

Architecture, Language, and Non-compositional Constraints

Erann Gat
Jet Propulsion Laboratory
California Institute of Technology
gat@jpl.nasa.gov

Abstract— In the realm of building construction and computer hardware, the word “architecture” means a set of features shared by a class of designs, or equivalently, a set of constraints on a class of designs (e.g. gothic architecture or RISC architecture). But in the realm of software, the meaning of “architecture” has changed to be more or less synonymous with “design.” This is unfortunate because the concept of architecture-as-constraint is potentially useful: properly chosen constraints can guide engineers towards good designs and away from bad ones (and, of course, poorly chosen constraints can have the opposite effect).

We can identify two distinct classes of architectural constraints. *Compositional* constraints are constraints on the structure of a software system, while *non-compositional* constraints are constraints on the mechanisms for constructing that structure. On this view, “structured programming”, for example, is an architecture that imposes non-compositional constraints on the use of the GOTO statement.

The concept of non-compositional constraints has been largely ignored by the software engineering community. This has been in some cases a serious impediment to progress. For example, the JPL Mission Data System (MDS) is an architecture based on the non-compositional constraint that spacecraft should be controlled using goals, which are defined as constraints on state variables over time intervals. This is a very simple and straightforward concept with significant benefits, but because it is a non-compositional constraint and therefore unfamiliar, MDS has not been received with unalloyed enthusiasm.

This paper offers an informal discussion of non-compositional constraints (NCC) in hopes of making the concept more familiar and accessible. It describes the relationship of NCC and programming languages, and the applicability of the concept to the problem of verification and validation of spacecraft autonomy software.

TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. COMPOSITIONAL AND NON-COMPOSITIONAL CONSTRAINTS...	2
3. NON-COMPOSITIONAL CONSTRAINTS AND LANGUAGE	2
4. APPLICATION TO AUTONOMOUS SPACECRAFT CONTROL ARCHITECTURES.....	2
5 RECAP	4
6 CONCLUSION	5
7 ACKNOWLEDGEMENTS	5
REFERENCES.....	5

1. INTRODUCTION

In the realm of software, the word “architecture” has come to be more or less synonymous with “design.” (The dictionary definition of architecture in the context of software is “The overall design or structure of a computer system, including the hardware and the software required to run it.”) But in the realm of building construction and computer hardware, the word “architecture” has a very different meaning, namely, a set of common features shared by a class of designs, or equivalently, a set of *constraints* on a design. For example, Romanesque architecture is characterized by round arches and narrow windows. RISC architecture is characterized by simple instruction sets and large numbers of equipotent registers.

On this view of architecture (architecture-as-constraints), designs *conform to* or are *instances of* architectures. The realm of computer software also has this concept, but there isn’t a term to describe it any more, the word “architecture” having been largely co-opted.

¹ IEEEAC paper #1107, Updated October 15, 2002. 0-7803-7651-X/03/\$17.00 © 2003 IEEE

There are a few residual examples of the word “architecture” retaining its original meaning. For example, a *three-tier* architecture is not a single design but a class of designs². Brooks’s *subsumption* architecture [Brooks86] originally referred to a class of designs³. But these are the only two examples I was able to find of architecture-as-constraint that were actually called architectures.

There are also examples of architecture-as-constraint in software that are not referred to as “architecture.” For example, *structured programming* can be viewed as a set of constraints on the use of GOTO statements. *Unix* was originally a single design, but now refers to a class of designs that share common constraints on the mechanisms for component interactions. *Object-oriented programming* is a set of constraints on the relationship between data and the operations on that data. All of these are examples of architecture-as-constraint. So the concept has not been entirely lost, but it has become moribund. It is much harder to think about a concept when there is no word for it.

The concept of architecture-as-constraint is potentially useful because well-chosen constraints can lead designers away from poor designs and towards good ones. But it is also potentially harmful if one attempts to design under poorly chosen constraints. It is therefore worthwhile to reintroduce this concept into the lexicon so that it can be discussed.

2. COMPOSITIONAL AND NON-COMPOSITIONAL CONSTRAINTS

When discussing software architecture-as-constraints it is useful to distinguish between *compositional* constraints, which constrain the *structure* of subsystem decomposition, and *non-compositional* constraints which constrain composition *mechanisms*. For example, the constraints of the three-tier architecture are compositional constraints because they constrain how to compose a system from subsystems. As a rule-of-thumb it is possible to express a compositional constraint as a block diagram (see figure 1).

In contrast, the constraints of *structured programming* are non-compositional. They constrain the *mechanism* for composing subsystems (limiting the use of the GOTO statement in favor of function calls, FOR and WHILE loops) but they do not constrain the *structure* of the composition. While it is possible to draw a block diagram of a system built according to the constraints of structured

² Actually, it is *two* classes of designs. It usually refers to an autonomy architecture consisting of a controller, sequencer, and deliberator [Gat98]

³ In Brooks’s seminal paper the term is introduced with the article “a”, not “the”, as in “a subsumption architecture.”

programming, it is not possible to draw a block diagram of structured programming itself, the way it was possible to draw a block-diagram of the three-tier architecture.

3. NON-COMPOSITIONAL CONSTRAINTS AND LANGUAGE

An important observation about non-compositional constraints is that they usually require some form of *language support* in order to realize their full benefit. For example, it is possible to do structured programming in assembly language (this is how structured programming was in fact originally done) but the full benefit of structured programming is usually not realized unless one uses a higher-level programming language like C that allows structured programming constructs to be entered directly. The reason for this is obvious: by using a higher-level language, the parser and compiler for that language can automatically enforce the non-compositional architectural constraints and relieve the programmer from the burden of having to enforce those constraints manually.

Another example: it is possible to do *structured error handling* in C despite the fact that the language provides no direct support for it. Idioms such as the following are ubiquitous in C code:

```
if ((result = f()) == ERROR) {
    print_error_message();
    exit();
}
else if ((result = g(result)) == ERROR) {
    print_error_message();
    exit();
}
else if ((result = h(result)) == ERROR) {
    print_error_message();
    exit();
}
```

This leaves the burden of enforcing non-compositional architectural constraints on the programmer, and so results in higher development costs and greater probability of error. The programmer can be relieved of this burden by a language like C++ that provides direct support for structured error handling in the form of exceptions. The result is much cleaner, more reliable code, e.g.:

```
try {
    result = h(g(f()));
}
catch (...) {
    print_error_message();
    exit();
}
```

4. APPLICATION TO AUTONOMOUS SPACECRAFT CONTROL ARCHITECTURES

The concept of non-compositional constraints has practical application in the area of autonomous spacecraft

control. Traditionally spacecraft have been controlled using time-based sequences of commands. These sequences are essentially little programs where time serves as the “program counter”. (Notice that this is a non-compositional constraint.) The advantage of this design is predictability: the spacecraft’s behavior can be modeled on the ground to insure that a command sequence will have its intended effect.

The disadvantage of the traditional architecture is a lack of robustness in the face of unexpected events, particularly hardware faults, which are usually handled by entering safe mode. Unfortunately, there are cases where safe mode isn’t safe (during an orbit insertion for example, where entering safe mode can result in loss of mission). Such cases require *critical sequences* which can run with the usual fault-protection disabled. Critical sequences are very difficult and expensive to generate. A single critical sequence can consume a significant portion of a mission’s total development budget.

(Incidentally, note that although the command-and-control architecture is based on a non-compositional constraint – the use of time as a program counter -- the traditional spacecraft *software* architecture (within which the traditional spacecraft *command and control* architecture is embedded) is compositional, as shown by the fact that you can draw a block diagram of it (see figure 2).)

New classes of missions currently in design require greater degrees of autonomy than the traditional control architecture can easily provide. In some cases, like comet rendezvous and sample return missions, most of the mission becomes one big critical sequence.

In such cases there is reason to believe that the traditional control architecture will fail, and that this failure cannot be easily repaired. I will support this claim by appealing to an analogy, to which I now make a brief digression.

In ancient times the craft of constructing buildings out of stone reached dramatic heights, both literally and figuratively. But the physics of stone as a building material place a fundamental limit on how tall a stone building can be before it will collapse under its own weight. To build a skyscraper requires a paradigm shift: new building materials, and new construction methods.

I argue that an analogous situation exists in spacecraft software. The fundamental informatics (the “physics” of software) of the traditional control architecture place fundamental limits on what can be achieved. To illustrate this, consider the following simple command sequence:

At time T1 do A
At time T2 do B

(Assume $T1 < T2$.)

Now consider the question of what happens if A fails. How does this impact the execution of B? There is no way to know; the information just isn’t there. It has been *compiled away* during the sequence design process. Without more information there is just no way for a control system, no matter how cleverly designed, to deal intelligently with a failure in a traditional spacecraft control sequence. If we want to build a command and control architecture that can deal intelligently with failures and other unexpected events we must *add information* somehow.

There are at least two possible ways to add the required information. One is to augment the traditional sequencing ontology with additional control constructs, such as conditionals and loops. For example:

At time T1 do A.
If that succeeds then
 At time T2 do B
otherwise
 at time T3 do C.

The second approach is to add declarative information, e.g.:

Step 1: At time T1 do A.
Step 2: At time T2 do B.

NOTE: The successful completion of Step 1 is a necessary precondition for Step 2.

Or:

At time T1 do A.
The result should be that the camera power state is ON.

Or:

Make the power state of the camera be ON.

Note that in the last example no actions have been specified, only a desired end-state.

The JPL Mission Data System (MDS) [Dvorak00, Gat00b] currently under development includes a new command and control architecture based on an extension of the last example. In MDS, spacecraft are commanded in terms of *goals*, which are defined as *constraints* on *state variables over time intervals*. States in turn are defined as *properties of objects*, e.g. the *power-state* of the *camera*, or the *position* of the *spacecraft*. An MDS time interval is defined in terms of start and end *time points*, which are not necessarily fixed in time, but can “float” subject to *temporal constraints*, which specify

minimum and maximum elapsed time relative to other time points. A set of goals whose time points are related to one another by temporal constraints is called a *goal net*.

Goal nets subsume traditional time-based sequences as a special case, but they are vastly more expressive than sequences in ways that can be exploited for many purposes. For example, goals include their own correctness specification, so merely by adopting the ontology of goals one has already made significant progress towards software verification and validation, since the first step of any V&V effort is to specify what the behavior of the system ought to be when it is operating correctly.

5 RECAP

So far this paper has advanced the following argument:

1. There is utility in the concept of architecture as a set of constraints on a class of designs. Well-chosen constraints can lead designers towards good designs and away from bad ones.
2. Architectural constraints in the realm of software can be divided into two broad categories: compositional constraints, which govern the structure of a system, and non-compositional constraints, which govern the mechanisms used to achieve that structure.
3. Non-compositional constraints often require language support to realize their full benefit.

The support for this argument is weak, appealing mainly to anecdotal evidence and intuition. At best, points 1-3 constitute a hypothesis. But this hypothesis makes a testable prediction:

An effort to construct an autonomous spacecraft control architecture will progress more quickly if it employs either 1) a programming language whose constructs match the non-compositional constraints of the architecture or 2) a programming language that allows such constructs to be added.

MDS is currently being developed in C++, which does not have constructs that match the fundamental non-compositional constraint of MDS, namely, that spacecraft should be commanded in terms of goals. C++ does have the ability to construct object-based representations of goals, but it can not be easily extended to directly *represent* a goal. In other words, the only way to create a goal within C++ is to write C++ code that constructs an instance of a goal class.

To see why this may not be the best option, consider what it would be like if one had to do structured programming in this manner. Instead of:

```
While (condition) do { procedure; }
```

One would have to write something like:

```
Class MyTest {
    Boolean testIt() { return
condition; }
}
Class MyBody {
    Void doIt() { procedure; }
}
WhileLoop w = new WhileLoop();
w.test = new MyTest();
w.body = new MyBody();
w.run();
```

There are other shortcomings of C++ that have motivated the MDS project to seriously consider replacing it with Java, but note that Java does not help at all with respect to the issue at hand. In fact, the Java thread model and the “Runnable” interface exist precisely because Java does not have a “spawn” construct, and also does not have the meta-linguistic abstraction capabilities needed to add such a construct to the language.

At this point it would seem that we have reached an impasse. Programming languages like C++ and Java are the results of hundreds of work-years of design and implementation effort. Their designs are set by standardization committees, and making changes or additions takes years. It would seem that we have no alternative but to simply make the best of the situation, and perhaps hope that something better will come along in ten years or so.

But there are actually three alternatives:

The first is to use one or more small custom-design domain-specific languages (DSLs) and write interpreters (or compilers) for them in the base implementation language. This approach is being used on MDS, where a small DSL called GEL (Goal Elaboration Language) has been designed specifically to express goals. Another example is the Virtual Machine Language (VML) under development at JPL [ref##].

The second is to use automatic code generation to compile a superset of an existing language down into the base language. This approach has been used in numerous contexts, including the DS1 fault protection system [Rouquette99].

The third is to use languages with meta-linguist abstraction capabilities such as Scheme [Kelsey01] or Goo [Goo]. This approach was used on the Remote Agent Executive [Pell98] where a domain-specific language called ESL was implemented as a direct extension to Common Lisp. The efficacy of this approach is illustrated by the fact that ESL consists of only about 2000 lines of source code. There is also some statistically significant evidence that Lisp can significantly reduce schedule risk

in software development projects [Gat00].

6 CONCLUSION

The purpose of this paper has been to suggest some directions for future research, and to argue why I believe those directions might prove fruitful. In particular, I argue it is beneficial to think explicitly about non-compositional architectural constraints, and about language constructs to provide support for those constraints. The cost of providing such support can be much lower than is commonly believed. I have supported my argument with anecdotal evidence only. Accordingly, I do not claim to have demonstrated anything here.

7 ACKNOWLEDGEMENTS

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[Brooks86] Rodney A. Brooks, "A Robust Layered Control System for a Mobile Robot", *IEEE Journal on Robotics and Automation*, vol RA-2, no. 1, March 1986.

[Dvorak00] Dan Dvorak and Robert Rasmussen. "Software Architecture Themes in JPL's Mission Data System." *Proceedings of the IEEE Aerospace Conference*, March 2000.

[Gat97] Erann Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. *Proc. of IEEE Aeronautics (AERO-98)*, Aspen, CO, IEEE Press, 1997.

[Gat98] Erann Gat. Three-Layer Architectures. in D. Kortenkamp et al. eds. *AI and Mobile Robots*. AAAI Press, 1998.

[Gat00a] Erann Gat. Lisp as an Alternative to Java. *Intelligence* 11(4): 21-24, 2000.
Scheme

[Gat00b] Erann Gat. The MDS Autonomous Control Architecture. World Automation Congress (WAC), 2000.

[Goo] Goo (Generic Object Orientator) is a programming language invented by Jonathan Bachrach. There are no formal publications about Goo, but there is extensive documentation on the Web at <http://www.googoo.org/>

[Kelsey01] R. Kelsey, W. Clinger, J. Rees (eds.), Revised⁵ Report on the Algorithmic Language Scheme, Higher-

Order and Symbolic Computation, Vol. 11, No. 1, September, 1998.

[Pell98] Barney Pell and Erann Gat. Smart Executives for Autonomous Spacecraft. *IEEE Intelligent Systems*, October 1998.

[Rouquette99] Nicolas Rouquette. The 13th Technology of Deep Space One. *Proceedings of the IEEE Aerospace Conference*, 1999.

[Steele90] Guy L. Steele Jr., *Common Lisp: The Language*, Second Edition, Digital Press, 1990.



Erann Gat is a principal scientist at the Jet Propulsion Laboratory, California Institute of Technology, where he has been working on autonomous control architectures since 1988.

A Three-Layer Architecture

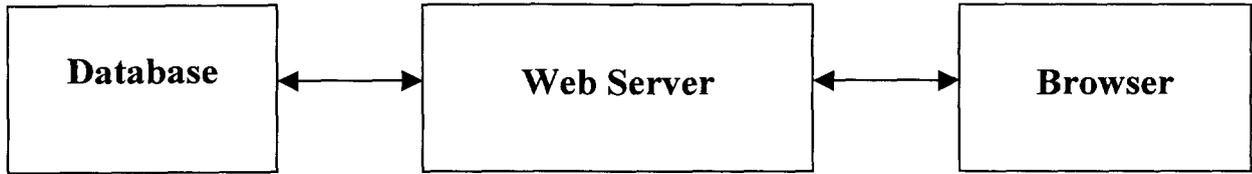


Figure 1: Compositional constraints can be expressed as block diagrams.

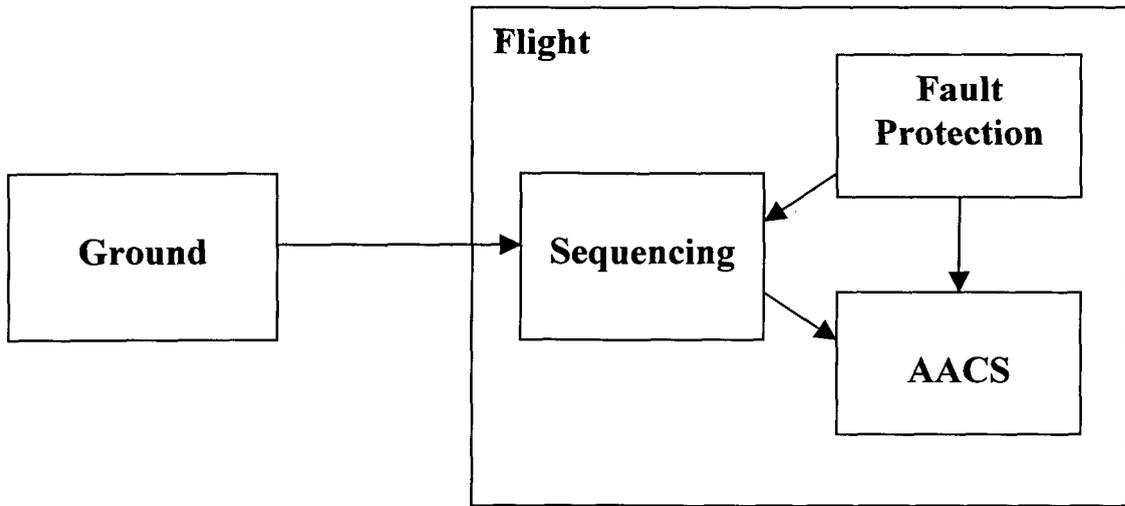


Figure 2: The traditional spacecraft software architecture is based on compositional constraints.