2014 Second International Workshop on Software Engineering for High Performance Computing in Computational Science & Engineering

Climate Models: Challenges for Fortran Development Tools

Mariano Méndez
III-LIDI, Facultad de Informática
Universidad Nacional de La Plata
50 y 120, 1900, La Plata, Argentina

Fernando G. Tinetti
III-LIDI, Facultad de Informática
Universidad Nacional de La Plata
50 y 120, 1900, La Plata, Argentina
and
Comisión de Investigaciones Científicas
de la Prov. de Bs. As.

Jeffrey L. Overbey
Department of Computer Science
and Software Engineering
Auburn University, AL, USA

***Abstract*—Climate simulation and weather forecasting codes are among the most complex examples of scientific software. Moreover, many of them are written in Fortran, making them some of the largest and most complex Fortran codes ever developed. For companies and researchers creating Fortran development tools—IDEs, static analyzers, refactoring tools, etc.—it is helpful to study these codes to understand the unique challenges they pose. In this paper, we analyze 16 well-known global climate models and collect several syntactic metrics, including lines of code, McCabe cyclomatic complexity, presence of preprocessor directives, and numbers of obsolescent Fortran language constructs. Based on these results, we provide some guidelines for people wishing to develop software development tools for Fortran. Notably, such tools must scale to million-line code bases, they must handle constructs that the ISO Fortran standard has deemed obsolescent, and they must work fluently in the presence of C preprocessor directives.**

I. Introduction

Fortran is one of the most widely used programming languages for developing scientific software [1]–[5]. Fortran programs consume a significant fraction of the hours on the world's largest supercomputers, and many of the largest, most complex scientific models—including earth system models key to understanding climate change—are written in Fortran.

While Fortran has evolved into a niche language for scientific computing, it was once used more generally; in fact, it is one of the oldest high-level languages. The FORTRAN project began at IBM in 1954, with the first language reference book published in 1957 [6]. It became an American National Standard in 1966 [7]; this language was known as FORTRAN 66. The standard was subsequently revised, creating FORTRAN 77 [8]. This was followed by Fortran 90 [9] and Fortran 95 [10]. Object-oriented features were introduced in Fortran 2003 [11]. Fortran 2008, was finalized in 2010 [12].

Over time, Fortran evolved and adapted to support several programming paradigms: structured programming, object-oriented programming, and parallel programming. While the standards committee deleted some features from the language standard, compatibility still remained: "Unlike Fortran 90, Fortran 95 was not a superset; it deleted a small number of so-called obsolescent features. However, the incompatibility is more theoretical than real as all existing Fortran 95 compilers include the deleted features as extensions" [13].

For practicing programmers, "Fortran" is not the language the ISO standard prescribes but rather the language their compiler accepts. This means that the Fortran language, in practice, consists of more than just the the features in the latest revision of the ISO standard: it is the union of that standard and all previous revisions, and any vendor-specific language extensions supported by their particular compiler.

Furthermore, many Fortran programs make extensive use of the C preprocessor (CPP), which provides the ability to customize Fortran code according to the needs of a particular user or a particular system. For example, CPPs `#ifdef` directive can be used to include a GPU-specific computational kernel on target systems that support it while omitting it on systems that do not. Thus, CPP can allow a level of portability and configurability that is essential for large codes.

Among Fortran programmers, climate simulation and weather forecasting codes are notorious for their size and complexity. One of the largest climate simulation codes, CESM, is well over 1 million lines of code, and even "small" climate simulations consists of tens of thousands of lines of code.

Maintaining large, complex codes requires software engineering tools, but advanced software development tools for Fortran programmers are sparse. Most Fortran programmers work with little more than a text editor, compiler, debugger, and perhaps some performance analysis tools. Meanwhile, software engineers working in languages like Java and C# have access to IDEs, static analysis tools, refactoring tools, testing tools, code understanding tools, and the like. Such tools can be made available to Fortran programmers, but to be useful on climate models (and similarly complex codes), tool developers must be aware of the challenges posed by such codes.

In this paper, we analyze 16 well-known global climate models and report several syntactic metrics, including lines of code, McCabe cyclomatic complexity, presence of preprocessor directives, and numbers of obsolete Fortran language constructs. Based on these results, we provide some guidelines for people wishing to develop software development tools that analyze Fortran source code. In particular, these tools must handle million-line codes, they must handle constructs the ISO standard deems obsolescent, and they must be able to operate correctly on codes making extensive use of the C preprocessor.

978-1-4799-7035-3/14 $31.00 © 2014 IEEE
DOI 10.1109/SE-HPCCSE.2014.7

Table I. CLIMATE MODELS ANALYZED

Modeling Group	Model(s)	Institution
BCC	BCC-CSM1.1	Beijing Climate Center, China Meteorological Administration
CSIRO-QCCCE	CSIRO-Mk3.6.0	Commonwealth Scientific and Industrial Research Org. and Queensland Climate Change Centre of Excellence
INM	INM-CM4	Institute for Numerical Mathematics
IPSL	IPSL	Institut Pierre-Simon Laplace
MOHC	HadGEM2, HadGEM3	Met Office Hadley Centre
MPI-M	MPI-ESM-LR	Max Planck Institute for Meteorology (MPI-M)
NASA GISS	ModelE, GISS	NASA Goddard Institute for Space Studies
NASA GMAO	GEOS-5	NASA Global Modeling and Assimilation Office
NCAR	CCSM3, CCSM4	National Center for Atmospheric Research
NOAA GFDL	GFDL-CM2.1	Geophysical Fluid Dynamics Laboratory
NSF-DOE-NCAR	CESM1	National Science Foundation, Department of Energy, National Center for Atmospheric Research
CMCC-CESM	CMCC-CESM	Centro Euro-Mediterraneo per I Cambiamenti Climatici
COLA and NCEP	CFSv2-2011	Center for Ocean-Land-Atmosphere Studies and National Centers for Environmental Prediction

II. MODELS, METRICS, AND METHODOLOGY

A. Models

In 2008, driven by the challenges of climate change, the World Climate Research Programme's Working Group on Coupled Modelling promoted a new set of coordinated climate model experiments. These experiments involved 20 climate modeling groups from around the world and comprise the fifth phase of the Coupled Model Intercomparison Project (CMIP5). According to Taylor et al. [14], "CMIP5 will ... provide a multimodel context for 1) assessing the mechanisms responsible for model differences in poorly understood feedbacks associated with the carbon cycle and with clouds; 2) examining climate 'predictability' and exploring the predictive capabilities of forecast systems on decadal time scales; and, more generally, 3) determining why similarly forced models produce a range of responses." All of the involved models are compute-intensive, since a climate simulation for a period of one hundred years can be very time-consuming even on supercomputers with thousands of processors [15].

We originally aimed to analyze all of the climate models involved in the CMIP5 experiments. In some cases, model source code was publicly available; in cases where it was not, we requested the source code from the corresponding modeling group. Ultimately, we were able to obtain source code for the 16 models listed in Table I.

B. Metrics

For each model, we collected the following measurements. (The justification for these measurements is included with more detailed explanations in Section III.)

- Code size and complexity measurements:
  - Source lines of code (LOC): total lines, logical executable LOC (LELOC), and non-blank non-comment LOC (NbNcLOC).
  - Number of subprograms.
  - Lines of code per subprogram.
  - Cyclomatic complexity [16] of each subprogram, as detailed in [17].
- Use of old vs. new language features:
  - Number of occurrences of obsolescent and deleted language features.
  - Frequency of old- vs. new-style DO loops.
  - Percentage of subprogram arguments with explicit intent specifications.
- Measurements related to the use of preprocessors:
  - Total number of preprocessor directives.
  - Number of C preprocessor #include directives.
  - Number of C preprocessor conditional compilation directives.
  - Number of Fortran INCLUDE lines.

C. Methodology

To collect the aforementioned measurements, we used the parser from Photran, an Eclipse plug-in for Fortran development [18]. For each file, the parser produced an abstract syntax tree, a data structure representing the syntactic structure of the source code. We traversed this tree to identify constructs of interest (subprograms, assigned GO TO statements, etc.), and output data to a CSV (comma-separated values) file. We then imported the CSV file into a SQL database for data analysis.

III. RESULTS

A. Overall Size

As a first step, we quantified the size of each model by counting lines of source code. Table II shows the results, where the columns are as follows:

- KLOC corresponds to total source lines of code, in thousands.
- KLELOC denotes logical executable lines of code, in thousands.
- KNbNcLOC indicates non-blank, non-comment lines, in thousands.

Table II. LINES OF SOURCE CODE

Model	KLOC	KLELOC	KNbNcLOC
GISS	40	15	20
CSIRO-Mk3.6.0	86	35	53
INM-CM4	91	47	74
GFDL-CM2.1	288	94	146
CCSM3	361	103	186
GEOS-5	367	145	212
IPSL	375	116	181
ModelE	380	166	279
CMCC-CESM	380	150	218
BCC-CSM1.1	451	152	236
MPI-ESM-LR	478	185	283
CFSv2-2011	478	209	297
HadGEM2	634	188	344
HadGEM3	737	241	439
CCSM4	822	262	416
CESM1	1371	482	803
total	7340	2589	4186

In total, we analyzed about 7.3 million lines of code, of which 4.2 million lines (57%) were non-blank/non-comment lines; of those, 2.6 million were logical executable lines.

We also counted the number of subprograms (functions and subroutines) in each model. Table III shows the results, ordered by the number of subprograms. The "In Mod." columns show how many functions/subroutines are defined in modules.

Table III. NUMBER OF SUBPROGRAMS

Model	Subpr.	Func.	In Mod.	Subr.	In Mod.
GISS	143	18	0	125	0
CSIRO-Mk3.6.0	299	3	0	296	0
INM-CM4	739	42	0	697	45
GFDL-CM2.1	2012	331	329	1681	1670
HadGEM2	2032	89	14	1943	319
HadGEM3	2566	222	184	2344	1219
CCSM3	2740	327	320	2414	1950
CMCC-CESM	2822	524	451	2298	1360
GEOS-5	3171	721	487	2450	1645
IPSL	3361	441	413	2920	2222
MPI-ESM-LR	3410	453	393	2957	2198
CFSv2-2011	3774	1113	818	2661	1248
BCC-CSM1.1	3784	802	781	2982	2090
ModelE	3944	619	474	3325	1697
CCSM4	6424	1150	1117	5274	4649
CESM1	9832	1852	1809	7980	7516
total	51053	8707	7590	42534	29828

By all of the above measures, GISS is the smallest code, and CESM1 is the largest. HadGEM2 and HadGEM3 are in both large, in terms of lines of code (Table II), but they contain relatively few subprograms (Table III); therefore, it is possible to conclude that they have larger subprograms than the other models, on average.

Some models have extensively taken advantage of modules, such as GFDL-CM2.1, CCSM4, and CESM1, while others have only a small fraction of subprograms in modules or do not use modules at all, such as GISS, CSIRO-Mk3.6.0, INM-CM4, and HadGEM2.

B. Subprogram Size and Complexity

Software development tools perform many static analyses intraprocedurally; that is, they analyze each subprogram separately. The complexity of such analyses is a function of the size of the subprogram. Thus, subprogram size is a concern, particularly for analyses with superlinear complexity.

Table IV shows the models ordered according to the size of the largest subroutine, along with their average subroutine size. The average subroutine size is in the range of 75–275 lines, depending on the model, although subroutines with over 1,000 lines of code exist in every model. The largest subroutine (in CESM1) is almost 12,000 lines of code, although it is actually quite simple (most of the lines are hard-coded constant array literals, used to populate arrays of coefficients).

Also of interest is the complexity of the control flow within each subprogram. Cyclomatic complexity (CC) measures the number of independent paths through a subprogram [16]. Although this analysis has its detractors [19], [20], some researchers claim that higher CC values indicate a higher probability of finding bugs in a procedure [21]. Some common thresholds for CC values are [22, p. 147]:

- 1–10: Simple subprogram without much risk

Table IV. LINES OF CODE PER SUBPROGRAM

Model	(Subroutine Name) Maximum LOC	Avg LOC
GISS	(SURFCE) 1333	156
CSIRO-Mk3.6.0	(TRACER) 1667	243
INM-CM4	(SEAICE_MODULE_SPLIT) 2018	128
CCSM3	(radcswmx) 2406	99
BCC-CSM1.1	(radcswmx) 2448	88
GEOS-5	(diaglist) 2602	81
ModelE	(init_ijts_diag1) 2975	83
CFSv2-2011	(INITPOST_GFS) 3267	108
CMCC-CESM	(DGESVD) 3409	123
MPI-ESM-LR	(datasub) 3504	123
IPSL	(physiq) 3848	75
HadGEM2	(GLUE_CONV) 4161	275
CCSM4	(hist_initFlds) 4623	99
GFDL-CM2.1	(morrison_gettelman_microp) 5124	108
HadGEM3	(conv_diag) 5602	261
CESM1	(lw_kgb03) 11975	115

- 11–20: More complex, moderate risk
- 21–50: Complex, high risk
- 51+: Untestable program (very high risk)

Table V shows the number of subprograms in each model with cyclomatic complexities in each of these ranges. By the above characterization, every model has large numbers of subprograms that would be classified as high risk and many that would be classified as untestable. Of course, it is debatable whether the above characterization is fair—do CC values above 50 really indicate "untestable" code?—but nevertheless, it is helpful to know that complex control flow is common and is present in every model.

Table V. NUMBER OF SUBPROGRAMS IN EACH CYCLOMATIC COMPLEXITY RANGE

Model	0–10	11–20	21–50	>51
GISS	62	26	34	21
CSIRO-Mk3.6.0	116	63	65	55
INM-CM4	459	135	93	52
GFDL-CM2.1	1442	249	212	109
HadGEM2	1003	361	383	285
HadGEM3	1318	421	453	274
CCSM3	2053	335	289	63
CMCC-CESM	1806	438	381	197
GEOS-5	2301	427	308	135
IPSL	2573	375	284	129
MPI-ESM-LR	2191	531	454	234
CFSv2-2011	2397	511	603	263
BCC-CSM1.1	2705	527	422	130
ModelE	3026	459	312	147
CCSM4	4682	803	701	238
CESM1	7312	1223	947	350

C. Decremental Language Features

Revisions of the Fortran language standard have, for the most part, retained backward compatibility with all previous revisions [23]. Since the Fortran 90 standard, some features of the language have been marked as obsolescent, and some have been deleted, but most compilers still accept them anyway.

An appendix of the Fortran standard [12] lists all of the features that are either obsolescent or deleted; it collectively calls these "decremental features." Assigned GO TO statements, the PAUSE statement, and Hollerith constants have been deleted (among other features). Obsolescent features—features that have not yet been deleted but will be at some point in the future—include arithmetic IF statements, computed GO TO statements, CHARACTER* declarations, and fixed source

form (where characters in columns 1–6 have special meaning, and statements are written in columns 7–72).

Table VI shows the number of occurrences of deleted and obsolescent features in each model's source code. Every climate model makes use of at least one of these features. Two deleted features—PAUSE statements and assigned GO TO statements—are found in the source code of 6 models.

Table VI. OCCURRENCES OF DECREMENTAL LANGUAGE FEATURES

Model	Arith IF	Asgn. GO TO	Comp. GO TO	ENTRY	PAUSE	Total
HadGEM3	0	0	0	0	1	1
CCSM3	0	0	1	0	0	1
GFDL-CM2.1	1	0	1	0	0	2
HadGEM2	0	0	0	0	2	2
IPSL	0	0	0	0	2	2
BCC-CSM1.1	0	0	3	0	0	3
CCSM4	1	0	10	8	1	20
CMCC-CESM	4	0	11	5	1	21
MPI-ESM-LR	3	4	9	4	1	21
CSIRO-Mk3.6.0	13	0	0	16	0	29
CESM1	16	6	9	8	1	40
INM-CM4	35	1	4	0	0	40
GEOS-5	37	0	22	13	0	72
GISS	33	0	6	36	0	75
CFSv2-2011	31	0	24	94	0	149
ModelE	23	0	4	139	0	166

D. Adoption of Newer Language Features

The Fortran 2008 standard [12] is large—over 600 pages—and the size of the language poses a challenge to tool developers. The size of the language is partly due to the retention of outdated language features. The set of language features deleted or deemed obsolescent is actually very conservative; many other features remain in the language, even though "better" alternatives are also present in the language. Therefore, for tool developers, it is helpful to know: Have climate models mostly moved to "new-style" Fortran, abandoning older language constructs in favor of their recently-added replacements? If so, it would be possible for tools to process only the newer subset of the language; this could make such tools cheaper to build and easier to test.

It is clear from Table VI that *some* outdated features appear in every model, but it is not clear if these exist in isolated modules or if they are representative of an older coding style throughout the system. It is difficult to give a single metric that quantifies whether a system uses an "old" or "new" Fortran coding style. However, looking at some representative language constructs can provide some insight. Tables VII and VIII attempt to do this.

Table VII lists the models sorted by the proportion of new- vs. old-style DO loops. New-style DO loops have a typical DO/END DO form, whereas in old-style DO loops,

```
      DO 999 I = 1, 10
          DO 999 J = 1, 10
              WRITE (*,*) I*10+J
999   CONTINUE
```

each DO statement contains the label of its terminating statement (999 in the example above). As in this example, two DO loops can share the same terminating statement (although this has been an obsolescent feature since Fortran 90 [9]).

Table VII. NUMBER OF DO STATEMENTS

Model	Count	% New	% Old	Shared
HadGEM3	30240	100	0	0
CCSM3	7613	99	1	2
HadGEM2	23251	98	2	60
GFDL-CM2.1	6367	98	2	6
CESM1	23316	97	3	14
CCSM4	16981	97	3	16
BCC-CSM1.1	11289	95	5	125
GEOS-5	7287	86	14	127
ModelE	13371	82	18	956
IPSL	11505	80	19	206
MPI-ESM-LR	11502	78	21	75
INM-CM4	5269	72	23	278
CFSv2-2011	14556	75	24	312
CMCC-CESM	9614	72	27	69
CSIRO-Mk3.6.0	4583	40	60	1375
GISS	1325	2	98	388

Table VIII. OCCURRENCES OF ARGUMENT INTENT SPECIFICATIONS

Model	Arguments	In	Out	InOut	Total	(%)
GISS	380	0	0	0	0	(0%)
INM-CM4	1748	0	0	0	0	(0%)
CSIRO-Mk3.6.0	2209	19	7	0	26	(1%)
IPSL	8988	1679	677	287	2643	(29%)
HadGEM2	38353	8368	1614	1953	11935	(31%)
CFSv2-2011	22661	8111	1676	860	10647	(47%)
CMCC-CESM	10068	3984	611	654	5249	(52%)
MPI-ESM-LR	18167	7131	1600	1186	9917	(55%)
HadGEM3	9829	4353	718	1045	6116	(62%)
ModelE	14508	6640	1263	1186	9089	(63%)
BCC-CSM1.1	19650	10747	2405	1361	14513	(74%)
GEOS-5	12712	6306	2617	1156	10079	(79%)
CCSM3	11708	6971	1825	845	9641	(82%)
CCSM4	28230	16499	4269	3038	23806	(84%)
CESM1	41744	24663	6653	5200	36516	(87%)
GFDL-CM2.1	11390	7756	1923	1339	11018	(97%)

Argument intents were added in Fortran 90; they specify whether subprogram arguments are input-only, output-only, or both (INTENT IN, OUT, and INOUT, respectively). Explicitly-labeled intents are useful as documentation, but they also allow additional compile-time checks to be performed. Table VIII shows the usage of intent specifications in each model.

As a proxy for determining the "age" of each model's coding style, Tables VII and VIII are not conclusive. GISS clearly has an old coding style, lacking argument intent specifications and using old-style DO loops almost exclusively. GFDL-CM2.1 is nearly the opposite: 97% of its arguments have explicit intent specifications, and only 2% of its DO loops use the old style. The other codes fall somewhere in between.

In sum, most of the models contain a non-negligible percentage of old-style DO loops, and none of the models have 100% explicit argument intents. For tool builders, this indicates that while new features are used, legacy Fortran features are still very much a part of how the language is used in practice.

E. Preprocessing Directives

The C preprocessor, also known as *cpp*, is the macro preprocessor for the C language [24], although it is widely used in Fortran codes as well. Preprocessing is carried out before the compilation stage by replacing some specially-identified regions of text in the program's source code. The C preprocessor poses a particular challenge for source code analysis tools because it is a *lexical* preprocessor, so programs containing preprocessor directives do not conform to the grammar of the underlying language (C or Fortran) until after

Table IX. PREPROCESSING DIRECTIVES PER MODEL

Model	Total	Conditional	Fortran Inc.	#include
GISS	0	0	0	0
INM-CM4	18	2	16	0
CSIRO-Mk3.6.0	37	6	31	0
GEOS-5	1795	1446	184	165
CFSv2-2011	1827	1243	65	519
GFDL-CM2.1	2056	1834	138	84
ModelE	3316	3202	75	39
CMCC-CESM	3616	3350	139	127
IPSL	5121	3346	1315	460
MPI-ESM-LR	5550	5223	145	132
CCSM3	5671	4166	207	1298
HadGEM3	6390	4575	927	888
BCC-CSM1.1	8375	6093	474	1808
CCSM4	10283	9466	465	352
HadGEM2	12893	5852	3501	3540
CESM1	13300	12563	430	307

preprocessing is performed. This makes programs containing *cpp* directives difficult to parse. Moreover, *cpp* provides conditional compilation directives, which allow portions of source code to be included or excluded at compile time based on the user's preferences. This makes analysis tasks (e.g., type checking) difficult. Parsing and analyzing code containing C preprocessor directives has been an active area of research in recent years (e.g., [25]–[29].

Table IX shows the number of preprocessing directives in each climate model, including the number of Fortran INCLUDE lines. (Fortran INCLUDE lines are not C preprocessor directives but are handled similarly to *cpp*'s #include directive.)

Obviously, the models with fewer lines of code also contain fewer preprocessing directives (e.g., GISS, INM-CM4, and CSIRO-Mk3.6.0). For four of the models—INM-CM4, CSIRO-Mk3.6.0, IPSL, and HadGEM2—at least a quarter of the total preprocessing directives were Fortran INCLUDE lines. Conditional directives (#ifdef, #ifndef, #if, #else, #elif, #endif) constitute the largest fraction of the total number of preprocessing directives in almost all models. Other preprocessor directives such as #error and #line are not found in any model. Moreover, we did find unorthodox uses of the C preprocessor, such as the following.

```
subroutine1 ( parameter1,parameter2
 #include additionalParameters
 )
```

Perhaps the most important conclusion for tool developers is that C preprocessor directives are present in every model, and most models use them extensively. Therefore, research on *cpp*-aware program analyses will have applications beyond C and C++: it will be applicable to Fortran programs as well.

IV. DISCUSSION

Based on the data we have presented, we can make several recommendations for developers building software engineering tools for Fortran.

The code complexity of climate models makes them suitable targets for software maintenance tools, such as code understanding tools, static analyzers, and refactoring tools. Most of the climate models were 300–500 KLOC, while the largest was 1.3 MLOC. Thirteen of the 16 models contained over 2,000 subprograms, and all of the models contained multiple subprograms with a cyclomatic complexity over 50. At this scale, no single developer can reasonably be expected to fully understand every detail of the model's source code. Sweeping, system-wide source code modifications are often daunting and may be prohibitively expensive. Although the *adoption* of software maintenance tools depends on the culture and the individuals involved, the scale and complexity of climate codes has certainly reached a point where such tools could be valuable.

Unfortunately, making such tools work effectively on these climate models can be an extremely challenging task.

Fortran development tools must scale to $10^5 - 10^6$ *lines of code.* The developers of Photran [18] (including the third author of the present paper) originally designed its indexer to handle codes up to about 50,000 lines. Based on their prior experience working on astrophysics codes, this seemed like a reasonable upper bound on the size of the codes most Fortran developers would be working with. Unfortunately, for developers working on climate models, this is a severe underestimate. Of the models we studied, the *smallest* was close to that size, and most were several times larger.

Fortran development tools must handle both new and old language constructs. Deprecation and deletion from the ISO standard have not made outdated language features disappear in practice. Most of the climate models we studied contained both new- and old-style Fortran code, in nonnegligible amounts. Several codes contained constructs (like arithmetic IF statements and computed GO TOs) that predate structured programming.

Tools that analyze Fortran source code must be able to handle embedded C preprocessor directives. C preprocessor directives were present in every model but one. Every model that made use of the C preprocessor contained conditional compilation directives; most contained 1,000–6,000 such directives. File inclusion was also used ubiquitously.

Together, the last two points reiterate the fact that the Fortran language, in practice, is not the language specified in the latest ISO standard. Rather, it is a mixture of that and all previous revisions of the Fortran language, together with macros, conditional compilation directives, and file inclusions afforded by the C preprocessor.

Although the data collected in this paper is intended primarily to benefit tool developers, it also seems to validate broader results obtained by previous researchers on the development of scientific software. Some language features were deprecated in the Fortran standard more than 20 years ago [9], but they are still being used. This may reinforce the idea that programming skills and knowledge are passed on from scientist to scientist [1], [30]. Also, several researchers have noted the existence of a gap between the scientific software community and the software engineering community [1], [31]. It is worth noting that, in the models we studied, the number of lines of code per subroutine would be quite large by most software engineers' standards (see Table IV). This is due partly due to the verbosity of the Fortran language, but subprograms were quite complex by other metrics as well (Table V). This suggests that computing scientists might benefit from some modularization techniques [1], [3], [32].

V. Related Work

In 1971, Donald Knuth analyzed a set of Fortran programs "in an attempt to discover quantitatively what programmers really do" [33]. He studied a variety of programs written by different people: a set of 400 programs distributed among 250,000 punched cards. Knuth provided a thorough study of the FORTRAN 66 constructs used on those 400 programs. Remarkably, he performed this analysis by hand.

Later research attempted to analyze a set of 255 Fortran programs by using a tool called FORTRANAL, which collects data for 31 metrics [34]. The aim of this work was to find some correlation among different groups of metrics.

The study carried out by Shen *et al.* [35] focused on finding Fortran programs' characteristics that were relevant in the parallelization process, from a compiler writer's view point, in order to contribute to research on data dependence analysis and program transformation.

A series of articles studying the relationship between Fortran programs and complexity metrics were published by Victor R. Basili. His 1981 article [36] used complexity metrics to evaluate software quality, to estimate software cost, and to evaluate the stability and quality of a software product during the development process. A follow-up article [17] analyzed several metrics' (lack of) correlation with effort and development errors, as well as with each other. Software was analysed using an automated tool called SAP.

Software metrics have also been used to evaluate system maintainability. One such study is due to Coleman *et al.* [37]; that work indicates that automated analysis can be used to "evaluate and compare software."

Scientific software has been the subject of several studies [1]–[3], [30]–[32], [38]–[41]. These articles all shed light on the fact that there is a mismatch between well-known, broadly accepted software engineering practices and the methods and practices used in the construction of scientific software.

Two of these articles focus specifically on the engineering of climate models. Easterbrook and Johns [39] studied climate scientists at the Met Office Hadley Centre, comparing and contrasting their practices with those of agile software development teams and open source software teams. Pipitone and Easterbrook [41] studied three climate models, finding that their defect densities were low when compared to open source software projects of comparable size.

In studies of scientific software development, a major recurring theme is that scientific software exists to help scientists achieve *scientific* goals. The software is not the product; science is. For those wishing to develop software engineering tools for scientific programmers, it is important to view such tools from this perspective. Many software engineers are willing to adopt new tools, libraries, and languages simply because they are new or interesting. Scientists are far less likely to do so, unless they have a specific need that it meets. Their goal is to produce science, not software. As Easterbrook and Johns [39] observed, "scientists ... are skeptical of most claims for software engineering tools. However, where such tools meet their needs ... they are readily adopted."

VI. Conclusions

In this article, we studied 16 global climate models in order to analyze the way the Fortran language is used in large-scale scientific applications. Our results have implications for those wishing to develop software development tools for Fortran. Notably, a market for Fortran software maintenance tools may exist, due to the size and complexity of these models. However, making such tools work effectively across a model's entire codebase is likely to prove challenging: Such tools must be able to process $10^5 - 10^6$ lines of code, including outdated features that have been removed from the ISO standard, and they must be able to handle source files with embedded C preprocessor directives.

References

[1] V. R. Basili, D. Cruzes, J. C. Carver, L. M. Hochstein, J. K. Hollingsworth, M. V. Zelkowitz, and F. Shull, "Understanding the high-performance-computing community," 2008.

[2] R. Kendall, J. C. Carver, D. Fisher, D. Henderson, A. Mark, D. Post, C. E. Rhoades, and S. Squires, "Development of a weather forecasting code: A case study," *Software, IEEE*, vol. 25, no. 4, pp. 59–65, 2008.

[3] J. C. Carver, L. Hochstein, R. P. Kendall, T. Nakamura, M. V. Zelkowitz, V. R. Basili, and D. E. Post, "Observations about software development for high end computing," *CTWatch Quarterly*, vol. 2, no. 4A, pp. 33–37, 2006.

[4] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 550–559.

[5] M. Schmidberger and B. Brugge, "Need of software engineering methods for high performance computing applications," in *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*. IEEE, 2012, pp. 40–46.

[6] J. Backus, "The History of Fortran I, II, and III," *ACM SIGPLAN Notices*, vol. 13, no. 8, pp. 165–180, 1978.

[7] A. FORTRAN, "X3. 9-1966," *American National Standards Institute Incorporated, New York*, 1966.

[8] American National Standards Institute, "X3. 9-1978," *American National Standards Institute, New York*, 1978.

[9] ——, *American National Standard for programming language, FORTRAN — extended: ANSI X3.198-1992: ISO/IEC 1539: 1991 (E)*. American National Standards Institute, Sep. 1992.

[10] ISO, "ANSI/ISO/IEC 1539-1:1997: Information technology — programming languages — Fortran — part 1: Base language."

[11] ——, "ANSI/ISO/IEC 1539-1:2004(E): Information technology — Programming languages — Fortran Part 1: Base Language," pp. xiv + 569, May 2004.

[12] ——, "ANSI/ISO/IEC 1539-1:2010: Information technology — Programming languages — Fortran Part 1: Base Language," p. 603, 2010.

[13] C. Malcom, "Fortran: A Few Historical Details," http://www.nag.co.uk/nag ware/NP/doc/fhistory.asp, Oct. 2004.

[14] K. Taylor, R. Stouffer, and G. Meehl, "An overview of cmip5 and the experiment design," *Bulletin of the American Meteorological Society*, vol. 93, no. 4, p. 485, 2012.

[15] W. M. Washington, L. Buja, and A. Craig, "The computational future for climate and earth system models: on the path to petaflop and beyond," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 367, no. 1890, pp. 833–846, 2009.

[16] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[17] V. R. Basili, R. W. Selby Jr, and T. Phillips, "Metric analysis and data validation across Fortran projects," *Software Engineering, IEEE Transactions on*, no. 6, pp. 652–663, 1983.

[18] "Photran, an Integrated Development Environment and Refactoring Tool for Fortran," http://www.eclipse.org/photran/.

[19] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, 1988.

[20] M. Shepperd and D. C. Ince, "A critique of three metrics," *Journal of Systems and Software*, vol. 26, no. 3, pp. 197–210, 1994.

[21] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 1963–1969.

[22] J. Foreman, D. A. Fisher, M. Bray, K. Brune, and M. Gerken, "C4 software technology reference guide-a prototype," 1997.

[23] M. Metcalf, "The seven ages of Fortran," *Journal of Computer Science and Technology*, vol. 11, no. 1, pp. 1–8, 2011.

[24] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C programming language*. prentice-Hall Englewood Cliffs, 1988, vol. 2.

[25] A. Garrido, "Program refactoring in the presence of preprocessor directives," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.

[26] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 805–824.

[27] C. Kästner, K. Ostermann, and S. Erdweg, "A variability-aware module system," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY: ACM, 2012, pp. 773–792. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384673

[28] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 81–91.

[29] P. Gazzillo and R. Grimm, "SuperC: Parsing all of C by taming the preprocessor," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY: ACM, 2012, pp. 323–334. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254103

[30] G. V. Wilson, "Where's the real bottleneck in scientific computing?" *American Scientist*, vol. 94, no. 1, p. 5, 2006.

[31] D. F. Kelly, "A software chasm: Software engineering and scientific computing," *Software, IEEE*, vol. 24, no. 6, pp. 120–119, 2007.

[32] J. Segal, "Scientists and software engineers: A tale of two cultures," *Proceedings of the Psychology of Programming Interest Group, PPIG 08*, pp. 10–12, September 2008.

[33] D. Knuth, "An empirical study of fortran programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.

[34] H. F. Li and W. K. Cheung, "An empirical study of software metrics," *Software Engineering, IEEE Transactions on*, no. 6, pp. 697–708, 1987.

[35] Z. Shen, Z. Li, and P.-C. Yew, "An empirical study of Fortran programs for parallelizing compilers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 3, pp. 356–364, 1990.

[36] V. R. Basili and T.-Y. Phillips, "Evaluating and comparing software metrics in the software engineering laboratory," *ACM SIGMETRICS Performance Evaluation Review*, vol. 10, no. 1, pp. 95–106, 1981.

[37] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[38] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. IEEE Computer Society, 2009, pp. 1–8.

[39] S. M. Easterbrook and T. C. Johns, "Engineering the software for understanding climate change," *Computing in Science & Engineering*, vol. 11, no. 6, pp. 65–74, 2009.

[40] J. Segal, "Some challenges facing software engineers developing software for scientists," in *2nd International Software Engineering for Computational Scientists and Engineers Workshop (SECSE '09), ICSE 2009 Workshop, Vancouver, Canada*, May 2009, pp. 9–14.

[41] J. Pipitone and S. Easterbrook, "Assessing climate model software quality: a defect density analysis of three models," *Geoscientific Model Development Discussions*, vol. 5, no. 1, pp. 347–382, 2012.