Summary

This paper discusses a set of experiences I had in developing ROCKET, an early application generator in the area of rocket flight mechanics at the Rand Corporation. It begins by describing a precursor system I worked on at General Dynamics. It also covers some experiences with the interactive Graphic ROCKET followon, and POGO, an early GUI builder. It concludes by relating these to the recent Megaprogramming and Domain Specific Software Architecture programs I was involved in at DARPA.

1. Rocket Trajectory Programs at General Dynamics

I entered the computer field in 1955 as a summer programming trainee at General Dynamics in San Diego. I joined General Dynamics full-time in 1957, working primarily on simulation of Atlas rocket trajectories. I didn’t realize it at the time, but several of the regular-guy programmers and engineers I worked with day-to-day would become heavyweights in the profession. For example, my carpool to work included Bob Price, who became the CEO of Control Data, and Donn Parker, who did a lot of the pioneering work on computer crime and computer ethics. In 1957-58, Parker was the leader of the Fortran movement within General Dynamics. Its adherents came in wearing Fortran T-shirts, and joked about the Neanderthal assembly-language programmers. I was one of the Neanderthals, still believing that Fortran wasted microseconds and restricted my essential access to the machine’s registers and exotic instructions.

It was an exciting time to be both in the computer business and the rocket business. One of my assignments involved the Atlas-Mercury astronaut program. The General Dynamics Atlas rocket engineers wanted to put an escape tower on top of the Atlas, that would blast free of the Atlas if it started getting into trouble. They weren’t sure that the escape tower would work in all the situations in which it was needed, and I got the job of analyzing it. This involved modifying a big assembly-language simulation of the Atlas rocket, running a lot of cases that might cause problems, and working with the rocket engineers to fix the problem situations. It really made me identify with the astronauts, who were putting their lives on the line that things like this would work. The escape tower did become part of the Atlas-Mercury flight vehicle, but fortunately never had to be used.

Again, I didn’t realize it at the time, but the big Atlas rocket simulation program was an early example of domain engineering, software product line architecting, and software reuse. The Atlas rocket product line included a number of options for rocket engines, aerodynamic configurations, and flight guidance algorithms. Their performance could be represented by either mathematical functions, tables, or polynomial approximations to the table values. Led by an unsung hero named Herb Hilton, the team developing the Atlas rocket simulation had encoded the most likely options as numerical sequence numbers. Thus, for example, the rocket engine simulation options were encoded roughly as:

00 - no thrust and fuel flow
01 - constant thrust and fuel flow
02 - constant thrust and fuel flow modified by atmospheric pressure
03 - Atlas V.1 booster thrust and fuel flow tables vs. time
04 - Atlas V.2 booster thrust and fuel flow tables vs. time
05 - etc.

Other sequences covered aerodynamic, guidance, and special output options. Thus, a simple unpowered coasting trajectory with a simple aerodynamic model and no
guidance could be represented by the sequence 00-01-00, while an Atlas V.1 booster with simple aerodynamics and a simple guidance scheme would be represented as 03-01-01.

Using this approach, most of the Atlas trajectory analyses desired by the rocket engineers could be accommodated by preparing input cards with these option sequences, plus some general parameters describing the rocket's launch location, initial weight, initial fuel weight, etc. It was also fairly easy to add new standard options. Exotic options, such as the Atlas-Mercury escape tower, involved special programming efforts. But overall, the approach provided great labor savings and rapid service via reuse of the standard Atlas rocket model components.

1.1 Rocket Trajectory Programs at Rand

In 1959, I left General Dynamics and went to the Rand Corporation in Santa Monica, working primarily as an engineering programmer-analyst. My primary motivation was that Rand was willing to have me work full-time while taking graduate courses toward a Ph.D. in math at UCLA (UC-San Diego didn't exist at the time, nor did any computer science departments).

My job at this time involved developing computer models for Rand's engineers and physicists. Unlike the Atlas rocket engineers, Rand's rocket engineers did characteristically exotic analyses. Some examples were rockets fired out the side of tall mountains, rockets carried above most of the atmosphere on huge airplanes, and aerospaceplanes cruising around converting atmospheric oxygen into rocket fuel before taking off into orbit.

After programming some of these special cases, I had a pretty good idea of how to develop a general-purpose rocket simulation to handle Rand's wide range of analyses. Fortunately, Rand's Engineering Department was interested in having such a program, and this became my main job in 1961. At this point, I'll spent some time describing the resulting rocket domain model and rocket trajectory application generator, as they formed the experience base for things I tried to do more recently with the DARPA Megaprogramming, STARS, and Domain Specific Software Architecture initiatives.

2. Rocket Trajectory Domain Architecting

2.1 Desired Technical Capabilities

The first step in domain architecting was to interview the Rand rocket engineers to determine the range of their desired capabilities. These are summarized below by contrasting them with the capabilities of the General Dynamics Atlas rocket model.

1. Wider ranges of aerodynamic, propulsion, and guidance models. Atlas was a liquid-fueled, basically cylindrical rocket with booster, sustainer, and vernier engines; and with basically preprogrammed guidance options. The Rand engineers needed to investigate solid rockets, various rocket shapes, various engine combinations, and various advanced guidance, navigation, and control options.

2. Wider range of environment models. The Atlas simulation included a fairly extensive set of Earth gravity and atmosphere models. The Rand engineers also needed to simulate rockets in the vicinity of the Moon and other planets, and to include such effects as solar radiation pressure on large, lightweight satellites.

3. Wider range of initiation and termination conditions. The Atlas simulation basically assumed a ground launch. It typically changed stages as a function of time or weight, or of altitude when the rocket finally came back to earth. The Rand engineers needed to be able to start in mid-flight or from aboard an aircraft. They had needs to change stages as functions of such quantities as velocity, flight path angle, relative time from the beginning of the stage, or the point at which the rocket's extrapolated trajectory
would reach a certain distance or altitude. A related need was for an alternate termination condition in case the primary termination condition could not be met.

4. Abilities to iterate on a control variable (e.g., a guidance parameter) to determine the value at which a resulting dependent variable (e.g., the extrapolated flight distance or range) would reach a desired value or reach its optimum value.

5. Abilities to investigate several options during a single pass on the computer. In the batch-sequential computer operations of the early 1960's, engineers wanted to get as much mileage as possible from each computer pass.

6. Abilities to provide special outputs, such as orbital parameters or the rocket's range, azimuth, and elevation with respect to one or more tracking stations.

7. Ability to specify higher or lower levels of simulation accuracy.

8. Ability to simulate backwards in time.

9. Ability to simulate rocket vibration effects.

10. Ability to simulate airplanes and helicopters as well as rockets.

It turned out to be feasible to accommodate all of these desired capabilities except the final two, which would have added significant complexity to the system.

2.2 Desired Operational Capabilities: User View

Besides the technical capabilities, there were also operational needs to make the program as easy to use and easy to modify as possible. This was also one of my main win conditions, as I would have to do all of the software maintenance myself.

Rand had an IBM 704 computer which ran the Share Operating System. It still ran in batch-sequential mode, in which engineers submitted card-deck inputs and got back printouts either a few hours later or the following morning. It had a number of amenities, such as the ability to compile programs and link them to system or programmer libraries of relocatable object modules. It did not have a job control language, but applications could be programmed to provide multiple simulation runs within a single pass on the computer. The primary programming languages supported were Fortran and COBOL. By this time, I was no longer preoccupied with saving microseconds, and did most of my programming in Fortran. For this program, I chose Fortran both for ease of maintenance and for portability, just in case Rand's next computer might not be assembly-language compatible with the IBM 704.

The users were primarily aerospace engineers. They were not programmers, but were largely willing and able to learn some simple programming constructs if it helped expedite getting their analysis jobs done. They wanted simple, problem-oriented input forms. They wanted the inputs to be concise, but not so concise as to be hard to read (e.g., not 030101). They wanted a good users' manual to explain the inputs, outputs, and control options. They would eventually like graphic outputs, but were satisfied with numerical printouts. They wanted an easily extensible system for their unpredictable new analyses. And they wanted to be able to reuse inputs as files or macros.

2.3 Domain Model and Interface Specifications

Accommodating all of the desired capabilities above appears complicated. Fortunately, the rocket trajectory simulation domain can build on a relatively simple and elegant central domain model:

1. The propulsion, aerodynamic, gravitational, and other forces on a rocket are basically a function of its position vector \( \mathbf{P} \) and velocity vector \( \mathbf{V} \); of the orientation of its body axis \( \mathbf{A} \); and of time. So is the rocket's fuel flow or rate of change of mass \( \frac{dm}{dt} \). The individual forces can be added together into an overall force vector \( \mathbf{F} \).
2. The rocket obeys Newton’s Second Law, $F=ma$, where $m$ is the rocket’s mass and $a$ is the rocket’s acceleration vector, or rate of change of velocity.

3. Given that the rocket’s position $P$, velocity $V$, and mass $m$ are known at time $t_0$, their values at time $t_0 + \Delta t$ can be determined by integrating their rates of change from $t_0$ to $t_0 + \Delta t$:

$$P(t_0 + \Delta t) \text{ by integrating } V(t)$$
$$V(t_0 + \Delta t) \text{ by integrating } a(t) = \frac{F(t)}{m(t)}$$
$$m(t_0 + \Delta t) \text{ by integrating } \frac{dm}{dt}.$$

There are elaborations on this domain model, such as the need to determine the rocket’s orientation vector $A(t)$, and the fact that multistage rockets have discontinuities in the nature of their propulsion and aerodynamic forces. But, overall, these elaborations can be added to the central domain model fairly cleanly.

2.4 Core Domain Architecture

Based on this domain model, the core domain architecture involved a trajectory initialization step, the central trajectory simulation loop, and a loop-termination step. The initialization step involved the assimilation of input parameters provided by the engineer. This was followed by a loop involving the three steps of the central domain model above: the calculation of rocket forces and attitude angles; the resolution of these forces along the rocket’s body axes; and the integration of rates of change to determine the rocket’s future position, velocity, attitude, and weight. Finally, the loop was terminated with the use of termination conditions supplied by the engineer. Figure 1 summarizes this core domain architecture.

Figure 1. Rocket Program Core Domain Architecture

2.5 Published Interface Specifications

The key to the practical success of the core domain architecture was the set of published interface specifications for the inputs to and outputs from the Flight Program of
propulsion, aerodynamic, guidance, and other routines specified or supplied by the engineer.

The set of inputs that any flight program subroutine could count on being available were defined in terms of their names, units, and coordinate systems. They included the current time, the rocket's position vector, the rocket's velocity vector, and the rocket's current weight (its mass times the gravitational constant).

The set of outputs that every flight program (combination of computational subroutines specified by the engineer) was expected to produce were similarly defined. They included the components of the rocket's thrust vector along the body axes, the corresponding components of the rocket's aerodynamic force vector, any other non-gravitational forces, such as solar radiation pressure, the angles relating the rocket's body axis to its velocity vector, and the rocket's rate of change of weight.

Note that the output interface specifications indirectly indicated force quantities that the engineer did not have to specify, such as the gravitational force. Such forces were calculated in the same way across the entire trajectory, and thus were taken care of by the main program (subject to some parameters that the engineer could specify for the entire run).

Also, the input interface specifications indirectly indicated input quantities that the engineer would have to calculate if he or she needed them for subsequent calculations. Examples were the local atmospheric pressure and density, often used for propulsion or aerodynamic calculations, but not automatically furnished because their calculation took considerable time and was unnecessary for major portions of rocket trajectories above the atmosphere.

The "published interface specifications" were eventually published in a book constituting the users' manual for the program, which was used at its peak by over 50 organizations besides Rand. The book, ROCKET: Rand's Omnibus Calculator of the Kinematics of Earth Trajectories [Boehm, 1964], is the source of several further tables and figures referenced in this paper.

2.6 Examples of Flight Program Subroutines

Three examples of propulsion flight program subroutine descriptions are provided in Table 1 to give a feel for their nature. The first one, CONTFL(TH,FL), assumes that the rocket's thrust and fuel flow are constants specified by the values TH and FL. The subroutines are invoked by a standard Fortran CALL statement. Thus, CALL CONTFL(300000., 1000.) specifies a rocket engine which will deliver a constant thrust of 300,000 lb, and a constant fuel flow of 1000 lb/sec. Each standard ROCKET subroutine is described by a common schema specifying the inputs needed which are not automatically furnished via the standard input interface, the outputs produced, and the action performed by the subroutine.

The second subroutine is TBTFTM(N), which assumes that thrust and fuel flow will be determined from a table as a function of the current time, using Nth order interpolation. The third subroutine, CONTAU(TA, TB), assumes that the rocket engine is offset from the body axis by two angles, $\tau_a$ and $\tau_b$. It takes the total THRUST determined by subroutines like CONTFL and TBTFTM and distributes it across the three body-axis components TAX, TBT, and TAL expected by the output interface specification.
<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTFL (TH,FL)</td>
<td>None</td>
<td>THRUST, total thrust T (lb); TAX, axial thrust $T_A$ (lb); WD, weight derivative $\dot{w}$ (lb/sec).</td>
</tr>
<tr>
<td>CONTAU (TA,TB)</td>
<td>THRUST, total thrust T (lb)</td>
<td>TAX, thrust along A-axis $T_A$ (lb); TBT, thrust along B-axis $T_B$ (lb); TAL, thrust along Al-axis $T_{Al}$ (lb).</td>
</tr>
<tr>
<td>TBTFTM (N)</td>
<td>None</td>
<td>THRUST, total thrust T (lb); TAX, axial thrust $T_A$ (lb); WD, weight derivative $\dot{w}$ (lb/sec).</td>
</tr>
</tbody>
</table>

Table 1. Propulsion Flight Program Subroutine Descriptions

3. Integrated Architecting: Control Structures, Data Structures, and User Interface

The overall architecting process involved elaborating the core domain architecture into a set of control structures, data structures, and user interface capabilities. These were determined by trying to accommodate as many as possible of the desired technical and operational capabilities above without overly compromising performance or ease of use and extension. The process thus involved a simultaneous determination of requirements and system architecture, including iteration of prototypes of the input forms and output formats with the prospective users.

3.1 Control Structures

Some of the main control structures involved the support of simulating multiple options or branches of a trajectory during a single run. For example, an engineer might want to experiment with range-vs.-payload-weight tradeoffs using more or less lofted trajectories.

The overall ROCKET program control structure accommodating the section and branching capabilities is shown in Figure 2. At the bottom of Figure 2, it shows that the central trajectory simulation loop from Figure 1 is extended to accommodate the printing of trajectory outputs. Proceeding upward in Figure 2, there are loops to cover multiple branches, multiple trajectory sections, and multiple runs per computer pass. The “Compile
Fig. 2-ROCKET Program Basic Flowchart
Flight Programs” and other initialization activities will be covered under the User Interface discussion below.

3.2 **Data Structures**

The ROCKET data structures relied heavily on the Fortran Global COMMON and EQUIVALENCE capabilities. These have been accurately analyzed as being highly risky and easy-to-misuse data structures [Wulf-Shaw, 1973]. Global COMMON enables any routine to modify a shared variable (e.g., the force of gravity) without anyone else’s knowledge. The EQUIVALENCE capability enables the same storage location to have multiple names. This also can create dangerous side effects if one of the names is used to modify a variable which is assumed to be stable under its other name.

However, these capabilities provided powerful support for the services ROCKET needed to provide, and ROCKET used them in ways that reduced the risks of misuse. One such strategy was to provide the data definitions as further published interface specifications [Boehm, 1964; p. 223]. Also, portions of the input and shared-variable areas were provided to accommodate extra user-defined capabilities. For example, 5 spaces were available for users to specify additional derivatives they wanted to integrate (e.g., heating rates, pressure buildups, etc.). These controlled discretionary areas further reduced the risks of misusing the shared variables.

3.3 **User Interfaces**

3.3.1 **Input Form**

The user interface for the ROCKET inputs included a standard form for specifying identifiers and trajectory parameters [Boehm, 1964; p. 74]. The form included entries for descriptive remarks; earth-model parameters (rotating or not; spherical vs. oblate); initial conditions (position, velocity, weight); and parameters for terminating a section (by time, weight, altitude, etc.) It also provided a set of spare input fields to enable users to easily extend the program’s input capabilities.

3.3.2 **Flight Programming Form**

The user interface for specifying the Flight Program used to “Compute Forces and Angles,” as discussed with Figure 1, was as a set of standardized Fortran subroutines such as those given in Table 1. The engineers placed references to these subroutines in standard CALL locations on another specialized form. These became part of the inputs, and were compiled by the Fortran compiler at the beginning of each run.

Having learned something about programming languages and compilers by this time, I was tempted to invent a special-purpose Flight Programming Language and compiler for it, rather than using Fortran capabilities. I decided against this, for the following main reasons:

- Using Fortran, it would be easy to interface existing Fortran programs (e.g., detailed engine models of heating calculations) to the ROCKET structure.
- The simple Fortran CALL’s on the Flight Programming Form covered most uses. Any extensions could be naturally specified via additional Fortran statements.
- Fortran was widely known by engineers; a special language would be one more thing to learn.
• Despite predictions of its demise as early as 1958, Fortran appeared to have a lot of staying power.

• The Fortran compiler and related utilities (e.g., diagnostics, debugging aids) provided a lot of capabilities that would be time-consuming to develop -- and maintain--by myself for a special-purpose language.

I never regretted this decision.

3.3.3 Output Formats

ROCKET user output began with a printout of the user’s descriptive remarks and inputs. It then proceeded to print lines containing standard quantities (time, position, velocity, altitude, weight, etc.) and user-specified optional quantities (guidance, tracking, orbital, etc.), at user specified time intervals.

The outputs were the most awkward part of the ROCKET user interface. The engineers were accustomed to reading them, and engineering assistants were generally available to turn the results into graphs, but a graphical output capability was much needed. Section 5 summarizes a followon user-interactive program called Graphic ROCKET, which provided such capabilities.

4. ROCKET Development and Usage

The overall ROCKET development took 6 months of roughly full-time effort. About 4 months involved architecting, which included prototypes of key capabilities. Coding and integrating took about a month, as did testing. Testing of complex numerical programs is quite difficult; one large trajectory program operated for two years before it was discovered that one minor gravitational parameter had been entered with a positive rather than a negative sign.

For ROCKET, there were several trajectory programs I could use for test oracles, such as the General Dynamics Atlas simulation, an Aerospace Corp. rocket simulator, and some special-purpose models I had done at Rand. Fortunately these initial tests picked up the serious defects in the program. The residual defects found during ROCKET usage were special cases which gave quite obviously erroneous outputs.

After about a year’s use of ROCKET at Rand, I prepared an export version for the IBM SHARE users’ group library. Before exporting it, I did the equivalent of a Post-Deployment Review. This picked up a number of shortfalls. For example, I had written a “Common Pitfalls” section of the users’ manual, including such advice as “Don’t leave the rocket’s initial weight input blank; it will make the weight equal to zero and cause an abort when the program tries to compute a = F/m.” On review, it made a lot more sense to just test for such inputs before starting the simulation.

Eventually, the program was used regularly at over 50 installations. It turned out to be quite portable, running on IBM (7000 and 360 series), Control Data, DEC, GE, Honeywell, and Univac machines. It was not a big burden to maintain, although there were occasional peaks of activity to add new capabilities, such as aerodynamic heating and non-stationary trackers.

4.1 Some Lessons Learned

A few lessons learned were contained in a paper on ROCKET [Boehm, 1965], which contained a section of “Remarks on Development of General Computer Programs.”
Here they are:

“Experience with the ROCKET program suggests the following conclusions for developers of similar large general-purpose computer programs:

1. Use a general programming language which is not tied to a particular machine;
2. Slight gains in efficiency, purchased at the cost of logical simplicity of the program, are a poor bet in the long run;
3. Develop the sections of the program in modular form;
4. Thorough documentation with numerous examples saves everybody’s time in the long run;
5. An extensive field-test period for both program and documentation eliminates a lot of embarrassing situations;
6. Anticipate the direction of extensions to the program and provide a clean, well-defined interface for tying them into the program.”

On reviewing these now, they seem like good remarks, but well short of what they could have been. It was only years later, when I read papers like David Parnas’ “Designing Software for Ease of Extension and Contraction,” [Parnas, 1979] that I could see and appreciate how much more could be done via more thorough definitions of “in modular form” and “clean, well-defined interface” to anticipated extensions.

5. Some Followons: Graphic ROCKET and POGO

5.1 Graphic ROCKET

In the mid-1960’s, Rand developed a set of powerful interactive graphics capabilities, and was looking for useful application areas for them. ROCKET’s lack of a graphic output capability made it a good candidate application.

John Rieber, Vivian Lamb and I developed Graphic ROCKET in 1966-1967. It ran on a dedicated IBM 360-40 that Rand was using for interactive graphics research for ARPA. Graphic input-output was done via an IBM 2250 display with both light-pen and freehand Rand Tablet inputs, plus a Stromberg-Carlson SC-4060 graphic output device.

Figure 3 shows the general nature of the user interaction. The program turned out to be quite popular with Rand engineers, but not very popular elsewhere. The main reason was economics. At Rand, the engineers were research subjects operating on a paid-for computer. Elsewhere, IBM 360-40 prime-shift time cost $50/hour, a difficult figure to justify when engineering assistants were paid around $5/hour.

For comparison, the Conclusions from an early paper on Graphic ROCKET [Boehm-Rieber, 1967] are provided below.

“1. The major benefit of interactive operation is the reduction of calendar time required to analyze a mission or the increased number of alternatives which can be investigated in a given time.
2. Even with its higher overhead rate, interactive operation can provide more efficient machine usage, since human judgment can reduce the number of runs required to establish a result.
3. Even a good batch-mode program needs considerable redesign to reorient it toward interactive processing.
4. User enthusiasm for interactive operation depends mainly on two factors: the degree of user-orientation of the language, and the degree of the user’s involvement with his problem.”
Figure 3 - User Viewing Graphic ROCKET Display
5. Man-machine interfacing for problems involving design creativity is still a little-understood subject. In designing such systems, one shouldn’t try to build a closely-optimized system around anticipated usage patterns. Instead, one should build a flexible system, wait and see how the analyst uses it, then modify it to serve him better.”

Conclusions 1 and 2 provide some rationale for benefits and savings due to interactive operation, but this rationale was not sufficiently persuasive for most organizations. Conclusions 3 and 5 are the strongest in retrospect. I was particularly surprised by the amount of breakage in converting ROCKET to Graphic ROCKET. Some capabilities, such as branching, just weren’t needed. Others needed total reorganization, such as batch vs. interactive input validation.

5.2 Programmer-Oriented Graphic Operation (POGO)

Another unpleasant surprise with Graphic ROCKET was the amount of effort required for maintenance. Just adding a single option or parameter to an input screen involved reorganizing the screen “real estate” and reworking a number of graphic system calls whose parameters involved the numerical raster coordinates of the boxes, character strings, or input fields.

After a few months of doing this, we decided to use the Rand Tablet’s dragging and dropping features to develop a screen-building and modifying capability for Graphic ROCKET. At the time, we were limited to using integers to indicate where to place parameters when a user entered them into an input slot, and where to transfer control when a box was clicked on. But with the ability to move the slots and boxes around, even this level of capability improved our maintenance productivity by more than a factor of 2.

It turned out that there were other Rand models and simulations that were looking for easy ways to develop interactive graphic interfaces. So we packaged the screen-builder and -linker capability with Graphic ROCKET’s curve-display capabilities to provide a support tool for such applications. The result was POGO, the Programmer-Oriented Graphic Operation [Boehm-Lamb-Mobley-Rieber, 1969]. It was used to provide graphic user interface (GUI) capabilities for several Rand models, particularly in the medical area.

5.2.1 Composing POGO Graphic User Interfaces (GUI’s)

POGO GUI’s were composed with a combination of the IBM 2250 light pen and function keypad, and the Rand Tablet stylus for freehand inputs. The GUI developer used function keys to navigate among POGO’s capabilities for creating and repositioning control boxes, parameter-input fields, text displays, and various geometric figures. The POGO paper’s explanations for some of the functions are given below.

2. Touch-Up - The console is placed in the character recognition mode. The user may modify any character on the screen by writing over it freehand with the tablet stylus.

3. Move - The user points to the character string or geometric entity he wants to move with the tablet stylus and drags it around the screen with the stylus until it is in the desired position. Lifting the stylus completes the action.

7. Values - The user points with the stylus to define a place to store the value of a variable; the position is denoted by underscores.

8. Fancy Boxes - These are similar to plain boxes, except that they have a dot at the center to serve as a target for the light pen.
9. **Insert Codes** - By each box and each "values" position in the current display, the user is presented with a line of underscores to furnish a numerical code that will identify this box or value to his FORTRAN control program.

12. **System Gronk** - If the system becomes unresponsive during a session, POGO will save as much as possible of programmer's work, and reinitialize the system.

15. **Recall Files** - These allow the user to recall previously created display for review or modification.

16. **Output Display** - POGO asks the user to provide a name for the current display. When this is done, POGO punches out a set of cards with the information necessary to re-create the display and identify its components to the FORTRAN control routines.

The Output Display function illustrates the primitive nature of the linkage between the POGO GUI composition mode and the GUI execution mode: a set of punched cards to be read by the GUI execution control programs. As with ROCKET and Graphic ROCKET, these linkages were defined by open published interfaces, to enable GUI designers to develop and integrate GUI capabilities not included in the POGO capability set.

### 5.2.2 An Example POGO GUI Screen

Figure 4 shows a representative POGO GUI screen for an interactive biomedical application. The top of the screen, down to "Initial Conditions," and the descriptions of the five initial-conditions parameters below, are created using the first four function keys: Small Characters, Large Characters, Touch-Up, and Move.

The five slots for entering initial condition values are created and positioned using the Values, Insert Codes, and Move buttons. The numbers 1, 2, 3, 4, and 14 in the slots indicate the positions in the POGO standard input parameter array D into which user-supplied values would be placed. Thus, during execution, if a user entered the number 5.0 in the top slot for "Initial time," POGO would convert it to floating point format and store it in location D(1). The programmer's Fortran execution-control program would then use this value (5.0 minutes) as the initial time for starting the fluid balance simulation program.

The remaining boxes define the user's control buttons. These were created and positioned using the Plain Boxes, Small or Large Characters, Insert Codes, and Move buttons. The number 500 was always keyed to a POGO program called SAVAL which checked the validity of the user's parameter inputs, converted them, and stored them in the D array. The numbers above 1000 were sequence numbers for the Fortran subroutines supplied by the programmer to perform the functions indicated in the boxes' titles (input curves; input control parameters; compute and display; quit). In the POGO execution mode, if a user selected one of the boxes, control would then be transferred to the appropriate programmer-supplied subroutine.

### 5.2.3 POGO Conclusions

The main conclusions from the POGO paper are given below.

"Total development time for POGO to date has been approximately one man-year. Machine usage on the 360/40 was about 100 hours for development. As mentioned above, the ability to bootstrap some curve input and output pages with the DESIGN program reduced the development time by approximately two months and made these pages much easier to modify."
FLUID BALANCE MODEL

With this model, you can

(1) Trace in curves describing the interaction of water, solute, and hormone levels in the human body

(2) Type or write in values of initial levels of these quantities, and trace in curves of external inputs

(3) View curves showing the resulting simulated evolution of the fluid balance of the body

INITIAL CONDITIONS

Initial time \( t_0 \) (min) *
Initial water level \( w_0 \) (moles) *
Initial solute level \( s_0 \) (moles) *
Initial hormone (ADH) level \( a_0 \) (moles) *
Number of significant digits *

Check inputs

Quit 
To input curves page

To control parameters page

To compute and display

Fig. 4 -- Initial Conditions Page for POGO Example
On the Graphic ROCKET application, the POGO DESIGN page is estimated to have cut control-page development times by factors of four to ten below those required for the manual layout-paper approach. Also, the work is far more palatable, and the error rate is virtually nil.

The most important consequence of the above factors is that they have allowed more response in providing users with additional capabilities not in the basic Graphic ROCKET package. On most interactive graphic systems this extension-threshold is quite high and constitutes a major usage bottleneck.

If any reason for this exists, it must be due to a tendency to design complete interactive graphics systems by deductive inference from an abstract model of typical user performance at a console; this produces “closed” systems that are quite responsive in the small (when the system is on the air), but quite unresponsive in the large (when trying to extend or modify the system). Experience with Graphic ROCKET and POGO users indicates that general characterizations of user activity are still quite risky, and that more overall responsiveness is gained by the prototype approach: deliberately designing an austere but extendable prototype and refining it by inductive inference from observed usage patterns.

The final conclusion on GUI prototyping is similar to Conclusion 5 of the Graphic ROCKET paper. This conclusion came back to the fore in the late 1970’s and early 1980’s, when I was trying to figure out why TRW’s waterfall model worked well for batch programs and ran into trouble on interactive programs. This eventually led to TRW’s experimentation with user interface prototyping and the Spiral Model [Boehm, 1988].

For similar economic reasons to those for Graphic ROCKET, POGO was little-used outside Rand. It was only about 10 years later that similar GUI-builders began to be economically feasible at TRW and elsewhere on machines like the DEC VAX, and about 10 years after that before the term “GUI builder” became widespread through emerging workstation and PC capabilities.

6. Broader Lessons Learned: Domain Engineering, DARPA, and Megaprogramming

A broader set of lessons I learned from the ROCKET, Graphic ROCKET, and POGO experiences involved the power of domain engineering and domain architecting in achieving savings via software reuse. The domains could be oriented around applications, such as rocket trajectory simulation, or around support elements, such as GUI’s.

There was a negative part of these lessons learned as well. My experiences with the IBM SHARE library involved successful sharing of Fortran mathematical subroutines as well as ROCKET, and I was convinced that many more software programs could be successfully shared. This led to my developing a catalog of potentially reusable software programs at Rand [Boehm, 1966].

Unfortunately, the programs in the Rand Computer Program Catalog did not get reused very much at all. In trying to analyze why, I discovered how many incompatible assumptions could be built into software programs. Some of the reasons for failure became clear to me somewhat later when I read Larry Constantine’s excellent analysis of module cohesion and coupling as critical success factors for software reuse [Constantine, 1967].
But beyond these general criteria, it appeared that there were further keys to
domain-specific reuse involved with shared assumptions and domain-specific interface
specifications. These were worked out for the ROCKET components, but were absent for
the programs in the Rand Catalog.

From time to time afterwards, I came upon people and organizations who had
developed domain architectures and successful associated software component libraries,
such as Toshiba in industrial process control [Matsumoto, 1984], Raytheon in business
data processing [Lanergan-Grasso, 1984], and a signal processing group at TRW in the
1980's led by Lyndon Hardy and Roger Vossler.

These success stories came to the fore again when I went to DARPA in 1989 and
was looking for themes for the DARPA software research and technology program. They
provided an experience base and example set of success stories for the DARPA
Megaprogramming initiative [Boehm-Scherlis, 1992], the Domain Specific Software
Architectures program [Mettala-Graham, 1992], and the domain-oriented reuse component
of the STARS program [Kramer-Foreman, 1992-96]. These in turn led to the successful
establishment of larger DoD initiatives such as the Army Software Reuse Initiative [Hess,
1992] and the DoD Software Reuse Initiative [Reifer, 1994].

Acknowledgements

This paper is sponsored by the Affiliates of the USC Center for Software Engineering.
The current Affiliates are Aerospace Corporation, Air Force Cost Analysis Agency, AT&T,
Bellcore, DISA, E-Systems, Electronic Data Systems Corporation, GDE Systems, Hughes
Aircraft Company, Institute for Defense Analysis, Interactive Development Environments,
Jet Propulsion Lab, Litton Data Systems, Lockheed Martin Corporation, Loral Federal
Corporation, Rockwell International, Science Applications International Corporation,
Software Engineering Institute (CMU), Software Productivity Consortium, Sun
Microsystems, Inc., Texas Instruments, TRW, U.S. Army Research Laboratory, and
Xerox Corporation.
References


