasoning systems for automating system assembly was prophesied to occur within the next ten years (*i.e.*, by 2012).

- The CoBaSA tool, which implements our approach and can efficiently scale to industrial-strength problems.

[1]Similar trends can be seen in other industries, *e.g.*, the automotive industry has also moved in this direction.

- An industrial case study showing the effectiveness and applicability of our work and the CoBaSA tool, which was able to solve in under a minute problems that require multiple person-weeks for Boeing engineers to just state and check.

The rest of the paper is organized as follows. We start in Section 2 by presenting the CoBaSA language, using examples to illustrate language features. Due to space constraints, we cannot give the full syntax and semantics of the language here, but these are described in the CoBaSA documentation. In Section 3, we describe our constraint solving work and show how to compile CoBaSA problems to PBSAT problems. In Section 4, we present the industrial case study where we applied CoBaSA to problems from the avionics domain. We discuss related work in Section 5 and then turn our attention, in Section 6, to several opportunities for future research. We conclude in Section 7.

2. CoBaSA LANGUAGE

CoBaSA uses its own modeling and constraint language for describing components and their interaction. Our goal was not to define a general-purpose language, but rather to define a language that can be used to model components, their interfaces, their resource needs, their capabilities, etc. We wanted expressive power, but more important was the requirement that the language have a formal semantics and that the system assembly problem should not only be decidable, but practically solvable for realistic, industrial examples.

The part of the CoBaSA language used for defining system-level requirements can be thought of as a declarative constraint specification language. Here, our goal was to design general and powerful abstractions with enough expressive power to declaratively describe system-level requirements and component interfaces, including dependencies, resource requirements, safety properties, separation constraints, objective functions, etc.

The modeling part of the CoBaSA language is based on object oriented concepts, like more complex languages such as UML [27, 26] and OCL [23]. This makes the modeling of systems convenient and flexible. It also provides many opportunities for reuse. A design goal was to develop a language that can function as the target language to which a variety of other languages, such as UML and OCL, can be compiled.

In fact, this is how we approached the Boeing case study for CoBaSA, which is presented in Section 4. In response to the specific needs of the domain in which we were working, we developed a domain-specific language which simplified the model construction and constraint specification tasks and which we compiled to CoBaSA. After we understood the domain, the development of the domain-specific language took a few days to implement, in part because CoBaSA provides many powerful abstractions.

We now give a brief overview of the core CoBaSA language, using examples to illustrate its features.

2.1 Component Modeling

We start by giving an overview of the part of the language used for modeling components. The language is object-based, but since our focus is on structural properties of components, not on their behaviors, our language is not used to describe computation. Therefore, there is no notion of a method. Instead, variables are write once; that is, once they are assigned a value, the user is not allowed to assign them a different value.

Our language supports as basic types all of the following: Booleans, integers (including bounded integer types), and strings. In addition, data can be organized into arrays and entities, which resemble Java classes without methods. A component specification can be expressed through an entity definition and an instance of the component can be created by defining an object for the entity definition. The syntax of an entity definition is illustrated by the following example.

```
entity server {
  ;name string
  ;memory int
  ;num-processors 4
}
entity foo-server extends server {
  ;name "foo"
  ;neighbor foo-server
}
```

The first specification defines a server component by declaring an entity of type `server`. The fields within the entity definition represents the interface and extra-functional properties of the component. The entity `server` contains three fields: a name field of type string, a memory field of type integer, and a num-processors field which is 4 for all `servers`. Note that every field in an entity definition begins with a ';', and specifies a field name and either a type or value for the field.

The second definition creates a `foo-server` type which extends the `server` entity. This means that foo includes all the fields contained in a server, plus the fields defined in the body of the definition of `foo-server`. The example demonstrates how inheritance is implemented in CoBaSA. Note that entities can be recursive, as is the case with the `foo-server` entity, which contains a field that is of type `foo-server`.

Declaring a variable can be thought of as either defining an instance of an entity type or defining a basic data-type variable. The following example illustrates the syntax of a variable declaration.

```
var ;int num = 5 ;string bar
var ;int[4] x = [1, 2, 3, 4]
var ;foo-server x = { ; ; ;"foo" ;1024 ;}
```

The first line defines an integer called `num` with value `5` and a string called `bar` with no value specified. The second line creates an array of integers with values `1`, `2`, `3`, and `4`. The third line creates an object x, which is an instance of component `foo-server`. Each ';' in the definition represents a field of the entity. The order of the fields in the object definition correspond to their order in the entity definition. Hence, the first field corresponds to the name field, and the second field corresponds to the neighbor field. Once the fields of the current type are exhausted, the definition fills the fields of the immediate ancestor type and so on until all the ancestors are exhausted. Thus, in our example, the third field corresponds to the name field of the server entity, the fourth corresponds to the memory field of the server entity, and so on. If no text appears after a semicolon, then no value is specified for the corresponding field. Any field can be left blank, and any field that is given a value in the entity

definition can also be left blank (*e.g.*, the name field of the `foo-server` entity).

To assign values to variables or fields of objects that have been declared but not assigned a value, our language provides an `assign` statement. The following code assigns values to `bar` and the `neighbor` field of `x`.

```
assign bar = "hello world"
assign x.neighbor = { ; ; ;"server-001" ;2048 ;}
```

Note that a '.' is used between the object name and the field name, to access the value of that field within the object.

2.2 Constraint Modeling

A key concept of the CoBaSA constraint modeling language is that of *maps*. These are simply functions or relations, generally mapping components to components. They are used to place restrictions on what constitutes a solution to the system assembly problem. For example, we may want to map processes to servers and we can use the notion of a map to correspond to such a mapping. But, maps are really existentially quantified in the sense that the user typically does not fully define a map, rather she outlines constraints describing what relationships must exists between components in the domain and range. The user claims to CoBaSA that maps satisfying these constraints exist and it is then up to CoBaSA to then find a definition for the maps that satisfies these constraints and optimizes the objective function.

There are several mechanisms for defining constraints. The first is the map definition itself, which allows users to specify a domain and range for an existentially quantified function. Once a map is defined, users can use field constraints, which further restricts the associated map so that it pairs resource consumers with resource providers. This means that a solution to the generated constraints will specify how the needs of the resource consumers are met by the resource providers. The implicit constraint of legal solutions is that providers must have enough of every resource to meet the needs of all the consumers that get mapped to them. In addition, the user can specify arbitrary arithmetic and Boolean constraints over maps and pseudo-Boolean variables (*i.e.*, variables that take the values 0 or 1 in an arithmetic context, and `False` or `True` in a Boolean context). This gives the user the flexibility to specify a variety of constraints on how components interact. In this section, we describe these mechanisms in more detail and make use of various illustrative examples.

2.2.1 Maps and Field Constraints

Consider the task of mapping process components to server components. We might expect entity and object definitions such as the ones given in Figure 2. Servers have resources such as memory, disk space, and the number of processors. They also have a security predicate (*e.g.*, this may indicate whether the server is behind a firewall or is configured for high security). The server entity is extended by Linux and Win servers, which have operating system and version strings.

We also have an entity definition for processes. Processes have minimum memory and disk requirements, as well as operating system, version, and security requirements. There are 20 Linux servers, 20 Win servers, and 1,000 processes. We wish to map each of the processes to one of the servers

```
entity server {
  ;mem    int
  ;disk   int
  ;procs  int
  ;secure bool
}

entity linux-server extends server {
  ;os "linux"
  ;version string
}
entity win-server extends server {
  ;os "win"
  ;version string
}
entity process {
  ;mem-req  int
  ;disk-req int
  ;os-req   string
  ;ver-req  string
  ;sec-req  bool
}

var linux-server[20] ls =
  [ {; ;"2.4.22" ;1024 ;250000 ;2 ;False},
    ...
    {; ;"2.6.11" ;512 ;160000  ;2 ;True} ]
var win-server[20] ws =
  [ {; ;"2k" ;1024 ;250000 ;2 ;True},
    ...
    {; ;"xp sp2" ;2048 ;250000 ;4 ;False} ]
var process[1000] pr =
  [ {;32 ;100 ;"win" ;"2k" ;True},
    ...
    {;128 ;4000 ;"linux" ;"2.6.10" ;False} ]
```

Figure 2: An example of software component matching in CoBaSA.

such that all the processes' extra-functional requirements are met.

We start by defining a map from processors to servers, which can be done as follows.

```
map proc-serve pr (ls, ws)
```

In general, a map command takes a name, followed by an optional relation $R \in \{$`<=`, `>=`, `=`$\}$ and natural number, k, followed by the domain of the map, followed by the range. Both domains and ranges can be a variable (indicating a singleton set), an array (indicating the set consisting of all the elements in the array), or a sequence of variables and/or arrays (the union of all corresponding sets).

A map has an implicit constraint. If R and k are specified, then for each component in the domain, the number of components to which it is mapped is at most, at least, or equal to k if R is `<=`, `>=`, or `=`, respectively. If R and k are not specified, then each component in the domain is mapped to exactly 1 component of the range. Thus `map m c p` is equalivalent to `map = 1 m c p`.

Often, it is the case that the elements in the domain of a map are resource consumers and the elements in the range of the map are resource providers. For example, `proc-serve` maps processes, which require computing resources, to the servers that will provide them.

To express such relationships in the CoBaSA language, one can use *field constraints*. In our case, we wish to indicate that the memory and disk requirements of the processes are

provided by the servers to which they are mapped. This is denoted by the following statement.

```
constraint proc-serve ((mem-req disk-req))
                      ((mem disk) (mem disk))
```

Each constraint statement is given a map name, and two lists. The first list is the same length as the list used to define the domain of the map (*i.e.*, the set of resource consumers). Similarly, the second list is the same length as the list used to define the range of the map (*i.e.*, the set of resource providers). The i^{th} member of the first list in the constraint statement specifies a list of fields in the i^{th} collection of elements specified by the domain list of the map. The situation for the range and the second list are analogous. All of these domain/range lists need to be exactly the same length. The consumer field lists correspond to the "needs" of the consumer components, and the provider field lists correspond to the "resources" that the provider components have available to meet these needs. The implicit constraint for these statements is that, for each provider, there must be enough of every resource to meet the needs of the consumers that are mapped to it. In our example, every server must have at least as much memory and disk space as all the processes running on it require.

2.2.2 Relational and Boolean Constraints

To express reliability and safety constraints, more expressiveness is needed than that offered by the map and field constraints. CoBaSA allows the designer to specify explicit constraints in the form of arbitrary Boolean formulas. In addition, CoBaSA allows formulas relating two arbitrary arithmetic expressions over pseudo-Boolean variables and what we call *map references*, which indicate whether a domain element gets mapped to a range element. CoBaSA also includes the powerful and convenient `For_all` and `Sum` statements, which allow us to easily apply constraints to entire arrays.

Consider the following CoBaSA example, which states that secure processes must be mapped to secure machines.

```
For_all p in pr {
  For_all s in ls {
      (proc-serve(p,s) and p.sec-req) implies s.secure}
  and
  For_all s in ws {
      (proc-serve(p,s) and p.sec-req) implies s.secure}}
```

In the above example, `proc-serve(p,s)` is a map-reference and is `True` if `proc-serve` maps `p` to `s`. The `For_all` statement applies the body to each of the values in the array. In addition, `For_all` statements can range over a natural number, meaning they range from zero to one less than the number. For more complicated computations, the designer can give arbitrary code in the Lisp programming language, which must return a value of the appropriate type. For Boolean expressions, the result is cast to `False` if the value is `nil`, and `True` otherwise. For example, we can use this feature to talk about OS and version constraints as follows.

```
For_all p in pr {
  For_all s in ls {
      proc-serve(p,s) implies
       (let ((pos p.os-req)
             (sos s.os)
             (pver p.ver-req)
             (sver s.version))
```

```
entity server {;bandwidth int}

entity proxy {
  ;bw-provide int
  ;bw-req int
  ;percent-overhead [0..100]
}

entity process {;bw-req int}

var proxy[100]    px = [ {;512 ; ;12} ... ]
var server[10]    sv = [ {;2048} ... ]
var process[1000] pr = [ {;32} ... ]

map px-sv px sv
constraint px-sv bw-req bandwidth
map pr-px pr px
constraint pr-px bw-req bw-provide
```

Figure 3: An example of solving chains of maps using CoBaSA.

```
         _(and (string-equal pos sos)
               (string-equal pver sver
                             :end1 3 :end2 3))_)}}
```

This constraint says that if a process maps to a Linux server, its operating system requirement must match the operating system of the server, and the first 3 characters of its version requirement must match the first 3 characters of the server OS version (*i.e.* a process that requires a 2.4 kernel gets matched to a server running a 2.4 kernel).

Another way to constrain maps is through relation constraints. These involve arithmetic expressions over Booleans and integers, where all the integer values must be known at compile time. In this context, Boolean values are viewed as 1 or 0 rather than `True` or `False`. As with Boolean expressions, Lisp code may be used for arbitrary computation. However, in this context the Lisp code must return an integer. A simple example of a relational constraint is the following, which limits the number of processes mapped to any Linux server to 100.

```
For_all s in ls {
  Sum p in pr proc-serve(p,s) <= 100 }
```

2.2.3 Interdependent Maps

CoBaSA provides a special kind of relational constraint that is used to set an unknown integer value to some arithmetic expression over pseudo-Boolean variables. This provides a mechanism for solving for arbitrary integers, not just Booleans, and is useful when there are "interdependent" maps. For example, suppose there were proxies fielding requests from the processes and assigning them to the servers. A simple example of this in CoBaSA appears in Figure 3.

Now suppose that the proxy incurred an overhead when negotiating between processes and a server. We can represent this using the CoBaSA special form of the relational constraint as follows.

```
For_all x in px {
  x.bw-req
  =
  Sum r in pr
    (* pr-px(r,x)
       (let ((bwr r.bw-req)
             (po  x.percent-overhead))
         _(ceiling (* (+ 1 (/ po 100)) bwr))_))}
```

This sets the bandwidth requirement of the proxies to be that of the processes mapped to them, but increased by the percent overhead specified in the proxy specification. The solver can then simultaneously solve the two maps, generating a solution to the system assembly problem that provides the necessary bandwidth to the proxies and the processes.

2.2.4 Optimization

A component based system assembly problem can have many solutions. It might be desirable to obtain a solution that maximizes or minimizes a given objective function. CoBaSA allows objective functions that can be expressed as an arithmetic or relational expression. For instance, in the proxy example, we may want to minimize the overhead, while setting a maximum of 75Mbps. We can do this with the following command.

```
Minimize Sum x in px Sum r in pr
          (* pr-px(r,x) (let ((bwr r.bw-req)
                             (po  x.percent-overhead))
                         _(* (/ po 100) bwr)_))
        <= 75
```

3. CoBaSA COMPILER

The component based system assembly problem is reducible to the 0-1 Integer Programming Problem (also known as Pseudo-Boolean Satisfiability, PBSAT), which involves solving two sets of constraints. The first is a set of linear constraints of the form $\sum_{i=1}^{n} c_i x_i \ R \ c$ where c and all the c_i are constant integer values, each x_i is a variable that ranges over the values $\{0, 1\}$, and R is either '$\leq$', '$\geq$', or '$=$'. The second is a Boolean formula over the same variables in conjunctive normal form (CNF). In this context, variables are viewed as Boolean, *i.e.*, `False` for 0 and `True` for 1. The problem is satisfied if there is an assignment for the variables (`False`/0 or `True`/1) such that all of the linear constraints as well as the Boolean formulas are satisfied. In addition, PBSAT allows for an optimization sum of the form $O \ \sum_{i=1}^{n} c_i x_i$ (or sometimes $O \ \sum_{i=1}^{n} c_i x_i \ R \ c$), where O is either `Maximize` or `Minimize`. In this case, the solution returned must be optimal, *i.e.*, the solution must maximize or minimize the objective function as specified by the user.

The CoBaSA compiler performs the transformation from a component based system assembly problem to a PBSAT problem. The goal of CoBaSA is to find a definition for each map that satisfies the constraints. In order to convert this into a PBSAT problem, we must first represent the maps using pseudo-Boolean variables. For each map, $M : C \rightarrow P$, we create $|C|*|P|$ Boolean variables $\{M_p^c | c \in C, \ p \in P\}$. Intuitively, M_p^c represents the Boolean expression $M(c) = p$. We then compile the user provided constraints and optimization function to pseudo-Boolean constraints over these variables such that the following two conditions hold:

1. A satisfying assignment to the PBSAT problem exists if and only if a satisfying assignment to the original component based system assembly problem exists,

2. Given an optimal satisfying assignment to the pseudo-Boolean problem, the assignment such that each function, M, is defined by $M(c) = p$ iff $M_p^c = \mathit{true}$, and each Boolean variable, b is *true* iff it is *true* in the pseudo-Boolean solution, is an optimal satisfying assignment to the original component based system assembly problem.

3.1 Implicit Map Constraints

For each map, we have the implicit constraint that each element of the domain maps to the appropriate number of elements in the range, according to the user specified relation, R, and natural number, k (or `=` and 1 if the user did not specify any). To represent this in the pseudo-Boolean format, we add the following constraint for each consumer component, $c \in C$, of a map $M : C \rightarrow P$:

$$\sum_{p \in P} M_p^c \ R \ k.$$

For example, the implicit map constraint of `proc-serve` from Section 2.2.1 for each process $p \in$ `pr` would be

$$\sum_{s \in (\texttt{ls} \cup \texttt{ws})} \texttt{proc-serve}_s^p = 1$$

3.2 Implicit Field Constraints

As with each map definition, there is an implicit pseudo-Boolean constraint implied by each field constraint. Namely, all the "needs" of all the consumers that map to a given provider – by any map – need to be met by the appropriate fields of the provider. Given a producer component, p, let $M_p = \{M : C \rightarrow P \mid p \in P\}$. That is, M_p is the set of all maps with a range containing p. We need to assure that p can satisfy all of the field constraints, as defined by the `constraint` statements of the given component based system assembly problem definition. Recall that such constraints tell us which fields of the consumers are supplied by which fields of the producers. So, for each field, f, of p, we add the following constraint:

$$\sum_{i=1}^{n} c_i.f_i * (M_i)_p^{c_i} \leq p.f$$

where $M_i \in M_p$, and each c_i and f_i is a pair such that c_i is a consumer of M_i and f_i is a field of c_i that is a need supplied by $p.f$ according to some field constraint. Note that neither the M_i nor the c_i are necessarily distinct, since each map can map several consumers to p, and each consumer can have several fields that draw from field f of p.

Consider the `proc-serve` field constraint from Section 2.2.1. In this case, for each $s \in (\texttt{ls} \cup \texttt{ws})$, we have two constraints: one for the memory requirement and one for the disk requirement.

$$\sum_{p \in \texttt{pr}} (p.\texttt{mem-req} * \texttt{proc-serve}_s^p) \leq s.\texttt{mem}$$

$$\sum_{p \in \texttt{pr}} (p.\texttt{disk-req} * \texttt{proc-serve}_s^p) \leq s.\texttt{disk}$$

Note that if all the fields involved have concrete integer values, these are proper pseudo-Boolean constraints. However, recall that fields can also be set to a pseudo-Boolean arithmetic expression using the special relational constraint as defined in Section 2.2.3. In this case, we substitute the expression for the field, giving us a term in the field constraint sum in which a Boolean variable is multiplied by some arithmetic expression. For example, consider the constraints generated by the first field constraint in Figure 3. For each $s \in$ `sv`, we have a constraint of the form:

$$\sum_{p \in px} (p.\texttt{bw-req} * \texttt{px-sv}_s^p) \leq s.\texttt{bandwidth}$$

However, each p.`bw-req` was defined in Section 2.2.3 as the sum of the bandwidth requirements of the processes

mapped to p multiplied by some overhead. If we substitute this into the above equation, we get something different than the required form for pseudo-Boolean constraints. Similarly, our language allows relational constraints comparing two arbitrary arithmetic expressions over pseudo-Boolean variables involving addition, subtraction, and multiplication. We deal with all of these constraints in a similar way.

3.3 Arbitrary Arithmetic Relation Constraints

Given an expression of the form $E_1\ R\ E_2$, we subtract E_2 from both sides, giving us $E_1 - E_2\ R\ 0$. We then focus on the left hand side of the equation. We turn expressions of the form $-E_1$ into $-1 * E_1$ and expressions of the form $E_1 - E_2$ into $E_1 + (-1 * E_2)$. We then push all the sums outward and all the products inward by distributing multiplication over addition in the usual way. Collecting all the coefficients together, we get a constraint of the form $c + \sum_{i=1}^{n}(c_i v_{i,1} v_{i,2} \ldots v_{i,m_i})\ R\ 0$. Subtracting c from both sides, we get $\sum_{i=1}^{n}(c_i v_{i,1} v_{i,2} \ldots v_{i,m_i})\ R\ -c$. This is almost the correct form, except that each term in the sum has multiple variables, and pseudo-Boolean constraints are only allowed to have one variable per term. But notice that $v_{i,1} v_{i,2} \ldots v_{i,m_i}$ is 1 if and only if $v_{i,1} \wedge v_{i,2} \wedge \ldots \wedge v_{i,m_i}$ is `True`. Therefore, we create a new Boolean variable, v_i for each term and add the following Boolean constraint:

$$v_i \Leftrightarrow (v_{i,1} \wedge v_{i,2} \wedge \ldots \wedge v_{i,m_i})$$

Finally, we substitute each v_i into the formula, giving us an expression of the desired form: $\sum_{i=1}^{n}(c_i v_i)\ R\ c$.

4. INDUSTRIAL CASE STUDY

CoBaSA has already been used to solve large component assembly problems in an industrial setting. Boeing has used CoBaSA in the development of their forthcoming 787 Dreamliner passenger jet. In this setting, CoBaSA was used to find the optimal component configurations in an avionic system as described by the Integrated Modular Avionics (IMA) standard. The problems solved involved hundreds of components and equally many constraints. For such problems, Boeing requires half a person-month to create a CoBaSA model from well-understood data, and over a person-week to verify that a given configuration satisfies the constraints. Boeing has requested that we do not disclose how long these problems take to solve using conventional methods, but it suffices to say that finding the optimal assembly is significantly more difficult than creating the problem or verifying a given solution. In this section, we describe the case study. We begin by giving some background on the domain of the problem, IMA.

4.1 Domain: Integrated Modular Avionics

Assembling components in avionic systems is tedious; there are typically hundreds of connections that have to be considered, in addition to an equal number of safety and reliability constraints imposed both at the component-level and system-level. Further complicating factors include scarcity of computation, memory, and bandwidth resources available on-board an aircraft. Straight-forward assembly techniques are insufficient for modeling and satisfying the complex assembly constraints.

Historically, the avionics system architecture involved a federated architecture of black-boxes called Line Replaceable Units (LRUs), each of which was specifically designed

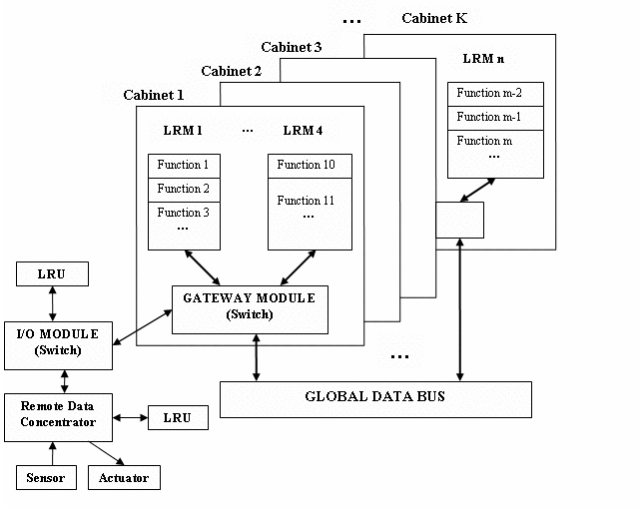

Figure 4: Diagrammatic Representation of Integrated Modular Avionics

to perform an individual function. Due to growing complexity of equipment and advancements in technology, the avionics industry has now moved to implementing open architectures that employ highly-integrated digital avionics under software control. This approach, referred to as Integrated Modular Avionics (IMA), is based on modular design, generic resources and multiplexed communication buses; it has resulted in the development of smaller, lighter and more cost-effective avionics equipment.

The IMA platform is illustrated in Figure 4. It reflects the current standards in the avionics industry [25, 19, 28]. It is made up of cabinets containing sets of modules, called Line Replaceable Modules (LRMs). An LRM can be either a memory module or an Avionics Computing Resource (ACR) module. In the figure, we concern ourselves only with ACR modules. An ACR is a computer which performs a variety of tasks (refered to as avionic functions), each of which would have been performed by a separate LRU in older systems. Several software applications (or avionic functions) that were originally implemented as independent LRUs, now face the possibility of interacting with one another while sharing common resources. To avoid unintended interactions, software applications running on the ACRs must follow strict guidelines for memory partitioning and timing.

The cabinets communicate with each other via a Gateway Module, which multiplexes communications through the Global Data Bus. The Gateway Modules also handle communication with elements external to the cabinets. These include some LRUs, which take care of tasks that cannot be integrated into the cabinets.

Also included in the external elements are sensors and actuators which collect data and carry out commands throughout the aircraft. These elements are tied to the system using Remote Data Concentrators (RDCs) that regulate the data collected from and distributed to these elements. All the external elements communicate with the cabinets via additional I/O modules. The data collected by a sensor is passed on as a message to an RDC, which through an I/O module switch passes the message to the gateway module switch, which through the global data bus passes the message to

Version	1	2	3	4	5
# Cabinets	2	2	2	2	2
# LRMs	16	16	22	22	16
# Avionic Functions	237	257	245	245	237
# Linked Memory	70	88	88	88	70
# Constraints	224	268	271	272	251

Table 1: Description of IMA Model Versions

the specific avionic function (or software application) in an LRM in a cabinet.

Since IMA is based on an open architecture scheme, it allows the system designer to shop around for COTS components and makes component changes and upgrades nearly effortless. The difficulties of the IMA approach have to do with the assembly of the components so as to satisfy the numerous and restrictive FAA guidelines. In order to minimize the risk of critical system failures, avionic functions must be redundant and spread out among the LRMs and cabinets in an appropriate way. Additionally, bandwidth and resource constraints must be met to guarantee proper timing of critical systems.

The assembly problem in IMA involves figuring out which cabinet each LRM maps to, which LRM each avionic function maps to, which switch each of the RDCs and LRUs map to, and which RDC each of the sensors, actuators, and LRUs map to. These are only the basic concerns when assembling IMA components. In addition, system architects must consider worst-case execution time, context switching time, I/O time, latency, network jitter, cache flushing time, memory latencies, and so on. All of these are expressible using CoBaSA.

4.2 The Study: Boeing

We used CoBaSA to solve component assembly problems that are part of the 787 Dreamliner IMA models. The IMA models we obtained from Boeing focused on the assembly of black-box components like avionic functions, LRMs, cabinets, *etc.*, as shown in Table 1. All the components except linked memories are shown in Figure 4; linked memories are memories, allocated within the LRMs, that are linked to either a specific avionic function or a set of avionic functions. The 5 versions in Table 1, either differ in number of components and constraints, or differ in the nature of the constraints.

There are over 200 constraints for each model. These include map and field constraints. A sanitized version of CoBaSA code for the map that maps avionic functions to LRMs appears as follows.

```
Map func-lrm func lrm
Constraint func-lrm
           ((processor-time-req, memory-req,
             bandwidth-req-on-global-data-bus))
           ((processor-time-available, memory-available,
             bandwidth-available-on-global-data-bus))
```

A similar map and field constraint exists for mapping linked memories to LRMS.

For the purposes of safety and reliability considerations, partitions are built into the IMA model components by design [25]. These partitions ensure that the data of one partition does not damage the data of another partition and that each partition holds its own execution window on the LRM to which it is mapped. In addition to these partitions, there are stringent safety and separation requirements that are imposed on the placements of the avionic functions and linked memories within the LRMs, and the placement of the LRMs within the cabinets. These requirements can be modeled as separation constraints in CoBaSA and these can be used by the solver to obtain an allocation that meets the critical safety requirements imposed on the aircraft. When the number of avionic functions is more than 230, and almost each avionic function needs to be separated from 4 (actually this number ranges from 1-10) other avionic functions, then the number of separation constraints that are generated is large. A sanitized example of such a constraints appears as follows.

```
for_all i in lrm {
  (func-lrm(func[1], i)
   implies ((not func-lrm(func[2], i)) and
              ((not func-lrm(func[3], i)) and
              (not func-lrm(func[4], i)))))}
for_all i in lrm {
  (func-lrm(func[2], i)
    implies ((not func-lrm(func[3], i)) and
             (not func-lrm(func[4], i))))}
for_all i in lrm {
   (func-lrm(func[3], i)
     implies (not func-lrm(func[4], i)))}
```

The above constraint says that the first four avionic functions cannot be mapped to the same LRM. Since linked memories are linked to avionic functions, it is desirable to allocate a linked memory to the same LRM, to which the avionic functions that link to that linked memory are allocated. For instance if the first linked memory is linked to the 12^{th} avionic function, then both of them must be allocated on the same LRM. This is indicated by the following sanitized CoBaSA constraints.

```
for_all i in lrm {
  (mem-lrm(mem[1], i) implies func-lrm (func[12], i))}
```

where `mem-lrm` refers to the map that maps linked memories `mem` to LRMs.

As described in Section 4.1, there are many other constraints that we cannot show here, including load-balancing constraints for processor time, and intricate constraints relating the memory regions to the avionic functions and the LRMs. In addition, the constraints that we have shown here are significantly simplified from those actually found in the Boeing model.

Results

The results of running CoBaSA on the Boeing examples are summarized in Table 2. For each version of the model, the original CoBaSA description file size is shown. In addition, the size, number of variables, and number of clauses for the CNF portion, and the size and number of constraints of the linear constraints portion of the pseudo-Boolean problem generated are given. These give an idea of the relative complexity of the problems given to PBSAT solver. Note that the CNF portions of the pseudo-Boolean problems are relatively small, but the linear constraints portions are significantly larger.

Note that Table 2 includes two extra models, models 6 and 7, that are significantly more complicated than the previous five models. For example, the size of model 7 is over half

Version		CNF			Linear		Results		
#	File Size	File Size	Vars	Clauses	File Size	Constraints	Compilation	Solving	Result
1	45.9K	24.7K	2704	1768	133.2K	371	16.55s	0.05s	SAT
2	59.7K	49.9K	2952	3748	138.3K	409	35.76s	0.00s	UNSAT
3	57.7K	66.0K	3971	4861	180.9K	421	42.59s	0.15s	SAT
4	59.9K	70.7K	3773	5620	175.3K	421	57.92s	2.3s	SAT
5	47.9K	28.3K	2592	2312	133.7K	371	21.68s	0.00s	UNSAT
6	364.5K	34.1K	12938	2018	331.5K	2320	181.50s	2.82s	SAT
7	521.3K	31.2K	12781	1876	463.9K	2339	367.7s	1.87s	SAT

Table 2: Summary of Experimental Results

a megabyte. Models 6 and 7 are even more complete and complex versions of the previous five models.

The results section of the table show the compilation time, PBSAT solving time, and satisfiability results for each model. Parsing and compiling take up the bulk of running time, due to the fact that our compilation code has not been optimized for performance. In addition to "SAT", CoBaSA gave us the assignment in human readable format, which the Boeing engineers confirmed was a correct satisfying assignment. None of the problems, satisfiable or unsatisfiable, took longer than 370 seconds to complete. This is in stark contrast to the much greater time taken by Boeing to find satisfying assignments using their current techniques.

With such low solving times, we were able to make significant architectural changes and find new configurations for the models. For example, the original CoBaSA problem that we received for version 3 of the model contained only 16 LRMs. There was no satisfying assignment, so we increased the number of LRMs to 18 and so on until we found the satisfying assignment using 22 LRMs. This entire exercise took only several minutes using CoBaSA, but would have taken orders of magnitude longer if done using Boeing's current methods.

5. RELATED WORK

Component-based software development (CBSD) is a well-studied area, and overviews are available in standard textbooks [31, 16, 12, 14, 32]. While there has been work on checking configurations of systems developed using CBSD (*e.g.*, [5, 10]) and on verifying component assemblies [33], automated component assembly requires further attention [15, 13].

Software architecture [24] has evolved to the point that effective tools for the development and analysis of software architectures are available [4, 29]. These include tools and techniques for automatically assembling components when given a detailed architecture description [17, 11]. While these tools can resolve the semantic issues of integrating components whose interfaces mismatch, they again do not generate the architecture themselves. For example, the GenVoca tool [7, 6], is a domain independent tool that generates a hierarchical software system from component specification and composition rules. The tool is effective as long as the software architecture specifies how each component is connected to other components. These tools have no way of dealing with architecture descriptions expressed as assembly constraints.

Automatic component deployment [18] and dynamic component redeployment [21] make adjustments or modifications to the components after they have been deployed in order to satisfy the run-time constraints of the deployment architecture. This approach can be used when sufficient runtime environment information is unavailable at components assembly time. However, these approaches do not address the problem of automatic static assembly.

Another challenge that automated component based system assembly faces is the development of specifications for components. There has been considerable work on this, including work on architecture description languages (ADL). After comparing various ADLs, Medvidovic et al. [20] make a point that existing ADLs lack in their support for expressing extra-functional properties. Assembly constraints are essentially constraints on extra-functional properties of components, hence there is need to develop specifications languages with full support for expressing these properties. Unified modeling language (UML) is the de-facto standard for modeling software applications [12]. While UML is a good modeling language for describing component functional properties, tools for automating system assembly for UML do not exist. An interesting research problem would be to compile UML system assembly problems to CoBaSA.

The Artificial Intelligence community has studied the problem of enforcing constraints for system assembly. Most of this work falls either checks existing configurations [3, 8] or develops ontologies and frameworks for automatic configuration [22, 30]. None of this work automatically gives optimal configurations given a set of constraints, as CoBaSA does. There is, however, one notable exception, which converts the Feature Model problem to the Constraint Satisfaction Problem (CSP), for which there are AI-based solvers [9]. These give optimal solutions based on a user-provided optimization function. However, this technique does not scale to large industrial systems such as those found in IMA problems. The experimental results show that the authors' technique can handle only up to 25 components efficiently.

6. FUTURE WORK

In this section we outline some of the numerous extensions and potential applications we see for CoBaSA.

When analyzing models arising in practice, it is not uncommon that constraint satisfaction problems generated can be decomposed into several independent subproblems. This arises in cases where components are integrated in a hierarchical fashion and the integration choices at the lowest level are independent of the integration choices at higher levels. This suggests performing a static analysis to determine if an iterative decomposition is possible. We believe that this can lead to significant savings in running times.

Another possible extension is to use more sophisticated encoding schemes. For example, consider the number of

Boolean variables we introduce for maps: if M is a map from A to B, we introduce $|A| \times |B|$ variables. However, the number of functions from A to B is $|B|^{|A|}$ so by a suitable encoding scheme, we can make do with $O(|A| \log |B|)$ variables. Given the input restrictions imposed by pseudo-Boolean solvers, we will have to introduce extra variables, so it is not clear how much of a savings alternate encoding schemes will provide.

If a solution to the system assembly problem cannot be found, then it is important to provide an explanation to the user showing, in as simple a fashion as possible, why this is not the case. In general, this is a hard problem that is related to the fact that due to NP-completeness, we do not know how to provide a short certificate of unsatisfiability. However, in practice simple explanations often exist and it would be useful to develop techniques that find and report such explanations.

7. CONCLUSION

Component-based software development and design has the potential to significantly impact software development in the large, but several technical challenges remain. One of them is the component-based system assembly problem: which components should be selected and how should they be assembled so that the overall system requirements are satisfied? We propose what we believe is the first approach to automatically solving the system assembly problem directly from the system requirements. Our framework includes an expressive language for declaratively describing system-level requirements, component interfaces and dependencies, resource requirements, safety properties, objective functions, and various types of constraints, including replication, timing, scheduling, and separation constraints. We developed constraint solving algorithms that transform problems expressed in our language to PBSAT problems, allowing us to leverage the recent and on-going advances in SAT solving technology. We have implemented these techniques in the CoBaSA tool. To evaluate our framework, we presented an in-depth case study in which we successfully applied our work to several large industrial examples arising in the design of the Boeing 787 Dreamliner. Our techniques and the CoBaSA tool were able to easily handle the Boeing problems, solving in under a minute problems that require multiple person-weeks for Boeing engineers to just state and check. Our approach is applicable to a wide class of component-based system assembly problems, and for future work, we plan to explore applying it to other domains.

8. REFERENCES

[1] C. A. 651. Arinc report 651, draft 9. Technical Report 91-207/SAI-435, Airlines Electronic Engineering Committee, September 1991.

[2] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo-boolean solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[3] T. Asikainen, T. Männistö, and T. Soininen. Using a configurator for modelling and configuring software product lines based on feature models. In *Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC3)*, 2004.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[5] D. Batory and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering (IEEE TSE)*, pages 67–82, 1997.

[6] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, Volume 1(4):355–398, 1992.

[7] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *Software, IEEE*, Volume 11:89–94, Sep 1994.

[8] D. S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines, SPLC 2005*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[9] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2005.

[10] A. Bertolino and R. Mirandola. Modeling and analysis of non-functional properties in component-based systems. In *TACoS 2003: Proc. International Workshop on Test and Analysis of Component Based Systems*, volume 82 of *Electronic Notes in Theoretical Computer Science*, April 2003.

[11] F. Cao, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, and M. Auguston. A component assembly approach based on aspect-oriented generative domain modeling. *Electronic Notes in Theoretical Computer Science*, Volume 114:119–136, 2005.

[12] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[13] I. Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, December 2001.

[14] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002.

[15] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau. Automated component-based software engineering. *Journal of Systems and Software*, 74(1), January 2005.

[16] G. T. Heineman and W. T. Councill. *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[17] P. Inverardi and M. Tivoli. Software architecture for correct components assembly. *Lecture Notes in Computer Science*, Volume 2804, Nov 2003.

[18] S. Lacour, C. Perez, and T. Priol. A software architecture for automatic deployment of CORBA components using grid technologies. In *In Proceedings of the 1st Franco-phone Conference On Software Deployment and (Re)Configuration (DECOR 2004)*, Oct. 2004.

[19] F. Martin and C. Fraboul. Modeling and simulation of integrated modular avionics. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998. PDP '98.*, pages 102 – 110, 1998.

[20] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[21] M. Mikic-Rakic, S. Malek, N. Beckman, and N. Medvidovic. A tailorable environment for assessing the quality of deployment architectures in highly distributed settings. In *Component Deployment, Second International Working Conference, CD 2004*, volume 3083 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2004.

[22] S. Mittal and F. Frayman. Towards a generic model of configuraton tasks. In *IJCAI*, pages 1395–1401, 1989.

[23] Object Management Group (OMG). Response to the UML 2.0 OCL RfP Revised Submission, Version 1.6, 2003. `http://www.omg.org/docs/ad/03-01-07.pdf`.

[24] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, Vol. 17(4):40–52, 1992.

[25] P. J. Prisaznuk. Integrated modular avionics. In *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference (NAECON 1992)*, volume 1, pages 39 – 45, 1992.

[26] Rational Partners, Object Management Group. UML Notation Guide, Sept. 1997. `http://www.omg.org/docs/ad/97-08-04.pdf`.

[27] Rational Partners, Object Management Group. UML Semantics, Sept. 1997. `http://www.omg.org/docs/ad/97-08-04.pdf`.

[28] M. A. Sánchez-Puebla and J. Carretero. A new approach for distributed computing in avionics systems. In *ISICT '03: Proceedings of the 1st international symposium on Information and communication technologies*, pages 579–584. Trinity College Dublin, 2003.

[29] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, Vol. 21(4):314–335, 1995.

[30] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(4):357–372, 1998.

[31] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[32] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. SEI Series in Software Engineering. Addison-Wesley, 2002.

[33] K. Wallnau, J. Stafford, S. Hissam, and M. Klein. On the relationship of software architecture to software component technology. In *Proceedings of the 6th ECOOP Workshop on Component-Oriented Programming*, 2001.

[34] R. Warrilow. The avionics platform, 2004. See URL `www.smiths-aerospace.com/Press/TechPapers/`.

[35] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *18th International Conference on Automated Deduction, CADE'02*, pages 295–313, 2002.