Why Haven't We Automated Programming

Robert Balzer
Teknowledge
balzer@teknowledge.com

Categories and Subject Descriptors
D.1.2 Software, PROGRAMMING TECHNIQUES, Automatic Programming

General Term: Design

Keywords: Interactive, Refinement.

1. INTRODUCTION
The notion of automatic programming has fascinated our field since at least 1954 when the term was used to describe early Fortran compilers. The allure of this goal is based on the recognition that programming was, and still is, the bottleneck in the use of computers.

While we have witnessed phenomenal improvements in hardware capability with corresponding decreases in cost in the intervening decades, software productivity has progressed much more slowly. Despite the development of high-level, and then higher-level, languages, structured methods for developing software, and tools for configuration management and tracking requirements, bugs, and changes, the net gain is nowhere close to that realized in hardware.

More importantly, the basic nature of programming remains unchanged. It is largely an informal, person-centered activity which results in a highly detailed formal object – the program. This manual conversion from informal requirements to programs is error prone and labor intensive. Software is produced via some form of the Waterfall, Iterative, or Spiral development model in which a linear progression of phases is expected to yield correct results. Rework and maintenance are afterthoughts.

Unfortunately, this existing software paradigm contains two fundamental flaws which exacerbate the maintenance problem. First, there is no technology for managing the knowledge-intensive activities which constitute the software development processes. These processes, which convert a specification into an efficient implementation, are informal, human intensive, and largely undocumented. It is just this information, and the rationale behind each step, that is crucial, but unavailable, for maintenance. (This failure also causes problems for the other lifecycle phases, but is particularly acute for maintenance because of the time lag and personnel changeover normally involved, which precludes reliance on the informal mechanisms such as "walking down the hall," typically used in other phases.)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11...$10.00.

Second, maintenance is performed on source code (i.e., the implementation). All of the programmer's skill and knowledge has already been applied in optimizing this form (the source code). These optimizations spread information; that is, they take advantage of what is known elsewhere and substitute complex but efficient realizations for (simple) abstractions.

Both of these effects exacerbate the maintenance problem by making the system harder to understand, by delocalizing information , and by thereby increasing the dependences among the parts (especially since these dependences are implicit).

With these two fundamental flaws, plus the fact that we assign our most junior people to this onerous task, it is no wonder that maintenance is such a major problem with the existing software paradigm.

Against this background, it is clear why automatic programming commands so much interest and appeal. It would directly obviate both these fundamental flaws through full automation of the compilation process.

But programming is more than just compilation. It also involves some means of acquiring the specification to be compiled and some means of determining that it is the intended specification. Furthermore, if one believes, as we do, that optimization cannot be fully automated, it also involves some interactive means of translating this high-level specification into a lower-level one which can be automatically compiled.

This is the extended automatic programming problem, and the main issue is how to solve it in a way that still obviates the two fundamental flaws in the current paradigm.

2. The Extended Automatic Programming Problem
Automatic programming has traditionally been viewed as a compilation problem in which a formal specification is complied into an implementation. At any point in time, the term has usually been reserved for optimizations which are beyond the then current state of the compiler art. Today, automatic register allocation would certainly not be classified as automatic programming, but automatic data structure or algorithm selection would be.

Thus, automatic programming systems can be characterized by the types of optimizations they can handle and the range of situations in which those optimizations can be employed. This gives rise to two complementary approaches to automatic programming. In the first (bottom-up) approach, the field advances by the addition of an optimization that can be automatically compiled and the creation of a specification language which allows the corresponding implementation issue to be suppressed from specifications. In the second (top-down) approach, which gives up full automation, a desired specification language is adopted, and the gap between it and the level that can be automatically compiled is bridged

interactively. It is clear that this second approach builds upon the first and extends its applicability by including optimizations and/or situations which cannot be handled automatically.

Hence, there are really two components of automatic programming: a fully automatic compiler and an interactive front-end which bridges the gap between a high-level specification and the capabilities of the automatic compiler. In addition, there is the issue of how the initial specification was derived. It has grown increasingly clear that writing such formal specifications is difficult and error-prone. Even though these specifications are much simpler than their implementations, they are nonetheless sufficiently complex that several iterations are required to get them to match the user's intent. Supporting this acquisition of the specification constitutes the third and final component of the extended automatic programming problem.

The rest of this paper sketches the types of capabilities that would be required for such a front-end that interactively supports the acquisition of a high-level specification and its interactive refinement to a level that can be automatically compiled.

To maximize the separation between a specification and an implementation, we add uncomputable constructs to the specification language that allow it to infer predicates, achieve states, maintain predicates, trigger actions when predicates become true or false, and range across infinite and/or unspecified sets – without identifying (initially) how those constructs are implemented.

The power of this proposed approach lies in the generality of these constructs (i.e. their applicability to a wide range of domains) and the ability to interactively refine them into efficient computable implementations.

A syntax-free pseudo natural language will be used for the specification language to facilitate its understandability for both the writer and readers of the specification and to focus the discussion on the capabilities required to separate the specification from its implementation and which must be interactively refined away into an implementation of that specification that can be automatically compiled.

3. High Level Specification Language

For the example we will consider – an air traffic control system – only three of the suggested uncomputable specification constructs are required:

1. **Achieve X while preserving Y**: put the system into a state in which the predicate (X) of the achieve statement is satisfied and the constraints (Y) have not been violated in any of the intermediate states.
2. **Whenever X do Y**: in every state in which the predicate (X) becomes true, do Y.
3. **Find a/all X such that Y**: find an (or all) objects (X) that satisfy the predicate (Y).

Using these constructs, the highest level specification of an air traffic control system would be:

> **For each plane fly a safe route from its origin to its destination.**

> **A safe route maintains a minimum distance between the plane flying that route and all other planes.**

This specification uses the Find and Achieve constructs to find a route for the plane which maintains the minimum distance between it and other planes while traversing that route. It doesn't specify how such a route is found, whether it remains constant throughout the traversal, or how it or other routes are adjusted if the addition of new safe routes would cause the minimum distance constraint to be violated.

4. Interactive Implementation

Suppose the human designer decides to implement the safety constraint on routes by enclosing them in tubes of airspace that are separated from each other by (at least) the minimum distance. Then, the minimum distance safety constraint need only be verified when new airspace-tubes are added rather than every time the position of a plane changes.

This implementation decision would be recorded and would change the specification to:

> **For each plane fly a route from its origin to its destination within a safe airspace-tube.**

> **A safe airspace-tube is a cylinder of airspace between a source and a destination that maintains a minimum distance between itself and all other safe airspace-tubes.**

The specification doesn't specify how such airspace cylinders are found, whether they remain constant throughout the traversal of a plane through them, or how these cylinders are adjusted if the addition of a new one would cause the minimum distance constraint to be violated.

Now suppose that the human designer decides to reuse the same airspace tube for all (one-way) traffic from a source to a destination (thus greatly simplifying the selection of safe routes). With multiple planes within an airspace tube, the constraint separating the tubes from one another is no longer sufficient to ensure that the planes within a single tube maintain a safe distance from one another. This can be fixed by requiring each plane to remain a safe distance behind the plane, if any, within the tube that is immediately ahead of it (implicitly forcing the planes to fly single-file through the tube).

This implementation decision would be recorded and would change the specification to:

> **For each plane fly a route from its origin to its destination within a single safe airspace-tube from the origin to the destination.**

> **Each plane within an airspace-tube maintains a minimum distance from the plane, if any, immediately ahead of it in the airspace-tube.**

> **A safe airspace-tube is a cylinder of airspace between a source and a destination that maintains a minimum distance between itself and all other safe airspace-tubes.**

Hopefully the reader can image this process of iterative refinement continuing through many more iterations to reach a point at which none of the high-level constructs introduced to simplify the

description of the specification remain and the resulting program can be automatically compiled.

The rest of this paper focuses on two special types of changes – revision of the top-level specification and revision of an implementation approximation.

5. Specification Revision

Suppose that the specification is revised to say that whenever possible planes should avoid storms and strong headwinds and should take advantage of strong tailwinds. In the original specification, this just becomes another (soft) constraint on the route that a plane flies from its origin to its destination. The question is how this soft constraint percolates down through the iteratively refined layers of the specification.

At the first refinement level, where a safe airspace-tube encapsulates a single plane's route, the conditions to be avoided or sought are either known before the plane's route is selected or they arise during its flight. In either case, all that is required is to add a safe airspace-tube from the plane's current position to its destination that satisfies those known conditions. This change would be expressed as:

> **Whenever conditions to be avoided or sought develop within the unflown portion of an airspace-tube, fly a route from its current position to its destination within a safe airspace-tube.**

However, at the next refinement layer where all planes flying from the same origin to the same destination are encapsulated in a single airspace-tube, the changes are much more substantial.

The multiply used airspace-tube must be stretched to include the modified routes for all planes it encompasses. Once this stretching has occurred, then all the planes can migrate to their modified routes. Finally once they have all traversed to the common route flown by all planes in the airspace-tube, then the airspace-tube can be shrunk to just include that common route.

> **Whenever conditions to be avoided or sought develop within an airspace-tube modify the route through the airspace-tube to avoid or incorporate those conditions, stretch the air-space tube to encompass both the original and the modified route as well as the space between these routes, and fly each plane within the airspace-tube from its current position onto the modified route within the airspace-tube. Once all planes are flying the modified route, the airspace-tube can be shrunk to just encompass that route.**

6. Implementation Approximation Revision

When we communicate with each other, we limit complexity by using abstractions that are approximations and later describing that abstraction more completely and accurately. We have no mechanisms for structuring specifications into such "elaboration layers".

Nevertheless, our air traffic control example utilizes such approximations to facilitate its concise and comprehendible description. These approximations are the origin/destination locations of the planes. Presumably these are either the takeoff/landing locations of the planes (to cover their entire flight path), or their departure/arrival gates (to cover both the ground movement of the plane within the airport and its flight path between airports).

It is clear that within the confines of the airport that the safe minimum (flying) distance can not be maintained between multiple airspace-tubes.

Rather, these origin/destination locations must be decomposed into a series of locations between the origin and destination locations for which a safe traversal of the plane can be ensured.

Thus, the original path from the origin to the destination must have intermediate locations that are in the vicinity of the origin and destination airports and represent the actual start and end points of the safe airspace-tube between those airports.

A single "departure" airspace-tube connects the origin airport's take-off location to the entire set of airspace-tubes encompassing traffic from that airport to other airports. The destination side of this airspace tube is stretched to incorporate the origin of all of these airspace-tubes. Similarly, a single "arrival" airspace-tube connects the destination airport's landing location to the entire set of airspace-tubes encompassing traffic to that airport from other airports. The origin side of this airspace tube is stretched to incorporate the destination of all of these airspace-tubes.

Finally, because planes within these departure and arrival airspace-tubes are not flying the same route, the maintenance of a minimum distance behind the plane immediately ahead of it must be re-expressed as maintaining a distance from the arrival airspace-tube destination (or destination origin) that is the minimum distance greater (less) than the distance of the plane immediately ahead of it from this location.

This revision of the approximation which retains the separateness of this elaboration layer could be expressed as:

> **Replace the safe airspace-tube from an origin location to a destination location by a safe airspace-tube from a location in the vicinity of the origin airport to a location in the vicinity of the destination airport and connect it to the origin airport's departure airspace-tube and the destination airport's arrival airspace-tube.**
>
> **Within each arrival and departure airspace-tube each plane maintains a distance from the arrival destination (or destination origin) that is the minimum distance greater (less) than the distance of the plane immediately ahead of it from this location.**

7. Conclusion

We have identified several constructs that facilitate a concise implementation-neutral specification and sketched a facility for interactively refining such specifications into compliable implementations.

It should be noted that Sections 1 and 2 of this position paper were extracted from a paper I wrote 25 years ago[1] describing a space of automatic programming systems. They provide the context for the rest of the paper that describes and illustrates the portions that have yet to be built.

These portions focus on the gap between the expressibility of the proposed uncomputable constructs and the highest level specifications used today – such as the domain specific languages used in Model Based Development [2,3] – which are designed to be formal, computable, and are either automatically compliable or easily transformed into compliable code.

The proposed approach does not presuppose the existence of such models, but in domains where they exist they would be fine targets for the interactive refinement of the portion of the design space currently unexpressible in those formalisms.

8. REFERENCES

[1] Balzer, R. 1985. A 15 Year Perspective on Automatic Programming. In IEEE Transactions On Software Engineering, VOL. SE-ll, NO. 11, (NOVEMBER 1985), 1257-1268

[2] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003, ISBN 0-471-31920-1

[3] Stephen J. Mellor, Anthony N. Clark, Takao Futagami, *Guest Editors' Introduction: Model-Driven Development,* IEEE Software, vol. 20, no. 5, pp. 14-18, Sep./Oct. 2003.