Abstract

In any given epoch, simulation tools evolve to deliver what the professions are perceived to want within the constraints of the computational resources and computer languages of the day. From a current perspective, legacy tools tend to run quite quickly on current computers but the evolution of computer languages and operating systems place such tools at risk of being frozen in time when they can no longer be modified and re-compiled.

One example is DOE-2.1E. It reflects the concerns of the professions in the mid 1980s. Its mix of solution techniques, the extent of the physics it embodies, and the types of entities that can be represented are also typical of that epoch. However, it relies on language extensions and data types which are no longer supported. The paper reviews the steps needed to syntactically assess, update and test such legacy code. It also explores: a) whether sufficient clues from legacy documents and user community can support explorations of legacy tools, b) steps required to evolve legacy code so that it can be built with modern compilers and deployed across a range of computers and operating systems and c) options for community adoption of legacy tools. These observations are worth reporting because today's tools will be impacted by epochal changes in computer languages, operating systems and the erosion of knowledge of their design and their data structures.

Introduction

The term archaeology is not often used in the context of simulation. And yet there is a significant body of research which is passing beyond the memory of the simulation community. There are also repositories of code which can no longer be compiled and executables which are susceptible to operating system and hardware changes which will render them inoperable. This paper is a wake-up call for the community to consider how we might preserve aspects of our legacy.

For long time observers, simulation conferences and journals associated with simulation reveal how knowledge is extended. But also how it is lost and then re-discovered or re-invented. This also occurs within the source code. Ground-breaking methods from the 1980s and 1990s persist in the quiet corners many of our tools. Sometimes this is because the methods have stood the test of time and are very efficient. Often these quiet corners persist because we have no choice - no one would fund that kind of research or development today.

They work. Until they don't. Perhaps it was a ‘Patch Tuesday’ that did it, or we updated our compiler, or we want to port a tool to a different OS. If the fault is with those more archaic corners or involves legacy software, our investigations take on aspects of Archaeology.

Evolving languages and platforms

Computing languages evolve. Features get deprecated but may linger in tool-chains. At some point software bumps up against the evolution of the language it uses. Compiler warnings become errors and there is no executable until we act.

There are, however, changes which define computing epochs. If your code or file formats are based on the assumption that an integer, a real number and four characters take up the same space in memory and your executables assume a 32-bit computing platforms, the increasing rarity of 32-bit computing platforms is a real and present danger. If a syntax check indicates code constructs that are no longer a part of the language this is a real and present danger. If your code requires lots of shifting of position within input files, changes in how compilers implement rewind commands are a real and present danger. When compilers get pedantic about passing information to subroutines and functions we may be looking at the global replacement or upgrade of thousands of lines of code.

This is not simply an issue for legacy code. The evolution of computer languages and tools chains will impact the code we are writing today. There will be an epoch and the assumptions, code design patterns and data structures that we fail to document in the near-term will need to be discovered via archaeological methods at some future date. Let’s start with a straightforward case.

TOPAZ-SNLL case study

A colleague at XX needed to solve a problem involving fluid transport and shock propagation in a piping system and remembered that there was a tool called TOPAZ-SNLL (Winters, 1985). This tool is hosted in the OECD archive and is described as:

“TOPAZ-SNLL, the Transient One-dimensional Pipe flow AnalyZer code for modelling the heat transfer, fluid mechanics, and thermodynamics of multi-species gas transfer in arbitrary arrangements of pipes, valves, vessels, and flow branches.” TOPAZ is ~10000 lines of code.
Fortran-77 designed for deployment on a Cray X-MP with typical flow problems taking from 30 seconds to several hours to compute. Its distribution included documentation as well as example input and report files for several types of problems. TOPAZ appeared to fit the needs of the colleague’s project. There was no graphic interface to port and no library dependencies to complicate the task and thus “a nice challenge”.

Except it included pre-Fortran-77 entities as well as inconsistent dimension statements which give modern compilers considerable indigestion. To check the robustness of code its syntax was reviewed via the static syntax checker forcheck (2007). This indicated 881 Fortran errors and 10 warnings. Most of the errors were ‘subscript out of range’ or mis-matches in the definitions of parameters being passed to subroutines. There were also implicit data type conversions, comparisons between real numbers which are traditional vectors for logic failing to do what you expect. There was also no Makefile. Although trivial to create in this case, such omissions can pose an initial barrier to porting projects.

Fortran used to be quite relaxed about how arrays were dimensioned and passed to subroutines. For example, array $X(100,50)$ in parent code could be passed to a subroutine which defined $X$ as $X(100,1)$. Modern compilers are increasingly pedantic. Mitigating inconsistent dimension statements is straightforward - identify all instances of the flagged arrays and set them to consistent sizes. Dealing with implicit data type conversions is also straightforward – do explicit casts as in Figure 1:

```fortran
IE=CONST(1)
XR(NR,2)=IE
```

```fortran
IE=NINT(CONST(1))
XR(NR,2)=REAL(IE)
```

*Figure 1: Explicit casts.*

There was a time when Fortran really didn’t do text. Fortran-4 had so-called Hollerith characters which were integer representations of strings but nothing so simple as an ASCII character set. The four characters PRIN are represented by the integer 131342 8048 and PLOT by 1414483024. TOPAZ used Hollerith characters, but in simple ways that were straightforward to re-implement (see Figure 2).

```fortran
C EDIT SYMBOLS FOR CONTINUITY-ENERGY ELEMENTS.
100 IF(IEDIT.EQ.1) SYMBOL(NEL)=' T(''
   IF(IEDIT.EQ.2) SYMBOL(NEL)=' P(''
   IF(IEDIT.EQ.3) SYMBOL(NEL)=' U(''
   IF(IEDIT.EQ.4) SYMBOL(NEL)=' R(''
```

*Figure 2: Conversion of Hollerith.*

Like many tools of its epoch, TOPAZ is a batch application – it takes no command line instructions and provides no feedback. It reads in a file describing the problem and writes out report file(s) and if it gets into trouble it writes error messages into a file. Sound familiar?

About eight hours were required convert TOPAZ so that it could be compiled with the GNU tool chain for deployment across a range of operating systems and computing platforms. There were 336 code interventions. After editing forcheck reported 6 remaining Fortran errors and 9 warnings. These errors were not fatal to versions 4-7 of the compiler but are fatal for version 8. We have kicked the can down the road a bit.

Porting legacy code requires the establishment of trust e.g. establishing that the operations carried out are equivalent to the original. It was fortuitous that the ported version of TOPAZ could be seen to generate identical predictions for the test cases. It helped that the structure of the code and most of the logic was well documented, the code was not outrageously different from other legacy code that the author has worked with.

With more resources the readability of TOPAZ could be improved by indentation of the code, switching to mixed case characters for documentation and explicit typing of variables.

The repository at the OECD site [https://www.oecd-nea.org](https://www.oecd-nea.org) lists scores of archives dealing with the fundamentals of physics. Many would be applicable for current engineering projects. A sample of one is, of course insufficient to gauge the resources needed to re-enable legacy tools, but suggests that it is possible. However, time is not on our side.

**Case study: DOE-2.1E**

The author has published timings of a range of simulation tools across a matrix of computer hardware, operating systems, work-flows and model complexities Hand (2016). A goal of including DOE-2.1E was a logical extension. Its authors explored many assessment tasks within a highly constrained computing envelope. Opening up this legacy tool to a newer generation of researchers and users might allow these insights to be re-discovered. Unlike TOPAZ, porting the DOE-2.1E suite has proved to be a considerable challenge and required a range of archaeological techniques.

A magnetic tape archive was sourced from Joe Huang at White Box Technologies, who was a member of the Simulation Research Group at LBNL for many years,
and quite familiar with DOE-2. We truly step back in time. Except we are not working on a 1990s computer or a development tool chain from that time. Indeed, according to Joe Huang (2018), the only compiler capable of a fresh build on Windows is Digital/Compaq Fortran, which has not been distributed since ~2000 although it is possible that some older Watcom compilers might also be applicable.

The first steps involved understanding the multiple stages needed to build the DOE-2.1E suite. The archive includes the source of the software needed to unpack and patch the code to get it ready for compilation (a good example of future proofing the process). This proved to be an iterative process to establish relationships between source files, ancillary files and old DOS scripts.

The porting process relied heavily on emails with colleagues who use DOE-2.1E, had compiled it or who knew of its development history. A comment from Joe Huang (2018) provided important clues:

“DOE-2 was originally designed by Zulfi Cumali and the input processor (DOEBDL) containing these non-standard usage of Fortran was written by Ender Erdem. Zulfi has a background in Operations Research and Ender in Computer Science… Using the current lingo, they were "stretching the envelope" at a time when Fortran was still quite limited compared to later, and used a lot of computer "tricks" to get around the limitations…”

The author’s background was more than adequate for porting TOPAZ. However, the mix of archaic language elements and “tricks” often proved to be ‘beyond the author’s pay grade’. Certainly there is a massive Engineering Reference (1982) detailing the nature of the underlying calculations. For the design of the code and its data structures not so much. The suite included:

- syntax, data structures and parameter passing conventions the author had never encountered
- use of deprecated data structures and syntax
- logic obfuscated by chains of GOTO statements
- syntax and data structures which appeared to be contrary to the language standard
- arrays which hold multiple types of data
- inconsistent common block and array definitions

Without access to the original development team one can only inspect the code and create software tests to understand its intent and confirm the evolution of the virtual model and virtual physics at different stages of the application.

Some of the utilities do simple tasks, in straightforward ways. Their syntax reports suggested few porting issues and porting them first might allow the explorer to gain insights into the design and style of the code. For the more complex applications in the range and magnitude of errors suggested extreme care was required to prove correct operation of the code.

The next stage is to iteratively address the errors which prevent compilation. Successful compilation is a waymark which allows us to single step through the application and allow run-time glitches to express themselves.

Here the archaeological tool of choice is a debugger. It allows us to understand the logic flow, but also allows us to see the evolution of at least some of the data structures. The design of the code and data structures made it difficult to debug:

- use of in-line calculations in subroutine calls
- use of in-line calculations to specify array indices
- data structures which can not be explored in a debugger
- code that only works if range checking is disabled

In general, there appeared to be a good correlation between the issues highlighted in the syntax report and run-time crash points. However, there were also cases where faults were from a cascade of glitches as well as artefacts of epochal changes to the language and how code directives are implemented.

**The DOE-2 weather suite**

There are two utilities `Fmtwth.f` & `Wthfmt.f` which convert between and ASCII and binary representations of DOE-2 weather files. There were few syntax issues flagged and it was straightforward to prove that the utilities worked (a round trip conversion resulting in identical files).

However, their batch operation was frustrating and error prone. Forcing users to manage files or rely on decades old scripts because of hard-coded names for input and output files is easy to rectify. Code that parsed command lines for ESP-r was adapted to form a new `parse.f`. The addition of a `-v` verbose option simplifies checking that the transform was successful and a `-help` option reminds users of the command line syntax (see Figure 3).

Porting involved the following interventions:

- introduce command line parsing of files and user directives
- document the actions taken in the code and indent the code for clarity
- create Makefiles to simplify compilation across a matrix of computers and operating systems.

```
linuxdoe/test$ ./wthfmt2 --help
Wthfmt2 converts packed binary DOE-2.1E weatherfiles to a formatted ASCII file.
Batch Use: copy *.bin to WEATHER.BIN then output is placed in WEATHER.FMT
Command line use: -I <the_bin_file> -out <the_fmt_file>
Optional: -v = verbose mode
```

```
linuxdoe/test$ ./wthfmt2 -in chicagotry.BIN -out chicagotry.FMT
Starting wthfmt2 with in file chicagotry.BIN and output file chicagotry.FMT.
```

Figure 3: Command line options.
The \texttt{wth.f} weather file utility converts different weather file sources into the DOE binary form. It is designed to pass through the source weather data files several times into to generate statistical reports. The initial syntax check showed 155 instances of non-standard syntax, 307 deleted features (Hollerith characters are ubiquitous within \texttt{wth.f}) and a couple of undefined variables.

Adapting the code for successful compilation did not, however, result in a useable DOE-2 weather file – every month was the same and the application continued to run until manually killed. Time to exercise some archaeology.

The fixation with memory meant only a few days data was held and the weather file sources were repeatedly re-scanned and the read position within the files adjusted via hundreds of backspace and rewind calls. After several debug sessions it became clear that the implementation of these low-level functions had changed and the elaborate bookkeeping logic no longer worked. Ouch.

The fix was straightforward - arrays were created to hold the hourly weather data over the year as well as a hash to support quick access to the relevant array position for any given month/day/hour (see Figure 4). The revised code scans once and then accesses memory to support the statistical and reporting operations.

```fortran
C Julian hour weather data (8760). Hash array: julian hour for month/day/hour.
integer hashhr,ISNWJ,IRAINJ,INWDDRJ,
 real WBTJ,DBTJ,PATMJ,CLDAMTJ,HUMRATJ,
 ENTHALJ,SOLRADJ,DIRSOLJ,WNDSPDJ
common/julianweather/hashhr(12,31,24),
 WBTJ(8760),DBTJ(8760),PATMJ(8760),
 CLDAMTJ(8760),HUMRATJ(8760),
 INWDDRJ(8760),IWNDDRJ(8760),
 DENSTYJ(8760),ENTHALJ(8760),
 SOLRADJ(8760),DIRSOLJ(8760),
 ICLDTYJ(8760),WNDSPDJ(8760)
```

Figure 4: Weather arrays.

A summary of changes to \texttt{wth.f}:
- introduce command line parsing of files and user directives
- convert hollerith character messages and comparison logic to Fortran-77 standard syntax
- create a hash table and hours-of-the-year array to hold scanned weather data to replace hundreds of lines of file positioning logic
- adapt dimension statements to ensure equivalence between parent code and subroutines
- add common blocks to store meta data collected during scanning of weather source files
- indentation of code blocks and added documentation for clarity.

There were \textasciitilde300 code interventions and at the end of the process there were zero deleted features. The invocation of \texttt{doewth} takes the form:

```
./doewth -v -dir chicagotry.inp -in chicagotry.tpe -out chig.bin -report chig.txt
```

Figure 5: Doewth command invocation.

**Doebdl**

Doebdl creates the underlying materials libraries used by doesim, parses the model bdl files and computes response factors. It make frequent use of ‘tricks’, the code is dense with GOTO and EQUIVALENCE statements as well as HOLLERITH constructs and comparisons. An initial syntax check indicated 757 fortran errors (including 376 ‘data type inconsistent with specification’ and 109 ‘array vs scaler’ conflicts) and 743 warnings.

Much of this relates to efforts to reduce memory use. Doebdl stuffs ALL the form, fabric, schedules and macros for the building (integers, reals and characters as well as most of the bookkeeping) into one unnamed common block – the IA/AA array (see Figure 6). It is up to the calling code to know whether the data at a specific point in the array is to be treated as an integer or real or a block of four characters so bookkeeping is critical! Other peoples bookkeeping is, by definition, opaque.

```fortran
common // // AA(3000000)
integer IA(3000000)
EQUIVALENCE(IA(1),IA(1))
```

Figure 6: The unnamed common.

Each cell in the IA array might hold a real or an integer or 4 characters – a pattern the author had never encountered and there were four different sizes of IA in doesim which seemed to be tempting fate. One of the first runs of doebdl faulted because it is trying to access an IA memory location that does not exist. A paranoid archaeologist has good reason to assume that the bookkeeping has failed or some of the non-fatal errors was at fault. Bdl.f’s use of in-line calculations makes proving the bookkeeping is correct particularly difficult from within a debugging environment:

```
DO 300 IYSCED = 1,NYSCED
IAPTR=IA(IRTPTR+2*IYSCED-1)
JSYM=IA(IRTPTR+2*IYSCED-2)
NWSCED=IA(IAPTR,1)
DO 120 I = 1,4
120 IA(IAX+I) = IA(I,JSYM)
IA(IAX+5) = NWSCED
IA(IAX+6) = 0
IA(IAX+7) = NDSCED+24+NYSCED*8+NWS*10+1
IA(IAX+8) = 0
NWS = NWS + NWSCED
```
\[ IAX = IAX + 8 \]
\[ \text{DO 290 IWSCED} = 1, NWSCED \]
\[ \text{JSYMW} = I(A + 3 * (IWSCED - 1) + 1) \]
\[ \text{DO 200 I} = 1, 2 \]
\[ \text{200 IA(INSPTR+I)} = I(A + 3 * (IWSCED - 1) + 1 + I) \]

---

**Figure 7: In-line calculations.**

DOE-2.1E is not alone in its use of in-line calculations. ESP-r has several hundred fewer than in the past but there is quite a bit more work to do.

Obfuscation is a risk in legacy code. Tasks like acting on errors e.g. call msg(2) creates a string and sets an index to be passed to a subsequent facility. What could be simpler? Within subroutine \textit{msg} are the following lines:

\[
\text{dimension MSGDSC(11,105)} \\
\text{DIMENSION MSG1(110),MSG2(110), MSG3(110),} \\
\text{MSG4(110),MSG5(110),MSG6(110),MSG7(110),} \\
\text{MSG8(110),MSG9(110),MSG10(110)}. . . \\
\text{EQUIVALENCE} \\
\{ \text{MSGDSC(1,1), MSG1(1)}, \} \\
\{ \text{MSGDSC(1,11), MSG2(1)}, \} \\
\{ \text{MSGDSC(1,21), MSG3(1)}, \} \\
\{ \text{MSGDSC(1,31), MSG4(1)} \} . . . \\
\text{C} \\
\text{MESSAGES 1 - 10} \\
\text{DATA MSG1/} \\
\text{11, 4HPARA,4HMETR,4HIC-I,4HNPUT,4H REQ,} \\
\text{4HUIRE,4HS PR,4HEV10,4HUS I,4HNPUT,} \\
\text{11, 4HNAME,4H NOT,4H ALL,4HOWED,4H WIT,} \\
\text{4HH TH,4HIS C, 4HOMMA,4HND ,4H ,} \\
\text{.} . . \\
\]

**Figure 8: Obscure data structures.**

Goodness this is dense and it goes on for another 98 messages. And there are similar subroutines for systems messages and MACRO messages. If a new error needs to be created where would one start? A single-step debug session exposed the underlying pattern and it was rewritten as a simple else-if structure:

\[
\text{character emsg*40} \\
\text{integer elevel} \ ! \text{error level} \\
\text{C Set text and level based on M passed to subroutine.} \\
\text{if(M.eq.1)then} \\
\text{write(emsg,'(a)') 'PARAMETRIC-INPUT} \\
\text{REQUIRES PREVIOUS INPUT' } \\
\text{elevel=11} \\
\text{elseif(M.eq.2)then} \\
\text{write(emsg,'(a)') 'NAME NOT ALLOWED} \\
\text{WITH THIS COMMAND' } \\
\text{elevel=11} \\
\text{elseif(M.eq.3)then} \\
\]

**Figure 9: Else if equivalent.**

Sometimes neither the compiler or the explorer can parse code. Consider:

\[
\text{SUBROUTINE DUMPIT ( IT, IA, L )} \\
\text{INTEGER it(5), IA(1)} \\
\text{.} . . \\
\text{DO 5000 J = J1 , J2} \\
\text{IW = IA(J)} \\
\text{c---------- ck if a4 format} \\
\text{IX = IW} \\
\text{DO 200 K = 1 , 4} \\
\text{IXX = IX .and. 255} \\
\text{c-- ctrl chars} \\
\text{if ( (ixx .lt. 32) } \\
\text{.} . . \\
\]

**Figure 10: Compiler does not understand.**

The IXX=IX .and. 255 appears to be attempting a bitwise operation (which Fortran is not known for) and the GNU compiler faulted at this point. We also have an array IA being accessed with indices beyond its range. Discussions with the community identified a similar subroutine used to decode MACRO statements in EnergyPlus input files. That was grafted into the legacy code and thus avoided some of the ‘tricks’ via an iachar:

\[
\text{SUBROUTINE DUMPIT ( IT, IA, L )} \\
\text{character it*20} \\
\text{INTEGER IA(1)} \\
\text{integer t} \ ! \text{as in upper4} \\
\text{character c(4) ! 4 characters in array} \\
\text{equivalence(t,c(1)) ! c(1)same as t} \\
\text{character*4 ch !string to cast into} \\
\text{.} . . \\
\text{DO 5000 J = J1 , J2} \\
\text{IW = IA(J)} \\
\text{c---------- ck if a4 format} \\
\text{IX = IW} \\
\text{t = IX } \ ! \text{as in subroutine upper4} \\
\text{DO 200 K = 1 , 4} \\
\text{C Get the ASCII value of IX.} \\
\text{C iachar takes a char use C array.} \\
\text{IXX = iachar(c(k))} \\
\text{if ((ixx.lt.32) .or. ! ctrl chars} \\
\text{.} . . \\
\]

**Figure 11: Less clever but works.**

Passing parameters to subroutines can also be a source of confusion. Consider the following parent code and subroutine which returns an array index IAPTRL where a value IAX is to be stored into the IA array:

\[
\text{common // AA(3000000)} \\
\text{integer IA(3000000)} \\
\text{EQUIVALENCE (AA(1),IA(1))} \\
\text{.} . . \\
\text{C INCREMENT THE LAYERS COMMAND} \\
\text{CALL INCRRF(IA,JOPLAY,IRF,IAPTRL)} \\
\text{C STORE DATA POINTER IN REF TABLE} \\
\text{IA(IAPTRL) = IAX} \\
\text{.} . . \\
\]
The IA common block is passed to INCRRF but internally is defined as IA(8,1) and used as IA(8,JSYM) where the value of JSYM might approach 1000. I have never seen a one dimensional array become a two dimensional array in any language. And there was a runtime crash at the IA(8,JSYM)=NREF line stating that there was no such location in memory. No one else in the community had a clue. Various code patterns were tested. In the end it was necessary to dump file to the entire contents of the IA array including its three possible data types at multiple places in bdl.f and do file different tests to establish what was actually changing as the model bdl file was parsed:

```fortran
SUBROUTINE INCRRF (IA, JOPCDT, JSYM, IAPTR)
  C: INCRRF INCREASES THE COMMAND REF COUNT
     COMMON/JOPCDT/JOPCDT(16,40), NOPCDT
  DIMENSION IA(8,1)
  IAPTR = 0
  IF (IOPCDT(12, JOPCDT).EQ.0) RETURN
  C: IF NO REF TABLE RETURN
  IF (IOPCDT(14, JOPCDT).EQ.0) RETURN
  C: GET NUM OF REF
     NREF = IOPCDT (13, JOPCDT)
  C: SET PTR TO DATA IN IA
     IAPTR = IOPCDT(14, JOPCDT) + 2*NREF - 1
  C: SET SYMBOL TBL PTR
     IA(IAPTR - 1, 1) = JSYM
  C: CLEAR IA PTR
     IA(IAPTR, 1) = 0
  C: IF A UNAME, SET REF NO. IN SYMBOL TBL
     IF (JSYM.NE.0) IA(8, JSYM) = NREF
RETURN
END
```

Figure 12: One and two dimension arrays.

With this brute-force archaeology the author established that IA(8,JSYM) was equivalent to:

```fortran
iofset = 8 * JSYM
IA(iofset) = NREF
```

At the time of this writing, there are still many blocks of clever code hindering the correct reading of MACRO entries in bdl files. Only files without macros can currently be used by the ported version of doeBdl. As DOE-2.1E was one of the first simulation suites to include the concept of MACRO definitions, getting this working would allow a new generation to explore such descriptive language syntaxes and perhaps extend them. With these new debugging tools it should be possible to un-pick other coding which has been flagged by the syntax checker. Interventions in doesim are also still work in progress. One of the classic fears at this point is that an application may not crash but yield inaccurate thermal performance predictions.

### Transfer files

One classic software design decision is in the use of files as transient stores. For example:

- as a store of transient data between software agents,
- as a repository of data accessed by one or more software agents i.e. materials properties,
- as ad-hoc memory substitutes.

Files which substitute for memory can be found in any number of legacy tools (if you look hard enough). Reverse engineering transient files to understand their internal structure or to identify faults can be quite a challenge. Documentation of legacy transient files are often terse or non-existent and the underlying method may be undocumented.

The 3D graphing facility in ESP-r’s results analysis tool gathers performance data and writes it to a transient file which is scanned by the graphing facility (see Figure 15). But it would not deal with timesteps shorter than 15 minutes and version 8 of the GNU compiler objected to working would allow a new generation to explore such descriptive language syntaxes and perhaps extend them. With these new debugging tools it should be possible to un-pick other coding which has been flagged by the syntax checker. Interventions in doesim are also still work in progress. One of the classic fears at this point is that an application may not crash but yield inaccurate thermal performance predictions.

The 3D graphing facility in ESP-r’s results analysis tool gathers performance data and writes it to a transient file which is scanned by the graphing facility (see Figure 15). But it would not deal with timesteps shorter than 15 minutes and version 8 of the GNU compiler objected to working would allow a new generation to explore such descriptive language syntaxes and perhaps extend them. With these new debugging tools it should be possible to un-pick other coding which has been flagged by the syntax checker. Interventions in doesim are also still work in progress. One of the classic fears at this point is that an application may not crash but yield inaccurate thermal performance predictions.

Below is a section of the 112th debug file generated as a simple bdl file was scanned (1st column is the IA array index, 2nd is integer, 3rd is real and last is if the slot held Hollerith characters:

```
112 dumpia after line 10580 COMMD
IAX IADIM IADIMV IAXMAX 151961 300000
1 1145982787 825.238464 COND
2 1330205769 3.37774413E+09 ITIO
3 54345102 1.66247768E-19 NED
4 132071402 1.35631564E-19 D
5 152014 1.40129846E-43 IONE
6 -1 NaN
7 -1 NaN
8 1 1.40129846E-43 NaN
9 1329811029 3.27669069E+09 UNCO
10 1414087758 3.45773756E+12 NDIT
11 1162760009 3300.95532 IONE
```

Figure 14: Portion of dump file.
required and the documentation updated in a number of places. The method still uses a transfer file and the compiler complaint could not be resolved until the logic is better understood so the offending code is commented out for those using the latest compiler.

if (display_mode) {
    - XDrawString(disp, gr_w.win, gr_w.gc,
        - gr_x, gr_font->descent + 1, txt, len);
    + d = XftDrawCreate (disp, gr_w.win,
        + visual, gr_colormap);
    + XftDrawString8(d,&xftcol, gr_font,
        + gr_x,gr_font>descent+1,
        + (const FcChar8 *)txt,len);
    + XFlush(gr_disp);
}

Figure 15: ESP-r 3D temperature graph.

Library dependencies

Updating library dependencies is a common porting task. If the library being ported from is ancient, there may be several generations of changes needed to be applied. It may be pragmatic to port to an intermediate version of the library.

An example is a recent project to update of the ESP-r text rendering facilities from original (ancient) X11 bitmap fonts to more modern font rendering. Current font rendering approaches involve dozens of library dependencies and as considerable overhead and tend to be over-engineered for the requirements of simulation tools. A less ancient approach with very few dependencies is libXft. Converting to this API would involve global replacement many types of calls, altering data structures and creating new data structures (for example libXft requires a separate colour data structure for fonts than are used for line drawing. Finding examples of such a transition is difficult on the usual stackexchange web sites as most developers would have done this kind of transition a decade ago.

Thankfully web searches focused on specific API calls can expose archives of source code control commits. In the example below a 2009 contribution to the Fedora OS project archive indicates the types of changes required (+ lines are old code and - lines are their replacement):

“The attached patch changes the Graphics module to use modern X fonts. The changes are relatively simple - changing calls such as XDrawString to use the Xf equivalent (XftDrawString8).”

Observations

Observations from the porting case studies:

• The build process for simulation tools involves rituals that are not well documented. Relationships within source distributions are rarely clear to outsiders.

• Start simple, allow time to see if there are code and data patterns with straightforward mitigation approaches.

• There are benefits in working with arrays of data rather than use hundreds of lines of file positioning.
and bookkeeping code. It almost always simplifies and clarifies the logic of legacy applications.

- Many tools use in-line calculations which introduce a level of opacity. Some are problematic, especially when passing data to a library function or a function written in a different language.
- All developers generate pockets of dense code which have not yet broken but would be a serious headache if we needed to revise or extend. What effort would be required for someone else to understand your subroutine X sufficiently to extend or adapt it?
- Sometimes we might find that someone else has already ported a block of code.
- Clever is costly. It took longer to work out a way to prove what was happening than it took to port the suite of weather utilities. Even a brief notation in the source code can save much investigative frustration.
- Library porting can take longer than anticipated and often involve changes to seemingly unrelated code.

Conclusion

Several case studies have identified patterns in the porting of legacy simulation tools. We have seen that the resource required can escalate depending on the level of obfuscation in the code and the degree to which the design of code and data structures is documented. We have seen that developers can be overly clever and it can require software archaeology to discover the original intent and devise a successful port. Porting the DOE suite is indicative of the nature and extent of risk of software becoming unbuildable. Step changes in compilers have rendered thousands of lines of code as points of failure. Community support can often substitute for access to an original development team but we have also seen that time is eroding the communities ability to understand and work with legacy code. Quite how porting tasks might be taken on by the simulation community and how the community might support extending the life of legacy tools and, perhaps mining them for nifty ideas or choosing to upgrade them to reflect aspects of current practice is a discussion that needs to happen.

As a community we might choose to deprecate legacy tools but to loose them by accident? Current simulation tools will be at risk in the next epoch. The nature of epochal change is speculative, however there are tools which can give a heads-up of future points of failure. And we have seen that even debugging interventions and debuggers can assist in understanding of a prior generations work. Their fixation with efficiency often lead to useful methods which might still have application.

Lastly, as developers we can lessen the demands on and frustrations of future explorers of our code. Many of the mitigations involve small changes. Others require iterations of re-design to future proof our code but can ensure that ‘time is on our side’.

<table>
<thead>
<tr>
<th>Source file</th>
<th>purpose</th>
<th>Number of diffs</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp-r.c</td>
<td>X11 Font naming</td>
<td>71</td>
</tr>
<tr>
<td>esp_draw.c</td>
<td>GTK drawing</td>
<td>51</td>
</tr>
<tr>
<td>esru.x</td>
<td>X11 drawing</td>
<td>530</td>
</tr>
<tr>
<td>esru_libNonGTK_F</td>
<td>Intermediate X11</td>
<td>79</td>
</tr>
<tr>
<td>Esru_libGTK.F</td>
<td>Intermediate GTK</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 2: Application interventions.

<table>
<thead>
<tr>
<th>Application</th>
<th>purpose</th>
<th>Number of diffs</th>
</tr>
</thead>
<tbody>
<tr>
<td>clim</td>
<td>weather tool</td>
<td>65</td>
</tr>
<tr>
<td>bps</td>
<td>simulator</td>
<td>&lt;24</td>
</tr>
<tr>
<td>prj</td>
<td>project manager</td>
<td>~1700</td>
</tr>
<tr>
<td>res</td>
<td>Results display</td>
<td>125</td>
</tr>
<tr>
<td>common</td>
<td>Common code</td>
<td>175</td>
</tr>
</tbody>
</table>

Acknowledgement

This project would not have been possible without feedback from folk which were familiar with DOE-2.1E as users or recent compilers of the tool or who knew its development team. Joe Huang, Linda Lawrie, Dru Crawley and Christopher Jones responded to scores of emails and provided many example files as well as clues as to the intent particularly obscure coding conventions and information structures.

References


Huang J. (2018) email conversation.


OECD site for TOPAZ <https://www.oecd-nea.org/>
