How I hacked my way into academia

Diomidis Spinellis
Professor, Department of Management Science and Technology
http://www.dmst.aueb.gr/dds
@CoolSWEng

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

HACKER [originally, someone who makes furniture with an axe] n.

1. A person who enjoys learning the details of programming systems and how to stretch their capabilities, as opposed to most users who prefer to learn only the minimum necessary.
2. One who programs enthusiastically, or who enjoys programming rather than just theorizing about programming.
3. A person capable of appreciating hack value (q.v.).
4. A person who is good at programming quickly. Not everything a hacker produces is a hack.
5. An expert at a particular program, or one who frequently does work using it or on it; example: "A SAIL hacker". (Definitions 1 to 5 are correlated, and people who fit them congregate.)
6. A malicious or inquisitive meddler who tries to discover information by poking around. Hence "password hacker", "network hacker".

— Guy Steele et al *The Hacker's Dictionary*, 1988

Challenges are the rule


ΗΛΕΚΤΡΟΝΙΚΟΝ ΕΡΓΑΣΤΗΡΙΟΝ
ΠΟΛΥΚΥΚΛΩΜΑΤΙΣΤΗΣ
1002
ΜΠΟΡΕΙΣ
χωρίς νά είσαι ειδικός
μέ ένα μόνον κίτ
νά σχημάτισης
102
ηλεκτρονικά κυκλώματα
διασκεδάζοντας
ΜΑΘΕ
άπλά καί εύκολα
πώς λειτουργοϋν
καί κατασκεύασε
μέ τόν πολυκυκλωματιστή:
• Ραδιόφωνα
• Πομπούς
• Ενισχυτάς
• Ταλαντωτάς
• Βομβητάς
• Ραδιοτηλέγραφο
• Σηματοδότας
• Δονητάς
Καί άλλες
πρακτικές έφαρμογές
άπό 9 μέχρι 99 ετών
Ο ΠΟΛΥΚΥΚΛΩΜΑΤΙΣΤΗΣ
ΠΡΟΣΦΕΡΕΙ
ΕΥΚΟΛΟΝ ΤΡΟΠΟΝ ΑΝΑΚΑΤΑΣΚΕΥΗΣ
ΠΟΛΛΩΝ ΕΦΑΡΜΟΓΩΝ
ΧΩΡΙΣ ΝΑ ΧΡΕΙΑΖΩΝΤΑΙ
ΚΟΛΛΗΤΗΡΙΑ Η ΕΡΓΑΛΕΙΑ
ΠΑΡΕΧΕΙ ΑΣΦΑΛΕΙΑ ΧΕΙΡΙΣΜΟΥ
ΛΕΙΤΟΥΡΓΕΙ ΜΕ ΜΠΑΤΑΡΙΑ
ΚΑΙ ΕΙΝΑΙ ΑΚΙΝΔΥΝΟΝ
EDUCATIONAL · ELECTRONIC LABORATORY


ΜΟΡΦΩΤΙΚΟΝ
ΔΙΑΣΚΕΔΑΣΤΙΚΟΝ
ΗΛΕΚΤΡΟΝΙΚΟΝ ΕΡΓΑΣΤΗΡΙΟΝ
ΔΗΜΙΟΥΡΓΙΚΟΝ
ΕΠΙΣΤΗΜΟΝΙΚΟΝ
Build Over 102 Projects
ΕΦΑΡΜΟΓΑΙ
ΤΡΑΝΖΙΣΤΟΡ
TRANSISTOR
POLYKYKLOMATIC MODEL 1002
ΜΠΑΤΑΡΙΑ
BATTERY
Speaker M-300
EDUCATIONAL
CREATIVE
ΠΟΛΥΚΥΚΛΩΜΑΤΙΣΤΗΣ No 1002
ENTERTAINING
SCIENTIFIC
68Ω
1K
4.7K
4.7K
100
680K

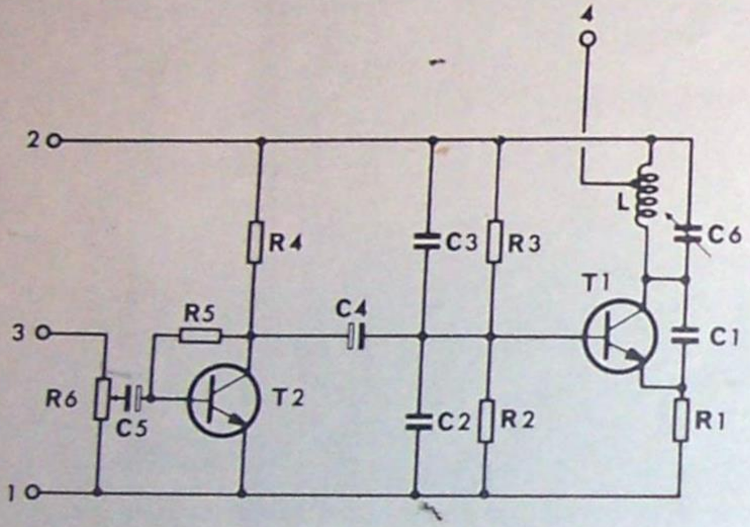


Fig.3.

Jasti Electronic

HF 65, anvendelse:

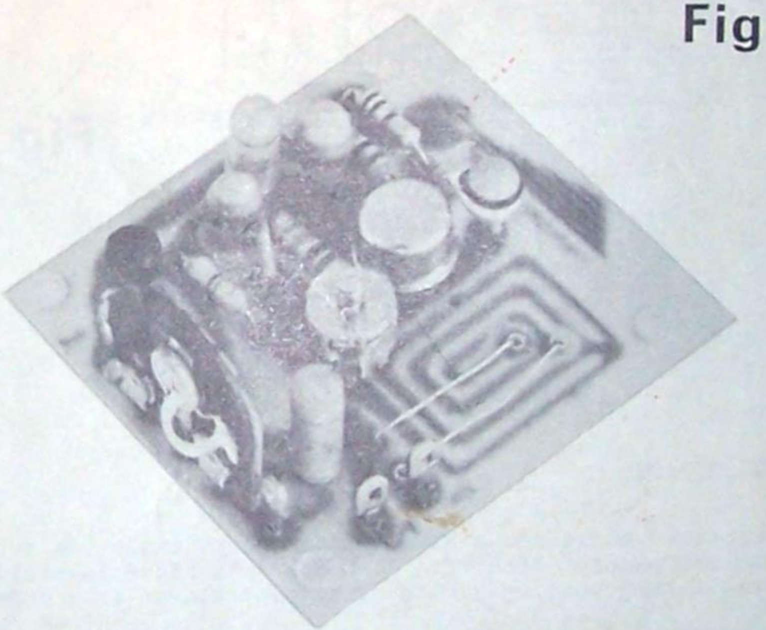
HF 65 kan anvendes af radioamatører til mikrofonsender på 144 M Hz. Det er nødvendigt at have senderlicens for at anvende senderen rigtigt.
HF 65 kan anvendes af sømænd, samt sejlsportsfolk i rum sø uden licens.
HF 65 er forsynet med en følsom forforstærker og en kraftig udgang. Istedet for en almindelig mikrofon kan man anvende en øreproptelefon, der kun koster få kroner i detailhandlen.
HF 65 kan også anvendes af radiomekanikere som målesender. Den skal så bygges ind i en tæt metalkasse, som hindrer uønsket senderudstråling. Istedet forat tilslutte en mikrofon til HF 65, kan man tilslutte en tonegenerator, og vil kunne få lodrette eller vandrette streger på TV. Hvis senderen ikke bygges ind i en metalkasse, vil den uden antenne række flere hundrede meter, og derved forstyrre FM eller TV båndene, hvilket ikke er tilladt.
Med en stor batterispænding vil senderen kunne række mere end lo Km. Udgangstransistoren kan tåle en tabseffekt på max. 5 Watt.

Jasti Electronic

DANMARK | HF 65

FM sender

Fig.1.

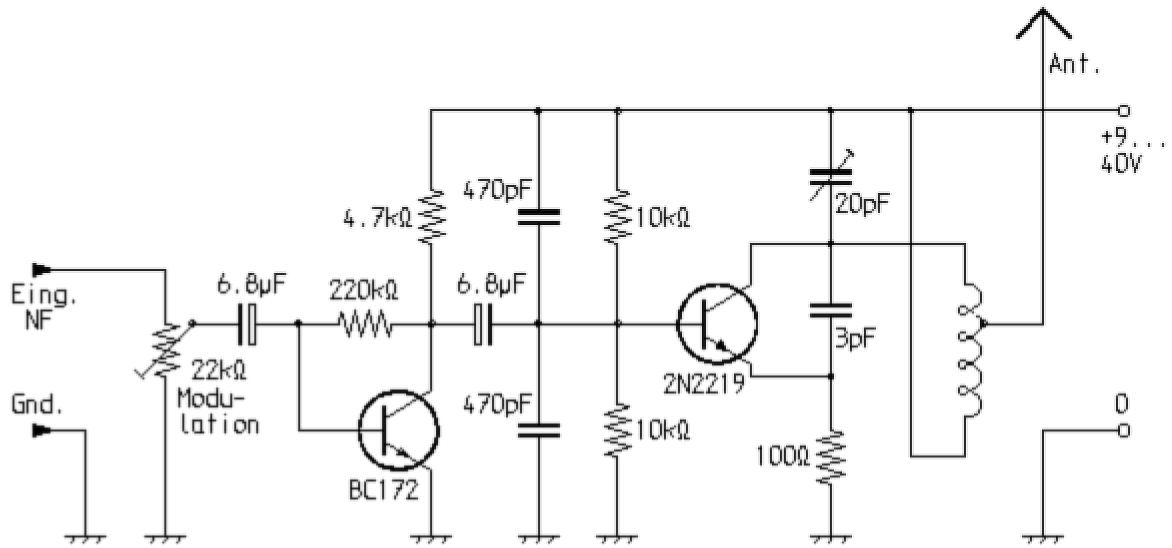


poweroscillator

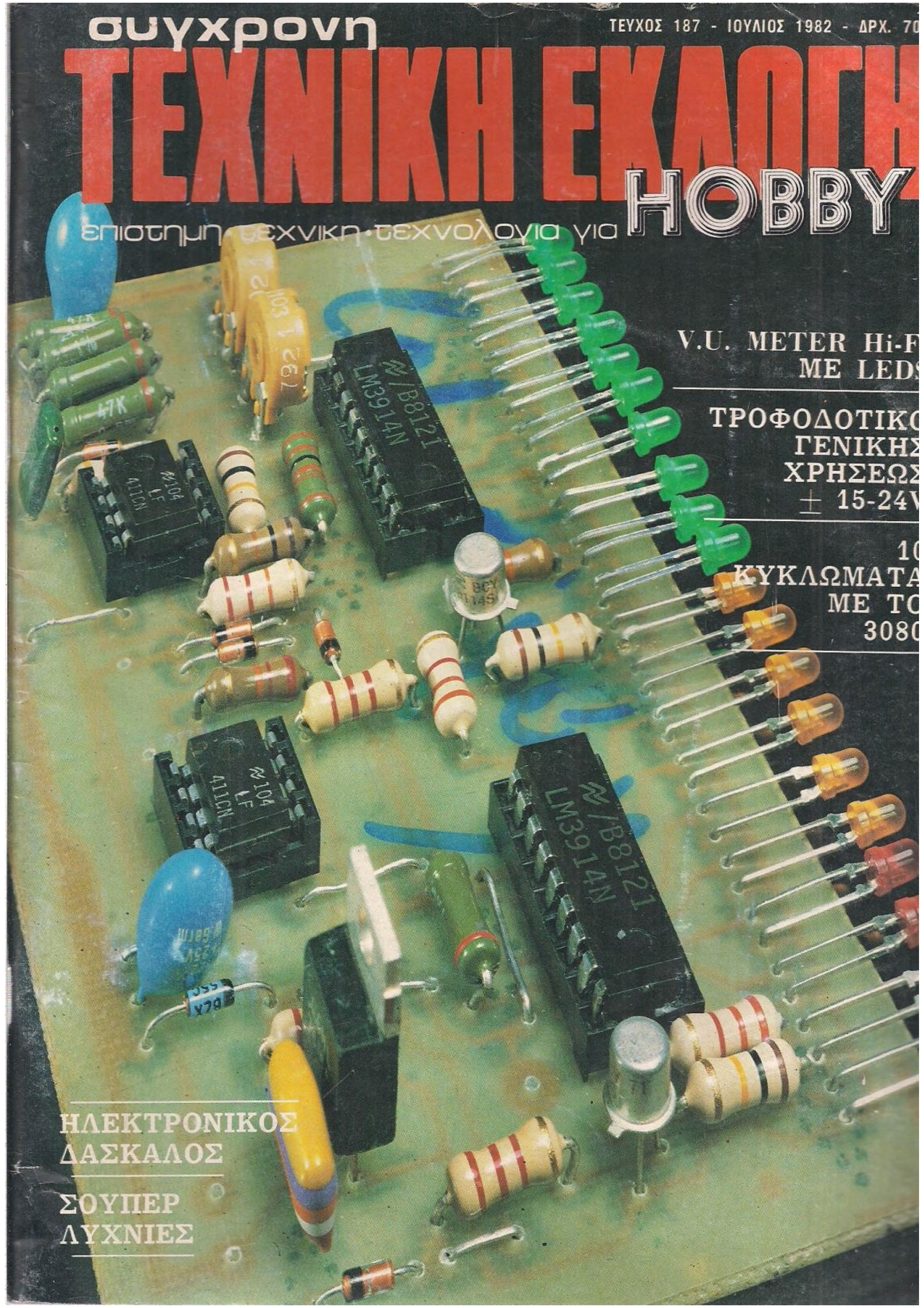
Tekniske data:

Udgangseffekt	:	Max. 1 Watt ved 45 Volt batterispænding
Udgangseffekt	:	Max. loo m Watt ved 9 volt batterispænding
Frekvensområde	:	Fra 6o M Hz til Ca. 145 M Hz
Spænding	:	4,5 volt til 45 Volt.
Strømforbrug	:	lo mA til 5o mA max.
Indgangsfølsomhed	:	Mikrofon, dynamisk, lo mV
Indgangsimpedans	:	Max. 22 K Ω

HF 65 bør kun afgive 1 Watt over en kortere periode uden køleplade

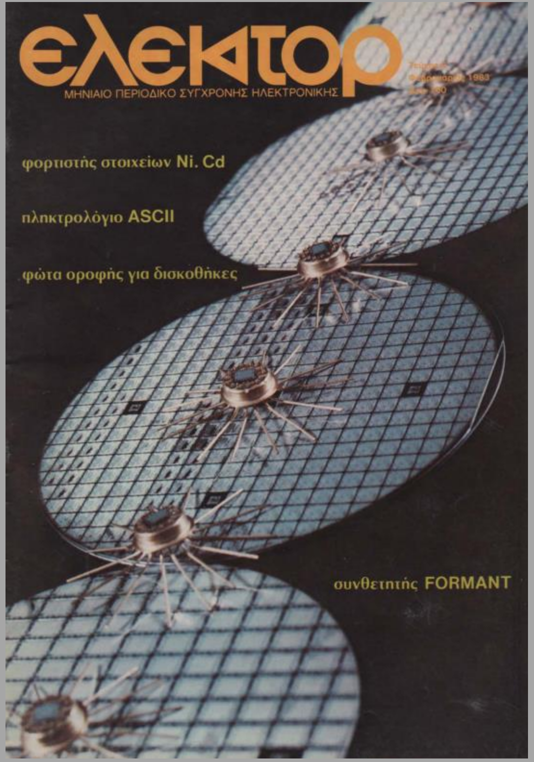

Ant.
+9...
40V
4.7kΩ
470pF
10kΩ
20pF
Eing.
NF
6.8µF
220kΩ
6.8µF
3pF
2N2219
22kΩ
Modu-
lation
Gnd.
470pF
10kΩ
0
100Ω
BC172


C5
R5
2
R4
C6
C3
R3
JOSTY KIT
R2
C2
HF65
R1
MADE IN DENMARK

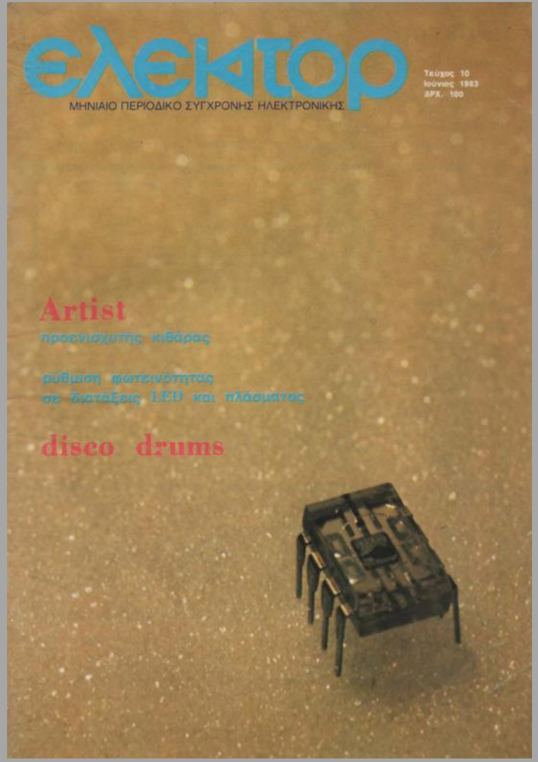

συγχρονη
ΤΕΧΝΙΚΗ ΕΚΛΟΓΗ
ΤΕΥΧΟΣ 187 - ΙΟΥΛΙΟΣ 1982 - ΔΡΧ. 70
HOBBY
επιστημη·τεχνικη·τεχνολογια για
V.U. METER Hi-F
ME LEDS
ΤΡΟΦΟΔΟΤΙΚΟ
ΓΕΝΙΚΗΣ
ΧΡΗΣΕΩΣ
± 15-24V
ΗΛΕΚΤΡΟΝΙΚΟΣ
ΔΑΣΚΑΛΟΣ
ΣΟΥΠΕΡ
ΛΥΧΝΙΕΣ

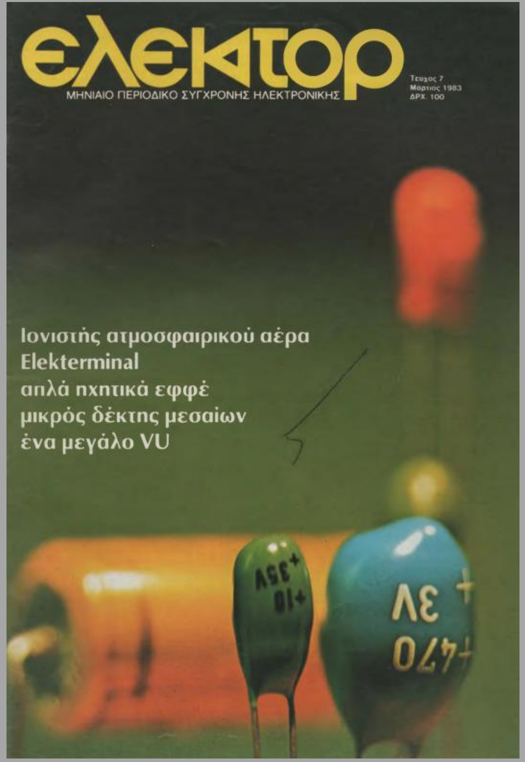

ΤΕΧΝΙΚΗ
ΕΚΛΟΓΗ 131
Ενισχυτής 100 Watt γιά κιθάρα καί Hi-Fi
Ενισχυτής 10 Watt μέ ολοκληρωμένο
ΤΕΧΝΙΚΗ
ΕΚΛΟΓΗ 138
ΕΓΚΥΚΛΟΠΑΙΔΙΚΗ ΤΕΧΝΙΚΗ ΕΠΙΘΕΩΡΗΣΗ
ΔΡΧ. 30
ΠΟΜΠΟΣ ΣΤΑ FM (κατασκευή)
Τεχνικά νέα
Φωτογραφία
Πρακτικές ιδέες


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
ΤΕΥΧΟΣ 2 ΟΚΤΩΒΡΙΟΣ 1982 ΔΡΧ. 100
70cm transverter για επικοινωνία μέσω δορυφόρου
Μετρητής στροφών
Μετρητής Geiger - Müller
Νέοι μέθοδοι ελαττώσεως θορύβου CX και DNR


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
φορτιστής στοιχείων Ni. Cd
πληκτρολόγιο ASCII
φώτα οροφής για δισκοθήκες
συνθετητής FORMANT


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
ΤΕΥΧΟΣ 4ο ΔΕΚΕΜΒΡΙΟΣ 1982 ΔΡΧ. 100
Το μίνι αρμόνιο
υπολογιστής Τζόζεφσον
συναγερμός αυτοκινήτου


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
Τεύχος 10 Ιούνιος 1983 ΔΡΧ. 100
Artist
προενισχυτής κιθάρας
ρύθμιση φωτεινότητας σε διατάξεις LED και πλάσματος
disco drums


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
Τεύχος 7 Μάρτιος 1983 ΔΡΧ. 100
Ιονιστής ατμοσφαιρικού αέρα
Elekterminal
απλά ηχητικά εφφέ
μικρός δέκτης μεσαίων
ένα μεγάλο VU


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
ΤΕΥΧΟΣ 1 ΣΕΠΤΕΜΒΡΙΟΣ 1982 ΔΡΧ. 100
ενισχυτής 100W
το OTA 13600
δέκτης υπεραναδράσεως 87 - 180MHz


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
Τεύχος 5 Ιανουάριος 1983 Δρχ. 100.
μετρητής αποστάσεων με υπερήχους
προενισχυτής μαγνητικής κεφαλής
αναλογικές γραμμές καθυστερήσεως
μετρητής στροφών για έλικες
πεδιόμετρο
Scrabler
μετρητής γύρων


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
ΤΕΥΧΟΣ 3 ΝΟΕΜΒΡΙΟΣ 1982 ΔΡΧ. 100
γαλάζια LED
ενισχυτές κεραιών
ψευδο ROM
δυναμική ελάττωση θορύβου DNR)


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
Σύστημα ασφαλείας «Κέρβερος»
Ξυλόφωνο
Τροφοδοτικό ακριβείας
Ψηφιακή ένδειξη ραδιοφωνικών σταθμών


ελεκτορ
ΜΗΝΙΑΙΟ ΠΕΡΙΟΔΙΚΟ ΣΥΓΧΡΟΝΗΣ ΗΛΕΚΤΡΟΝΙΚΗΣ
εισαγωγή στους μικροεπεξεργαστές παίζοντας στην TV
ενισχυτής τηλεφώνου
διαμορφωτής VHF/UHF

ΤΕΧΝΙΚΗ

ΕΚΛΟΓΗ

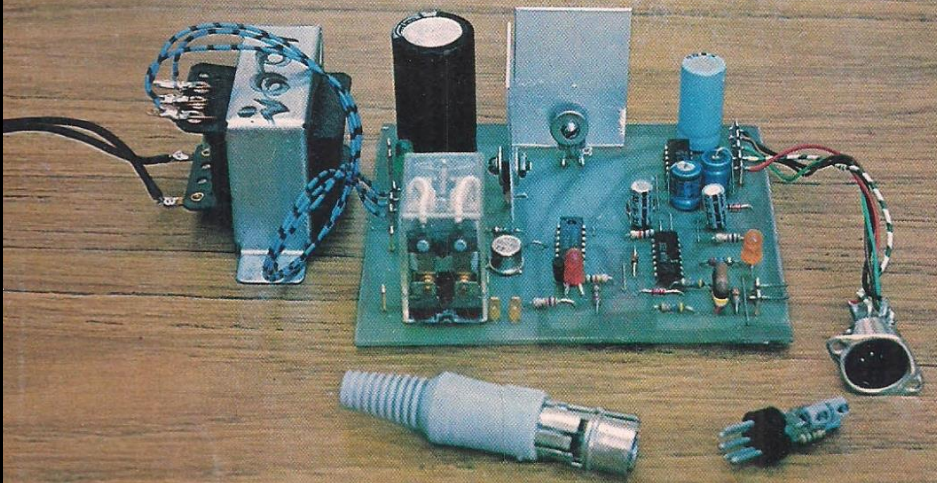
ΤΕΥΧΟΣ 159
ΜΑΡΤΙΟΣ

ΕΓΚΥΚΛΟΠΑΙΔΙΚΗ ΤΕΧΝΙΚΗ ΕΠΙΘΕΩΡΗΣΗ

ΔΡΧ. 35

BASIC ἡ ἁπλή γλώσσα τῶν ὑπολογιστῶν

ΣΥΣΤΗΜΑ ΣΥΝΑΓΕΡΜΟΥ ΓΙΑ ΥΠΕΡΑΣΦΑΛΕΙΑ



προενισχυτής γιά μαγνητική κεφαλή πικάπ

ΓΕΦΥΡΑ ΜΕΤΡΗΣΕΩΣ ΣΤΑΣΙΜΩΝ

BASIC
η απλη γλωσσα των υπολογιστων

ια νά άγοράσει κανείς ένα υπολογιστή γιά δική του χρήση έχει γίνει πολύ δύσκολο: διότι πολλοί κυκλοφορούν στην αγορά. Οι υπολογιστές έχουν γίνει σήμερα πολλοί απαραίτητοι και ο αριθμός των εργασιών που μπορούν νά κάνουν είναι φοβερά μεγάλος. Μερικές από τις δουλειές που μπορεί νά κάνει σήμερα ένας υπολογιστής είναι:

- Δημιουργία συστήματος οικονομίας ενέργειας στό σπίτι με τον έλεγχο από τόν υπολογιστή του καλοριφέρ ή αίρ - κοντίσιον
- Αύτόματο άνοιγμα και κλείσιμο παραθύρων
- Γράψιμο φορολογικών δηλώσεων
- Προγραμματισμός δίαιτας: πόσα κιλά επιθυμεί κάποιος νά χάσει, σέ πόσο χρόνο θέλει νά τά χάσει.
- Υπολογισμός προϋπολογισμών, ισολογισμών, κ.α
- Μετατροπή σε σύστημα συναγερμού
- Ρύθμιση φώτων (αναβοσβήσιμο).
- Καταγραφή αριθμών τηλεφώνου κ.ά.

Ο υπολογιστής είναι τό μηχάνημα τού μέλλοντος και αργά ή γρήγορα όλοι πρέπει νά μάθουν νά τόν χειρίζονται. Γιά νά μπορεί όμως κάποιος νά δώσει τις διάφορες εντολές στόν υπολογιστή, θα πρέπει νά επικοινωνήσει μαζί του, δηλ. θα πρέπει νά «μιλάει» την ίδια «γλώσσα» μέ τόν υπολογιστή. Υπάρχουν διάφορες γλώσσες. Μεταξύ αυτών μιά από τις σπουδαιότερες είναι τό "BASIC" σύστημα. Πρόκειται γιά πολύ εύκολη γλώσσα προγραμματισμού και προορίζεται γι' αυτούς πού ούτε ξέρουν ούτε έχουν την πρόθεση νά μάθουν πώς λειτουργεί ο υπολογιστής τους. Δημιουργήθηκε στό Κολλέγιο Dartmouth της Αμερικής στις αρχές του 1960 και από τότε έγινε μιά από τις σπουδαιότερες γλώσσες προγραμματισμού. Τό όνομα προέρχεται από τις λέξεις (Beginners All-purpose Symbolic Instruction Code), που σημαίνει συμβολικός κώδικας οδηγιών κάθε χρήσεως γιά αρχάριους.

Η BASIC μπορεί νά είναι απλή, αλλά είναι πολύ χρήσιμη και κυρίως εξυπηρετεί τούς αρχάριους πού έχουν λίγες γνώσεις γιά τη λειτουργία τών υπολογιστών.

Η γλώσσα χρησιμοποιεί αγγλικές προτάσεις και διαταγές. Γιά παράδειγμα ας δούμε πώς θα γραφόταν μιά πρόσθεση γιά νά εκτελεσθεί από τόν υπολογιστή και συγκεκριμένα η πρόσθεση τών αριθμών 2 και 3.

```
10A = 2 + 3
20 PRINT A
30 END
```

Expand your horizons

BASIC 4

```
>1REM THIS IS A GAME PROGRAMM
>10 PRINT "IF YOY WANT TO START THEGAME"
>20 PRINT "PRESS KEY MARKED 1"
>30 INPUT A
>40 IF A=1 THEN GO TO 60
>50PRINT"IF YOU DON'T WANT TO BESTARTED THATS"
>55 PRINT "YOUR PROBLEM"
>60 LET D=RND*10+1
>70 LET C=INTD
>80 LET D=RND*10+1
>90 LET E=INTD
>100 PRINT "PLEASE ENTER SHOT SPOT"
>110 INPUT F,G,
>120 IF F=C THEN 160
>130 IF G'=E THEN 160
>140 PRINT "I'M SORRY...!"
>150 GOTO 60
>160 IF(F^2/G^2)^2=(C^2/E^2)^2 THEN190
>170 PRINT"YOY HAVE SHOT THE HALFSPOT"
>180 GOTO 100
>190 PRINT YOY WINNED
>200 END
```

```
10 REM expression evaluator E$                S$ = ""
                                       IS DIM: P$ = ""
                                       DEF C$ = SEG$(E$,I,1)
20 FOR I = 1 TO LEN(E$)
30 IF C$ > CHR$(128) THEN 70
40 P$=P$ & E$                                  ! Next I
50
60                          GOTO 220           !   ..
70 IF C$ <> CHR$(183) THEN 110
80 PR=0
90
100 S$= C$ & S$
110 IF C$ <> CHR$(182) THEN 130
120 PR=0
130 IF S$ < C$ THEN 200
140 IF ASC(S$) <> 183 THEN 170
150 S$= SEG$(S$,2,255)
160 GOTO 220       ! NEXT I
170 P$=P$ & CHR$(ASC(S$))
180 S$= SEG$(S$,2,255)
190 GOTO 130
200 IF C$= CHR$(182) THEN 220! NEXT I
210 S$= C$ & S$
220 NEXT I
230 IF S$="" THEN 270
240 P$ = P$ & CHR$(ASC(S$))
250 S$ = SEG$(S$,2,255)
260 GOTO 270
270 ...
```

Ερέμου Πατρασσον 65
Αθηνα 11475

Programs can process and generate other programs

Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1156–1168
The Journal of Systems and Software
www.elsevier.com/locate/jss

A framework for the static verification of API calls

Diomidis Spinellis *, Panagiotis Louridas

Department of Management Science and Technology, Athens University of Economist and Business, Patision 76, GR-104 34 Athens, Greece

Received 20 February 2006; received in revised form 18 September 2006; accepted 30 September 2006
Available online 13 November 2006

Abstract

A number of tools can statically check program code to identify commonly encountered bug patterns. At the same time, programs are increasingly relying on external APIs for performing the bulk of their work: the bug-prone program logic is being fleshed-out, and many errors involve tricky subroutine calls to the constantly growing set of external libraries. Extending the static analysis tools to cover the available APIs is an approach that replicates scarce human effort across different tools and does not scale. Instead, we propose moving the static API call verification code into the API implementation, and distributing the verification code together with the library proper. We have designed a framework for providing static verification code together with Java classes, and have extended the FindBugs static analysis tool to check the corresponding method invocations. To validate our approach we wrote verification tests for 100 different methods, and ran FindBugs on 6.9 million method invocations on what amounts to about 13 million lines of production-quality code. In the set of 55 thousand method invocations that could potentially be statically verified our approach identified 800 probable errors.

Keywords: Static analysis; API; Library; Programming by contract; FindBugs

1. Introduction

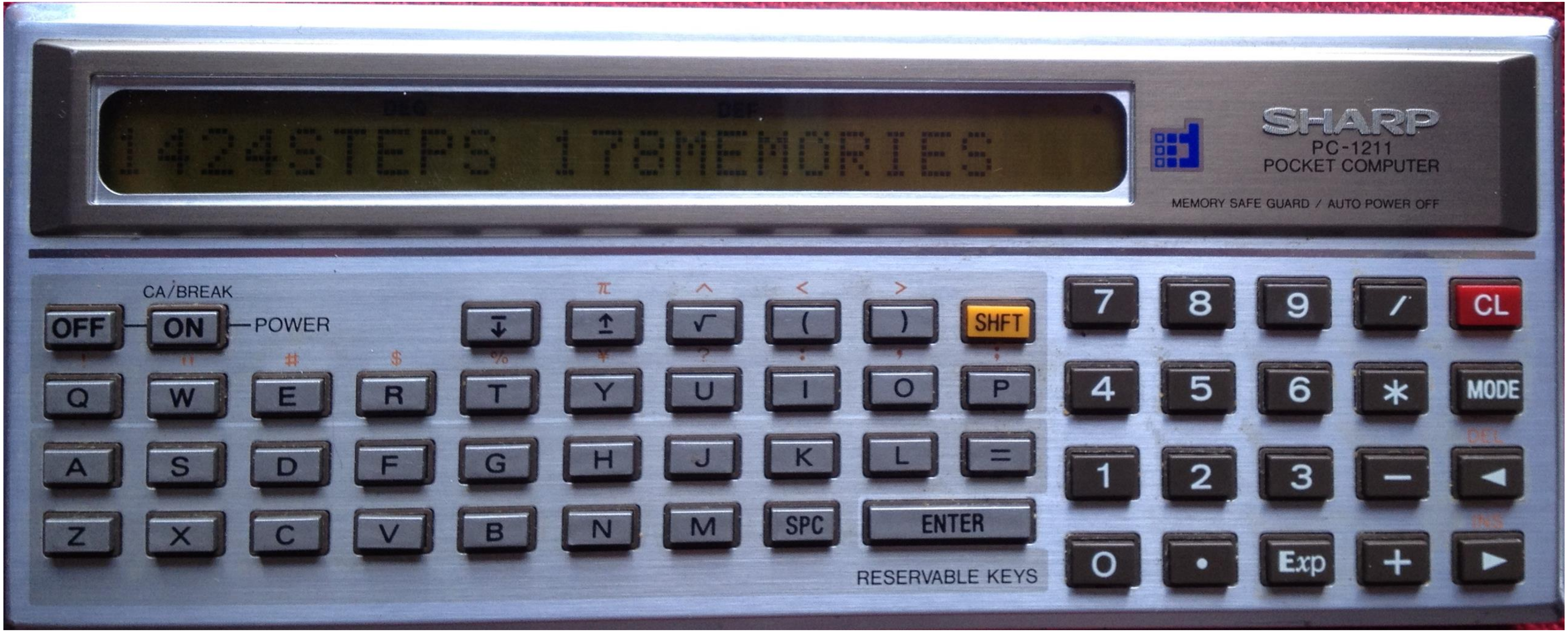
Automatic program verification tools have had a significant impact on software development, and are more and more used in practice to eliminate many errors that in the past would have caused program crashes, security vulnerabilities, and program instability (Johnson, 1977; Bush et al., 2000; Ball and Rajamani, 2002; Das et al., 2002; Csallner and Smaragdakis, 2005; Cok and Kiniry, 2005; Barringer et al., 2006). However, two software development trends are now hindering the applicability of automated program verification tools:

(1) the increasing use of binary-packaged components (for the most part libraries) through their application programming interface (API), and
(2) the increasing API sophistication, and in particular the embedding of many different domain-specific languages (DSLs) as strings in the program code.

Both trends reduce the efficiency of the current approaches. The use of feature-rich libraries in their binary form handicaps verification programs that require access to source code, such as ESC/Java (Flanagan et al., 2002), and also programs that contain a fixed-set of specific bug patterns, like ITS4 (Viega et al., 2000). Furthermore, the diversity of the libraries handicaps any tool that depends on a centralized repository of verification patterns. In addition, the embedding of DSLs, like SQL and XPath, in strings appearing in the program's source code can introduce bugs that are beyond the reach of the current breed of tools based on approaches like theorem proving (Flanagan et al., 2002), dataflow analysis (Jackson, 1995), and finite state machines (Ball and Rajamani, 2002). To overcome these difficulties we propose a framework for incorporating API call verification code within each library containing the corresponding API implementation. Through the use of reflection techniques program checkers can invoke this

* Corresponding author. Tel.: +30 2108203981; fax: +30 2108203370.
E-mail addresses: dds@aueb.gr (D. Spinellis), louridas@aueb.gr (P. Louridas).

0164-1212/$ - see front matter
doi:10.1016/j.jss.2006.09.040


1424STEPS 178MEMORIES
SHARP
PC-1211
POCKET COMPUTER
MEMORY SAFE GUARD / AUTO POWER OFF
CA/BREAK
OFF
ON
POWER
SHFT
CL
MODE
ENTER
SPC
Exp
RESERVABLE KEYS

PC-1211

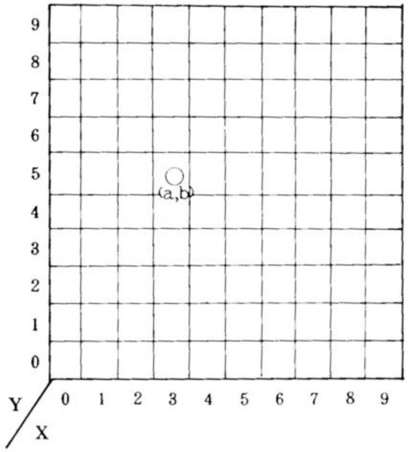
SHARP

POCKET COMPUTER

APPLICATIONS MANUAL

Title	TREASURE-HUNTING GAME	PROGRAM NO. P 4 – X – 11

[Formula]



This is a game in which the location of treasure is decided by random numbers and a player hunts out the treasure.

Your initial position is at (0, 0). Input a distance you want to cover in each of the x and y directions. If you get out of the matrix shown on the right, the error indication appears, so try again.

A distance between you and the treasure is indicated as a hint by the value of "ABS(x – a) + ABS(y – b)". You are allowed an energy of 60 at the start of treasure hunt. The energy decreases by a sum of distances in the x and y directions which you make. And when you locate the treasure, ▽HIT▽ is displayed and the energy increases by 5. Every time you input, the following indication appears:

Example:

4	5	4	51
x coordinate of present position	y coordinate of present position	Distance bet. you and treasure (Hint)	Remaining energy

Every time the treasure is hunted out, a new location of the treasure is fixed by random numbers. You continue to hunt the treasure until the energy you have exhausts. The number of hunted treasures is displayed at the end of the game.

[Operation]

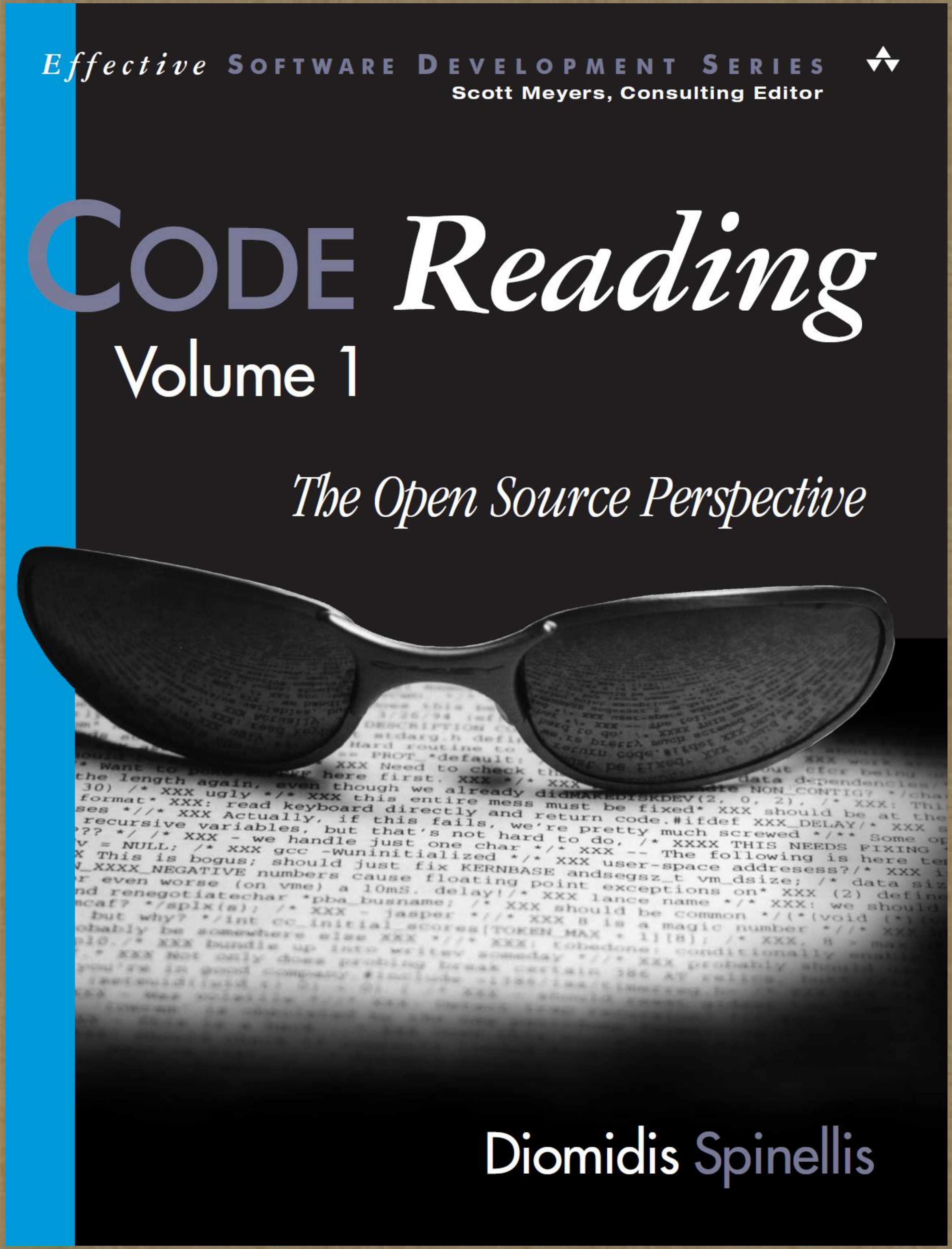
CLOAD ▽X11▽ ENTER

	Input	Display	Note		Input	Display	Note
1	SHFT A	INITIAL =		11	– 4 ENTER	HIT	
2	2345781 ENTER	0 0 9 60		12		6 3 6 44	
3		DX DY ?		13		⋮	
4	4 ENTER			14		⋮	
5	5 ENTER	4 5 4 51		15		Continue fill energy exhausts	
6		DX DY ?		16			
7	–2 ENTER	?		17			
8	2 ENTER	2 7 8 47		18			
9		DX DY ?		19			
10	4 ENTER	?		20			

Title	TREASURE-HUNTING GAME	PROGRAM NO. P 4 – X – 11

Memory content		
A	1	x coordinate of target
B	2	y coordinate of target
C	3	Distance
D	4	No. of targets found
E	5	√
F	6	√
G	7	√
H	8	√
I	9	
J	10	
K	11	
L	12	
M	13	
N	14	
O	15	
P	16	√
Q	17	√
R	18	
S	19	
T	20	
U	21	
V	22	
W	23	√
X	24	√
Y	25	√
Z	26	Energy

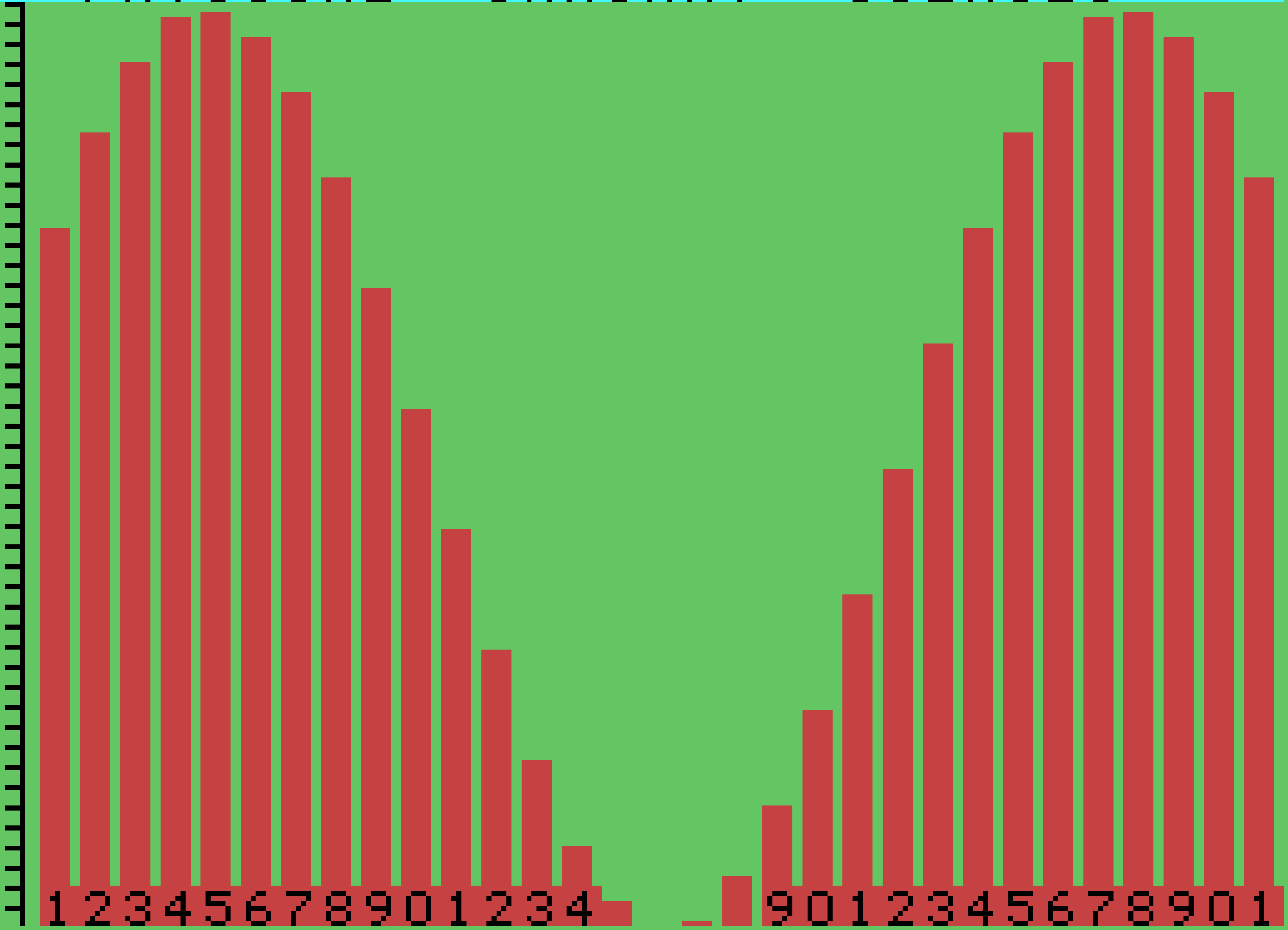
```
 10: "A":Z=60:X=0:Y=0:D=0
 20: INPUT "INITIAL=";H
 30: GOSUB 500
 40: A=W
 50: GOSUB 500
 60: B=W
 70: C=ABS (X-A)+ABS (Y-B)
 75: PAUSE USING "#####";X;Y;C;Z
 80: PAUSE USING "#####";X;Y;C;Z
 90: INPUT "DX DY ?",P,Q
100: Z=Z-ABS P-ABS Q
110: IF 0>=ZBEEP 4:PRINT "TARGET",D:END
120: IF (X+P>=0)*(9>=X+P)*(Y+Q>=0)*(9>=Y+Q)
    <>0GOTO 160
150: PAUSE "ERROR":GOTO 80
160: X=X+P:Y=Y+Q
170: IF X=AIF Y=BGOTO 190
180: GOTO 70
190: BEEP 2:PAUSE "HIT"
200: D=D+1:Z=Z+5:GOTO 30
210: END
500: E=ABS (439147+E+F)
510: G=E8+1:H=23*E
520: E=H-INT (H/G)*G
530: W=INT (10*E/G)
540: RETURN
  367
```


Effective Software Development Series
Scott Meyers, Consulting Editor
Code Reading
Volume 1
The Open Source Perspective
Diomidis Spinellis

Learn by reading code
(and studying systems)
that other people have written


TEXAS INSTRUMENTS
TI-99/4A
computer
DEL INS ERASE CLEAR BEGIN PROC'D AID REDO BACK QUIT
ENTER
SHIFT
ALPHA LOCK
CTRL
FCTN
COMMAND MODULE
TI Extended BASIC
Solid State Software

PHYSICAL JANUARY GIORGOS

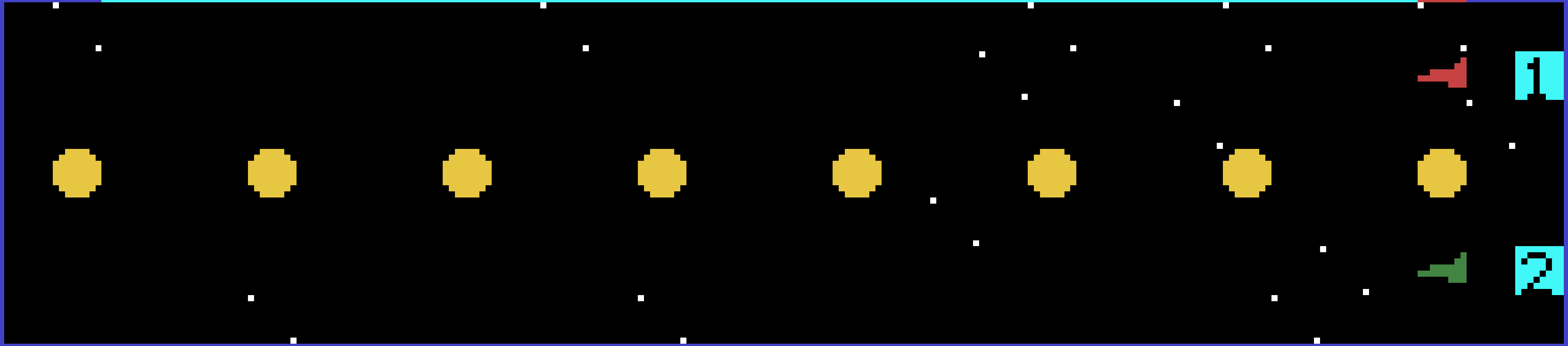


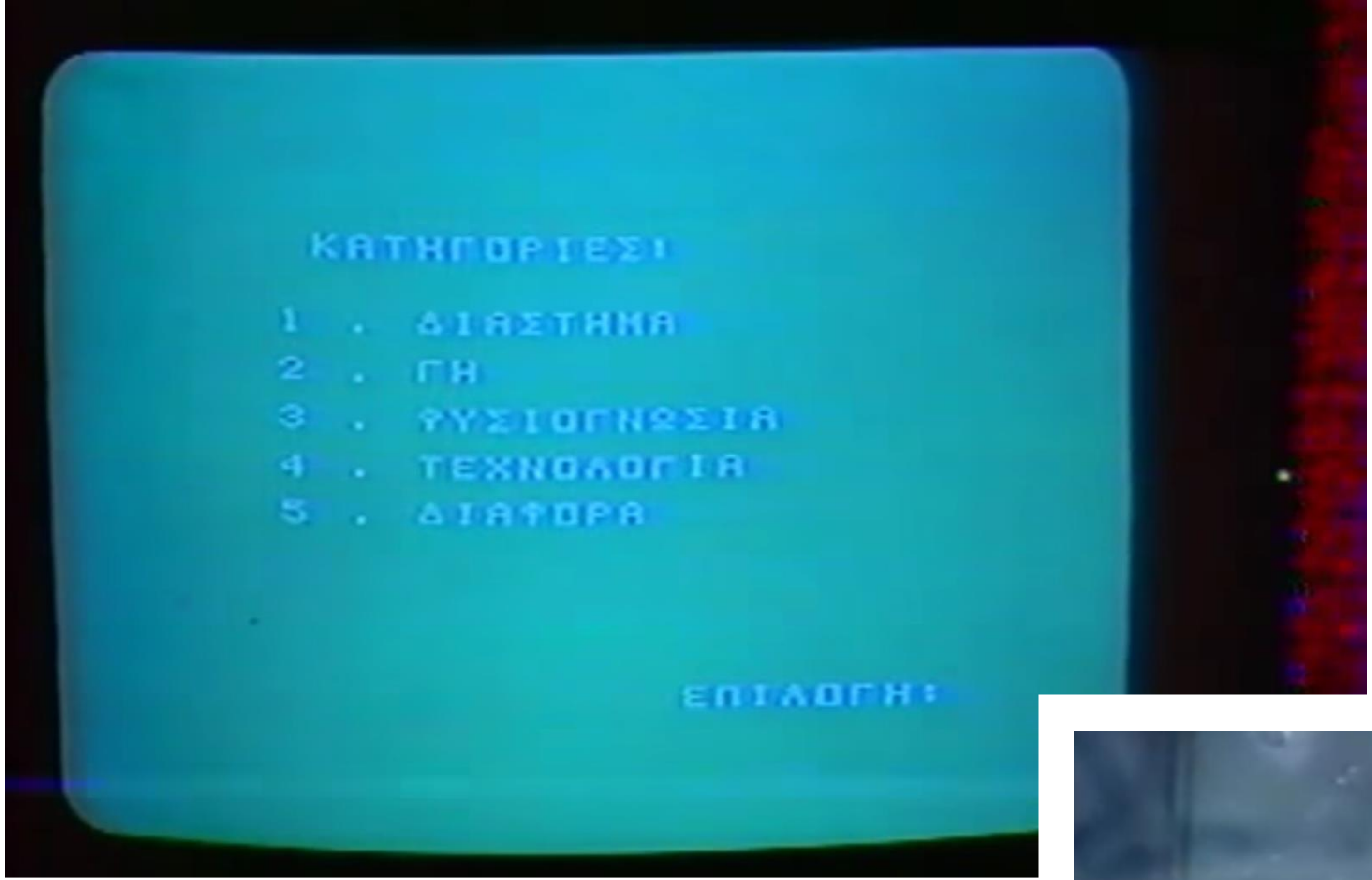
ΚΟΚΚΙΝΟΙ ΓΙΓΑΝΤΕΣ
ΑΣΠΡΟΙ ΝΑΝΟΙ

ΟΜΑΔΑ 1

ΠΟΙΑ ΕΙΝΑΙ Η ΜΕΓΑΛΥΤΕΡΗ Α
ΠΟΣΤΑΣΗ ΑΠΟ ΤΗΝ ΓΗ ΤΟΥ ΠΛΟ
ΥΤΩΝΑ

____




ΚΑΤΗΓΟΡΙΕΣ:
1 . ΔΙΑΣΤΗΜΑ
2 . ΓΗ
3 . ΦΥΣΙΟΓΝΩΣΙΑ
4 . ΤΕΧΝΟΛΟΓΙΑ
5 . ΔΙΑΦΟΡΑ
ΕΠΙΛΟΓΗ:


A=32
B=30

```
730 SUB LOC(E$,H)
770 DATA ASTRONOMIA,BIOLOGIA,GEVGRAFIA,FYSIKH,XHMEIA
780 CALL POSITION(#26,Y,X)
790 R=INT((Y+4)/8)+1 :: C=INT((X+4)/8)+1
800 IF (R>15)OR(R<7)OR(C>23)OR(C<15)THEN H=0 :: SUBEXIT
810 IF (C/2=INT(C/2))OR(R/2=INT(R/2))THEN H=0 :: SUBEXIT
820 CALL MOTION(#26,0,0)
830 CALL SOUND(100,1000,0)
840 CALL DELSPRITE(#26)
850 CALL SPRITE(#26,42,7,R*8-12,C*8-12)
860 CALL MAGNIFY(2)
862 IF P=1 THEN 870
863 RESTORE 770
865 FOR I=1 TO 5 :: READ P$(I):: NEXT I :: P=1
870 E$=P$(INT(R/2)-2):: H=INT(C/2)-6
880 SUBEND
640 SUB WHAT(E$,H)
650 DISPLAY AT(20,4):"EPISTHMH:" :: DISPLAY AT(21,4):"DYSKOLIA:"
660 CALL SPRITE(#26,140,2,32,64)
670 CALL JOYST(1,X,Y)
680 CALL MOTION(#26,05*(Y=+4)-05*(Y=-4),05*(X=-4)-05*(X=+4))
690 CALL KEY(1,RET,STA)
700 IF RET=18 THEN CALL LOC(E$,H)ELSE 670
710 IF H=0 THEN 670 ELSE DISPLAY AT(20,13):E$ :: DISPLAY AT(21,13):STR$(H):: SUB EXIT
720 SUBEND
730 SUB LOC(E$,H)
770 DATA ASTRONOMIA,BIOLOGIA,GEVGRAFIA,FYSIKH,XHMEIA
780 CALL POSITION(#26,Y,X)
790 R=INT((Y+4)/8)+1 :: C=INT((X+4)/8)+1
800 IF (R>15)OR(R<7)OR(C>23)OR(C<15)THEN H=0 :: SUBEXIT
810 IF (C/2=INT(C/2))OR(R/2=INT(R/2))THEN H=0 :: SUBEXIT
820 CALL MOTION(#26,0,0)
830 CALL SOUND(100,1000,0)
840 CALL DELSPRITE(#26)
850 CALL SPRITE(#26,42,7,R*8-12,C*8-12)
860 CALL MAGNIFY(2)
862 IF P=1 THEN 870
863 RESTORE 770
865 FOR I=1 TO 5 :: READ P$(I):: NEXT I :: P=1
870 E$=P$(INT(R/2)-2):: H=INT(C/2)-6
880 SUBEND
```

```
890 SUB SCR1
900 DEF CTR$(W$)=SEG$("            ",1,(28-LEN(W$))/2)&W$
910 DISPLAY AT(1,1):CTR$("KOKKINOI GIGANTES")
920 DISPLAY AT(2,1):CTR$("ASPROI NANOI")
930 DISPLAY AT(10,1):CTR$("@ 1983 DIOMHDHS SPINELLHS")
940 DISPLAY AT(23,1):CTR$("PATA ENA KOYMPI")
950 DISPLAY AT(24,1):CTR$("GIA NA ARXISEI")
960 CALL KEY(5,RET,STA):: IF STA=0 THEN 960
970 CALL CLEAR :: CALL SOUND(100,1000,0)
980 SUBEND
990 SUB ACC(N1$,N2$,N3$,N4$)
1000 CALL CLEAR
1010 DISPLAY AT(4,1):"OMADA 1"
1020 ACCEPT AT(5,1)VALIDATE(UALPHA)SIZE(10)BEEP:N1$
1030 ACCEPT AT(6,1)VALIDATE(UALPHA)SIZE(10)BEEP:N2$
1040 DISPLAY AT(8,1):"OMADA 2"
1050 ACCEPT AT(9,1)VALIDATE(UALPHA)SIZE(10)BEEP:N3$
1060 ACCEPT AT(10,1)VALIDATE(UALPHA)SIZE(10)BEEP:N4$
1070 CALL CLEAR
1080 SUBEND
1090 SUB MAIN(N1$,N2$,N3$,N4$,E$,H,W)
1100 CALL CLEAR :: CALL DELSPRITE(ALL):: CALL
CHAR(141,"18181818181818181818181818FFFF1818"):: CALL HCHAR(9,1,95,32):: CALL VCHAR(1,15,141,24)
1110 CALL VCHAR(9,15,142)
1120 DISPLAY AT(1,4)SIZE(7):"OMADA 1"
1130 DISPLAY AT(1,19)SIZE(7):"OMADA 2"
1140 DISPLAY AT(3,1)SIZE(2):"1:"
1150 DISPLAY AT(3,3)SIZE(10):N1$
1160 DISPLAY AT(3,16)SIZE(2):"1:"
1170 DISPLAY AT(3,18)SIZE(10):N3$
1180 DISPLAY AT(4,1)SIZE(2):"2:"
1190 DISPLAY AT(4,3)SIZE(10):N2$
```

Perseverance and discipline
can get you a long way

Write clear code, even when the environment doesn't make it easy for you

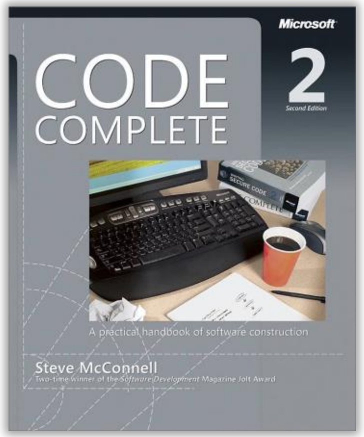

IBM
IBM
A
IBM
B
IBM

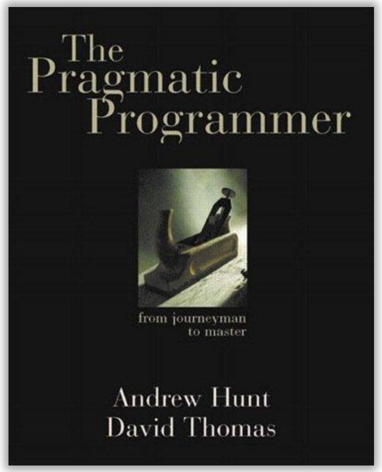

IBM
IBM
Portable
Personal
Computer

```pascal
procedure GenerateLabel(var Line : Linetype);
{Will remove the label from the line and add it to the  symbol table}
begin
  ConvertUppercase(Line);
  If Copy(Line,1,1)<>' ' then {Label exists}
  begin
    LabelName:=Copy(Copy(Line,1,Pos(' ',Line)-1),1,SymbolLength);
    CharacterizeSymbol(LabelName,PC,False,Relocatable,LabelS);
    {Labels become valid only after the directive is proved that it does no
    change their value i.e. it is not an EQU directive. This is done in the
    ObjectProcess procedure}
    Line:=Copy(Line,Pos(' ',Line),LineLength);
  end
  else
    LabelName:='$'; {For no label $ is imlied so it is a nice place holder}
  While Copy(line,1,1)=' ' do
    Line:=Copy(Line,2,LineLength);
end;
```

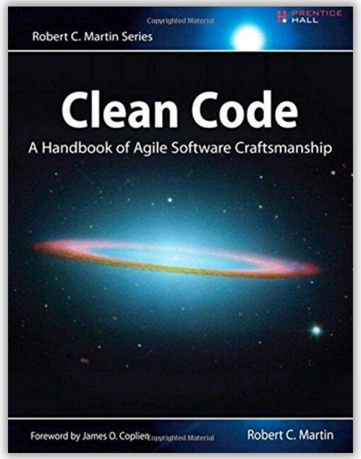
Structure your code into small units
Document your code with ample comments

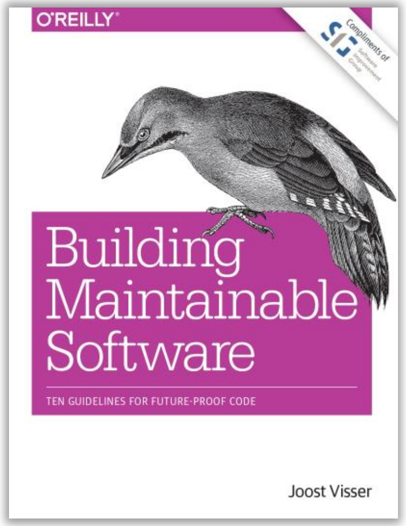
Software development practices



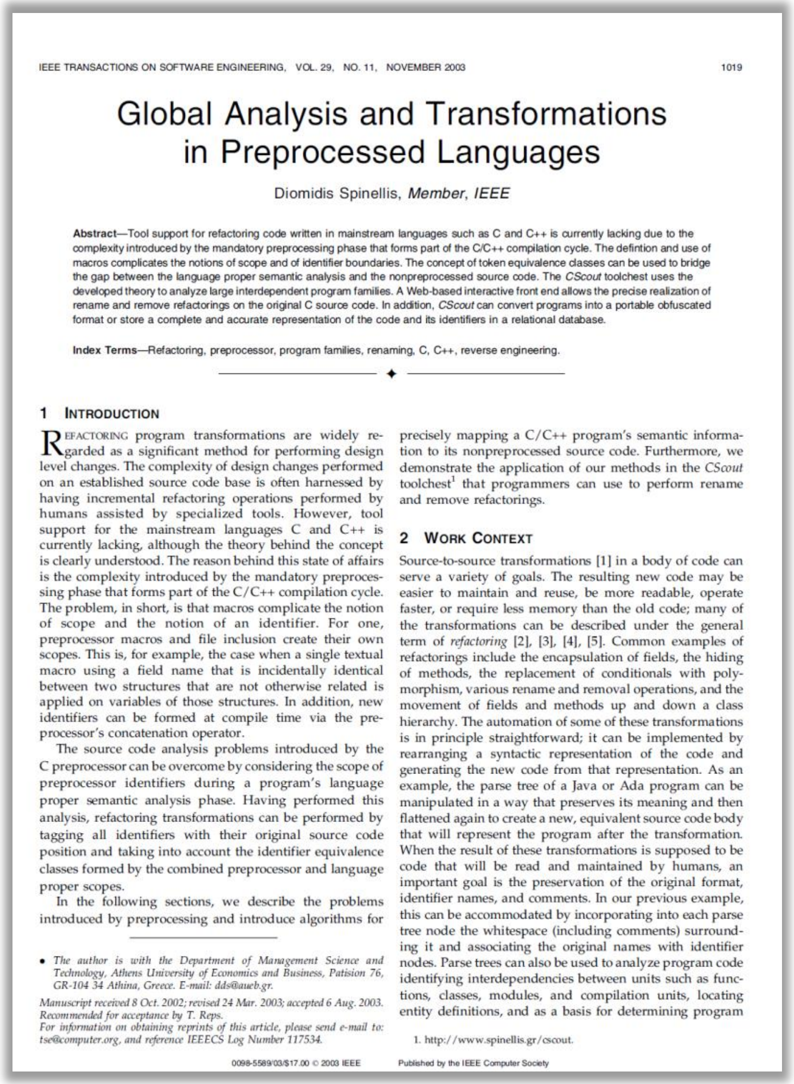


1. Put source code under revision control
2. Perform frequent small commits
3. Follow the language's style guide
4. Choose precise and consistent identifier names
5. Code in small units
6. Write unit tests
7. Separate concerns in module
8. Don't repeat yourself
9. Ensure compliance through continuous integration
10. Release your code as open source software





github.com/DSpinellis/CScout



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 29, NO. 11, NOVEMBER 2003 1019

Global Analysis and Transformations in Preprocessed Languages

Diomidis Spinellis, *Member, IEEE*

Abstract—Tool support for refactoring code written in mainstream languages such as C and C++ is currently lacking due to the complexity introduced by the mandatory preprocessing phase that forms part of the C/C++ compilation cycle. The definition and use of macros complicates the notions of scope and of identifier boundaries. The concept of token equivalence classes can be used to bridge the gap between the language proper semantic analysis and the nonpreprocessed source code. The *CScout* toolchest uses the developed theory to analyze large interdependent program families. A Web-based interactive front end allows the precise realization of rename and remove refactorings on the original C source code. In addition, *CScout* can convert programs into a portable obfuscated format or store a complete and accurate representation of the code and its identifiers in a relational database.

Index Terms—Refactoring, preprocessor, program families, renaming, C, C++, reverse engineering.

1 Introduction

Refactoring program transformations are widely regarded as a significant method for performing design level changes. The complexity of design changes performed on an established source code base is often harnessed by having incremental refactoring operations performed by humans assisted by specialized tools. However, tool support for the mainstream languages C and C++ is currently lacking, although the theory behind the concept is clearly understood. The reason behind this state of affairs is the complexity introduced by the mandatory preprocessing phase that forms part of the C/C++ compilation cycle. The problem, in short, is that macros complicate the notion of scope and the notion of an identifier. For one, preprocessor macros and file inclusion create their own scopes. This is, for example, the case when a single textual macro using a field name that is incidentally identical between two structures that are not otherwise related is applied on variables of those structures. In addition, new identifiers can be formed at compile time via the preprocessor's concatenation operator.

The source code analysis problems introduced by the C preprocessor can be overcome by considering the scope of preprocessor identifiers during a program's language proper semantic analysis phase. Having performed this analysis, refactoring transformations can be performed by tagging all identifiers with their original source code position and taking into account the identifier equivalence classes formed by the combined preprocessor and language proper scopes.

In the following sections, we describe the problems introduced by preprocessing and introduce algorithms for precisely mapping a C/C++ program's semantic information to its nonpreprocessed source code. Furthermore, we demonstrate the application of our methods in the *CScout* toolchest[1] that programmers can use to perform rename and remove refactorings.

2 Work Context

Source-to-source transformations [1] in a body of code can serve a variety of goals. The resulting new code may be easier to maintain and reuse, be more readable, operate faster, or require less memory than the old code; many of the transformations can be described under the general term of *refactoring* [2], [3], [4], [5]. Common examples of refactorings include the encapsulation of fields, the hiding of methods, the replacement of conditionals with polymorphism, various rename and removal operations, and the movement of fields and methods up and down a class hierarchy. The automation of some of these transformations is in principle straightforward; it can be implemented by rearranging a syntactic representation of the code and generating the new code from that representation. As an example, the parse tree of a Java or Ada program can be manipulated in a way that preserves its meaning and then flattened again to create a new, equivalent source code body that will represent the program after the transformation. When the result of these transformations is supposed to be code that will be read and maintained by humans, an important goal is the preservation of the original format, identifier names, and comments. In our previous example, this can be accommodated by incorporating into each parse tree node the whitespace (including comments) surrounding it and associating the original names with identifier nodes. Parse trees can also be used to analyze program code identifying interdependencies between units such as functions, classes, modules, and compilation units, locating entity definitions, and as a basis for determining program

- The author is with the Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, GR-104 34 Athina, Greece. E-mail: dds@aueb.gr.

Manuscript received 8 Oct. 2002; revised 24 Mar. 2003; accepted 6 Aug. 2003.
Recommended for acceptance by T. Reps.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117534.

1. http://www.spinellis.gr/cscout.

0098-5589/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

Science of Computer Programming 75 (2010) 216–231

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

CScout: A refactoring browser for C

Diomidis Spinellis

Athens University of Economics and Business, Department of Management Science and Technology, Patision 76, GR-104 34 Athens, Greece

ARTICLE INFO

Article history:
Received 30 November 2008
Received in revised form 14 August 2009
Accepted 3 September 2009
Available online 11 September 2009

Keywords:
C
Browser
Refactoring
Preprocessor

ABSTRACT

Despite its maturity and popularity, the C programming language still lacks tool support for reliably performing even simple refactoring, browsing, or analysis operations. This is primarily due to identifier scope complications introduced by the C preprocessor. The *CScout* refactoring browser analyses complete program families by tagging the original identifiers with their precise location and classifying them into equivalence classes orthogonal to the C language's namespace and scope extents. A web-based user interface provides programmers with an intuitive source code analysis and navigation front-end, while an SQL-based back-end allows more complex source code analysis and manipulation. *CScout* has been successfully applied to many medium and large-sized proprietary and open-source projects identifying thousands of modest refactoring opportunities.

1. Introduction

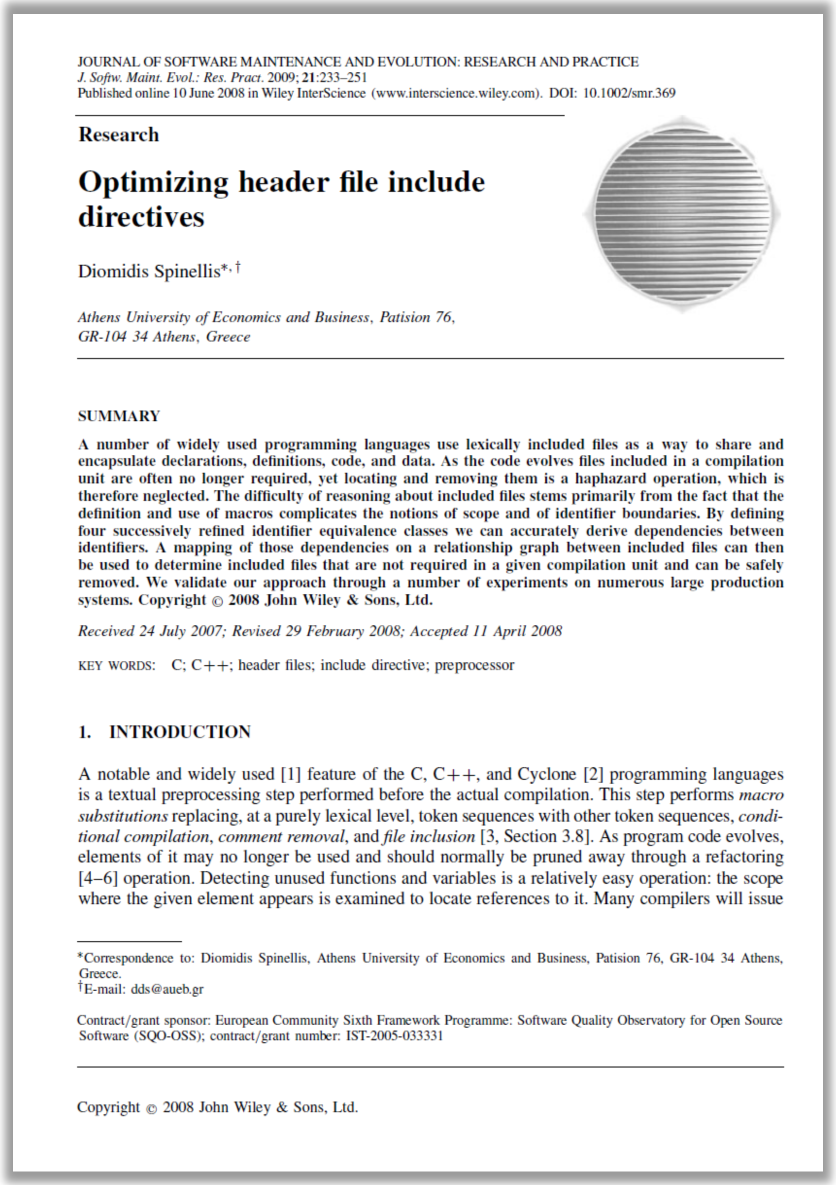
C remains the language of choice for developing systems applications, such as operating systems and databases, embedded software, and the majority of open-source projects [44, p. 16]. Despite the language's popularity, tool support for performing even simple refactoring, browsing, or analysis operations is currently lacking. Programmers typically resort to using either simplistic text-based operations that fail to capture the language's semantics, or work on the results of the compilation and linking phase that – due to the effects of preprocessing – do not correctly reflect the original source code. Interestingly, many of the tools in a C programmer's arsenal were designed in the 1970s, and fail to take advantage of the CPU speed and memory capacity of a modern workstation. In this paper we describe how the *CScout* refactoring browser, running on a powerful workstation, can be used to accurately analyze, browse, and refactor large program families written in C. The theory behind *CScout*'s operation is described in detail elsewhere [45]; this paper focuses on the tool's design, implementation, and application.

CScout can process program families consisting of multiple related projects (we define a project as a collection of C source files that are linked together) correctly handling most of the complexity introduced by the C preprocessor. *CScout* takes advantage of modern hardware (fast processors, large address spaces, and big memory capacities) to analyze C source code beyond the level of detail and accuracy provided by current IDEs, compilers, and linkers. Specifically, *CScout*'s analysis takes into account both the identifier scopes introduced by the C preprocessor and the C language proper scopes and namespaces.

The objective of this paper is to provide a tour of *CScout* by describing the domain's challenges, the operation of *CScout* and its interfaces, the system's design and implementation, and details of *CScout*'s application to a number of large software projects. The main contributions of this paper are the illustration of the types of problems occurring in the analysis of real-life C source code and the types of refactorings that can be achieved, the demonstration through the application of *CScout* to a number of systems that accurate large-scale analysis of C code is in fact possible, and a discussion of lessons associated with the construction of browsers and refactoring tools for languages, like C and C++, that involve a preprocessing step.

E-mail address: dds@aueb.gr.
URL: http://www.dmst.aueb.gr/dds.

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.09.003



JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2009; **21**:233–251
Published online 10 June 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.369

Research

Optimizing header file include directives

Diomidis Spinellis*,†

Athens University of Economics and Business, Patision 76, GR-104 34 Athens, Greece

SUMMARY

A number of widely used programming languages use lexically included files as a way to share and encapsulate declarations, definitions, code, and data. As the code evolves files included in a compilation unit are often no longer required, yet locating and removing them is a haphazard operation, which is therefore neglected. The difficulty of reasoning about included files stems primarily from the fact that the definition and use of macros complicates the notions of scope and of identifier boundaries. By defining four successively refined identifier equivalence classes we can accurately derive dependencies between identifiers. A mapping of those dependencies on a relationship graph between included files can then be used to determine included files that are not required in a given compilation unit and can be safely removed. We validate our approach through a number of experiments on numerous large production systems. Copyright © 2008 John Wiley & Sons, Ltd.

Received 24 July 2007; Revised 29 February 2008; Accepted 11 April 2008

KEY WORDS: C; C++; header files; include directive; preprocessor

1. INTRODUCTION

A notable and widely used [1] feature of the C, C++, and Cyclone [2] programming languages is a textual preprocessing step performed before the actual compilation. This step performs *macro substitutions* replacing, at a purely lexical level, token sequences with other token sequences, *conditional compilation*, *comment removal*, and *file inclusion* [3, Section 3.8]. As program code evolves, elements of it may no longer be used and should normally be pruned away through a refactoring [4–6] operation. Detecting unused functions and variables is a relatively easy operation: the scope where the given element appears is examined to locate references to it. Many compilers will issue

*Correspondence to: Diomidis Spinellis, Athens University of Economics and Business, Patision 76, GR-104 34 Athens, Greece.
†E-mail: dds@aueb.gr

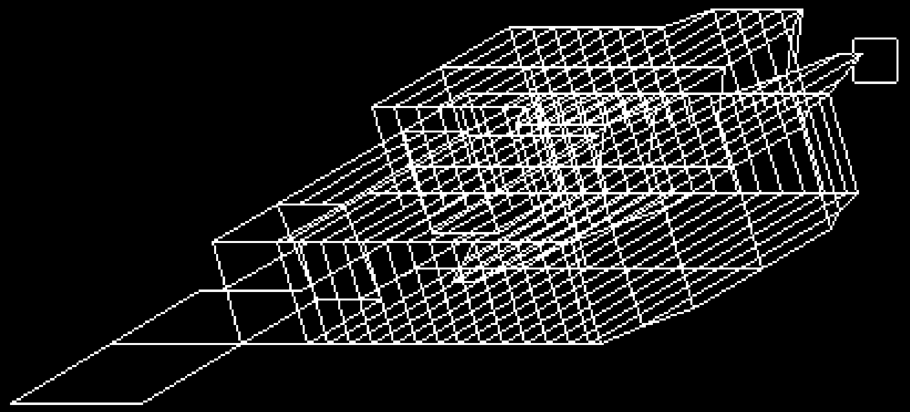
Contract/grant sponsor: European Community Sixth Framework Programme: Software Quality Observatory for Open Source Software (SQO-OSS); contract/grant number: IST-2005-033331

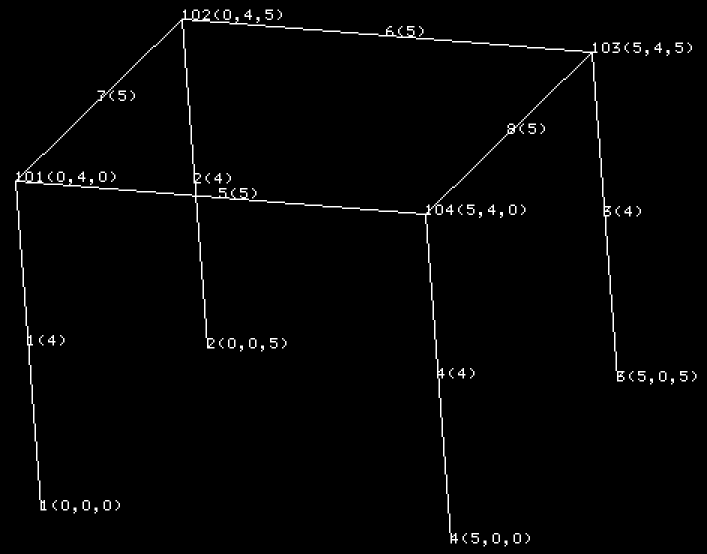
Build systems and tools

Αρχεία Στροφή Όψεις Επιλογές Αριθμοί Πληροφορία Εκτύπωση Σχεδίαση

√ Κόμβοι
√ Μέλη
Συντετ.
√ Μήκη
Παραμορφώσεις

Μεγέθ. X [500]
Μεγέθ. Y [50]
Μεγέθ. Z [500]

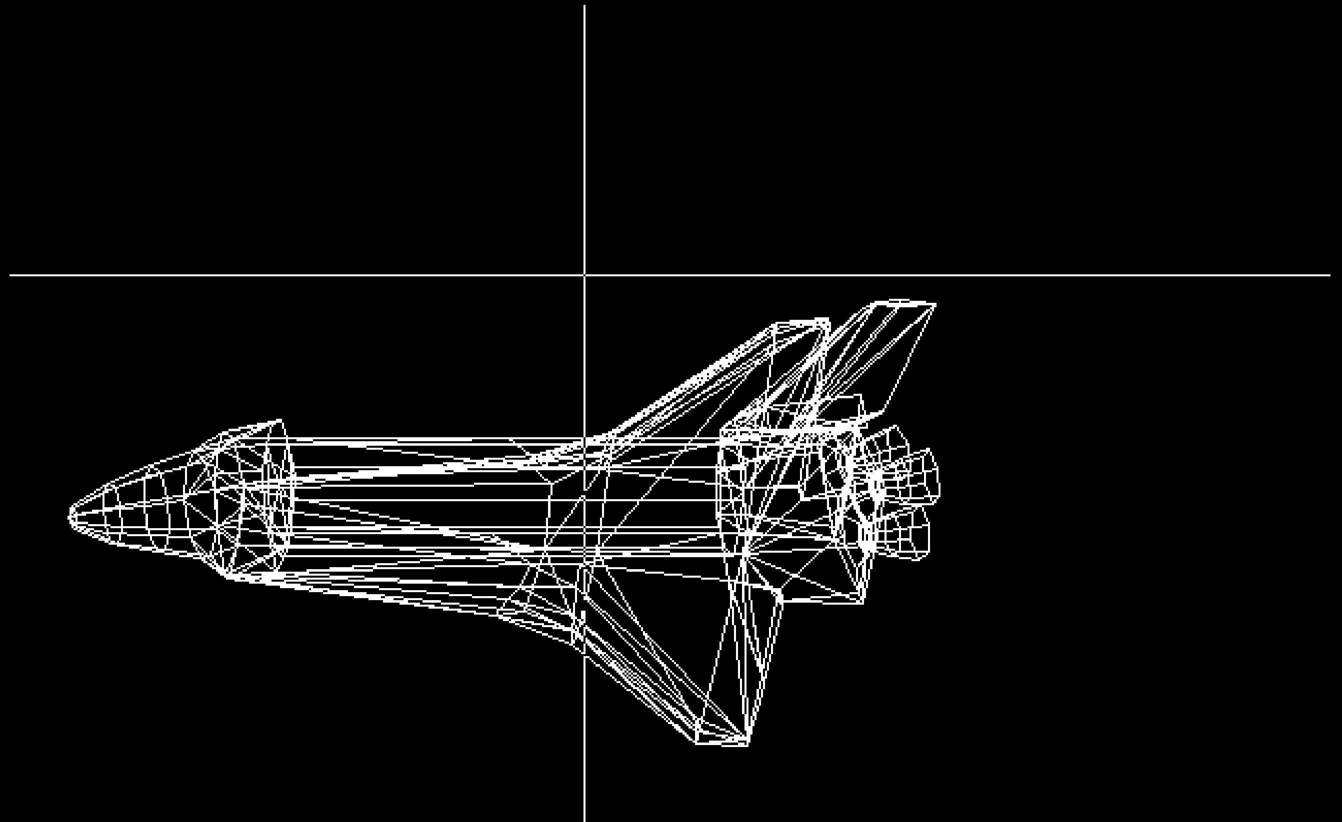




Κόμβος 135

Μέλος	←Κόμβος	Μήκος
163	134	0.68
164	136	1.55
235	100	2.20

Πατήστε ένα πλήκτρο για επιστροφή



Επιλέξτε το πρώτο σημείο
και πατήστε HOME

Keep in contact with industrial practice

Echoes from Space: Grouping Commands with Large-Scale Telemetry Data

Alexander Lattas
Department of Computing
Imperial College London
London, United Kingdom
alexandros.lattas17@imperial.ac.uk

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
dds@aueb.gr

ABSTRACT

Background: As evolving desktop applications continuously accrue new features and grow more complex with denser user interfaces and deeply-nested commands, it becomes inefficient to use simple heuristic processes for grouping GUI commands in multi-level menus. Existing search-based software engineering studies on user performance prediction and command grouping optimization lack evidence-based answers on choosing a systematic grouping method.
Research Questions: We investigate the scope of command grouping optimization methods to reduce a user's average task completion time and improve their relative performance, as well as the benefit of using detailed interaction logs compared to sampling.
Method: We introduce seven grouping methods and compare their performance based on extensive telemetry data, collected from program runs of a CAD application.
Results: We find that methods using global frequencies, user-specific frequencies, deterministic and stochastic optimization, and clustering perform the best.
Conclusions: We reduce the average user task completion time by more than 17%, by running a Knapsack Problem algorithm on clustered users, training only on a small sample of the available data. We show that with most methods using just a 1% sample of the data is enough to obtain nearly the same results as those obtained from all the data. Additionally, we map the methods to specific problems and applications where they would perform better. Overall, we provide a guide on how practitioners can use search-based software engineering techniques when grouping commands in menus and interfaces, to maximize users' task execution efficiency.

CCS CONCEPTS

• **Human-centered computing → Interaction design process and methods**; • **Software and its engineering → Software evolution**; *Search-based software engineering*;

KEYWORDS

Command grouping, menu layout, GUI optimization, telemetry, sampling

ACM Reference Format:
Alexander Lattas and Diomidis Spinellis. 2018. Echoes from Space: Grouping Commands with Large-Scale Telemetry Data. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183519.3183545

1 INTRODUCTION

Computer applications aimed at professional users are deemed to be at least as complex as the problem they aim to solve. Computer aided design applications, image and video editors, simulators and enterprise resource planners are just a few examples of programs whose interfaces have become unyieldingly complex. As developers struggle to publish frequent updates that introduce more and more integrated commands, tools and extensions, their graphical user interfaces (GUIs) become packed with icons impossible to memorize. Moreover, new, modern but niche features are placed at the center of a user's attention in order to justify the increasing costs of an update, while well-known and frequently used commands get buried in multiple nested hierarchical menus.

In the meantime, parallel efforts to improve the user's experience often are in vain, as they focus on the aesthetic aspect, or derive conclusions based on heuristics and small-scale experiments. Common user experience (UX) experiments, involving heuristics-based testing tools focus on specific scenarios that the developers think important. However, lacking a user-centric approach, the majority of the users are likely to face mental overhead and require more time when executing common tasks that involve deep-nested commands. Moreover, new users that are introduced to such complex applications will need much time to become comfortable in using them resulting, for example, in longer profitless training sessions.

We propose, evaluate and compare seven methods that exploit easily accessible program telemetry data to reorganize an application's command tree structure based on actual evidence. These methods involve a combination of command frequencies, domain-based heuristics, continuous training, and stochastic optimization. To train the algorithms, as well as to evaluate them, we use a large data set of telemetry data, created by real users of a fairly complex professional application. Additionally, we use experimental data we produced, to understand the data set and to eliminate the noise from the data.

The application studied is a CAD suite for architects and civil engineers. The architectural design functionality (TEKTON) supports 2D and 3D modeling, as well as photo-realistic rendering. On the civil engineering front (FESPA) the system supports the analysis and design of steel, concrete, and masonry structures, and also the evaluation, strengthening and repair of existing buildings under a variety of structural design standards, such as the Eurocodes and the corresponding national annexes.

Figure 1 is a screen dump depicting the application's salient characteristics and features. The top two windows show an architectural drawing floor plan and its photo-realistic rendering, while the bottom two windows show a civil engineering wood mould plan and the corresponding beam and column model, which is used for finite element analysis. The user interaction is based around entities, such as walls, windows, beam, slabs, columns, text, hatches, and stairs. Each entity has associated parameters (e.g. dimensions and material) and commands (e.g. add new, delete, extend, change properties). The current version of the application supports 13 general-purpose entities (e.g. spline or cross-section), 15 entities supporting architects (e.g. balustrade or roof), and 18 entities supporting civil engineers (e.g. column or footing). A few so-called entities relate to groupings of related commands and properties, without being associated with concrete elements appearing on a plan. Examples of these are the groupings of commands used for rendering and for global manipulations. In total, 48 entities are associated with 627 commands and 3735 properties.

The large number of available commands and properties is managed by having users interact with the application by first selecting the entity they want to manipulate. The corresponding entity icons appear in the top toolbars of Figure 1. Once an entity is selected, a toolbar with the commands associated with it appears on the left, while a separate dialog (not shown) provides access to the corresponding properties. For example, the toolbar on the left side of the window shown in Figure 1 contains the commands associated with the beam entity.

While the organization of commands and properties around entities provides a way to navigate through their large number, it also imposes a switching overhead. For instance, an architect wishing to design a house, might first employ the grid entity to draw the lines along which the house's elements will be aligned, then switch between the wall and the opening entity to add walls and windows, and then switch to the roof entity to add a tiled covering. Further adjustments to the drawing's elements (e.g. to change a window's size), would have the architect switch again to the corresponding entity.

Recently, staff dealing with the application's user experience (UX) asked us to explore alternative command arrangements that might enhance user productivity by reducing the cost of entity switching. The main idea was to make some commonly-used commands always accessible on screen. The design of a new arrangement proved to be controversial. Proposals based on the intuitive understanding of users' interactions were criticized as lacking empirical backing. On the other hand, proposed arrangements based on command frequency counts were considered unrealistic, because they ignored the sequence in which commands were issued.

Thankfully, in order to help debugging and to improve the users' experience, recent application versions can log the commands a user issues in a centralized database. It was obvious that these data could be used to create realistic command grouping optimization proposals, and also to evaluate the performance improvement associated with these proposals.

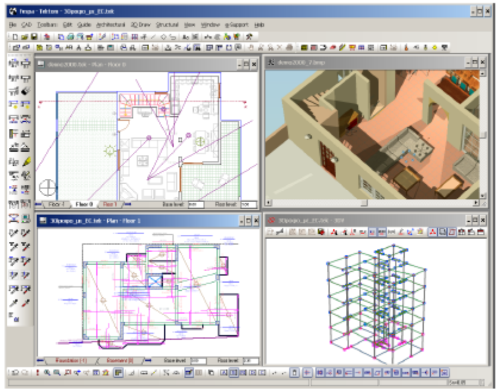


Figure 1: The CAD application in action.

Within this context we sought to answer the following research questions.

(1) What is the scope for increasing the CAD's user productivity through the optimization of command grouping?
(2) What is the relative performance of diverse command grouping optimization methods in terms of user productivity?
(3) What, if any, is the benefit of using comprehensive and detailed interaction logs, instead of sampling a few users or obtaining simple command invocation frequencies in terms of achievable command grouping performance optimization?

The two main contributions of this study are 1) the evaluation of seven command grouping optimization methods based on detailed actual interaction data, and, 2) results regarding the effect of data sampling as part of the evaluation. Our findings can guide software developers and UX designers on how to use telemetry data to optimize their applications. We therefore devise specific design suggestions for CAD applications that can be directly applied to improve user performance.

We begin this study by analyzing the preexisting quantitative and empirical approaches to command grouping optimization, as well as the relevant literature on which we are building upon (Section 2). Then, in Section 3, we outline how we obtained the data we used and describe the command optimization methods we propose in terms of their purpose and the algorithms used for training and evaluation. In Section 4 we discuss our results, comparing the methods and mapping them to specific use cases, while also offering advice regarding sampling and interface design techniques. Section 5 concludes the paper with an overview of our findings and pointers to future work.

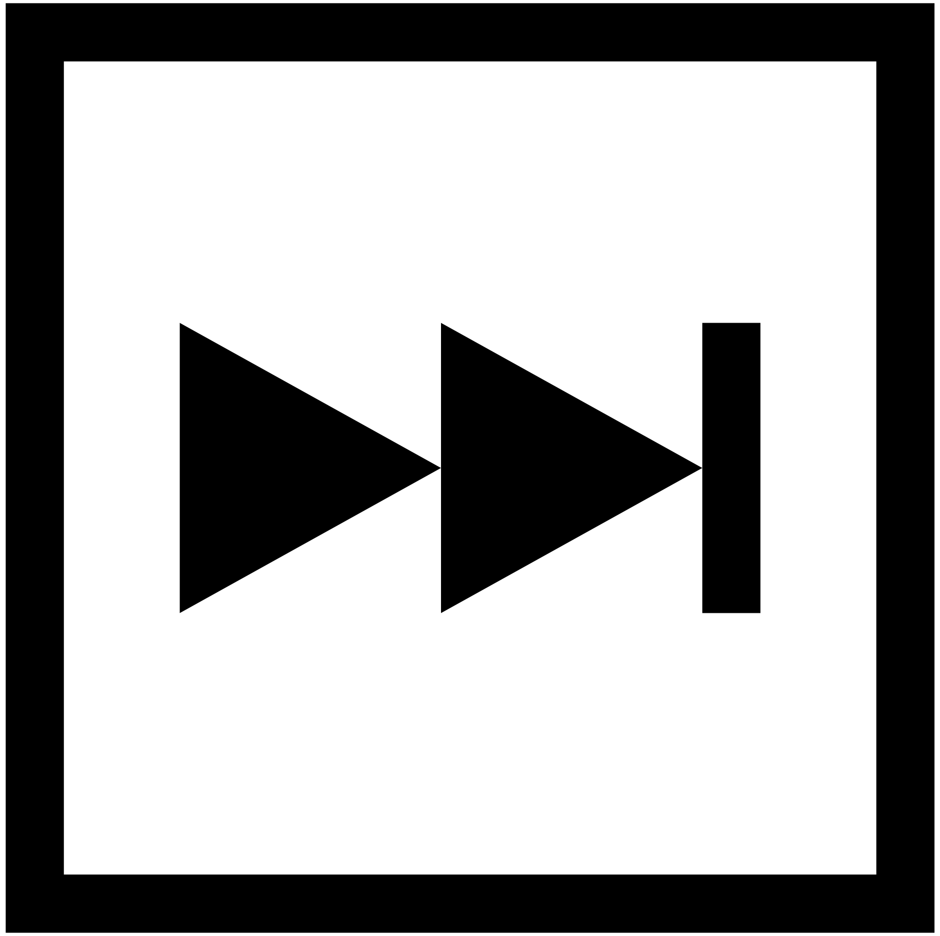
ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183545

```
static void check_joints()
{
        register i, j ;
        double maxx, minx, maxy, miny ;
        int p1, p2, p3 ;
        double d, dx, dy, dz, a, b, c ;
        int surfcount ;
        /* zplane is true when the surface contains the z coordinate.
         * vert is true when a line from the surface is vertical
         */
        int zplane ;
        double zmin ;
        struct surfacestruc *s ;

        for( surfcount = 0 ; surfcount < surfacenum ; surfcount++ ){
                s = surfaces[surfcount] ;
                /* In partial views eliminated surfaces don't obscure */
                if( partial || window )
                        for( i = 0 ; i < s->anglenum ; i++ )
                                if( ! (tag[s->joint[i]] & LINE ) )
                                        goto nextsurface ;
[80 more lines]
```

Assume responsibility for your learning

Fake it till you make it



A Dynamically Linkable Graphics Library

Diomidis D. Spinellis
Department of Computing, Imperial College of Science and Technology, 180 Queens Gate, London SW7 2BZ, U.K.

SUMMARY

The design issues behind the implementation of an efficient and portable graphics library are discussed. A description of its components is given and the constraints leading to dynamic linking are presented. Techniques allowing the transparent dynamic linking of library elements are analysed and two implementations of a system that automatically creates dynamically linkable code are presented. The one implementation is based on traditional UNIX tools and the other on the perl programming language. The two implementations are compared.

KEY WORDS : Dynamic linking   Graphics libraries   Perl

INTRODUCTION

During the design of an interactive graphics pre- and post-processor for a finite element analysis system, the problem of portably displaying the output on a wide variety of graphics output devices was encountered. The program, initially, had to run on IBM-PC class machines running the MS-DOS operating system. In a latter stage it was ported to run under the UNIX operating system on Sun and microVAX workstations.
The program is used to inspect structures represented by wire frames containing hundreds of elements in two distinct phases. First, before input to the finite element analysis program, the wire frame is examined in order to visually verify its form. After the analysis the program is used to inspect the distortions suffered under specific loads. The user may rotate the structure in three dimensions, view specific parts of it, label its joints and members and perform various other operations on it. The interactive nature of the program and the range of machines it was designed to operate on, made its design focus on a fast implementation. The main program consists of about 7000 lines of code written in the C[1] programming language.
MS-DOS does not provide an application graphics interface and the ROM *Basic Input Output System* (BIOS)[2] that is available on these machines does not support devices other than those manufactured by the machine vendor. In addition the functions it provides are minimal. Typical functions could display a character, set a point to a specified colour and set up the

A Dynamically Linkable Graphics Library

Diomidis D. Spinellis
Department of Computing, Imperial College of Science and Technology, 180 Queens Gate, London SW7 2BZ, U.K.

[INTRODUCTION]

During the design of an interactive graphics pre- and post-processor for a finite element analysis system, the problem of portably displaying the output on a wide variety of graphics output devices was encountered. The program, initially, had to run on IBM-PC class machines running the MS-DOS operating system.
In a latter stage it was ported to run under the UNIX operating system on Sun and microVAX workstations.
The program is used to inspect structures represented by wire frames containing hundreds of elements in two distinct phases.

[THE LIBRARY APPROACH]

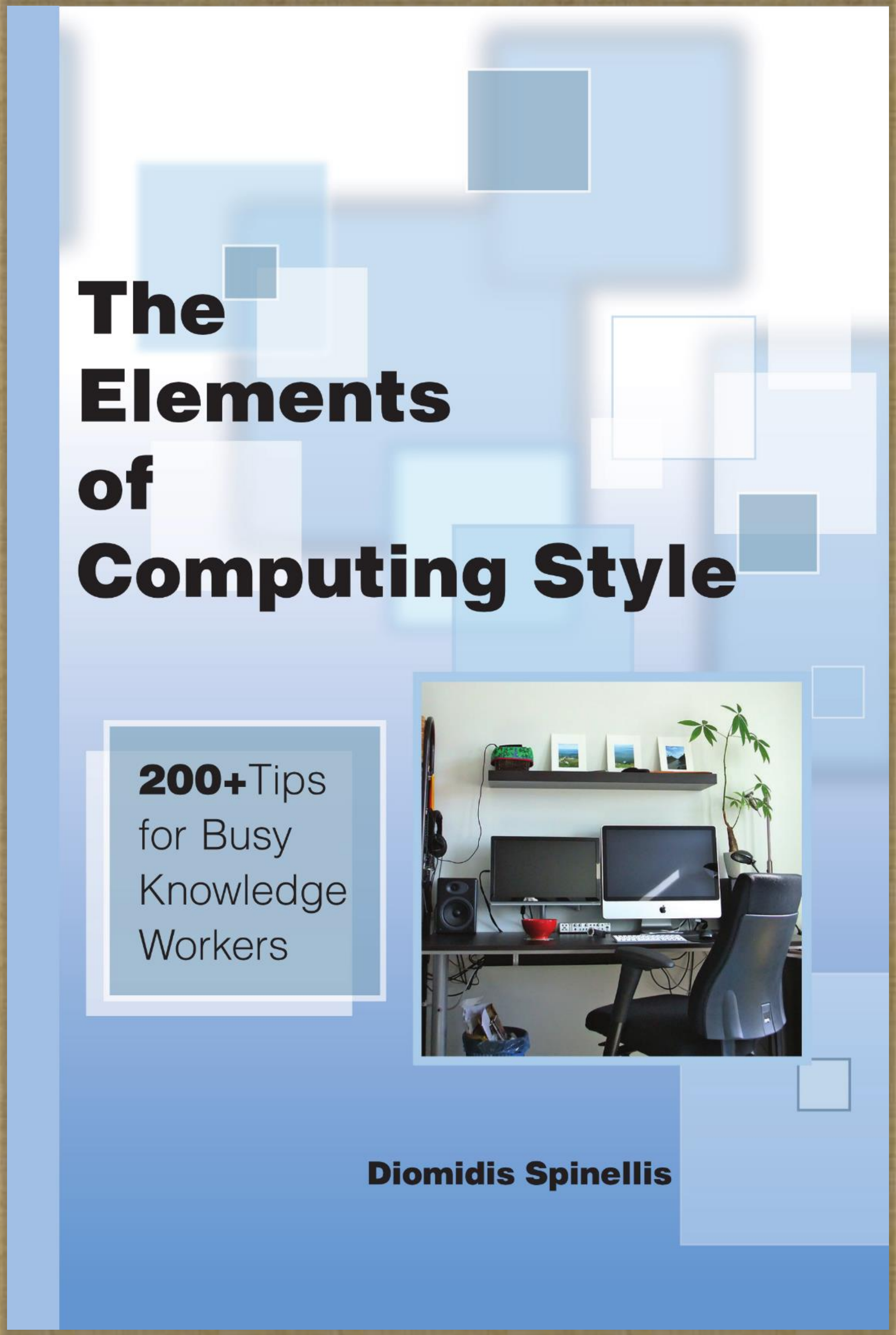
[Functions provided]

[The portable part]

[]

No global variables are defined by the library. The functions provided rely on functions from the device specific library. Th
dichotomy of the two libraries was established gradually and in the early phases of the development functions tended to migrate

Use computer tools
to amplify your potential


The Elements of Computing Style
200+Tips for Busy Knowledge Workers
Diomidis Spinellis

SOFTWARE
PRACTICE & EXPERIENCE
an international journal
published under the Wiley Interscience imprint
by John Wiley & Sons of Chichester New York Brisbane Toronto Singapore

PROFESSOR JOHN A. CAMPBELL
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, England.
Telephone (01) 380-7394
Telex 28722 UCPHYS G

PROFESSOR JOHN A. CAMPBELL
University College London, England.
PROFESSOR DOUGLAS E. COMER
Purdue University, West Lafayette, Indiana, U.S.A.

13 August 1988

Mr. Diomidis Spinellis,
[redacted]

Dear Mr. Spinellis,

I have now received the referees' reports on your manuscript "A dynamically linkable graphics library".

These reports are enclosed.

In view of the referees' comments, I regret to say that I shall not be taking up your offer of this work for publication. With this letter, I am returning all the manuscript material that is presently in my files. Thank you, nevertheless, for considering "Software" as a possible medium for publication.

Yours sincerely,

J.A. Campbell

J.A. Campbell

This paper describes a portable graphics library to run on the IMB-PC and on UNIX workstations. There is an ISO standard, GKS, aimed specifically at this area and implementations exist from a number of suppliers on both types of systems. The reason for not using it appears to be that the user base did not use GKS. As it did not use the new graphics library that was developed either, this seems a weak statement.

The Library produced has a set of primitives which muddle basic graphical output with the attributes that apply to it. It mixes in a random set of window management facilities.

My view is that the paper is unacceptable. I doubt if the work needed to be done. Assuming the need was there, it should have followed accepted graphical methodology.

R E P O R T

The idea of a dynamically linkable graphics library is nice but what is presented here is no solution.

The first part of the paper looks at the design of a graphics library and comes to the same conclusions as most implementators (eg NAG and NCAR) that it is necessary, to aid portability, to design a small device dependent set of primitives. The details given in Tables 1, 2 and 3 are unnecessary.

The author mentions (p 10) that "a portable windowing library is under consideration", it looks as though X-windows has beaten him to it!

The underlying theme of the paper is the need for portability and an easy solution to multiple devices (relinking is considered arduous and multiple executable modules difficult to maintain). The solution provided is operating system dependent (bottom p 12), 'highly compiler specific' (top p 13) (the code presented in this section even has embedded magic numbers!), linker dependent (p 14) and macro assembler dependent (p 19).

This 'solution' can only be described as a 'hack' which has been partially automated using the UNIX tools awk and sed. Apparently even the automation is ugly - 'Its interfaces are unclear and much of the work is done in a non-obvious and highly involved way!'

Finally the comparison with perl promised in the summary is very superficial and is compressed into under a page.

Continued ....

Publish often, be prepared to fail,

… until you succeed


digital
pdp11/20
digital equipment corporation · maynard, massachusetts
ADDRESS REGISTER
RUN
BUS
FETCH
EXEC
DATA
SOURCE
DESTINATION
ADDRESS
OFF
POWER
PANEL LOCK
SWITCH REGISTER
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
Ø
LOAD ADDR
EXAM
CONT
ENABLE
HALT
S-INST
S-CYCLE
START
DEP

Unix PDP-11 Emulator
(As11 & Em11)
User's Guide

Duncan White
Jan-Simon Pendry
Diomidis Spinellis

ABSTRACT

As11 and *em11* form an emulated PDP-11* environment which can be used on UNIX† systems to design and develop simple PDP-11 assembly language programs. The emulated PDP-11 includes 16K bytes of store, a screen, a keyboard, a line printer, and two random access disks.

1. Filename Conventions

Just as Modula-2 uses standard suffixes such as *.def* and *.mod* to identify files as belonging to Modula-2, so the PDP-11 system uses the suffixes *.a11* for an assembly language file, and *.e11* for an emulator input file.

2. The Assembler

As11 is a free-format assembler, accepting all the standard PDP-11 mnemonics and operand types. It is invoked by:

as11 file

The action of the assembler is to translate the single *.a11* file named on the command line [you may omit the .a11 suffix] into the corresponding *.e11* file.

Error messages and *warnings* during assembly are reported on the standard error stream. These are intended to be self-explanatory.

The assembler continues after a warning, but aborts after a fatal error.

* PDP-11 is a registered trademark of DEC
† Unix is a registered trademark of AT+T Bell Labs

2.1. An Example Assembly Language Program

To make the following discussion clearer, here is a simple example of a PDP-11 Assembly Language program.

```
;           An Example PDP-11 Assembly Language Program

; A useful ASCII char, newline
nl          =           12

;           Make space for the stack
            .org        500
stack:
;           then declare the startpoint:
            .org        1000
start:
;           initialise the stack ptr
            mov         #stack,sp

            mov         #greeting, -(sp)
            jsr         pc, scr_mesg
            add         #2, sp
            halt

greeting:
            .byte       nl, nl, "hello there everyone"
            .byte       / isn't it a lovely day ? /, nl, nl
            .byte       0
            .even
```

For the moment, let us not worry about the scr_mesg routine. Accept that it simply displays a null terminated message whose starting address is passed on the stack.

2.2. The Format of Assembly Language Programs

Most of the lines in the above program contain a single PDP-11 instruction. Some lines, however, declare labels, or perform assembler directives [known as *pseudo-ops*].

Any line may be terminated by a comment, introduced by a semi-colon which acts until the end of the current line. A line, if so desired, can contain nothing except a comment.

Between the various constituents of a line, you may place any number of **tabs** and **blanks** which act as separators.

The assembler is not sensitive to upper and lower case.

2.3. Basic Concepts

2.3.1. Symbols

A *symbol* is the assembler equivalent of a Modula-2 *constant*. That is, it is a name which is used to represent a particular numeric value, increasing the readability of a program.

It is an error to *redefine* a symbol.

The assembler accepts indefinite-length symbols, which are sequences of alphanumeric and underscore characters, where the first character is not numeric.

Partner with an experienced mentor

- In the Ads SRE FE team at Google: Mark Bean, Carl Crous, Alexandru-Nicolae Dimitriu, Fede Heinz, Lex Holt, Thomas Hunger, Thomas Koeppe, Jonathan Lange, David Leadbeater, Anthony Lenton, Sven Marnach, Lino Mastrodomenico, Trevor Mattson-Hamilton, Philip Mulcahy, Wolfram Pfeiffer, Martin Stjernholm, Stuart Taylor, Stephen Thorne, Steven Thurgood, and Nicola Worthington.
- At CQS: Theodoros Evgeniou, Vaggelis Kapartzianis, and Nick Nassuphis.
- In the Department of Management Science and Technology at the Athens University of Economics and Business, current and former research and lab associates: Achilleas Anagnostopoulos, Stefanos Androutsellis-Theotokis, Konstantinos Chorianopoulos, Marios Fragkoulis, Vaggelis Giannikas, Georgios Gousios, Stavros Grigorakakis, Vassilios Karakoidas, Maria Kechagia, Christos Lazaris, Dimitris Mitropoulos, Christos Oikonomou, Tushar Sharma, Sofoklis Stouraitis, Konstantinos Stroggylos, Vaso Tangalaki, Stavros Trihias, Vasileios Vlachos, and Giorgos Zouganelis,
- In the General Secretariat for Information Systems at the Greek Ministry of Finance: Costas Balatos, Leonidas Bogiatzis, Paraskevi Chatzimitakou, Christos Coborozos, Yannis Dimas, Dimitris Dimitriadis, Areti Drakaki, Nikolaos Drosos, Krystallia Drystella, Maria Eleftheriadou, Stamatis Ezovalis, Katerina Frantzeskaki, Voula Hamilou, Anna Hondroudaki, Yannis Ioannidis, Christos K. K. Loverdos, Ifigeneia Kalampokidou, Nikos Kalatzis, Lazaros Kaplanoglou, Aggelos Karvounis, Sofia Katri, Xristos Kazis, Dionysis Kefallinos, Isaac Kokkinidis, Georgios Kotsakis, Giorgos Koundourakis, Panagiotis Kranidiotis, Yannis Kyriakopoulos, Odyseas Kyriakopoylos, Georgios Laskaridis, Panagiotis Lazaridis, Nana Leisou, Ioanna Livadioti, Aggeliki Lykoudi, Asimina Manta, Maria Maravelaki, Chara Mavridou, Sofia Mavropoulou, Michail Michalopoulos, Pantelis Nasikas, Thodoros Pagtzis, Angeliki Panayiotaki, Christos Papadoulis, Vasilis Papafotinos, Ioannis Perakis, Kanto Petri, Andreas Pipis, Nicos Psarrakis, Marianthi Psoma, Odyseas Pyrovolakis, Tasos Sagris, Apostolos Schizas, Sophie Sehperides, Marinos Sigalas, George Stamoulis, Antonis Strikis, Andreas Svolos, Charis Theocharis, Adrianos Trigas, Dimitris Tsakiris, Niki Tsouma, Maria Tzafalia, Vasiliki Tzovla, Dimitris Vafiadis, Achilleas Vemos, Ioannis Vlachos, Giannis Zervas, and Thanasis Zervopoulos.
- At the FreeBSD project: John Baldwin, Wilko Bulte, Martin Cracauser, Pawel Jakub Dawidek, Ceri Davies, Brooks Davis, Ruslan Ermilov, Bruce Evans, Brian Fundakowski Feldman, Pedro Giffuni, John-Mark Gurney, Carl Johan Gustavsson, Konrad Jankowski, Poul-Henning Kamp, Kris Kennaway, Giorgos Keramidas, Boris Kovalenko, Max Laier, Nate Lawson, Sam Leffler, Alexander Leidinger, Xin Li, Scott Long, M. Warner Losh, Bruce A. Mah, David Malone, Mark Murray, Simon L. Nielsen, David O'Brien, Johann 'Myrkraverk' Oskarsson, Colin Percival, Alfred Perlstein, Wes Peters, Tom Rhodes, Luigi Rizzo, Larry Rosenman, Jens Schweikhardt, Ken Smith, Dag-Erling Smørgrav, Murray Stokely, Marius Strobl, Ivan Voras, Robert Watson, Peter Wemm, and Garrett Wollman.
- At LH Software and SENA: Katerina Aravantinou, Michalis Belivanakis, Polina Biraki, Dimitris Charamidopoulos, Lili Charamidopoulou, Angelos Charitsis, Giorgos Chatzimichalis, Nikos Christopoulos, Christina Dara, Dejan Dimitrijevic, Fania Dorkofyki, Nikos Doukas, Lefteris Georgalas, Sotiris Gerodianos, Vasilis Giannakos, Christos Gkologiannis, Anthi Kalyvioti, Ersi Karanasou, Antonis Konomos, Isidoros Kouvelas, George Kyriazis, Marina Liapati, Spyros Livieratos, Sofia Livieratou, Panagiotis Louridas, Mairi Mandali, Andreas Massouras, Michalis Mastorantonakis, Natalia Miliou, Spyros Molfetas, Katerina Moutogianni, Dimitris Nellas, Giannis Ntontos, Christos Oikonomou, Nikos Panousis, Vasilis Paparizos, Tasos Papas, Alexandros Pappas, Kantia Printezi, Marios Salteris, Argyro Stamati, Takis Theofanopoulos, Dimitris Tolis, Froso Topali, Takis Tragakis, Savvas Triantafyllou, Periklis Tsahageas, Nikos Tsagkaris, Apostolis Tsigkros, Giorgos Tzamalis, and Giannis Vlachogiannis.
- At the European Computer Industry Research Center (ECRC): Mireille Ducassé, Anna-Maria Emde, Alexander Herold, Paul Martin, and Dave Morton.
- At Imperial College London in the Department of Computer Science: Vasilis Capoyleas, Mark Dawson, Sophia Drossopoulou, Kostis Dryllerakis, Dave Edmondson, Susan Eisenbach, Filippos Frangulis Anastasios Hadjicocolis, Paul Kelly, Stephen J. Lacey, Phil Male, Lee M. J. McLoughlin, Stuart McRobert, Mixalis Melachrinidis, Jan-Simon Pendry, Mark Taylor, Periklis Tsahageas, and Duncan White.
- In the Computer Science Research Group (CSRG) at the University of California at Berkeley: Keith Bostic.
- At Pouliadis & Associates: Alexis Anastasiou, Constantine Dokolas, Noel Koutlis, Dimitrios Krassopoulos, George Kyriazis, Giannis Marakis, and Athanasios Pouliadis.
- At diverse meetings and occasions: Yiorgos Adamopoulos, Dimitris Andreadis, Yannis Corovesis, Alexander Couloumbis, John Ioannidis, Dimitris Kalogeras, Panagiotis Kanavos, Theodoros Karounos, Faidon Liampotis, Elias Papavassilopoulos, Vassilis Prevelakis, Stelios Sartzetakis, Achilles Voliotis, and Alexios Zavras.

Document everything you implement

TM

open source

Open Source Software Contributions

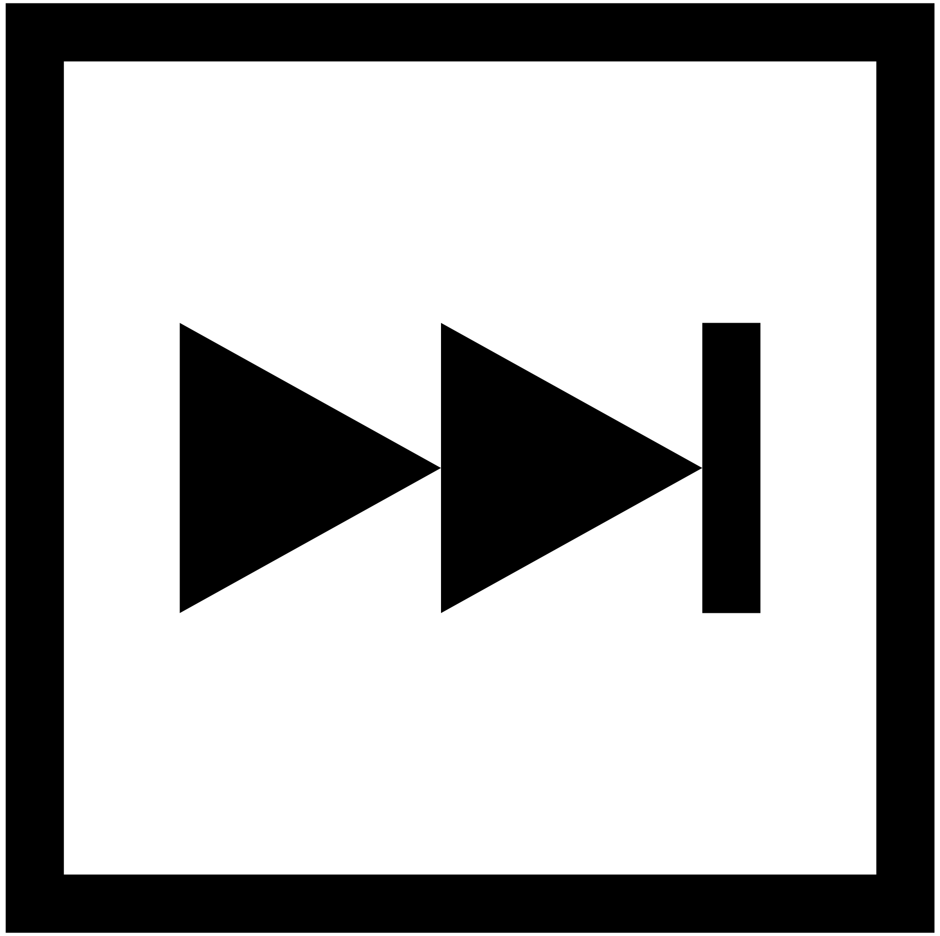
- Port Perl to MS-DOS
- Re-implement sed(1) for BSD Unix (now in macOS, FreeBSD)
- Trace tool for MS-DOS
- RCS utility functions
- zopen(3) compression interface
- NetPBM tools

www.spinellis.gr/sw/

Gains

- Fought boredom by working on challenging problems
- Honed coding skills
- Learned how to read/implement standards (POSIX)
- Networking (people)
- Established (a tiny) reputation

Contribute to / initiate
open source software projects



Programming Paradigms as Object Classes:
A Structuring Mechanism for
Multiparadigm Programming

by
Diomidis D. Spinellis

May 1993

A thesis submitted for the degree of
Doctor of Philosophy of the University of London
and for the Diploma of Membership of Imperial College

Department of Computing
Imperial College of Science, Technology and Medicine
University of London
London SW7 2BZ
United Kingdom

Chapter 2

Related Work: Multiparadigm Programming

In this chapter we begin our exploration of the area of multiparadigm programming by examining the work that has been done up to now. Research in the area of multiparadigm programming can be divided in three different areas:

1. Programming paradigms: work in this area examines the notion of programming paradigms, their relationship to language design, and their effect on the software production environment.

2. Multiparadigm languages: during the literature research for this thesis we found more than 90 languages supporting more than one programming paradigm. Although not all languages were explicitly directed towards multiparadigm programming *per se* we believe that there were lessons to be learned from their collective study.

3. Multiparadigm programming frameworks: some researchers have come up with suitable abstractions and systems that support multiparadigm programming in general without targeting specific programming paradigms. Again in this case at least one of the systems covered was not the result of explicit multiparadigm research effort although it supports programming in multiple programming paradigms.

Our approach is geared towards producing a multiparadigm design methodology, a prototype system based on that methodology, and multiparadigm programming environment built upon that system. Therefore, the two last research areas are directly relevant to our research. We also examine the first area, because we believe that the notion of a programming paradigm is central to the theme of this thesis. Mixed language programming environments which only deal with languages based on a single paradigm (such as [Ein84]), and generic concurrent, distributed, heterogeneous systems, and module interface languages [Tic92] that could potentialy be used as multiparadigm frameworks (such as [Bea92]) are not examined. A thorough survey of distributed system languages can be found in [BST89], and of concurrent logic programming languages in [Sha89]; a set of articles on concurrent object-oriented programming can be found in [CAC93], and a survey of specific concurrent Smalltalk implementations in

Name	References	Implementation
ALF	[Han90a, Han91]	WAM extension
ALICE	[CST87]	Meta-interpreter on top of 3-Lisp
Applog	[Coh86]	Interpreter written in Prolog
Bon87	[Bon87]	Meta-interpreter on Scheme
EqL, E	[JSG86, JS86]	Language
FGL+LV	[Lin85]	Extension to the graph reduction language FGL
FPL	[BDL82]	Extension to TEL functional language
Fresh	[Smo86]	Extensions to functional
Funlog	[SY86]	Interpreter implemented in Prolog
HASL	[Abr86]	Implemented in C-Prolog
HCPRVR	[Che80]	Implemented on top of Lisp
HHT82	[HHT82]	Extension to Prolog
Han90	[Han90b]	Theoretical framework
Id Nouveau	[JP91]	Operational Semantics
LML	[BMPT90]	Extension to functional
LOGLISP	[RS82]	Extension to Lisp
Leaf	[BBLM86, BBLM84]	Plan for hardware implementation
Nar85	[Nar85]	Technique
Qute	[SS86a]	Implemented in Prolog as a translator to Prolog
SProlog	[Smo84]	Implemented on top of Prolog
SchemeLog	[Bon91]	Meta-interpreter on Scheme
TABLOG	[MMW84, MMW86]	Language implemented in Lisp
Term Desc.	[Nak85]	Prolog extension
YS86	[YS86]	Semantic framework

Table 2.4: Implementations combining the functional and logic paradigms

Name	Characteristics										Control
	BR	DT	f	λ	$\|$	R	RT	∪	$\overline{DT}$	$\overline{RT}$	
ALF	√	√				√	√	√	√		SLD, narrowing
ALICE		√	√	√		√	√	√	√		SLD, FR
Applog		√	√	√		√	√	√	√		SLD, FR
Bon87			√	√		√		√	√		FR, SLD
EqL, E						√	√	√	√		*
FGL+LV		√	√	√			√	√			FR
FPL		√				√		√			*
Fresh		√	√	√			√	√	√		FR, SLD
Funlog		√	√	√		√	√	√	√		FR
HASL		√	√	√			√	√			FR
HCPRVR	√	√	√	√		√	√	√	√		SLD
HHT82	√	√				√	√	√	√		*
Han90	√	√	√	√		√	√	√	√		SLD, narrowing
Id Nouveau	√	√	√				√	√		√	*
LML	√	√	√			√	√	√	√		FR
LOGLISP		√	√	√			√	√	√		FR
Leaf		√				√	√	√	√		*
Nar85	√					√	√	√	√		SLD, FR
Qute		√	√	√	√		√	√			FR
SProlog	√	√				√	√	√	√		SLD
SchemeLog			√	√		√		√	√		*
TABLOG	√					√	√	√	√		*
Term Desc.	√		√			√	√	√	√		SLD
YS86		√				√	√	√	√		*

Table 2.5: Characteristics of functional and logic paradigm combinations

Name	References	Implementation
DSM	[Rum87]	Extension to C
Echidna	[HSS+92]	Implemented on top of Lisp
Educe	[Boc86]	Prolog DBMS
Enhanced C	[Kat83]	Compiler producing C
Fooplog	[GM87]	Language
Icon	[OG87, Gri84, GG83]	Language
KE88	[KE88]	LOOPS and Prolog
Kaleidoscope	[FBB92]	Language interpreter
Lex	[Les75]	C preprocessor
ML-Lex	[AMT89]	C preprocessor
ML-Yacc	[TA90]	ML preprocessor
SB86	[SB86]	Meta-interpreters on Prolog
SPOOL	[FiH86, Yok86]	Implemented on top of Prolog VM
Uniform	[Kah86]	Implemented on top of Lisp
Yacc	[Joh75]	C preprocessor

Table 2.24: Implementations combining the various paradigms

Name	Characteristics												Control
	BR	DT	f	IN	λ	MI	O	R	RT	∪	$\overline{DT}$	$\overline{RT}$	
DSM	√			√			√	√				√	
Echidna				√			√		√	√	√		*
Educe	√							√	√	√	√		SLD
Enhanced C	√											√	X
Fooplog		√	√	√	√		√	√	√	√	√		*
Icon	√								√		√	√	X
KE88	√	√	√	√	√		√	√	√	√	√		FR, SLD, X
Kaleidoscope	√	√		√		√	√					√	*, X
Lex	√											√	X, *
ML-Lex		√	√		√				√			√	*, FR
ML-Yacc		√	√		√				√			√	*, FR
SB86	√							√	√	√	√		*
SPOOL	√			√			√	√	√	√	√	√	SLD
Uniform		√	√		√	√	√		√	√			*
Yacc	√											√	X, *

Table 2.25: Characteristics of various paradigm combinations

Name	Characteristics							Control
	BR	$\|$	R	RT	∪	$\overline{DT}$	$\overline{RT}$	
2.PAK	√				√	√	√	*
C with Rule Extensions	√		√	√	√	√	√	*, X
Leda	√		√	√	√	√	√	SLD, X
Logicon	√		√	√	√	√	√	X, SLD
Modula-Prolog	√		√	√	√	√	√	SLD, X
PIC	√		√	√	√	√	√	SLD, X
Paslog	√		√	√	√	√	√	SLD, X
Planlog	√		√	√	√	√	√	SLD
Predicate Logic in APL	√		√	√	√	√	√	SLD, X
Strand	√	√	√	√	√	√	√	SLD

Table 2.7: Characteristics of imperative and logic paradigm combinations

2.PAK [Mel75] Block structured language offering user-defined pattern matching and backtracking.

C with Rule Extensions [MS90] Based on the C programming language [KR78] with an extended syntax, a richer set of data types, a flexible input/output system and a forward chaining [Ric83, p. 56] execution strategy.

Leda [Bud91] Language with syntax similar to that of Pascal, with an additional code abstraction facility, the *relation*. The data-space for all entities contains the *undefined* value. Relations are coded as Prolog rules, and allow backtracking.

Language	Unification	Backtracking	I/O extensions
Modula-Prolog	√	√	√
Planlog	√	√	
Predicates in APL	√	√	
Paslog	Explicit	√	√
C with Rules	?	√	√
PIC	√	√	
Leda	1 level	√	

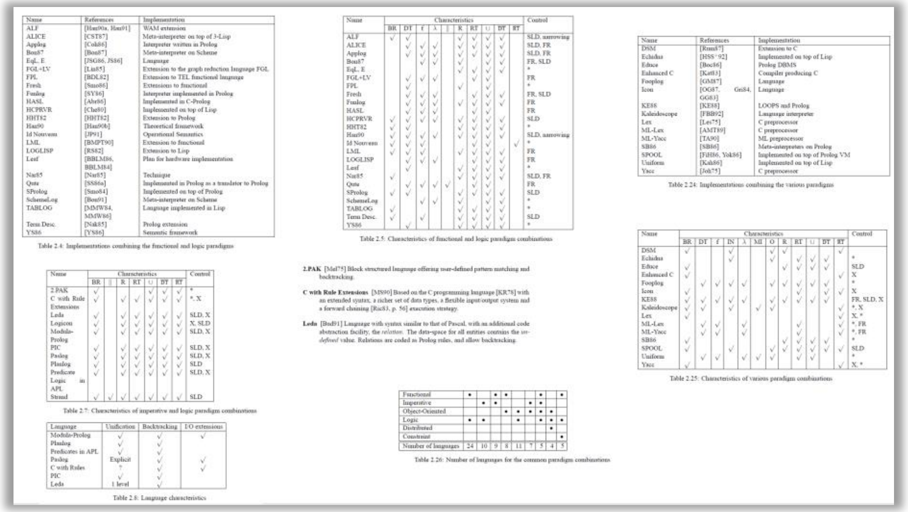
Table 2.8: Language characteristics

Functional	•		•	•			•		•
Imperative		•	•			•	•		
Object-Oriented				•	•	•	•	•	
Logic	•	•			•		•	•	•
Distributed								•	
Constraint									•
Number of languages	24	10	9	8	11	7	5	4	5

Table 2.26: Number of languages for the common paradigm combinations

```
%N Modula-Prolog
%R Mul86
%C UN RL RT nDT BR (SLD,EX) BR nRT
%I Run-time library for Modula-2
%P Logic Imperative
%D
The facilities of a Prolog interpreter are provided to a
Modula-2 programmer through a library.  Predicates, that can be
called from the Prolog interpreter, are written in Modula-2.
The library includes term handling procedures.
```

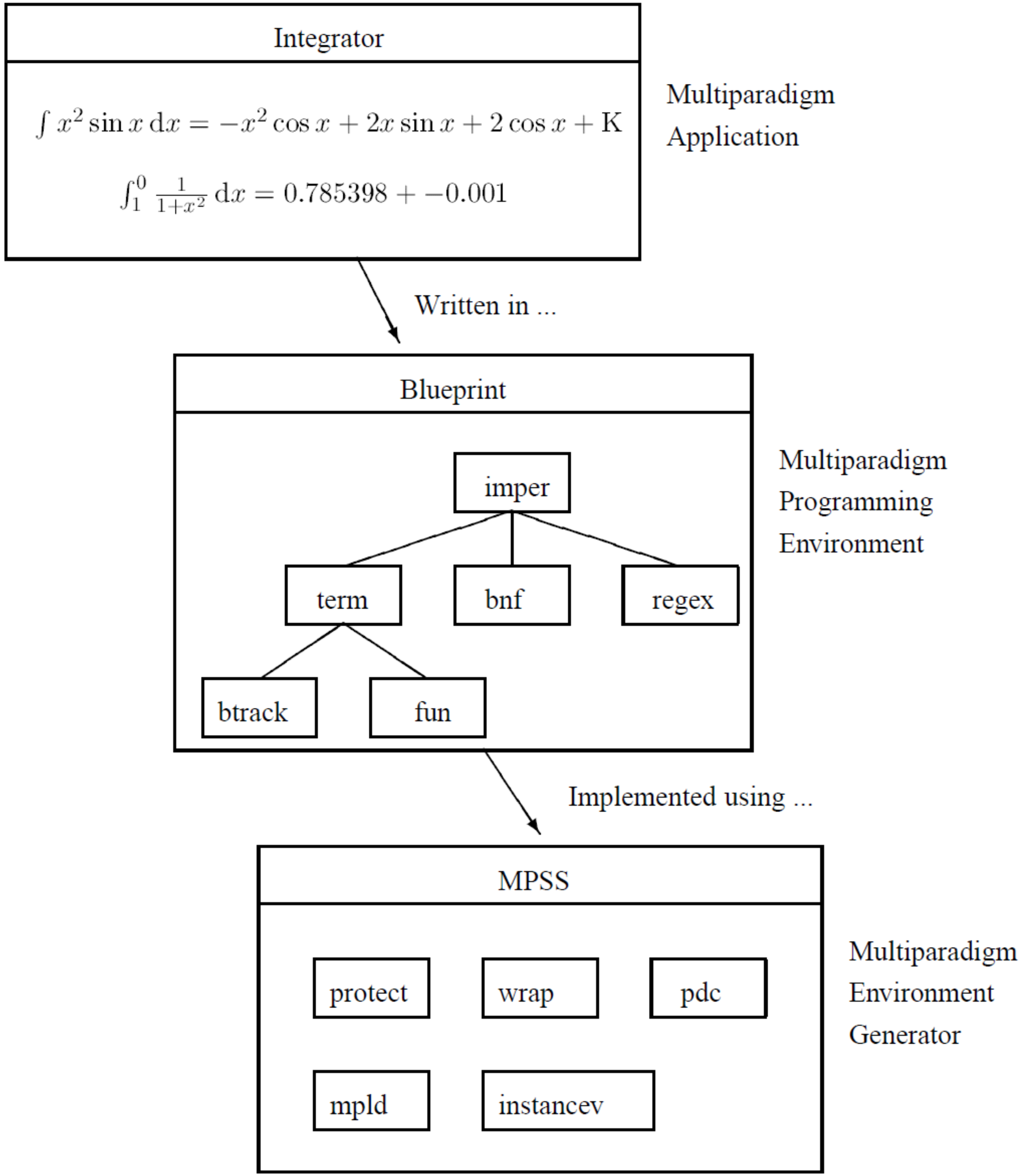
Program	Implementation	Lines
chars	Sh	13
chartabl	Perl	59
dbgrep	Perl	27
desclist	Perl	17
imptable	Perl	38
linesort	Perl	19
llinesor	Perl	26
maketext	Perl	117
pars	Sh	13
partable	Perl	27
Total	Perl	697



Automate

data collection, analysis, presentation

Prefer building plumbing to porcelain
Write small tools that do one thing well



Function	Paradigm	Module	Lines
Symbolic integration	btrack	sint.pb	127
Lexical analysis	regex	scan.pl	47
Expression parsing	bnf	parse.py	76
Numeric integration	fun	aint.pf	75
Interfacing	term	ui.pt	131
Graph creation	imper	main.c	51
Total	blueprint		507

Paradigm	PDF	*imper*	*bnf*	*regex*	*term*	*fun*	Total	%
imper	43						70	1.6
term	70	1192	119	84	666		2269	53.3
btrack	60				316		554	13.0
fun	140		305	59	237	43	840	19.7
bnf	95						121	4.3
regex	379						405	9.5
Total	787	1192	424	143	1219	43	4259	100.0
%	18.5	28.0	10.0	3.4	28.6	1.0	100.0	

Play on your strengths

[12] T. Kaida, S. Uehara, and K. Imamura, "Computation of the k-error linear complexity of binary sequences with period 2^n," in *Concurrency and Parallelism, Programming, Networking (Lecture Notes in Computer Science)*, J. Jaffar and R. H. C. Yap, Eds. Berlin, Germany: Springer-Verlag, 1996, vol. 1179, pp. 182–191.
[13] ——, "An algorithm for the k-error linear complexity of sequences over $GF(p^m)$ with period p^n, p a prime," *Inform. Comput.*, vol. 151, pp. 134–147, 1999.
[14] K. Kurosawa, F. Sato, T. Sakata, and W. Kishimoto, "A relationship between linear complexity and k-error linear complexity," *IEEE Trans. Inform. Theory*, vol. 46, pp. 694–698, Mar. 2000.
[15] J. L. Massey, "Shift register synthesis and BCH decoding," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 122–127, Jan. 1969.
[16] J. L. Massey, D. J. Costello, and J. Justesen, "Polynomial weights and code constructions," *IEEE Trans. Inform. Theory*, vol. IT-19, pp. 101–110, Jan. 1973.
[17] J. L. Massey and S. Serconek, "Linear complexity of periodic sequences: A general theory," in *Advances in Cryptology—CRYPTO'96 (Lecture Notes in Computer Science)*, N. Koblitz, Ed. Berlin, Germany: Springer-Verlag, 1996, vol. 1109, pp. 358–371.
[18] H. Niederreiter, "Some computable complexity measures for binary sequences," in *Sequences and Their Applications—Proc. SETA98*, C. Ding, T. Helleseth, and H. Niederreiter, Eds. Berlin, Germany: Springer-Verlag, 1999, pp. 67–78.
[19] K. G. Paterson, "Perfect maps," *IEEE Trans. Inform. Theory*, vol. 40, pp. 743–753, May 1994.
[20] M. J. B. Robshaw, "On evaluating the linear complexity of a sequence of least period 2^n," *Des., Codes Cryptogr.*, vol. 4, no. 3, pp. 263–269, 1994.
[21] R. A. Rueppel, *Analysis and Design of Stream Ciphers*. Berlin, Germany: Springer-Verlag, 1986.
[22] M. Stamp and C. F. Martin, "An algorithm for the k-error linear complexity of binary sequences of period 2^n," *IEEE Trans. Inform. Theory*, vol. 39, pp. 1398–1401, July 1993.

Reliable Identification of Bounded-Length Viruses Is NP-Complete

Diomidis Spinellis, *Member, IEEE*

Abstract—A virus is a program that replicates itself by copying its code [to] other files. A common virus-protection mechanism involves scanning [fil]es to detect code patterns of known viruses. We prove that the problem [of] reliably identifying a bounded-length mutating virus is NP-complete by [sh]owing that a virus detector for a certain virus strain can be used to solve [th]e satisfiability problem. The implication of this result is that virus iden[tif]ication methods will be facing increasing strain as virus mutation and [ho]sting strategies mature, and that different protection methods should be [de]veloped and employed.

Index Terms—Buffer overflow, complexity, detection, identification, [m]utation, NP-complete, security, virus.

I. Introduction

One often-used defense against computer viruses is the execution of [an] *anti-virus* program that detects and cleans programs that appear to be infected. Virus writers respond to this defense by trying to thwart anti-virus software through targeted attacks, mutations, or social engineering. Mutating viruses are a particularly insidious threat, because detection algorithms need to be constantly updated and to spend increasing processing time to identify new mutation types. The question of whether complexity theory is on the side of virus writers or the protection vendors could have important practical implications. In this correspondence we will prove that there exist realistic viruses whose reliable detection is of NP-complete complexity [1] and that, therefore, the general problem of reliable bounded-length virus identification is NP-complete.

Manuscript received June 14, 2002; revised August 22, 2002. This work was [su]pported by the IST Project mEXPRESS (IST-2001-33432), which is funded in [pa]rt by the European Commission.
The author is with the Department of Management Science and Technology, [At]hens University of Economics and Business, GR 104 34 Athens, Greece [(e-]mail: dds@aueb.gr).
Communicated by N. I. Koblitz, Associate Editor for Complexity and Cryp[togr]aphy.
Digital Object Identifier 10.1109/TIT.2002.806137

II. Viral Software

Intentionally created malicious software [2]—often termed *malware*—is typically classified into Trojan horses, viruses, and worms [3]. A Trojan horse is a program that exploits the rights of its user to perform an action its user does not intend, a virus is a Trojan horse that replicates itself by copying its code into other program files [4], while a worm is an independently running program that replicates through a network exploiting security weaknesses to invade other computers.

A number of virus-prevention and -detection methods have been proposed and are commonly implemented [5], [6]. Reference [7] contains an annotated bibliography of malware analysis and detection papers. Prevention methods involve limiting the flow of information between programs through the use of appropriate hardware and software protection domains, coupled with self-defense mechanisms, instrumentation, and fault-tolerance. Since the above methods will typically interfere with many legitimate operations (such as the installation of new software or the correction of an existing version) they need to be coordinated through carefully designed and executed security procedures. Unfortunately, current practice in system administration often renders these methods useless. A large percentage of users typically administer their personal workstations on their own, in most cases exercising the full rights of the system administrator, without sufficient training and diligence.

Therefore, as a secondary line of defense, detection measures are often employed to locate virus instances and infections. Two often used detection measures involve either the comparison of the system's programs against known-good versions (typically condensed in the form of a checksum or a cryptographically secure signature [8]) or the comparison of files against patterns of known viruses. Since the first method depends on a known-clean system and cannot be used to check software of unknown origin, the second, scanning, method is the one most commonly employed. A number of virus software vendors provide virus-scanning software that can search new and existing system files for patterns of all known viruses. The vendors regularly distribute updated versions of the virus patterns to keep the virus detection process up to date.

Virus writers however, have developed a series of countermeasures. Even early academic examples of viral code were cleverly engineered to hinder the detection of the virus [9]. Since the actual task of writing a virus is relatively simple [10], [11], modern virus code focuses on employing platform independence, stealth, effective replication, and detection countermeasures. Three pattern-matching detection countermeasures typically employed are the *encryption* of the virus body with a variable cryptographic key, the *polymorphic* generation of the decryption routine using equivalent code instructions, and, more recently, the *metamorphic* generation of the whole virus body through the addition, removal, permutation, and substitution of code sequences. Viruses that employ these techniques, such as W32/Simile [12], can be very difficult to identify. In the following section we establish that reliably detecting instances of such viruses is a problem of NP-complete complexity.

III. Identification Complexity

A virus is formally defined [13] by reference to a Turing Machine [14]

$$M: (S_M, I_M, O_M: S_M \times I_M \to I_M, \quad N_M: S_M \times I_M \to S_M, \quad D_M: S_M \times I_M \to d) \tag{1}$$

with a given set of states S_M, set of input symbols I_M, and maps (O_M, N_M, D_M) that, based on its current state $s \in S_M$ and input symbol $i \in I_M$ coming from a semi-infinite tape, determine: the output symbol $o \in I_M$ to write on the tape, the machine's next state $s' \in S_M$, and the tape's motion $d \in \{-1, 0, 1\}$.

Given the machine M, a sequence of tape symbols v: $v_i \in I_M$ can be considered as a virus *for that machine* iff processing the sequence v at time (sequence point) t implies that at a future time point t' a sequence v'—not overlapping with v—will exist on the tape, and that the sequence v' will have been written by M at a point t'' lying between t and t'

$$\begin{array}{ll} \forall \Box_M \, \forall t \, \forall j: & \\ S_M(t) = S_{M_0} & \wedge \\ P_M(t) = j & \wedge \\ \{\Box_M(t, j) \cdots \Box_M(t, j + |v| - 1)\} = v & \Rightarrow \\ \exists v' \, \exists j' \, \exists t' \, \exists t'': & \\ t < t'' < t' & \wedge \\ \{j' \cdots j' + |v'|\} \cap \{j \cdots j + |v|\} = \emptyset & \wedge \\ \{\Box_M(t', j') \cdots \Box_M(t', j' + |v'| - 1)\} = v' & \wedge \\ P_M(t'') \in \{j' \cdots j' + |v'| - 1\} & \end{array} \tag{2}$$

where

- $t \in \mathbb{N}$ stands for the number of times the machine has performed its basic operation—"move";
- $P_M(t) \in \mathbb{N}$ represents the machine's tape cell position number at time t;
- S_{M_0} is the machine's initial state;
- $\Box_M(t, c) \in I_M$ represents the content of cell c at time t.

Note that in the original seminal reference [13], the above virus definition appears in the context of a viral set $VS = (M, V)$: a tuple consisting of a Turing Machine M and a set of symbol sequences V: $v, v' \in V$. From the virus definition it is clear that the notion of a virus is intimately associated with its interpretation in a given context—environment. It has been shown [13] that "any self-replicating tape symbol sequence is a one element VS, that there are countably infinite VS's and non-VS's, that machines exist for which all tape sequences are viruses and for which no tape sequences are viruses, and that any finite sequence of tape symbols is a virus with respect to some machine." The same reference also proves that in the general case determining whether a given tuple (M, X): $X_i \in I_M$ is viral is an undecidable problem (i.e., that there is no algorithm that can reliably detect all viruses) through a reasoning similar to that employed to prove the undecidability of the Halting Problem [14]. Other researchers have shown that there are also virus types (viruses that evolve to contain an instance of the virus detection program) that cannot be detected by any error-free algorithm [15].

As is often the case, current practice differs from theory. Typical pattern-based virus-detection software scans a (relatively) known environment (processor architecture and operating system) to locate one of several (thousands in practice) *a priori* known viruses. In the following paragraphs, we will therefore establish the complexity of the more restricted problem of locating an instance of a known finite-length virus in a given execution environment. For instance, the virus programs we provide in the appendixes are only viral in the context of compila-

tion and execution following the rules of Haskell and ANSI C/POSIX, respectively.

The complexity of detecting a known fixed virus pattern of length M in a program of length N is harnessed by the Boyer–Moore string-searching algorithm [16] which never uses more than $N + M$ steps and under many circumstances (a small pattern and a large alphabet) can use about N/M steps. Unfortunately, as we saw in the previous section, virus writers are seldom thus accommodating; fixed search patterns are not any more a viable virus detection method. We will prove that the problem of reliably identifying a bounded-length mutating virus is NP-complete. Our proof is based on showing that a virus detector D for a certain virus strain V can be used to solve the satisfiability problem, which is known to be NP-complete [17]. (This approach works in the same way for any similar NP-complete problem; the satisfiability of the problem we are examining is not a special case.)

The virus V is a mutating self-replicating program. We assume that the virus detector D can reliably determine in P-time whether a given candidate program C is a mutation of the virus V. We will use the virus detector as an oracle for determining the satisfiability of an N-term Boolean formula S of the following type:

$$\begin{array}{rll} S = & (x_{a_{1,1}} \vee x_{a_{1,2}} \vee \neg x_{a_{1,3}} \vee \cdots) & \wedge \\ & (\neg x_{a_{2,1}} \vee x_{a_{2,2}} \vee \neg x_{a_{2,3}} \vee \cdots) & \wedge \\ & (x_{a_{3,1}} \cdots) & \wedge \\ & \vdots & \end{array} \tag{3}$$

$$0 \le a_{i,j} < N \tag{4}$$

and thereby show that a P-time reliable virus detector is equivalent to a P-time solution to the satisfiability problem.

We will use the satisfiability formula S to create a virus archetype A and a possible instance of a virus phenotype P. The virus is a triple

$$(f, s, c) \tag{5}$$

where

- f is the virus processing and replication function;
- s is a Boolean value indicating whether an instance of the virus has found a solution to S;
- c is an integer encoding the candidate values for S.

The function f maps a triple (f, s, c) into a new triple (f, s', c') and is defined as follows:

$$\lambda(f, s, c).\left(f, s \vee S, \text{ if } c = 2^N \text{ then } c \text{ else } c + 1\right). \tag{6}$$

Each S term x_n is calculated from c through the expression

$$\left\lfloor \frac{c}{2^n} \right\rfloor \bmod 2 = 1. \tag{7}$$

A new generation of the virus is generated by applying f to the current generation.

Expressed in words, each new virus generation

2) evaluates S by extracting successive Boolean value combinations from c;
3) increments c until it reaches 2^N;
4) passes the result of the S evaluation to the next generation.

We can now ask D whether the virus archetype A

$$(f, \text{ False}, 0) \tag{8}$$

will ever result in a virus mutation phenotype P

$$(f, \text{ True}, 2^N) \tag{9}$$

that is, whether one of the virus mutations will satisfy S.

We have thus proven that a reliable virus detector D operating in P-time can be used as a P-time satisfiability oracle and that therefore reliable virus detection is NP-complete.

As an example for the operation of the virus consider the satisfiability of the formula S

$$(x_0 \vee x_1) \wedge \neg x_0. \tag{10}$$

The virus replication function f—after omitting for simplicity of expression the conditional, which only serves to limit the number of virus mutations—will be

$$\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) \tag{11}$$

the corresponding archetype A

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1), = \mathrm{F}, 0) \tag{12}$$

and the phenotype P indicating satisfiability

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1), = \mathrm{T}, 4). \tag{13}$$

This particular virus will generate a mutation P—and thereby indicate that S is satisfiable—in four generations through the following sequence:

$$\begin{array}{ll} ffffA & \to \\ fff\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) & \\ \quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 0) & \xrightarrow{\beta} \\ fff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F} \vee (\mathrm{F} \vee \mathrm{F}) \wedge \neg \mathrm{F}, 0 + 1) & \xrightarrow{\delta} \\ fff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 1) & \to \\ ff\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) & \\ \quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 1) & \xrightarrow{\beta} \\ ff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F} \vee (\mathrm{T} \vee \mathrm{F}) \wedge \neg \mathrm{T}, 1 + 1) & \xrightarrow{\delta} \\ ff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 2) & \to \\ f\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) & \\ \quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 2) & \xrightarrow{\beta} \\ f(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F} \vee (\mathrm{F} \vee \mathrm{T}) \wedge \neg \mathrm{F}, 2 + 1) & \xrightarrow{\delta} \\ f(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T}, 3) & \to \\ \lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) & \\ \quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T}, 3) & \xrightarrow{\beta} \\ (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T} \vee (\mathrm{T} \vee \mathrm{T}) \wedge \neg \mathrm{T}, 3 + 1) & \xrightarrow{\delta} \\ (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T}, 4) \equiv P. & \end{array} \tag{14}$$

IV. Implications

The creation of metamorphic viruses is a relatively recent phenomenon that places a considerable threat on our information system infrastructures. From a theoretical point of view, the viruses bear remarkable similarities to the virus we have examined and the examples depicted in the appendixes to this correspondence. Virus detection programs, however, need not be 100% correct. Users can tolerate the (typically remote) possibility of some "noise" (false positives), because in practice it is quite rare for a nonviral program to match the detection pattern of a known virus. As an example, a virus detector that detected the viruses in this correspondence and also detected as a virus all triplets of the form (f, s, n): $\forall s \forall n$ (even cases where f is a nonsatisfiable formula and s is true) would probably be tolerated as a functioning "good-enough" virus detector, although strictly speaking it detects some false positives. Such a virus detector can be implemented to terminate in linear time and is not NP-complete.

Thus, given the difference between the theoretically perfect detection (which is in the general case undecidable, and for known viruses, as we demonstrated, NP-complete) and the practically sufficient identification (which is the basis for a number of working virus scanner implementations) two questions arise.

2) How can the notion of "sufficiently good detection" be formalized in information theory terms?
3) Can the increasing ability of metamorphic viruses to mutate move the identification threshold currently used by virus detection programs to the point where either numerous legitimate data sequences are falsely detected as viruses, or real viruses fail to be detected?

An interesting phenomenon affecting the above topics concerns the currently permeable boundary between code and data. *Buffer overflow attacks* [18] are based on data that overwrites a carelessly written program's return stack address lying at the end of a data buffer to cause the program to execute part of that data. This renders all data files (documents, images, music, video—many of them highly compressed) stored on a computer into potential carriers of viral code, and dramatically increases the data a virus detector has to scan and discriminate. Few viruses currently propagate through buffer overflows; these weaknesses have traditionally been mainly exploited by worms and Trojan horses [19]. However, once such viruses are released, the current virus detection approach will come under increasing strain, faced with the short pattern vectors of mutating viruses and orders of magnitude more data to scan; as an example a 18–Gbyte disk filled with MP3 files is likely to contain any 4-byte (virus) pattern. In the medium and long term, hardening our security defenses and developing software, procedures, and work practices that will stem the spread of malware seem to be the only reasonable alternatives.

Appendix I
Virus Code in Haskell

The following code defines the virus replication function and the respective archetype and candidate phenotype, for determining the satisfiability of the expression

$$(x_0 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_5) \wedge (x_2). \tag{15}$$

The satisfiability function candidate values are encoded using Haskell's arbitrary precision integers.

```
module Virus where
replicate :: (replicate, Bool, Integer)->
  (replicate, Bool, Integer)
replicate (v, b, i) = (v, b||
    (((bit 0 i) || (bit 3 i) ||
       not (bit 4 i)) &&
     (not (bit 1 i) || (bit 5 i)) &&
     ((bit 2 i)))
    , if i == 64 then i else i + 1)

-- Extract bit b out of the Integer n
bit :: Integer -> Integer -> Bool
bit b n = n `div` (2 ^ b) `rem` 2 == 1

virus_archetype = (replicate, False, 0)
virus_phenotype = (replicate, True, 64)
```

Appendix II
Virus Code in C

The following code is the virus archetype, again for determining the satisfiability of (15). The satisfiability function candidate values are encoded as elements of the array x.

```
#include <stdio.h>
#include <ctype.h>

/* Number of variables to satisfy */
#define N 6
int x[N] = {
0, 0, 0, 0, 0, 0,
};
void
advance(void)
{
  int i, j;
  for (i = 0; i < N; i++)
    if (x[i] == 0) {
      for (j = 0; j < i; j++)
        x[j] = 0;
      x[i] = 1;
      return;
    }
}
void
print_vector(FILE *f)
{
  int i;
  for (i = 0; i < N; i++)
    fprintf(f, "%c,", x[i] ? '1' : '0');
  fputc('\n', f);
}
main()
{
  char buff[1024];
  FILE* fi = fopen(__FILE__, "r");
  FILE* fo = fopen("new" __FILE__,"w");

  if ((x[0] || x[3] || !x[4]) &&
     (!x[1] || x[5]) && (x[2]))
    fprintf(fo, "/* Satisfied */\n");
  advance();
  while (fgets(buff, sizeof(buff), fi))
    if (isdigit(buff[0]))
      print_vector(fo);
    else
      fputs(buff, fo);
  fclose(fi); fclose(fo);
  system("cc new" __FILE__);
  return 0;
}
```

The candidate virus phenotype begins as follows:

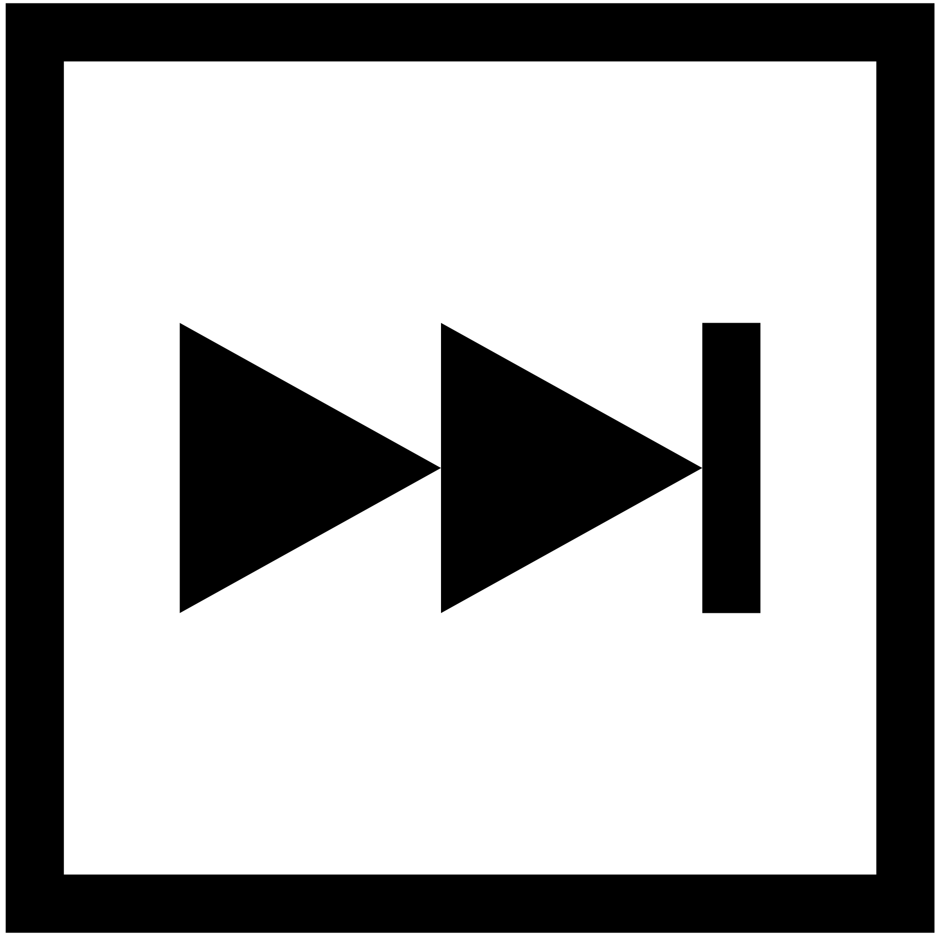
```
/* Satisfied */
[...]
int x[N] = {1, 1, 1, 1, 1, 1,
};
```

Acknowledgment

The author acknowledges the valuable suggestions of the anonymous referees on an earlier version of this correspondence.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
[2] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," *ACM Comput. Surv.*, vol. 26, no. 3, pp. 211–254, Sept. 1994.
[3] P. J. Denning, "Computer viruses," *Amer. Scientist*, pp. 236–238, May–June 1988.
[4] F. Cohen, "Computer viruses: Theory and experiments," *Comput. Security*, vol. 6, no. 1, pp. 22–35, Feb. 1987.
[5] E. H. Spafford, K. A. Heaphy, and D. J. Ferbrache, "A computer virus primer," in *Computers Under Attack: Intruders, Worms, and Viruses*, P. J. Denning, Ed. Reading, MA: Addison-Wesley, 1990, ch. 20, pp. 316–355.
[6] V. Prevelakis and D. Spinellis, "Sandboxing applications," in *USENIX 2001 Technical Conf. Proc.: FreeNIX Track*. Boston, MA: Usenix Assoc., June 2001.



A Simulated Annealing Approach for Buffer Allocation in Reliable Production Lines

*

Diomidis D. Spinellis [a,**] Chrissoleon T. Papadopoulos [b]

[a] *Department of Mathematics, GR-832 00 Karlovasi, University of the Aegean, Greece*
E-mail: dspin@aegean.gr

[b] *Department of Business Administration, GR-821 00 Chios Island, University of the Aegean, Greece*
E-mail: hpap@aegean.gr

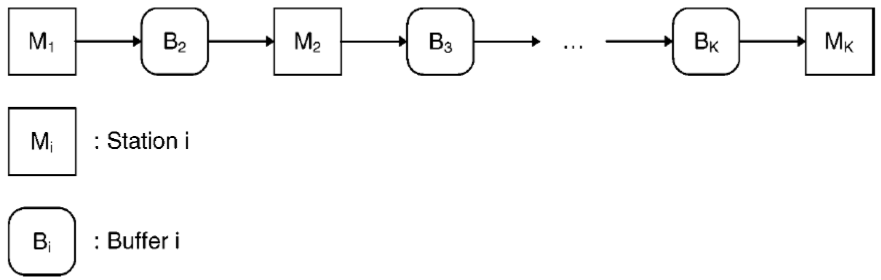
We describe a simulated annealing approach for solving the buffer allocation problem in reliable production lines. The problem entails the determination of near optimal buffer allocation plans in large production lines with the objective of maximizing their average throughput. The latter is calculated utilizing a decomposition method. The allocation plan is calculated subject to a given amount of total buffer slots in a computationally efficient way.

Keywords: Simulated annealing, production lines, buffer allocation, decomposition method

1. Introduction and Literature Review

Buffer allocation is a major optimization problem faced by manufacturing systems designers. It has to do with devising an allocation plan for distributing a certain amount of buffer space among the intermediate buffers of a production line. This is a very complex task that must account for the random fluctuations in mean production rates of the individual workstations of the lines. To solve this problem there is a need of two different tools. The first is a tool that calculates the performance measure of the line which has to be optimized (e.g., the average throughput or the mean work-in-process).

** Corresponding author.



$$\binom{N+K-2}{K-2} = \frac{(N+1)(N+2)\cdots(N+K-2)}{(K-2)!}.$$

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:57–71 (DOI: 10.1002/spe.428)

SP&E

Unix tools as visual programming components in a GUI-builder environment

Diomidis Spinellis*,†

Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, Athens GR-10434, Greece

SUMMARY

Development environments based on ActiveX controls and JavaBeans are marketed as 'visual programming' platforms; in practice their visual dimension is limited to the design and implementation of an application's graphical user interface (GUI). The availability of sophisticated GUI development environments and visual component development frameworks is now providing viable platforms for implementing visual programming within general-purpose platforms, i.e. for the specification of non-GUI program functionality using visual representations. We describe how specially designed reflective components can be used in an industry-standard visual programming environment to graphically specify sophisticated data transformation pipelines that interact with GUI elements. The components are based on Unix-style filters repackaged as ActiveX controls. Their visual layout on the development environment canvas is used to specify the connection topology of the resultant pipeline. The process of converting filter-style programs into visual controls is automated using a domain-specific language. We demonstrate the approach through the design and the visual implementation of a GUI-based spell-checker. Copyright © 2001 John Wiley & Sons, Ltd.

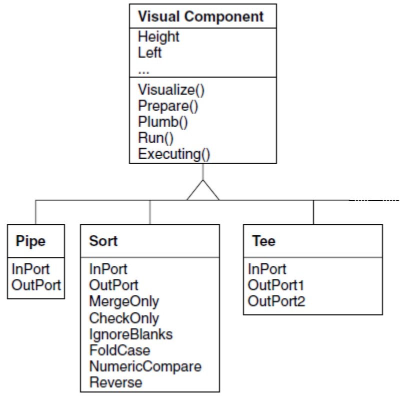
KEY WORDS: visual programming; components; reflection; Unix tools; pipe and filter architecture; reuse

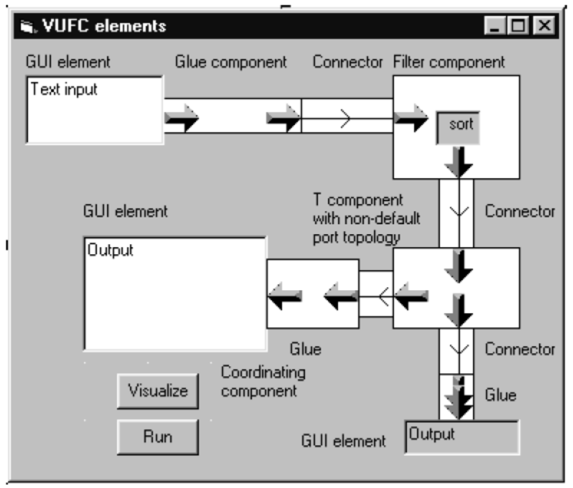
1. INTRODUCTION

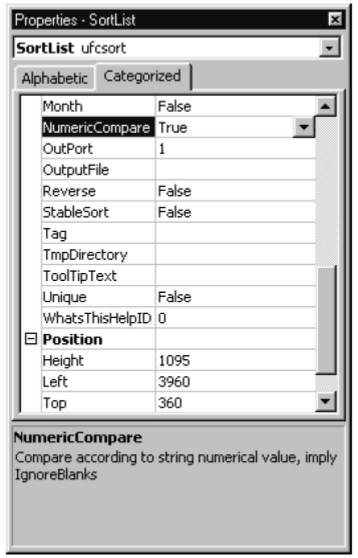
A number of environments support the visual composition of graphical user interfaces (GUIs) using components with a predefined set of interfaces. In addition, technologies such as ActiveX and JavaBeans allow the development of visual components (typically GUI elements) that can be seamlessly incorporated into an integrated development environment (IDE) and subsequently used in application development. In this article we present how visual IDEs and components can be extended beyond GUI development to support visual programming for a particular domain.

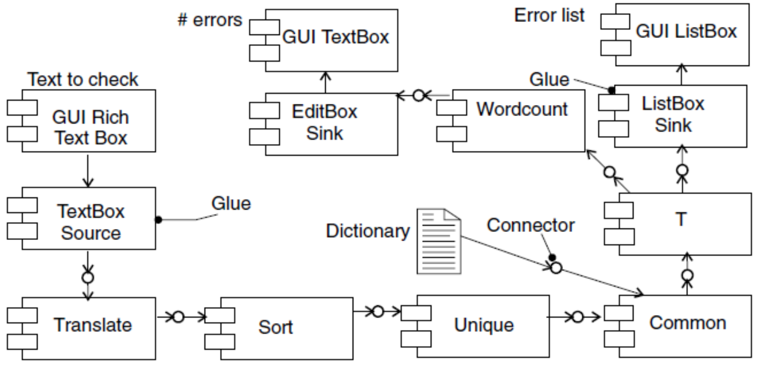
*Correspondence to: Diomidis Spinellis, Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, Athens GR-10434, Greece.
†E-mail: dds@aueb.gr

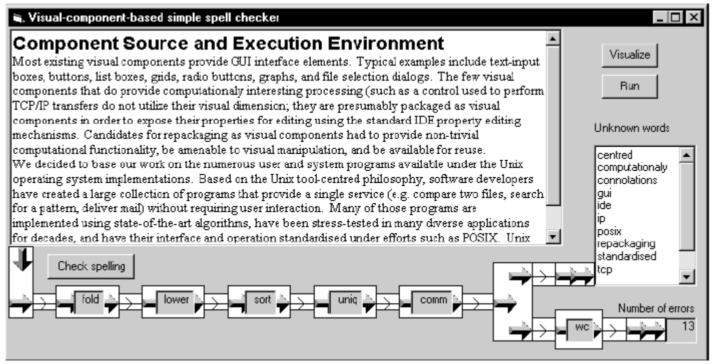
Received 10 November 2000
Revised 4 June 2001 and 17 September 2001
Accepted 18 September 2001











Make the most out of the time you have

github.com/dspinellis/latex-advice

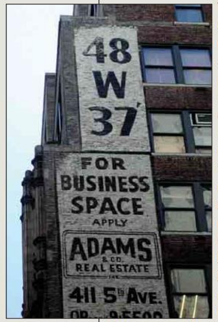
1. Put the document under version control
2. Write readable and maintainable LaTeX source code
3. Avoid explicit formatting
4. Automate the management of bibliographic references
5. Use symbolic references
6. Automate the document's build
7. Use Continuous Integration
8. Use third-party LaTeX packages
9. Use style files
10. Learn how to set text, mathematics, tables, figures, and floats

```
% Why obtaining metrics is difficult
Obtaining metrics from large code bodies is difficult
for technical and operational reasons~\cite{Moc09,GS13}.
On the technical side,
code dependencies make it difficult to establish
the full context needed in order to parse and semantically analyse the code.
This is especially true for C code,
where the compilation depends on
system header files,
compiler-defined macros,
search paths, and
compile-time flags passed through the build process~\cite{Spi03r,LKA11,GG12}.
The operational reasons are associated with the required throughput,
though due to the relatively small number of releases we examined,
this was not a major issue in this study.
```

The Decay and FAILURES of WEB REFERENCES

Attempting to determine how quickly archival information becomes outdated.

By Diomidis Spinellis



The widespread adoption of the Web as a mechanism for sharing information has brought with it the corresponding ubiquity of URL references and citations. URLs regularly appear on billboards, packages, business cards, print advertisements, clothing, and as references in scientific articles. Most readers have probably experienced a "dead link": a Web reference that for a variety of reasons will not lead to a valid or correct Web page. A dead link stemming from a URL appearing in the context of everyday life is usually a minor inconvenience that can be resolved by using a Web index or a search engine; it will seriously affect only the future archeologists trying to untangle the web of our daily lives. On the other hand, a dead Web link appearing in a scientific article has wider implications. Citations in scholarly work are used to build upon existing work, substantiate claims, provide the context in which research is performed, and present, analyze, and compare different approaches or methodologies. Therefore, references that cannot be located seriously undermine the foundations of modern scientific discourse.

The objective of this article is to examine, quantify, and characterize the quantity and quality of Web links used in computing literature. Our aim is to provide definitive information related to the availability of URL references as a function of their age, their domain, the depth of the path used, as well as the technical reasons leading to failed links. Our research has been greatly aided by the emergence of online versions of traditional paper-based publications [4]. By tapping into the online libraries of the ACM and the IEEE Computer Society we were able to download, extract, and verify 4,375 Web links appearing in print articles during the period from 1995–1999. Here, we describe the technologies related to Web references and retrieval, outlining the methodology we followed, presenting the results obtained, and discussing their implications.

Internet resources are typically specified using the string representation of Uni-

PHOTOGRAPHS BY MICHAEL KLOSE

The Journal of Systems and Software 85 (2012) 666–682

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

ELSEVIER

Organizational adoption of open source software

Diomidis Spinellis[a,*], Vaggelis Giannikas[b]

[a] Department Management Science and Technology, Athens University of Economics and Business, Patision 76, GR-104 34 Athens, Greece
[b] Institute for Manufacturing, University of Cambridge, 17 Charles Babbage Road, Cambridge CB3 0FS, United Kingdom

ARTICLE INFO

Article history:
Received 6 February 2011
Received in revised form 19 August 2011
Accepted 15 September 2011
Available online 22 September 2011

Keywords:
Open source software
Technology adoption
Industrial practice

ABSTRACT

Organizations and individuals can use open source software (OSS) for free, they can study its internal workings, and they can even fix it or modify it to make it suit their particular needs. These attributes make OSS an enticing technological choice for a company. Unfortunately, because most enterprises view technology as a proprietary differentiating element of their operation, little is known about the extent of OSS adoption in industry and the key drivers behind adoption decisions. In this article we examine factors and behaviors associated with the adoption of OSS and provide empirical findings through data gathered from the US Fortune-1000 companies. The data come from each company's web browsing and serving activities, gathered by sifting through more than 278 million web server log records and analyzing the results of thousands of network probes. We show that the adoption of OSS in large US companies is significant and is increasing over time through a low-churn transition, advancing from applications to platforms. Its adoption is a pragmatic decision influenced by network effects. It is likelier in larger organizations and those with many less productive employees, and is associated with IT and knowledge-intensive work and operating efficiencies.

1. Introduction

Thousands of volunteers and numerous companies develop, distribute, and license software in a way that allows others to freely use it, study it, modify it, and redistribute it. What are the prospects of the organizational adoption of this so-called open source software (OSS) and why should we care?

In this paper, through a novel application of web server log scanning and host fingerprinting techniques, we gather evidence of OSS adoption among the US Fortune-1000 companies, and use it to examine factors associated with OSS adoption. Our observations are statistically significant and span a wide sample of companies. However, although each research question we test is backed by existing theories, we freely admit that our study as a whole is data-driven rather than grounded on a single cohesive theoretical framework. Our main contributions are: (a) findings that theoretical frameworks of organizational OSS adoption could build upon and should be able to explain, and (b) the description and demonstration of powerful internet-based methods for collecting data about an organization's IT operations.

A commonly accepted OSS definition (Coar, 2006) specifies that complying software must be licensed for free redistribution (at no cost or for profit), must provide access to its source code, should allow the creation of derived works provided they respect the creation of the original author, and should not restrict the use of the software with reference to specific persons, groups, fields of endeavor, products, technologies, or other software. Well-known examples of open source software include the Linux operating system kernel, the Mozilla Firefox web browser, the OpenOffice.org office application suite, the MySQL relational database system, and the PHP programming language. Many OSS products offer plausible alternatives to the corresponding proprietary products, while some, like the the Apache web server, the Sendmail mail server, and the BIND domain name system server, are market leaders in their categories (Netcraft Ltd., 2009; E-Soft Inc., 2007; Simpson and Bekman, 2007; Kerner, 2007).

With its roots in the academic world OSS was initially viewed with suspicion by some companies. As a representative example, Microsoft openly attacked it citing problems related to version incompatibilities, intellectual property risks (especially in the context of copyleft licenses), lack of a credible business model, and an inability to fund innovation (Mundie, 2001; The Economist, 2001). However, other IT companies have embraced it for operational or strategic reasons. One example of operational use involves Google's thousands of servers, which work on a modified version of Linux, thus benefiting the company through the system's low cost and the ability to modify it to suit its needs (Weber, 2005, p. 6). As another example consider Apple, which has used OSS code from the Mach and FreeBSD operating systems to leapfrog in the development of its widely acclaimed Mac OS X operating system (West, 2003). On the

* Corresponding author. Tel.: +30 210 8203981; fax: +30 210 8203370.
E-mail addresses: dds@aueb.gr (D. Spinellis), eg366@cam.ac.uk (V. Giannikas).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.09.037

2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

The Evolution of C Programming Practices: A Study of the Unix Operating System 1973–2015

Diomidis Spinellis
dds@aueb.gr

Panos Louridas
louridas@aueb.gr

Maria Kechagia
mkechagia@aueb.gr

Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece

ABSTRACT

Tracking long-term progress in engineering and applied science allows us to take stock of things we have achieved, appreciate the factors that led to them, and set realistic goals for where we want to go. We formulate seven hypotheses associated with the long term evolution of C programming in the Unix operating system, and examine them by extracting, aggregating, and synthesising metrics from 66 snapshots obtained from a synthetic software configuration management repository covering a period of four decades. We found that over the years developers of the Unix operating system appear to have evolved their coding style in tandem with advancements in hardware technology, promoted modularity to tame rising complexity, adopted valuable new language features, allowed compilers to allocate registers on their behalf, and reached broad agreement regarding code formatting. The progress we have observed appears to be slowing or even reversing prompting the need for new sources of innovation to be discovered and followed.

CCS Concepts

•Software and its engineering → Software evolution; *Imperative languages; Software creation and management;* Open source model; •General and reference → Empirical studies; Measurement; •Social and professional topics → *Software maintenance; History of software;*

Keywords

C; coding style; coding practices; Unix; BSD; FreeBSD

1. INTRODUCTION

Tracking long-term progress in engineering and applied science allows us to take stock of things we have achieved, appreciate the factors that led to them, and set realistic goals for where we want to go. Progress can be tracked along two orthogonal axes. We can look at the processes (inputs) or at the resulting artefacts (outputs). Furthermore, we can examine both using either qualitative or quantitative means.

The objective of this work is to study the long term evolution of C programming in the context of the Unix operating system development. The practice of programming is affected by tools, languages, ergonomics, guidelines, processing power, conventions, as well as business and societal trends and developments. Specific factors that can drive long term progress in programming practices include the affordances and constraints of computer architecture, programming languages, development frameworks, compiler technology, the ergonomics of interfacing devices, programming guidelines, processing memory and speed, and social conventions. These might allow, among other things, the more liberal use of memory, the improved use of types, the avoidance of micro-optimisations, the writing of more descriptive code, the choice of appropriate encapsulation mechanisms, and the convergence toward a common coding style.

Here are a few examples. The gradual replacement of clunky teletypewriters with addressable-cursor visual display terminals in the 1970s may have promoted the use of longer, more descriptive identifiers and comments. Compilers using sophisticated graph colouring algorithms for register allocation and spilling [12] may have made it unnecessary to allocate registers in the source code by hand. The realisation that the overuse of the *goto* statement can lead to spaghetti code [13] might have discouraged its use. Similarly, one might hope that the recognition of the complexity and problems associated with the (mis)use of the C preprocessor [15,34,48,49] may have led to a reduced and more disciplined application of its facilities. Also, one would expect that the introduction and standardisation of new language features [2,23,45] would lead to their adoption by practitioners. Finally, the formation of strong developer communities, the maturing of the field, and improved communication facilities may lead to a convergence on code style.

In more formal terms, based on a simple-regression exploratory study [54], we established the following hypotheses, which we then proceeded to test with our data.

H1: Programming practices reflect technology affordances

If screen resolutions rise we expect developers to become more liberal with their use of screen space, as they are no longer constrained to use shorter identifiers and shorter lines. Higher communication bandwidth (think of the progress from a 110 bps ASR-33 teletypewriter, to a 9600 bps VT-100 character addressable terminal, to a 10MB Ethernet-connected

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884799

Create / collect your own data sets

Pers Ubiquit Comput (2003) 7: 53–69
DOI 10.1007/s00779-002-0213-8

ORIGINAL ARTICLE

Diomidis D. Spinellis

The information furnace: consolidated home control

Received: 1 June 2002 / Accepted: 14 August 2002

Abstract The Information Furnace is a basement-installed PC-type device that integrates existing consumer home-control, infotainment, security and communication technologies to transparently provide accessible and value-added services. A modern home contains a large number of sophisticated devices and technologies. Access to these devices is currently provided through a wide variety of disparate interfaces. As a result, end users face a bewildering array of confusing user-interfaces, access modes and price structures. In addition, as most devices function in isolation, important opportunities to exploit synergies between their functionalities are lost. The information furnace distributes data, provides services, and controls an apartment's digital devices. Emphasis is placed on accessibility and on exploiting the synergies that inevitably come up when these technologies and services are housed under a single roof. The prototype implementation I outline integrates on a FreeBSD server the distribution of MP3-encoded music to DNARD/NetBSD thin clients, an answering machine, a burglar alarm, an Internet router, a fax server, a backup server, and intelligent control of a PBX.

Keywords Automation · Consumer electronics · Home-control · Multi-modal interfaces

1 Introduction

Although our complex lives are not necessarily improved by each new technological widget we adopt, uncooperative devices and appliances with deficient user-interfaces can certainly conspire to frustrate us. Over the past three years I have experimented with a number of technologies that gave birth to the *information furnace* concept: a basement-installed PC-type device that integrates existing consumer home-control, infotainment, security, and communication technologies to transparently provide ubiquitous access and synergistic value-added services. In the following sections we will examine the devices and appliances lurking in the modern home, overview the problems associated with the current breed of devices, and go over the basic elements of the information furnace concept and its prototype implementation. Further implementation details on technologies behind the system we describe can be found in Spinellis [1]; this paper focuses on the system's concept, architecture, and evaluation.

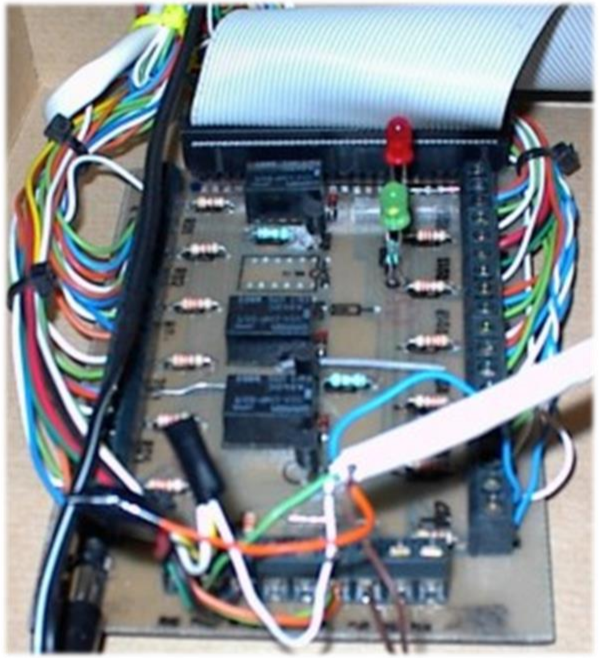
D.D. Spinellis
Department Management Science and Technology,
Athens University of Economics and Business,
Patision 76, GR-104 34 Athens, Greece
E-mail: dds@aueb.gr

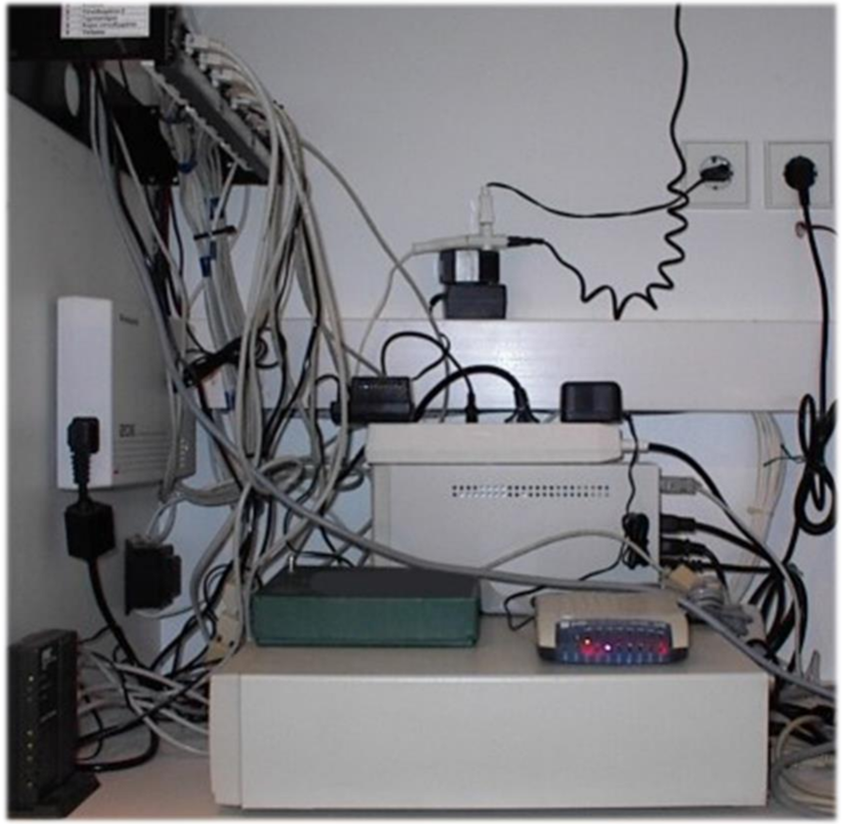
2 The modern home

A modern home contains a large number of sophisticated devices and technologies. Current and near future technologies and respective devices can be roughly categorised into the categories of home control, infotainment, security, communication and special-purpose devices.

2.1 Home control

Contemporary central heating systems are regulated by one external and a number of internal temperature sensors in conjunction with a control unit occupants use to set the desired room temperature. The system compares the internal room temperature to the setting of the control unit and, using the external temperature as a compensating factor, regulates the temperature of the water produced by the local heat-generating plant or the valve bringing remotely-heated water into the home. Burners often have their own control circuits based on target temperatures for the burner and the circulating pump, but we can regard them as a black box for the purposes of this article. Convenience elements associated





Position-Annotated Photographs: A Geotemporal Web

The GTWeb system exploits the synergies of integrating different information appliances and publicly accessible databases to create and present trip diaries.

Diomidis D. Spinellis
Athens University of Economics and Business

With the advent of digital cameras, photographs are no longer gathering dust, forgotten in old shoeboxes. Instead, they are lying unused in hard disk directories and on CDs. The Geotemporal Web system, belonging to the "capture and access" class of ubiquitous computing applications,[1] addresses this phenomenon by automatically converting raw data from the typical vacation trip into a lively Web site. Exploiting the synergies of integrating different consumer-grade information appliances and publicly accessible databases, a GTWeb site presents a trip overview, timelines, maps, and annotated photographs.

I first started working on GTWeb in the second half of 2001, in an effort to experiment with the presentation of GPS logs and digital photographs. Since then, I've been gradually adding features and maintaining its interfaces to keep up with technology evolution. Here, I discuss GTWeb's design and implementation and review what I've learned about integrating information appliances in general and presenting geotemporal data in particular.

Functional description

Initially constructing a Web site using GTWeb is fully automatic and involves integrating photographs from a consumer-grade digital camera, a track log recorded from a handheld GPS device,[2] and publicly accessible coastline, topography, and gazetteer data. Once created, you can manually edit and further enhance GTWeb HTML pages.

A GTWeb homepage (see Figure 1) displays a description of the trip, such as (underlining denotes hyperlinks)

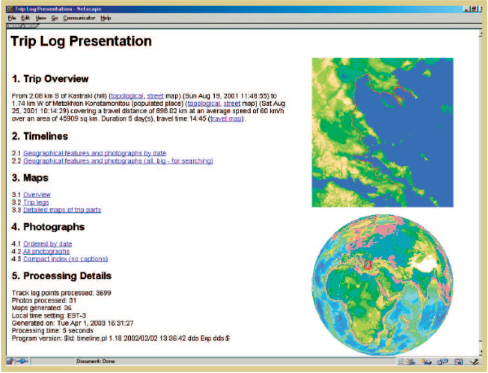
> *From 2.08 km S of Kastraki (hill) (topological, street map) (Sun Aug 19, 2001 10:48:55) to 1.74 km W of Metokhion Konstamonitou (populated place) (topological, street map) (Sat Aug 25, 2001 09:14:29) covering a travel distance of 898.02 km at an average speed of 60 km/h over an area of 45909 sq km. Duration 5 day(s), travel time 14:45 (travel map).*

The homepage also includes links to detailed timelines, maps, and photograph galleries (all presented in chronological order); a trip overview on a topographical map substrate; and the trip's location on an azimuthal orthographic projection of the earth globe. (See Figure 2 for a UML diagram of the GTWeb content tree.)

The timelines list information such as when the traveler approached a geographical feature or took a photograph (see Figure 3). GTWeb divides the maps into separate pages based on when the trip was made and presents a separate overview map for each trip leg and detailed maps covering smaller areas. Each detailed map shows the route traveled and geographic features (populated places, streams, hills, and so forth), annotated with the time they were approached (see Figure 4). Each map is prefixed by a textual description of the trip part it illustrates, such as

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/03/$17.00 © 2003 IEEE

Figure 1. A personal GTWeb's overview page.



> *Detailed Trip Part Map 40°19'N, 23°42'E – 40°24'N, 23°59'E Wed Aug 22, 2001. From 2.96 km SW of Pirgadhikia (populated place) (topological, street map) (11:15:44) to 0.96 km S of Prosforiou (populated place)(topological, street map) (12:20:23) covering a travel distance of 46.99 km at an average speed of 43 km/h over an area of 237 sq km. Duration 01:04, travel time 01:04.*

GTWeb indexes photographs using thumbprints and annotates them with a description of the time and place they were taken (see Figure 5). The same description, together with links to the corresponding trip leg map and detailed trip part map,

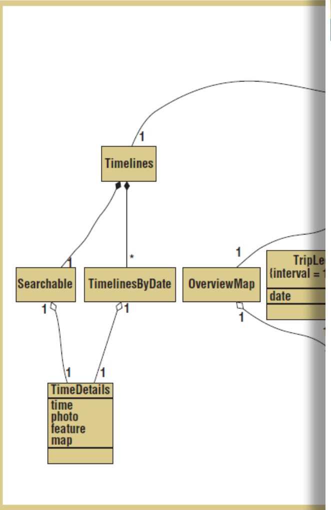


Figure 2. The GTWeb functional decomposition.

Wed Aug 22, 2001

[12:51:29] Approached (topological, street map) 2.95 km SW of Megali Vigla (hill) (topological, street map) traveling at a speed of 18 km/h.
[12:51:30] Photograph. About (most recent fix taken 1 seconds from the picture time) (topological, street map) 2.95 km SW of Megali Vigla (hill) (topological, street map) traveling at a speed of 18 km/h.
[…]
[12:57:53] Approached (topological, street map) 2.73 km W of Thivais (populated place) (topological, street map) traveling at a speed of 17 km/h.
[13:08:56] Photograph. About (most recent fix taken 5 seconds from the picture time) (topological, street map) 1.43 km SE of Thivais (populated place) (topological, street map) traveling at a speed of 18 km/h.
[13:10:25] Approached (topological, street map) 1.52 km SW of Monoxilitai (populated place) (topological, street map) traveling at a speed of 18 km/h.
[13:25:16] Approached (topological, street map) 5.48 km S of Moni Khiliandhariou (monastery) (topological, street map) traveling at a speed of 18 km/h.
[…]
[13:40:00] Photograph. About (most recent fix taken 3 seconds from the picture time) (topological, street map) 0.39 km NW of Moni Xenofondos (monastery) (topological, street map) traveling at a speed of 19 km/h.

Figure 3. A sample GTWeb timeline (underlining denotes hyperlinks).

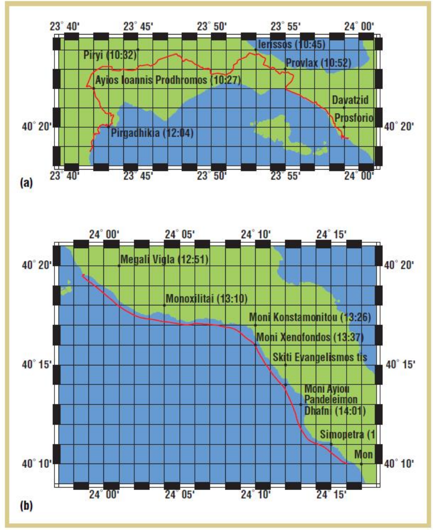


Figure 4. A detailed map of a trip leg over (a) land and (b) water.

also appears under the full-sized image of each photograph. All descriptions contain links leading to dynamically generated topological and street maps available on public Web sites. (See www.spinellis.gr/gtweb/Chalkidiki for a sample GTWeb site.)

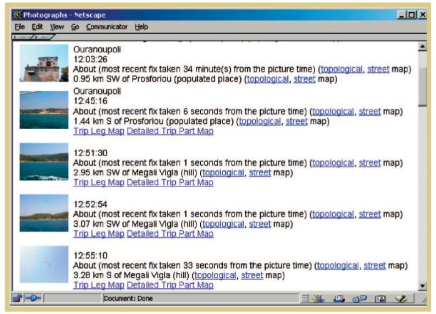
Application design

Figure 6 shows the dataflow diagram of the GTWeb creation process. The GTWeb software first processes a GPS track log together with the gazetteer database to annotate the track log with the nearest—in Euclidean distance—geographical features for each track point. GTWeb can then use topography (a grid of altitude points on the earth globe) and coastline data (closed polygons) to create the various maps. This phase superimposes the trip track and geographical features on the maps drawn by matching the respective longitude and latitude coordinates. Finally, GTWeb allocates the photographs into different maps, textually annotating them based on the time assigned by the respective appliance to each track log point and digital photograph. The availability of time information for both track log points and the photographs was the crucial factor that let me integrate the two different data sets.

Figure 7 depicts (as a UML diagram) the data model used to construct a GTWeb site. The primary types of data objects are

- Track points: latitude-longitude-time triples
- Photographs: the actual image plus an optional caption and the time each photograph was taken
- Gazetteer geographical features: coded references to each feature's geographical region, the feature's type (such as lake, town, or mountain), and the feature's name and coordinates

Figure 5. Index of boat-trip photographs.



To create a GTWeb site, the system extends the three data objects by combining features of their parent classes:

- Annotated track points might contain the details of a geographical feature (for example, a town) near a given point, together with its distance.
- Annotated features refer to the time the user's track passed near them (the user "visited" them) and the track's nearest distance.
- Annotated photographs contain the details of the nearest geographical feature and track point, together with the time difference between the photograph and the temporary closest track point.

The track log point with the smallest Euclidean distance to the given feature determines the time and location of the traveler's visit to the vicinity of a given geographical feature. We can formalize this as follows:

1. The coordinates of all known geographical features form a set F, and the coordinates of the track followed by the user form a set T.
2. Given two coordinate pairs (a_x, a_y), and (b_x, b_y), the notation $|a - b|$ denotes the Euclidean distance between a and b:

$$\sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}.$$

GTWeb forms an annotated track log A by associating each track point t with its nearest feature f:

$$A = \{(t,f) \mid t \in T \wedge f \in F \wedge \forall f' \in F \; |t - f| \le |t - f'|\}.$$

3. A set of "visits" V is formed from the annotated track log points that are nearest to each feature:

$$V = \{(t,f) \mid (t,f) \in A \wedge \forall (t',f) \in A \; |t - f| \le |t' - f|\}.$$

GTWeb uses most data in its native format, apart from photograph metadata where an intermediate program layer transforms file system resident information into XML, which is used for further processing. Thus, a photograph's details will appear as

```
<photo>
<name>DSC00007.JPG</name>
<time>998474606</time>
<caption>Ouranoupoli</caption>
<localtime>Wed Aug 22 13:03:26 2001</localtime>
<gmtime>Wed Aug 22 10:03:26 2001</gmtime>
</photo>
```

In a future version, I would probably use standardized schemas based on XML to

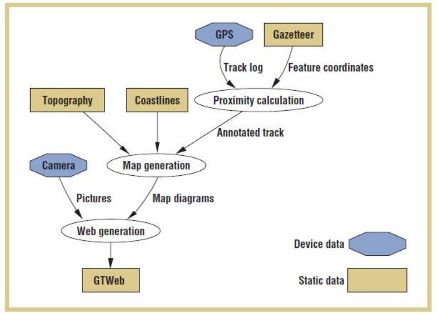


Figure 6. Data-flow diagram of the GTWeb generation process.

THE ATHENS AFFAIR

HOW SOME EXTREMELY SMART HACKERS PULLED OFF THE MOST AUDACIOUS CELL-NETWORK BREAK-IN EVER

By Vassilis Prevelakis & Diomidis Spinellis

ON 9 MARCH 2005, a 38-year-old Greek electrical engineer named Costas Tsalikidis was found hanged in his Athens loft apartment, an apparent suicide. It would prove to be merely the first public news of a scandal that would roil Greece for months.

The next day, the prime minister of Greece was told that his cellphone was being bugged, as were those of the mayor of Athens and at least 100 other high-ranking dignitaries, including an employee of the U.S. embassy.

The victims were customers of Athens-based Vodafone-Panafon, generally known as Vodafone Greece, the country's largest cellular service provider; Tsalikidis was in charge of network planning at the company. A connection seemed obvious. Given the list of people and their positions at the time of the tapping, we can only imagine the sensitive political and diplomatic discussions, high-stakes business deals, or even marital indiscretions that may have been routinely overheard and, quite possibly, recorded.

Even before Tsalikidis's death, investigators had found rogue software installed on the Vodafone Greece phone network by parties unknown. Some extraordinarily knowledgeable people either penetrated the network from outside or subverted it from within, aided by an agent or mole. In either case, the software at the heart of the phone system, investigators later discovered, was reprogrammed with a finesse and sophistication rarely seen before or since.

A study of the Athens affair, surely the most bizarre and embarrassing scandal ever to engulf a major cellphone service provider, sheds considerable light on the measures networks can and should take to reduce their vulnerability to hackers and moles.

It's also a rare opportunity to get a glimpse of one of the most elusive of cybercrimes. Major network penetrations of any kind are exceedingly uncommon. They are hard to pull off, and equally hard to investigate.

Even among major criminal infiltrations, the Athens affair stands out because it may have involved state secrets, and it targeted individuals—a combination that, if it had ever occurred before, was not disclosed publicly. The most notorious penetration to compromise state secrets was that of the "Cuckoo's Egg," a name bestowed by the wily network administrator who successfully pursued a German programmer in 1986. The programmer had been selling secrets about the U.S. Strategic Defense Initiative ("Star Wars") to the Soviet KGB.

But unlike the Cuckoo's Egg, the Athens affair targeted the conversations of specific, highly placed government and military officials. Given the ease with which the conversations could have been recorded, it is generally believed that they were. But no one has found any recordings, and we don't know how many of the calls were recorded, or even listened to, by the perpetrators. Though the scope of the activity is to a large extent unknown, it's fair to say that no other computer crime on record has had the same potential for capturing information about affairs of state.

While this is the first major infiltration to involve cellphones, the scheme did not depend on the wireless nature of the network. Basically, the hackers broke into a telephone network and subverted its built-in wiretapping features for their own purposes. That could have been done with any phone account, not just cellular ones. Nevertheless, there are some elements of the Vodafone Greece system that were unique and crucial to the way the crime was pulled off.

We still don't know who committed this crime. A big reason is that the UK-based Vodafone Group, one of the largest cellular providers in the world, bobbled its handling of some key log files. It also reflexively removed the rogue software, instead of letting it continue to run, tipping off the perpetrators that their intrusion had been detected and giving them a chance to run for cover. The company was fined €76 million this past December.

To piece together this story, we have pored through hundreds of pages of depositions, taken by the Greek parliamentary committee investigating the affair, obtained through a freedom of information request filed with the Greek Parliament. We also read through hundreds of pages of documentation and other records, supplemented by publicly available information and interviews with independent experts and sources associated with the case. What emerges are the technical details, if not the motivation, of a devilishly clever and complicated computer infiltration.

THE CELLPHONE BUGGING began sometime during the fevered run-up to the August 2004 Olympic Games in Athens. It remained undetected until 24 January 2005, when one of Vodafone's telephone switches generated a sequence of error messages indicating that text messages originating from another cellphone operator had gone undelivered. The switch is a computer-controlled component of a phone network that connects two telephone lines to complete a telephone call. To diagnose the failures, which seemed highly unusual but reasonably innocuous at the time, Vodafone contacted the maker of the switches, the Swedish telecommunications equipment manufacturer Ericsson.

We now know that the illegally implanted software, which was eventually found in a total of four of

CEOs, MPs & A PM

The illegally wiretapped cellphones in the Athens affair included those of the prime minister, his defense and foreign affairs ministers, top military and law enforcement officials, the Greek EU commissioner, activists, and journalists.

On 6 April 2006, BILL ZIKOU, CEO of Ericsson Hellas, was summoned to give evidence before a parliamentary committee looking into the scandal. His company provided the telecommunications switching equipment that rogue programmers broke into.

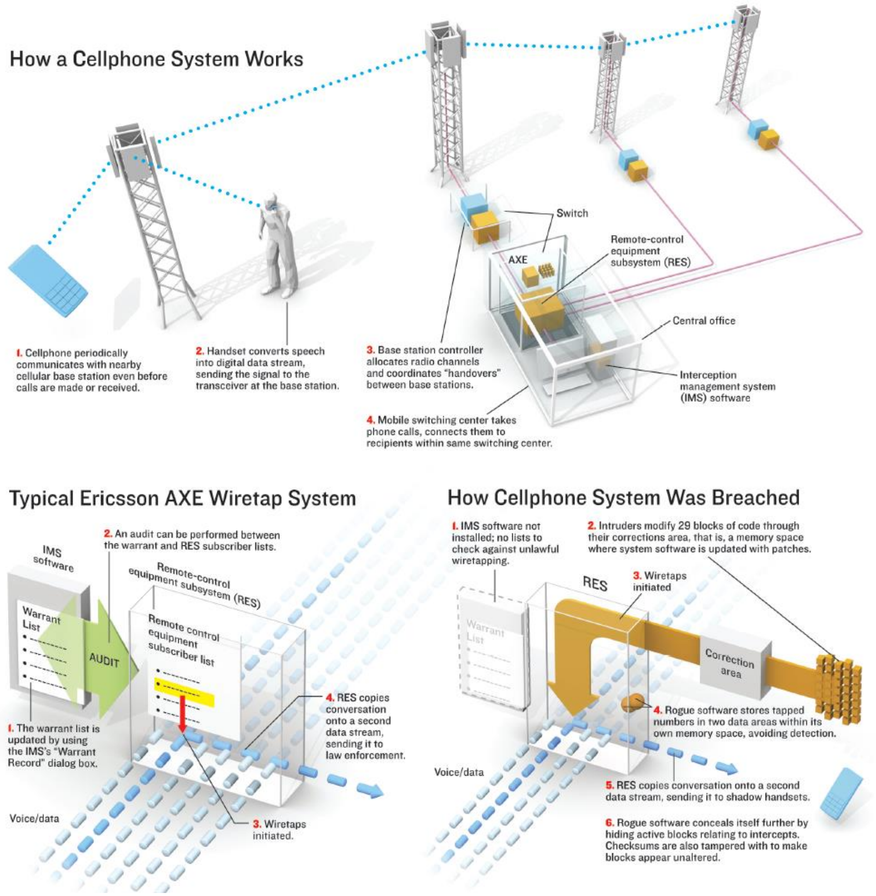
Vodafone Greece CEO GIORGOS KORONIAS ordered the removal of the surveillance program, because, as he explained in a February 2006 newspaper interview, "the company had to react immediately." Removing the program is thought to have tipped off the perpetrators and helped them evade capture.

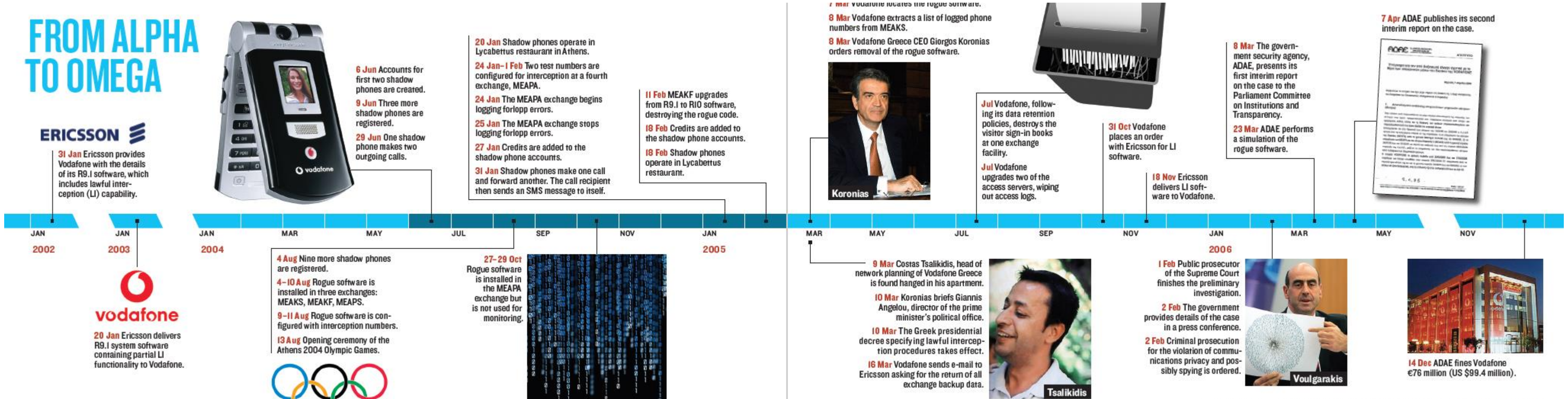
Greek Prime Minister COSTAS KARAMANLIS was only the most notable of the 100 or so individuals illegally wiretapped, which, besides the country's political, law enforcement, and military elite, included Karamanlis's wife.

COSTAS TSALIKIDIS was found hanged, an apparent suicide, just before the Athens affair became public. As a telecommunications engineer in charge of network planning at Vodafone, he was ideally placed to be either an inside accomplice or discoverer of the digital break-in. But his involvement in the case has never been established.

GIORGOS VOULGARAKIS was the first government official to whom Koronias disclosed the case. Giannis Angelou, the director of the Prime Minister's political office, was also present.

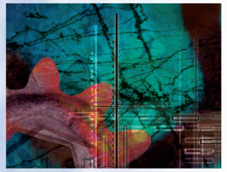
OPPOSITE PAGE: ARCHIVBERLIN FOTOAGENTUR/ALAMY; FROM TOP: KOSTAS TSIRONIS/AP PHOTO(2); JOHANNA LEGUERRE/AFP/GETTY IMAGES; AFP/GETTY IMAGES; LOUISA GOULIAMAKI/AFP/GETTY IMAGES





CLOCKWISE FROM TOP LEFT: ERICSSON; KOSTAS TSIRONIS/AP PHOTO; MICHAEL BROWN/ISTOCKPHOTO; ADAE; VODAFONE; LOUISA GOULIAMAKI/AFP/GETTY IMAGES; AFP/GETTY IMAGES; ANDREY PROKHOROV/ISTOCKPHOTO

The Antikythera Mechanism: A Computer Science Perspective



Diomidis Spinellis
Athens University of Economics and Business

The Antikythera mechanism is an ancient astronomical calculator that contains a lunisolar calendar, predicts eclipses, and indicates the moon's position and phase. Its use of multiple dials and interlocking gears eerily foreshadows modern computing concepts from the fields of digital design, programming, and software engineering.

Two thousand years separate us from an ancient Greek computing device known as the Antikythera mechanism. Here I explain the mechanism's operation based on its reconstruction in Squeak Etoys, a multimedia authoring environment primarily designed to help high school students learn scientific and engineering concepts.[1,2] The reconstruction relies on the recent findings that an international cross-disciplinary team of scientists obtained through surface imaging and high-resolution x-ray tomography. My work aims to present the functioning of this remarkable device using working code, the language of our community.

The complete image of this implementation is available online as open source software running on the Etoys environment (www.dmst.aueb.gr/dds/sw/ameso). I encourage readers to download an Etoys image and run the software on it, as they step through the descriptions in this article.

HISTORY

In 1900, a group of sponge fishers seeking shelter from the Kythera Sea's cruel weather anchored their boats on the barren island of Antikythera. Continuing their diving there, they discovered at a depth of 42 meters an ancient shipwreck with bronze and marble statues. For almost a year afterward, they worked with the Greek government to salvage the ship's contents. These artifacts were then transferred for preservation and study to the National Archaeological Museum in Athens, where they remain on display to this day. Among the recovered items, which dated from the first century B.C., were a beautiful nude bronze statue and a severely corroded lump of bronze clearly containing gear wheels.

Numerous scientists have devoted their lives to the study of this mysterious mechanism. Based on the few legible letters in fragments and descriptions of mechanical contraptions in ancient Greek and Roman texts, it was initially identified as an astrolabe or planetarium. Derek J. de Solla Price, the father of scientometrics, subsequently spent three decades analyzing and reconstructing the device. Using radiographs, he was able to count the teeth of most of the device's gears and construct a detailed model of their operation. In his seminal 1974 monograph, "Gears from the Greeks," he described the mechanism as a calendar computer.[3] Famously, his proposed model included a differential mechanism, similar to the one found in the drive trains of modern cars, apparently constructed scores of centuries before its reinvention.

Published by the IEEE Computer Society

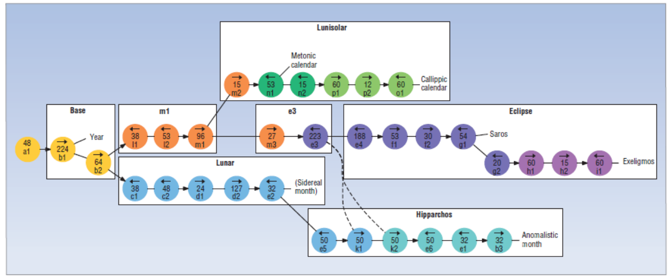


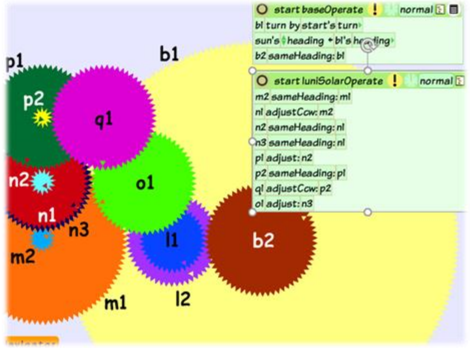
Figure 1. Antikythera mechanism operational model. Each circle represents a gear, with the arrow at the top indicating the direction the gear is turning, the number in the center how many teeth the gear has, and the letter-number combination at the bottom the name of the gear. Gears with the same letter are concentric, and numbers increase from the front to the back of the mechanism.

Recently, astronomers, archeologists, computer e
neers, and physicists from around the world collabor
on the Antikythera Mechanism Research Project (w
antikythera-mechanism.gr) to reconstruct a more
cise model. They used three computer-based imag
techniques—3D x-ray microfocus computed tomo
phy, polynomial texture mapping, and digitized h
quality photography—to study virtual cross-secti
of the device under various simulated lighting co
tions (samples of the images are available for inte
tive study at the project website). The project's resu
published in *Nature* in November 2006,[4] confirmed
the device was indeed a calendar computer. Howe
the new model proposes that the gears Price identi
as a differential instead operate in a distinct, but no
sophisticated, manner to calculate the anomaly in
moon's rotation.

CALCULATING WITH GEARS

The Antikythera mechanism is believed to consis
35 gears. Archeologists identified 30 in the surviv
fragments, while science historian Michael Wright
the authors of the *Nature* study introduced another
to explain the device's functionality.[4,5]

Figure 1 shows the relationship of the gears, e
represented by a circle. The arrow at the top of e
circle indicates the direction the gear is turning: →
clockwise and ← for counterclockwise. The numbe
the center indicates how many teeth the gear has.
studies of the mechanism name gears systematic
with a letter-number combination; the figure adopts
nomenclature used in the *Nature* article: Gears with
same letter are concentric, and numbers increase f
the front to the back of the mechanism.[4] A simple
connects gears that rotate together as one piece, w



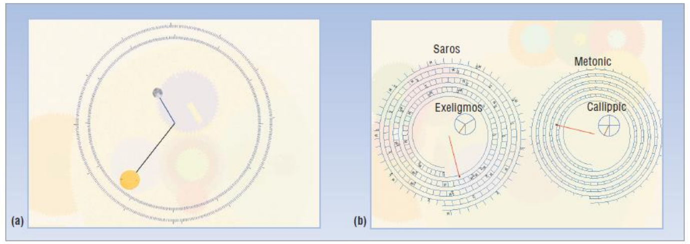


Figure 2. Antikythera mechanism's front and back faces. (a) A dial on the front face shows the sun's position throughout the year on the Zodiac cycle and a 365-day calendar. (b) The back face contains two lunicalendar dials (showing the Metonic and Callippic cycles) and two eclipse-prediction dials (showing the Saros and Exeligmos cycles).

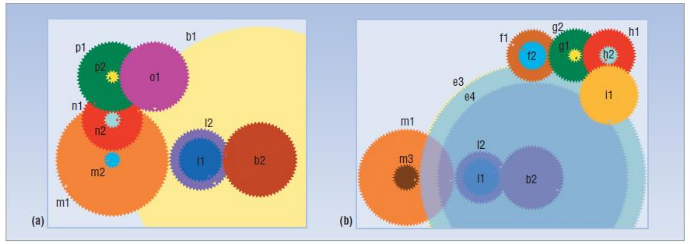


Figure 3. Emulator setup of (a) lunisolar calendar and (b) eclipse-prediction gears.

LUNISOLAR CALENDAR

Reckoning time progression through the moon's phases is convenient. It involves a calendar based on the visibly recurring lunar phases defining the 29½-day synodic month—the time from one full moon to the next. However, the months of such a calendar don't fit correctly in the seasonal year, which has practical significance for, say, agriculture.

In the fifth century B.C., the Athenian astronomer Meton devised a way around this problem by observing that 19 seasonal (known as tropical) years contain almost exactly 235 synodic months, and proposing a cycle containing 125 full months of 30 days and 110 hollow, 29-day, months.[6] A century later, Callippus further refined that model by proposing the removal of one day every four Metonic cycles. Two dials on the back of the Antikythera mechanism indicate each month in a Metonic cycle as well as track progress through the Callippic cycle.

To increase the Metonic display's resolution, the dial rotates five times in each cycle with a pointer tracking a five-turn spiral. As the pointer rotates, the spiral's grooves force it to move toward the outer turns of the spiral, similarly to a needle tracking a gramophone record. Once the pointer reaches the end of the spiral, the human operator would presumably return it to the beginning.

Gear n2 driving the Metonic calendar's dial must rotate 5 times in 19 years, thus the ratio between gear b1 tracking the tropical years and n2 should be 5/19. Indeed, the sequence b2-l1-l2-m1-m2-n1 calculates this ratio: $64/38 \times 53/96 \times 15/53 = 960/3648 = 5/19$. Further, gear o2 driving the Callippic cycle's dial must turn at 1/20 of the Metonic dial: once every four cycles of five turns each. The sequence n2-p1-p2-o1 calculates the required ratio: $15/60 \times 12/60 = 1/20$. Figure 3a shows the emulator setup of these gears.

ECLIPSE PREDICTION

The Antikythera mechanism predicts eclipses by means of the Saros cycle established by ancient Babylonian astronomers: a period of 223 and 1/3 synodic

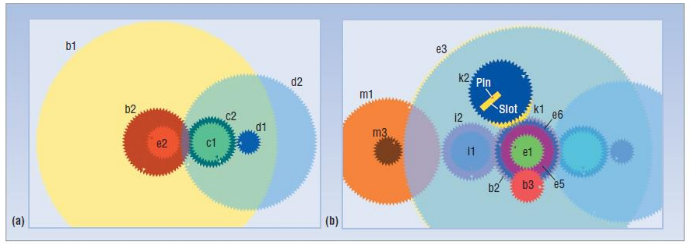


Figure 4. Emulator setup of gears to (a) calculate the sidereal month and (b) modify this calculation using Hipparchos's lunar mechanism to model the moon's elliptical orbit.

months in which identical moon and sun eclipses occur. Glyphs on the 223-month divisions of the plate where the Saros dial rotates indicate each eclipse type. The letter Σ (for ΣΕΛΗΝΗ—moon) indicates a lunar eclipse, while the letter H (for HΛΙΟΣ—sun) a solar one. Like the Metonic display, the Saros display is laid out in a four-turn spiral. Interestingly, its construction involves two elements of modern software engineering: the use of a lookup table (the Saros eclipse data) to aid computation, and the adoption of a design pattern (a spiral for increasing the display's resolution).

Because the Saros cycle contains a 1/3-day fraction, it's necessary to wait three Saros cycles to witness an eclipse at the same time. Thus, a separate dial indicates the Exeligmos cycle, which comprises three Saros cycles and can be used to predict the time of each eclipse.

Figure 3b shows the emulator setup of the eclipse-prediction gears. We already know from the Metonic calendar that there are 235 synodic months in 19 years. For the Saros display, we need four revolutions in 223 synodic months, a ratio of $4/223 \times 235/19$. The sequence b2-l1-l2-m1-m3-e3-e4-f1-f2-g1 establishes this ratio, which can be easily verified with a calculator. Further, the Exeligmos dial must turn once every three four-turn Saros cycles, thus at a rate 1/12 of Saros. The sequence g2-h1-h2-i1 calculates this ratio.

LUNAR CALCULATIONS

The Antikythera mechanism's front dial indicates the moon's *anomalistic month*—its position on the celestial sphere taking into account both the moon's elliptical orbit and the additional rotation of the ellipse's two extreme points. This anomaly is caused by the solar tide, and one full rotation takes nine years to complete. Gears b0 and q1 combine the moon's position with that of the sun to show the moon's phase. The three-step calculation of the moon's position is the most sophisticated of the mechanism's known parts.

The first step involves calculating the *sidereal month*, the moon's period in a fixed frame of reference. In a period of 19 years, the moon performs 235 synodic rotations (from the Metonic calendar) and another 19 due to its rotation around the sun—a total of 254. The sequence b2-c1-c2-d1-d2-e2, shown in Figure 4a, calculates the required yearly rotation ratio 254/19.

Next, the Antikythera mechanism models the moon's elliptical orbit through an ingenious device known as *Hipparchos's lunar mechanism*. The sidereal rotation established on gear e2 is transferred to gear e5, which is mounted on the same axle, as Figure 4b shows. Gear e5 in turn turns k1, which has a pin mounted a small distance from its center. Gear k2 is mounted below k1, but its center is slightly displaced from k1's center. The pin moves within a slot cut into gear k2, and, because the two gears are eccentrically mounted, harmonically varies k2's rotation rate. Running the Antikythera emulator demonstrates that k2's rotational speed is high when the slot is at the top of the screen and low when it's at the bottom; this models the corresponding variation of the moon's speed between its perigee and apogee.

Finally, the Antikythera mechanism models the rotation of this elliptical orbit by mounting k1 and k2 on e3. The gear e3 rotates at the rate of the elliptical orbit's rotation—the precession period of the moon's long axis—through the sequence b2-l1-l2-m1-m3-e3. Note that this rotates k2's axis and thereby complicates driving a dial with it. Thus, k2 drives e6, which is on a fixed axis. Gear e6 in turn drives e1, which is located on the front-dial side of e3, and b3 moves the rotation clockwise to the front dial's center. In computer engineering terms, the sequence e6-e1-b3 interfaces the processing unit to the display unit.

Another parallel with modern computing technology is the dual role of some gears: e3 in the calculation of both the Saros and the anomalistic month, and m1 in the calculation of Saros and the lunisolar calendar. This is

RESOURCES

A DIY LEGO CONTROLLER

A LOW-COST WAY TO PROGRAM LEGO MACHINES

DIOMIDIS SPINELLIS

RESOURCES_HANDS ON

If you want to explore coding with Lego bricks, there's one major option: to use a kit from the company's well-known Mindstorms robotics line. Mindstorms-based machines are built around the Intelligent Brick, which can be programmed using Lego's graphical programming environment or one of a number of third-party alternative languages. But Lego also makes a collection of motors, connectors, lights, and infrared receivers collectively sold under the label of Power Functions. In place of a programmable brick, the Power Function line includes a handheld controller for transmitting command signals. • I wondered if it was possible to use a Raspberry Pi to replace the handheld controller, taking on the role of an Intelligent Brick. This would have some advantages. With programs being created on the same device used to control Lego constructions, it would eliminate the need to download the programs to the brick, speeding up development. The US $40 Pi is also a lot cheaper than the $190 Intelligent Brick. I also wondered if such a setup could be used with MIT's Scratch, a free visual programming environment aimed at children. Scratch extensions are available for use with the Mindstorms brick, but they require altering the brick's firmware, and I wanted to try something simpler. • As I discovered, most of the code required for controlling Lego toys using Scratch is already available as open source software. What was

needed was integration, configuration, and some glue software.

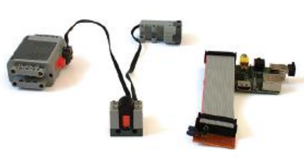
First, I needed to build an infrared control link, which is basically two infrared LEDs operated via the Raspberry Pi's general purpose input/output (GPIO) connector and Lego's receiver. I used schematics and instructions by Alex Bain to build the hardware. For the software, I downloaded and installed LIRC, a package that has support for decoding and transmitting signals used by over 2,500 different infrared remote controls.

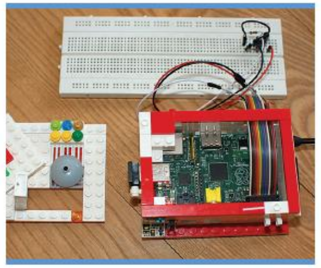
Getting the LIRC package to work with my home-brew infrared link was a simple matter of editing some configuration files and specifying which GPIO pins I had wired up for input and output.

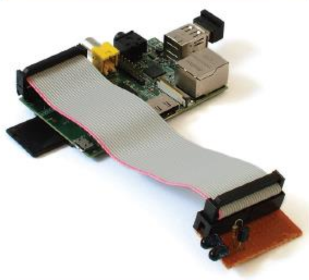
Now I needed to get LIRC to send valid Lego command signals. This means specifying the waveform—a pattern of infrared pulses—that must be sent for each Lego command. Fortunately, Lego has released a document specifying the protocol and format of all commands (for example, a binary value of 1 is transmitted by six pulses of IR light at a frequency of 38 kilohertz, followed by a pause of 553 microseconds). The Lego Power Functions system supports up to four receivers working on different channels, and each receiver has a red side and a blue side, each of which can independently control a motor.

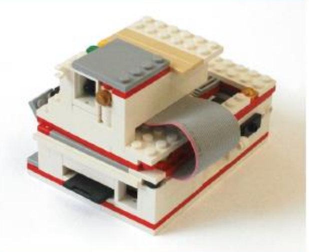
Building on this information, Conor Cary created lego-lirc, a Java program that generates command waveforms, complete with the correct checksums, in a format that LIRC understands. I downloaded lego-lirc and, with the Lego documentation in hand, created additional waveforms that allow the transmission of PWM (pulse-width modulation) commands. These commands allow precise speed adjustment of Power Function motors without requiring timing loops in the application software. (To avoid the hassle of running lego-lirc, you can just download my file of generated LIRC waveforms directly from my GitHub repository under the username of dspinellis.) To configure LIRC to use the Lego commands, I copied the waveform to the LIRC configuration directory. I could then send Lego commands from the Pi's command line through LIRC's irsend program.

BLOCK BY BLOCK









Lego Power Functions allow motors to be controlled with infrared signals [top]. Signals can be generated by connecting infrared LEDs to a Pi [middle images]. A Lego enclosure holds the components [bottom].

The final step was to issue the LIRC commands from the Scratch environment. I enabled "remote sensor connections" in Scratch. This makes Scratch behave like a local server running on the TCP port 42001. Client software can connect to Scratch using this port and listen for messages from Scratch programs. (It's also possible to have the client software and Scratch environment run on separate machines, so you could have the Raspberry Pi–based infrared interface controlled by a Scratch program running on a desktop computer, for example.) I then installed Phillip Quiza's excellent scratchpy library, which allows you to write Scratch clients in the Python programming language.

Finally, I wrote a Python script that receives Scratch broadcast messages specifying Lego remote commands, and runs the LIRC command-line client to send them (this is also available from my lego-power-scratch GitHub repository). To run the script, run the control.py program in a separate terminal window and launch the Scratch environment. While control.py is running, it will display on its standard output the remote control messages it sends or the errors it detects on the incoming Scratch messages.

In Scratch, programs are constructed by chaining together graphical blocks on screen. Blocks perform functions such as program-flow control and graphics manipulation. To send a message to a Lego Power Functions receiver, a "broadcast" block is used, with a simple text string of the form "Lego <channel> <Blue|Red> <power level>." So, for example, the message "Lego 2 blue -7" will send a signal by way of the Python client and my transmitter to turn the motor connected to the blue side of the receiver on channel 2 at full speed, backward.

How does the system work in practice with its intended audience? I tried it out with a young budding engineer—who quickly wrote a Scratch program to control Lego's Volvo Wheel Loader kit with a computer's arrow keys. **—DIOMIDIS SPINELLIS**

DIOMIDIS SPINELLIS (4)

Takeaway?

Be curious, have fun!

Thank you!

dds@aueb.gr

www.spinellis.gr

@CoolSWEng

github.com/dspinellis

Image Credits

- HF65 instructions: Bruker
- HF65 image: DL4CS German Amateur Radio Station
- Sharp PC-1211: Denisfo - Own work CC BY-SA 3.0
- TI-99/4A: Rama & Musée Bolo - Own work CC BY-SA 2.0 fr
- IBM PC: Rama & Musée Bolo - Own work CC BY-SA 2.0 fr
- IBM Portable PC: Hubert Berberich (HubiB) CC BY-SA 3.0
- Τεχνική Εκλογή magazine: salax54

References

- Diomidis Spinellis and Panagiotis Louridas. A framework for the static verification of API calls. *Journal of Systems and Software*, 80(7):1156–1168, July 2007. doi:10.1016/j.jss.2006.09.040
- Diomidis Spinellis. *Code Reading: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2003.
- Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, November 2003. doi:10.1109/TSE.2003.1245303
- Diomidis Spinellis. CScout: A refactoring browser for C. *Science of Computer Programming*, 75(4):216–231, April 2010. doi:10.1016/j.scico.2009.09.003
- Diomidis Spinellis. A dynamically linkable graphics library. Unpublished article, Imperial College, London, UK, March 1988.
- Alexander Lattas and Diomidis Spinellis. Echoes from space: Grouping commands with large-scale telemetry data. In *40th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '18, New York, NY, USA, May 2018. Association for Computing Machinery. doi:10.1145/3183519.3183545
- Diomidis Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, January 2003. doi:10.1109/TIT.2002.806137
- Duncan White, Jan-Simon Pendry, and Diomidis Spinellis. Unix PDP-11 emulator (as11 & em11) user's guide. Laboratory documentation, Imperial College, London, UK, January 1989.
- Diomidis Spinellis. *Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming*. PhD thesis, Imperial College, London, UK, February 1994.
- Diomidis Spinellis and Chrissoleon T. Papadopoulos. A simulated annealing approach for buffer allocation in reliable production lines. *Annals of Operations Research*, 93:373–384, 2000. doi:10.1023/A:1018984125703
- Diomidis Spinellis. Unix tools as visual programming components in a GUI-builder environment. *Software: Practice and Experience*, 32(1):57–71, January 2002. doi:10.1002/spe.428
- Diomidis Spinellis. *The Elements of Computing Style: 180+ Tips for Busy Knowledge Workers*. Leanpub, Vancouver, BC, Canada, 2014.
- Diomidis Spinellis. The decay and failures of web references. *Communications of the ACM*, 46(1):71–77, January 2003. doi:10.1145/602421.602422
- Diomidis Spinellis and Vaggelis Giannikas. Organizational adoption of open source software. *Journal of Systems and Software*, 85(3):666–682, March 2012. doi:10.1016/j.jss.2011.09.037
- Diomidis Spinellis, Panos Louridas, and Maria Kechagia. The evolution of C programming practices: A study of the Unix operating system 1973–2015. In Willem Visser and Laurie Williams, editors, *ICSE '16: Proceedings of the 38th International Conference on Software Engineering*, pages 748–759, New York, May 2016. Association for Computing Machinery. doi:10.1145/2884781.2884799
- Diomidis Spinellis. The information furnace: Consolidated home control. *Personal and Ubiquitous Computing*, 7(1):53–69, 2003. doi:10.1007/s00779-002-0213-8
- Diomidis Spinellis. Position-annotated photographs: A geotemporal web. *IEEE Pervasive Computing*, 2(2):72–79, April-June 2003. doi:10.1109/MPRV.2003.1203756
- Vassilis Prevelakis and Diomidis Spinellis. The Athens affair. *IEEE Spectrum*, 44(7):26–33, July 2007. doi:10.1109/MSPEC.2007.376605
- Diomidis Spinellis. The Antikythera mechanism: A computer science perspective. *IEEE Computer*, 41(5):22–27, May 2008. doi:10.1109/MC.2008.166
- Diomidis Spinellis. A DIY Lego controller: A low-cost way to program Lego machines. *IEEE Spectrum*, 53(11):21–22, November 2016. doi:10.1109/MSPEC.2016.7607018

www.spinellis.gr/pubs/