PAUL ANDERSON

programming the virtual infrastructure

Paul Anderson is a researcher at the University of Edinburgh. He has a background in practical system administration and he is particularly interested in bridging the gap between theory and practice. His homepage is http://homepages.inf.ed.ac.uk/dcspaul.

dcspaul@ed.ac.uk

OVER THE PAST FEW YEARS, THERE seems to have been a growing demand for practical system configuration tools but depressingly little progress. The recent arrival of virtual machines seems only to have increased the difficulties, as well as the need. However, the "programmable" nature of these "virtual infrastructures" made me wonder whether there was anything to be learned from the corresponding development of programming languages for the early computers. It seems that there are some very interesting analogies, but in the end, the problem might be sufficiently different that a good solution is not likely to be available for some time.

In my early teens, I was fascinated by electronics. I've always liked creating things, and I could build devices that seemed to have a life of their own—even if they just played random musical notes or flashed colored lights. I met my first computer when I was around fifteen—we had a school trip to a local engineering firm that ran an Elliot 905, and I was hooked. We had a chance to write and run our own programs, and I remember writing Fortran code to do some numerical integration. The Fortran compiler had to be loaded off paper tape and followed by the source code. This then produced a paper tape of the object code, which you could run. The machine flashed its lights and hummed a tune as it moved the data around, then the answer came out on a teletype roll. Sometimes you could fix small errors by manually punching extra holes in the tape.

I went on to work for ICL, a UK computer company formed in 1968, during the design of the 2900 series, learning more about hardware and how to program in machine code. But it was around 1979 before I actually got my hands on my own computer hardware—I built an Acorn System I from a kit, soldering in the chips and programming in hex. Then I started working with larger microprocessor systems that supported higher-level languages and more sophisticated software. This was an interesting time because the hardware was still simple enough that it was possible to understand the whole process. I designed and built a video framegrabber, including the analog electronics—but I also wrote the LISP interpreter that processed the images.

As things became more complex, this kind of generalization got a lot harder. I was seduced by the possibilities of the first Apple Macintosh and the Sun 3/50. I wrote software in high-level languages that ran at a layer once removed from the hardware. Even the hardware design people were using software and simulations and programmable chips.

Eventually, I became interested in managing lots of machines. All of the associated system administration problems are now well documented, but at the time this was uncharted territory—how do we get the software onto all these machines without loading each one individually? How do we set up the configuration files so that the clients talk to the right servers? How do we make sure it all stays up-to-date and correct? How do we make sure that the whole installation actually does what it is intended to do?—if we even have a clear idea of what it is supposed to do in the first place!

For the past few years I've been working on "autonomic systems" and trying to create infrastructures that can reconfigure automatically in response to failures or problems with loading. But the recent explosion of interest in virtualization has increased the complexity by another order of magnitude, with modern datacenters supporting virtual machines that migrate around the physical hardware. The network connections are established by programming VLANs, and the storage comes from network-attached devices. This means that the entire datacenter is now programmable—we can rewire network connections, change storage sizes, and replace failed hardware, all by "reprogramming the virtual infrastructure."

This all begins to sound very like the early days of computer programming when software replaced the hard-wired connections. So I started to think about the analogy, and I wondered what we could learn from the way in which computer programming has evolved.

> The charm of history and its enigmatic lesson consist in the fact that, from age to age, nothing changes and yet everything is completely different.
>
> *Aldous Huxley*

It seems clear that there have been definite steps in the evolutionary process. When a particular level becomes complex enough, a new layer of abstraction develops to advance the technology to the next stage—transistors, chips, assembler code, high-level languages, operating systems, etc.—and each stage comes with new techniques, new theories, and a new generation of specialists. Yet, over the past 10–15 years, I've spent a lot of time thinking about the problems of configuring large computing installations, and I am often frustrated by how little progress there seems to be. Compared to the rigor in designing a new chip, most computing installations are set up in a very ad hoc way—there is no systematic process nor a way of demonstrating the correctness, nor is there even a clear idea of the overall specification. Why is this?

Some History

Wikipedia has a good description of some of the early computers. EDVAC was one of the first computers to support "stored programs." Before that, machines such as ENIAC had used switches and patch leads to set up the instructions. This sounds familiar to me—there was a time when I could change the network port in my office by walking down the hall and changing the patch panel. Now I have to find someone who knows how to make the right incantation to the switches! But these early computers were still programmed by the engineers with an intimate knowledge of the hardware.

In 1953, John Backus proposed the idea of a what we would now call a "higher-level" language for the IBM 704. His team created the Fortran language and the first compiler. It's fascinating to read about this process and think about the analogy with today's system configuration tools. The initial motivations were very similar—for example, efficiency. The cost of the programmers associated with a computer installation was higher than the cost of the computer itself:

> The programmer attended a one-day course on Fortran and spent some more time referring to the manual. He then programmed the job in four hours, using 47 Fortran statements. These were compiled by the 704 in six minutes, producing about 1000 instructions. He estimated that it might have taken three days to code this job by hand. [1]

Correctness (hence reliability) was also a manual process:

> He studied the output (no tracing or memory dumps were used) and was able to localise his error in a Fortran statement he had written. He rewrote the offending statement, recompiled and found that the resulting program was correct. He estimated that it might have taken three days to code this job by hand, plus an unknown time to debug it. [1]

Of course, there were other benefits too; programs were now portable between different machines, and the language was much closer to the statement of the problem to be solved. This meant that users themselves could learn one language and their programs would run on almost every computer created since that time. However, this new concept of "automatic programming" wasn't universally accepted; many people were concerned about the efficiency of the code. Backus and Heising emphasized how much the fear of not being able to compete with hand-coded assembler code influenced the design of the first Fortran compiler. And my favorite quote comes from Irvine Ziller—one of the original Fortran team:

> And in the background was the scepticism, the entrenchment of many of the people who did programming in this way at that time; what was called "hand-to-hand combat" with the machine. [2]

This definitely reminds me of the time I have spent trying to convince people to configure systems by using the tools, rather than simply hand-editing some configuration file because "it is an emergency" or "it is only a one-off."

Languages

Most practical configuration languages have not been specifically "designed"—their functions are often related closely to the operations provided by a particular tool, and their syntax and semantics have not been carefully thought out. It is interesting that the same was true of the early programming languages. John Backus writes:

> We simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs. [3]

Of course, this was all to change—people soon realized that translating the problem description into a usable program was the biggest source of effort and errors. In the fifty years since then, there has been a huge amount of work on programming languages and their related theory. (See Figure 1 for a timeline.) Backus himself gave his initial to the BNF notation, which he co-developed for describing the formal syntax of programming languages. Periodically, different paradigms have appeared. Over the years, some of these have become accepted, and others have faded away.

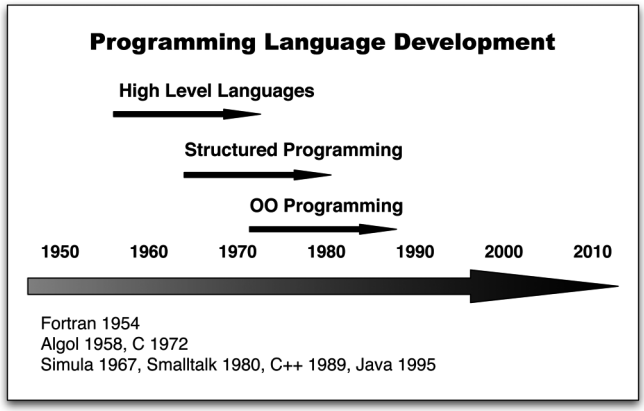

FIGURE 1: TIMELINE FOR DEVELOPMENT OF PROGRAMMING LANGUAGES, SHOWING APPROXIMATE DATES

I was quite surprised when I looked into this to see the amount of time between the "invention" of a language and its acceptance as a common production tool. Something like 10–15 years doesn't seem to be atypical. It appears to take this long for people to become comfortable with a new approach, for the features of the language to be refined, and for implementations to be accepted as stable. Of course, the increasing power of the machines along with the increasing complexity of typical code also changes the balance. If we look at the development of configuration languages on the same scale (Fig. 2), perhaps we shouldn't be surprised at the apparent lack of progress.

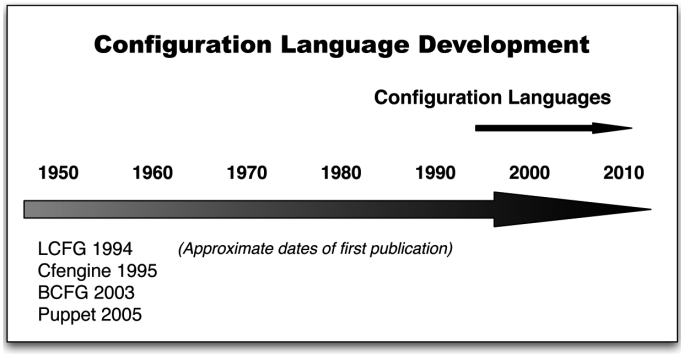

FIGURE 2: TIMELINE FOR DEVELOPMENT OF CONFIGURATION LANGUAGES, SHOWING APPROXIMATE DATES

It would be a mistake to try and draw too much of an analogy between programming and configuration languages, but it is interesting to look briefly at a few ideas and see what we can learn—both from the features that evolved and from the process itself.

Raising the Level of Abstraction

Early programming languages were based on a model that was close to the operation of the hardware. Large arithmetic expressions could be translated into multiple instructions, but the flow of control had to be specified explicitly (using conditionals and branches) in a way that mapped fairly directly onto the underlying hardware. The next significant step was the introduction of structured programming. Here, the explicit control flow was replaced with control structures; these mapped more closely onto the kind of operations that people wanted to model. Independent routines with local variables also made it easier to reuse code and for multiple programmers to work on the same project. Modern programs are supported by frameworks and op-

erating systems that deal with entities at a much higher level of abstraction, such as files and windows.

Current configuration languages still seem to operate at a level close to the hardware—manipulating files and processes, for example. There is a big gap between this and the level at which a system administrator normally wants to talk about the infrastructure—in terms of services (mail, Web, database, etc.) and the migration strategies for the virtual machines, for example. It is certainly possible to configure multiple systems with a single statement, and it is common to have constructs that encapsulate concepts such as "Web server." But even the ability to exchange a "Web server" configuration between two sites is rare—certainly if we include all the associated consequences, such as DNS entries and firewall holes.

It is not entirely clear why there has been so little progress in raising the abstraction level. The virtual infrastructure certainly presents a few problems, such as the distributed and unreliable nature of the underlying system, that are not present when programming a single machine. However, CIM, for example, provides one possible way of modeling entities at a much higher level. Perhaps one difficulty is the relative complexity of the underlying "machine"—the infrastructure is complex and it changes rapidly as new software and services are added. System administrators tend to need a more agile approach, preferring Perl to Java and Cfengine to CIM. Or perhaps it will simply take a few more years for the appropriate paradigms to emerge.

Declarative Programming

Existing configuration languages are often "declarative" (to varying degrees). This means that the user specifies the desired configuration, and the tool works out what it needs to change to make this true. For example, you might specify that a configuration file should contain a certain line. The tool will then add that line, only if it is not already present. There isn't space here to go into detail, but declarative configuration languages have a lot of practical advantages. The problem, though, is that the tool has to work out the necessary steps by itself. This is fine when things are simple (as in the example here), but if we are specifying, say, the relationship between a set of virtual services, then working out the deployment steps can be much more complex; the placement of the virtual machines, their configuration, and the order in which we move things are all important. This may be too complex or too critical to leave completely to some automatic process.

General-purpose declarative languages such as Prolog have been around since the 1970s, but they remain confined to a comparatively small number of applications for similar reasons. Indeed, the configuration situation is actually more difficult, because the intermediate states of a configuration change may be important, whereas the intermediate states of a computation are purely internal.

Conclusion

"Automatic programming" of the virtual infrastructure is hard. It is not easy to specify correctly what is required. Translating high-level requirements into implementable specifications is hard. The languages are immature and contain considerable accidental complexity. The solutions can be difficult to compute, and automatic solutions may be difficult to understand and trust. I suspect that the idea of fully "automatic configuration," from a declarative service description, is really a myth. At several points in the history of programming, new approaches have led to talk of the "death of the program-

mer." Certainly the programming problems change, but they don't really become easier—they just enable a new level of abstraction.

I am still interested in languages and ways of specifying configurations. In the future, there may well be completely new approaches that provide a new degree of automation. But recently I've become interested in frameworks that might support a better integration of manual and automatic processes—this is inspired by a similar approach in AI research [4], and it may provide a smoother transition toward more automation, as the techniques become available and accepted.

REFERENCES

[1] J.W. Backus et al., "The FORTRAN Automatic Coding System": http://archive.computerhistory.org/resources/text/Fortran/102663113.05.01.acc.pdf.

[2] *The Bulletin of the Computer Conservation Society*, Number 41, Autumn 2007: http://www.cs.man.ac.uk/CCS/res/res41.htm.

[3] Computer History Museum, Fellow Awards, 1997—John Backus: http://www.computerhistory.org/fellowawards/index.php?id=70.

[4] I-X: Technology for Intelligent Systems: http://www.aiai.ed.ac.uk/project/ix/.