Enabling Variability-Aware Software Tools

Paul Gazzillo

Yale University

*Funded in Part by the Yale Postdoctoral Scholars Travel Fund

About Me

About Me

- Currently: postdoctoral researcher
  - Program analysis for security
  - Concurrency and distributed computing

About Me

- Currently: postdoctoral researcher
  - Program analysis for security
  - Concurrency and distributed computing

?

About Me

- Currently: postdoctoral researcher
  - Program analysis for security
  - Concurrency and distributed computing

?

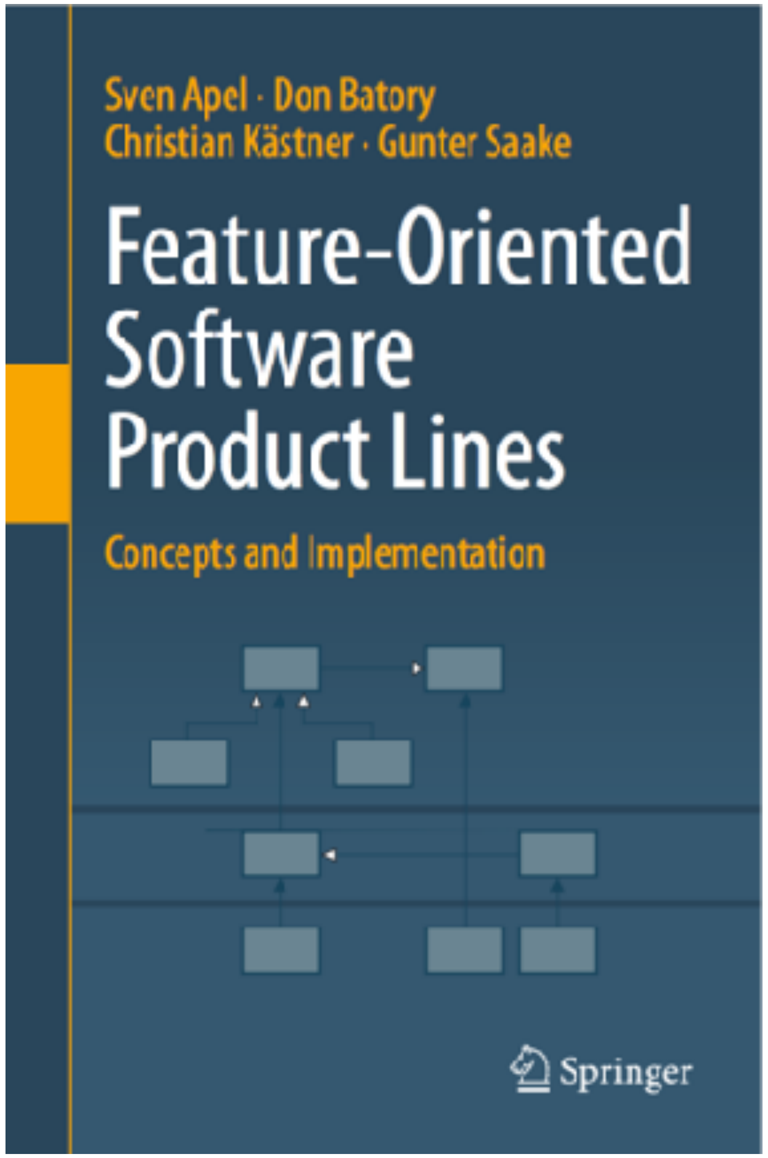
- Graduate school
  - Parsing all of C (including preprocessor usage)

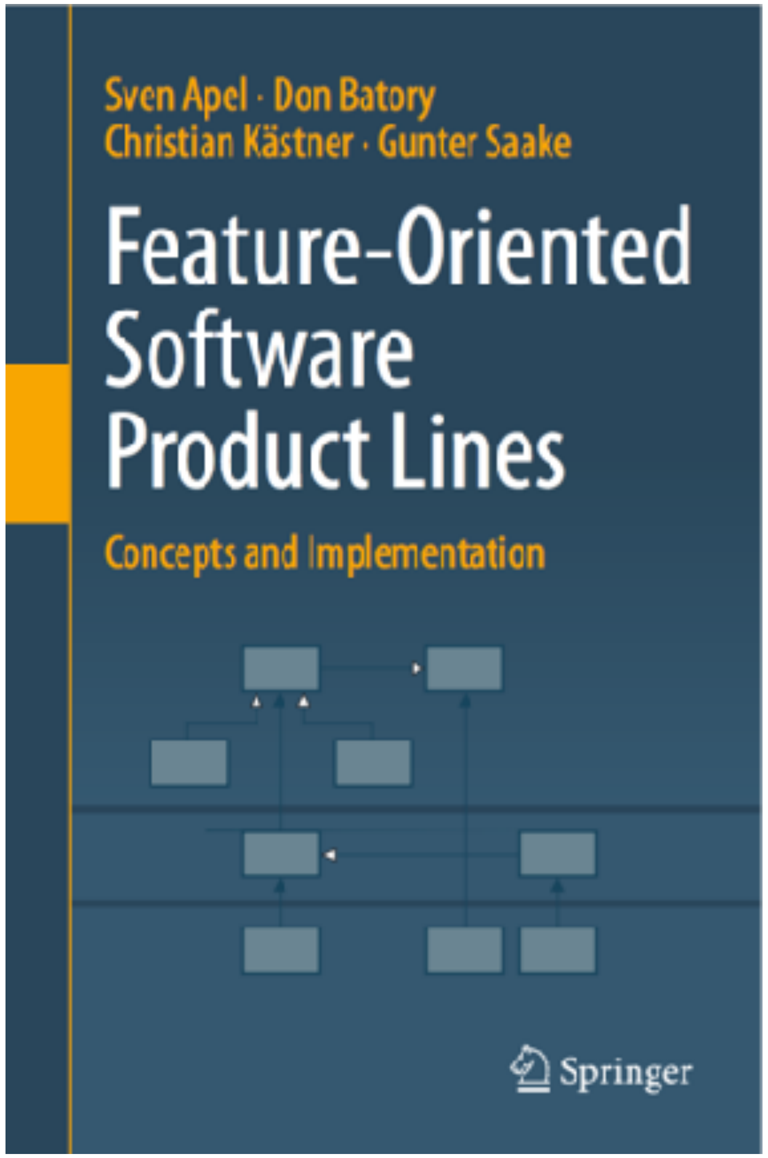
About Me

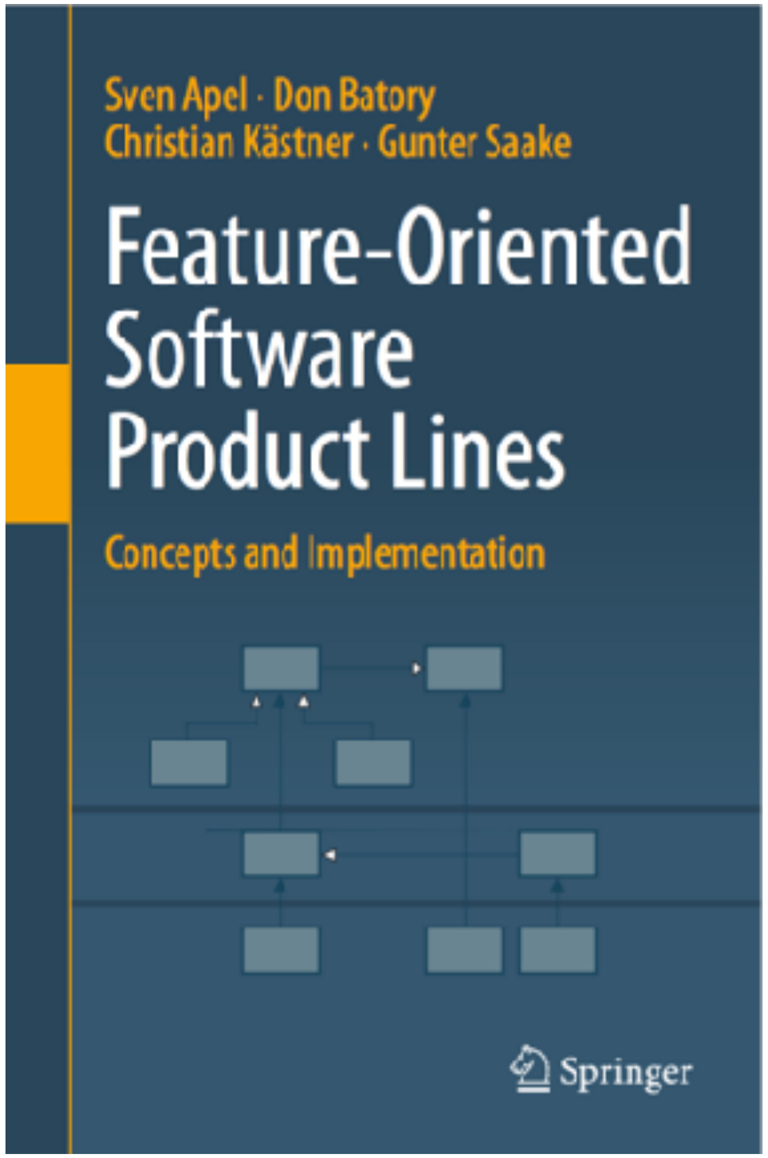
- Currently: postdoctoral researcher
  - Program analysis for security
  - Concurrency and distributed computing
- Graduate school
  - Parsing all of C (including preprocessor usage)

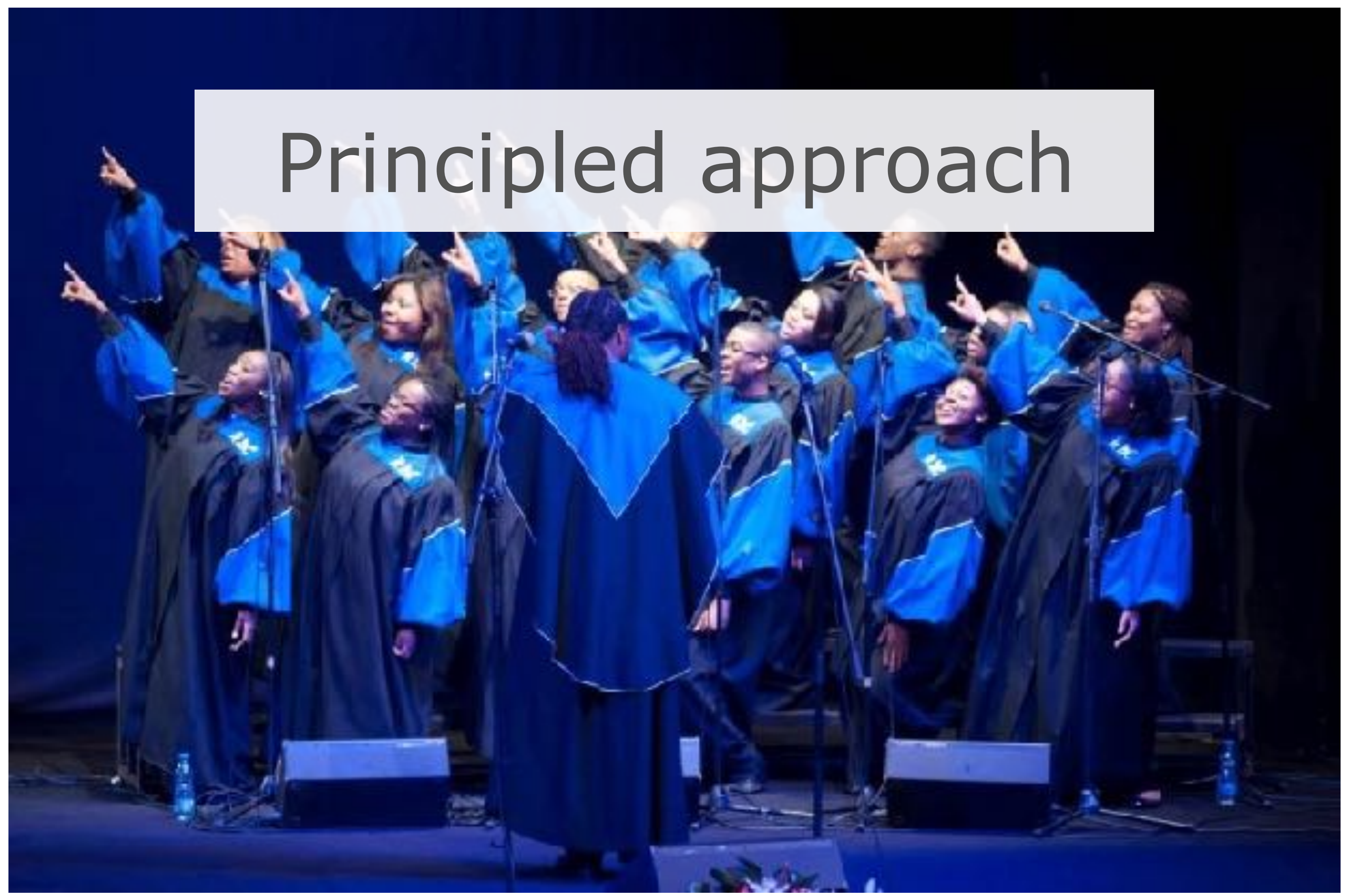
?

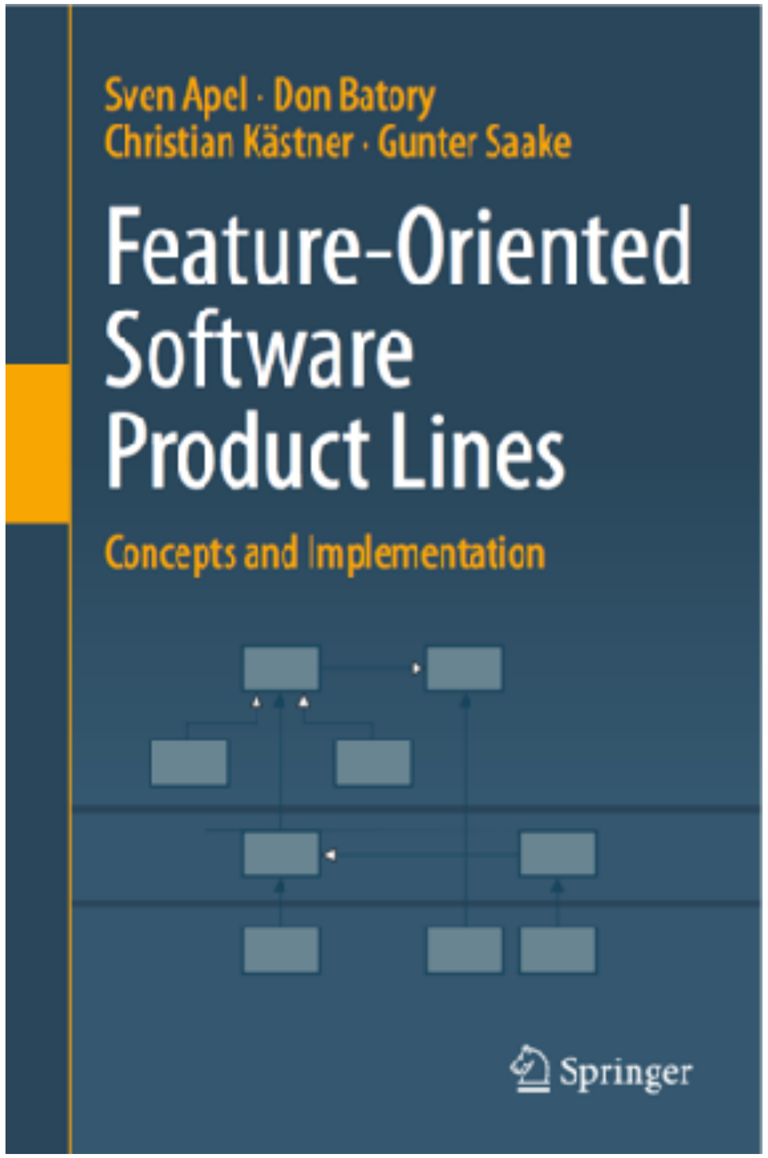
Encountered SIGSOFT and FOSD research

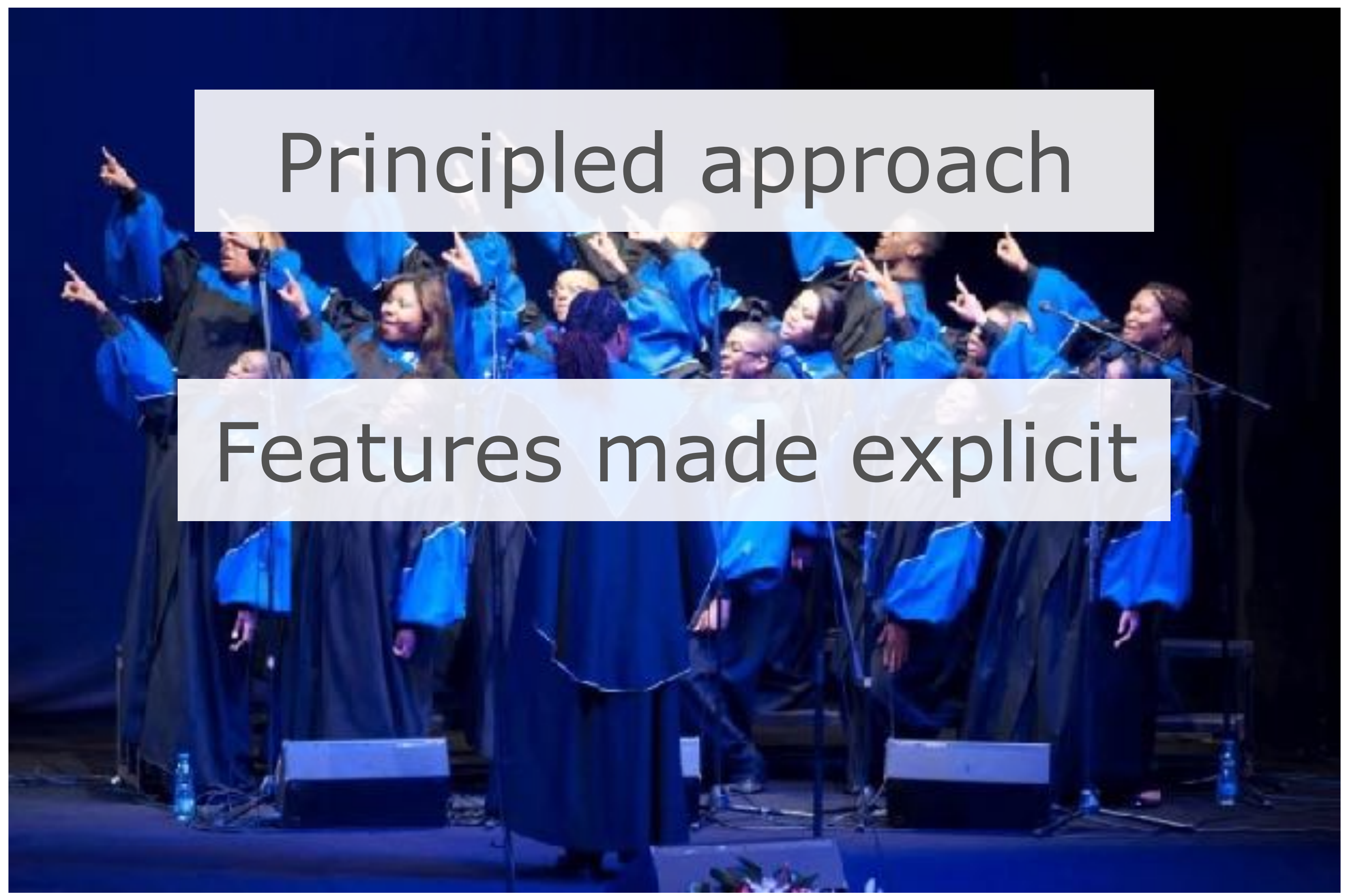

Sven Apel · Don Batory
Christian Kästner · Gunter Saake
Feature-Oriented Software Product Lines
Concepts and Implementation
Springer

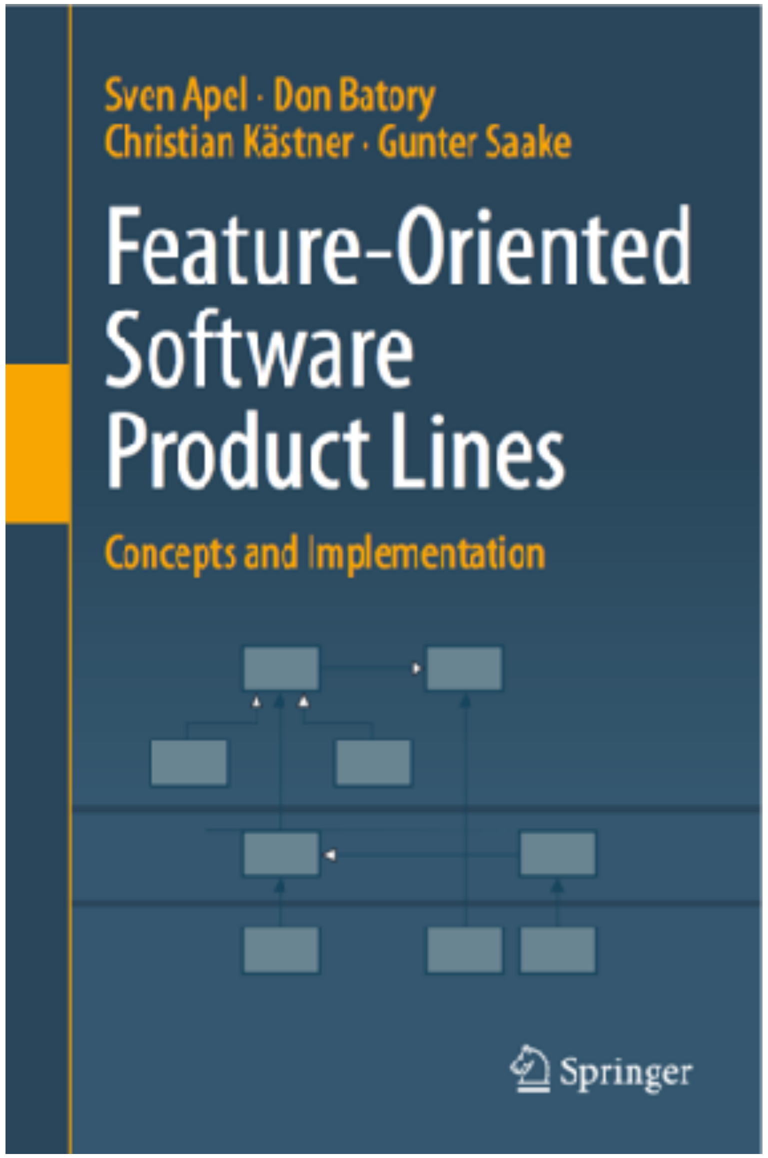

Sven Apel · Don Batory
Christian Kästner · Gunter Saake
Feature-Oriented Software Product Lines
Concepts and Implementation
Springer

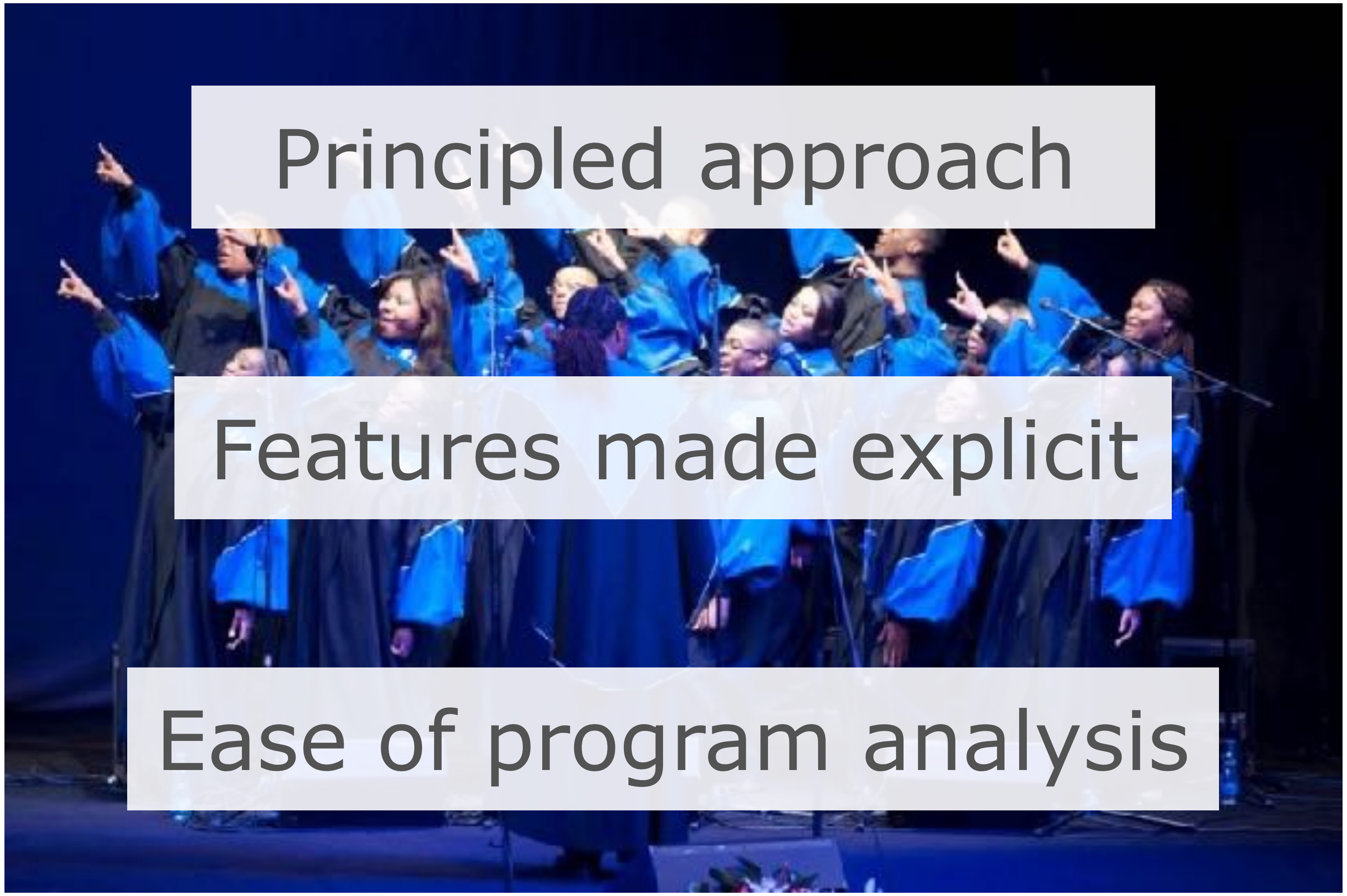

Sven Apel · Don Batory
Christian Kästner · Gunter Saake
Feature-Oriented Software Product Lines
Concepts and Implementation
Springer


Principled approach


Sven Apel · Don Batory
Christian Kästner · Gunter Saake
Feature-Oriented Software Product Lines
Concepts and Implementation
Springer

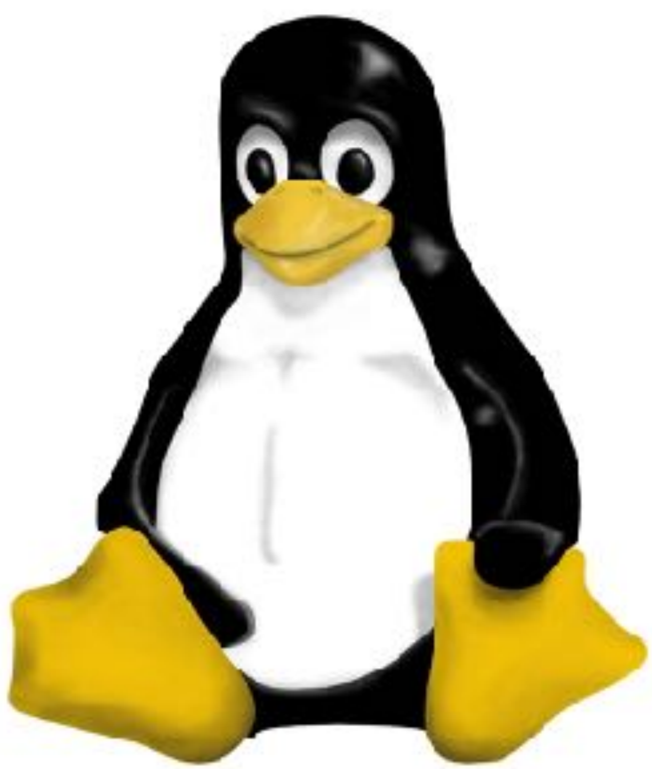

Principled approach
Features made explicit


Sven Apel · Don Batory
Christian Kästner · Gunter Saake
Feature-Oriented Software Product Lines
Concepts and Implementation
Springer


Principled approach
Features made explicit
Ease of program analysis

Apache

GNU



Linux

Apache

GNU

Linux

Apache

GNU

Millions SLoC

Linux

Apache

GNU

Linux

Millions SLoC

>14,000 features

Apache

GNU

Linux

Millions SLoC

>14,000 features

1000s of Makefiles

Apache

GNU

Linux

Millions SLoC

>14,000 features

1000s of Makefiles

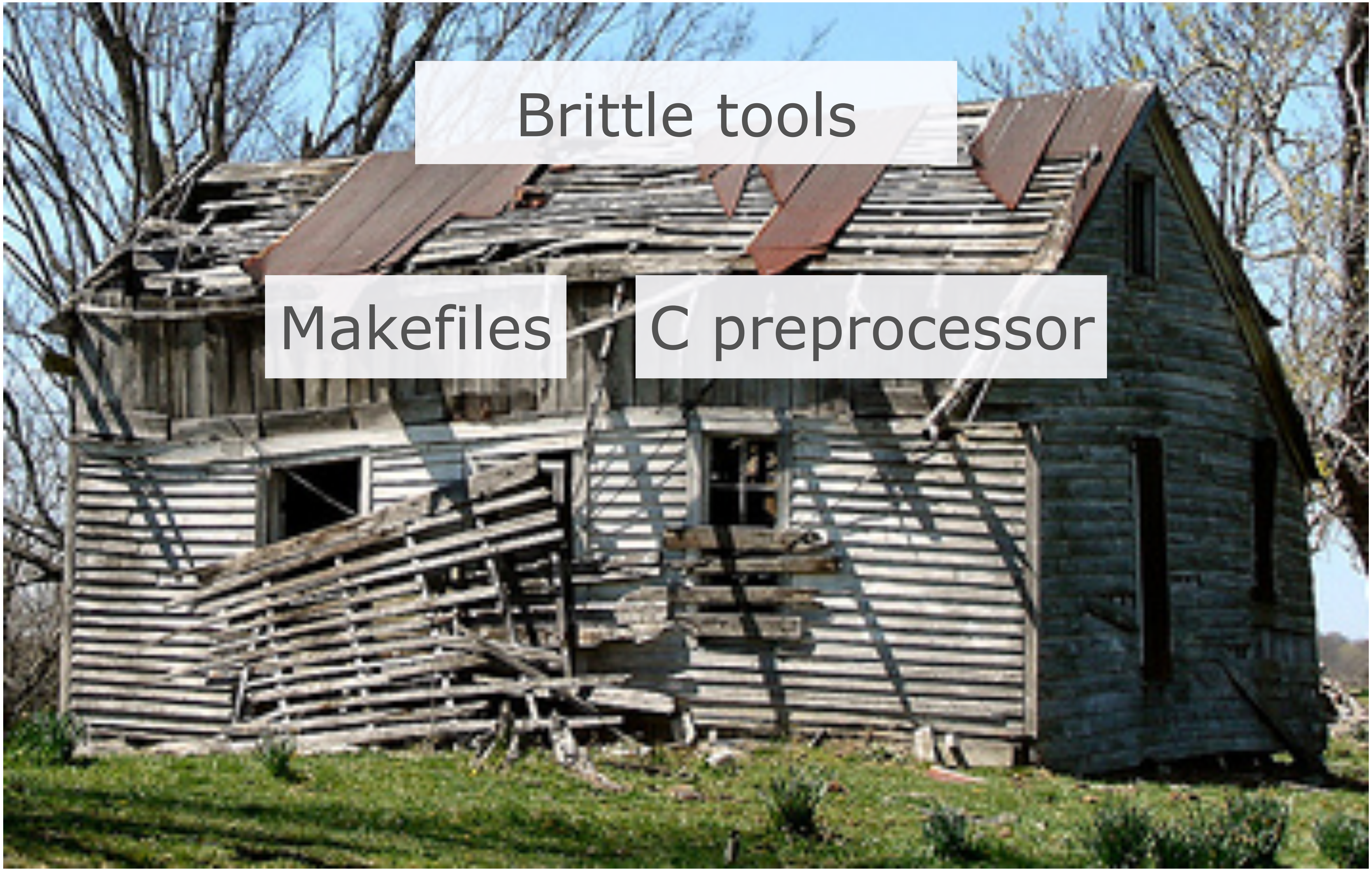
1000s of C files

Linux Variability

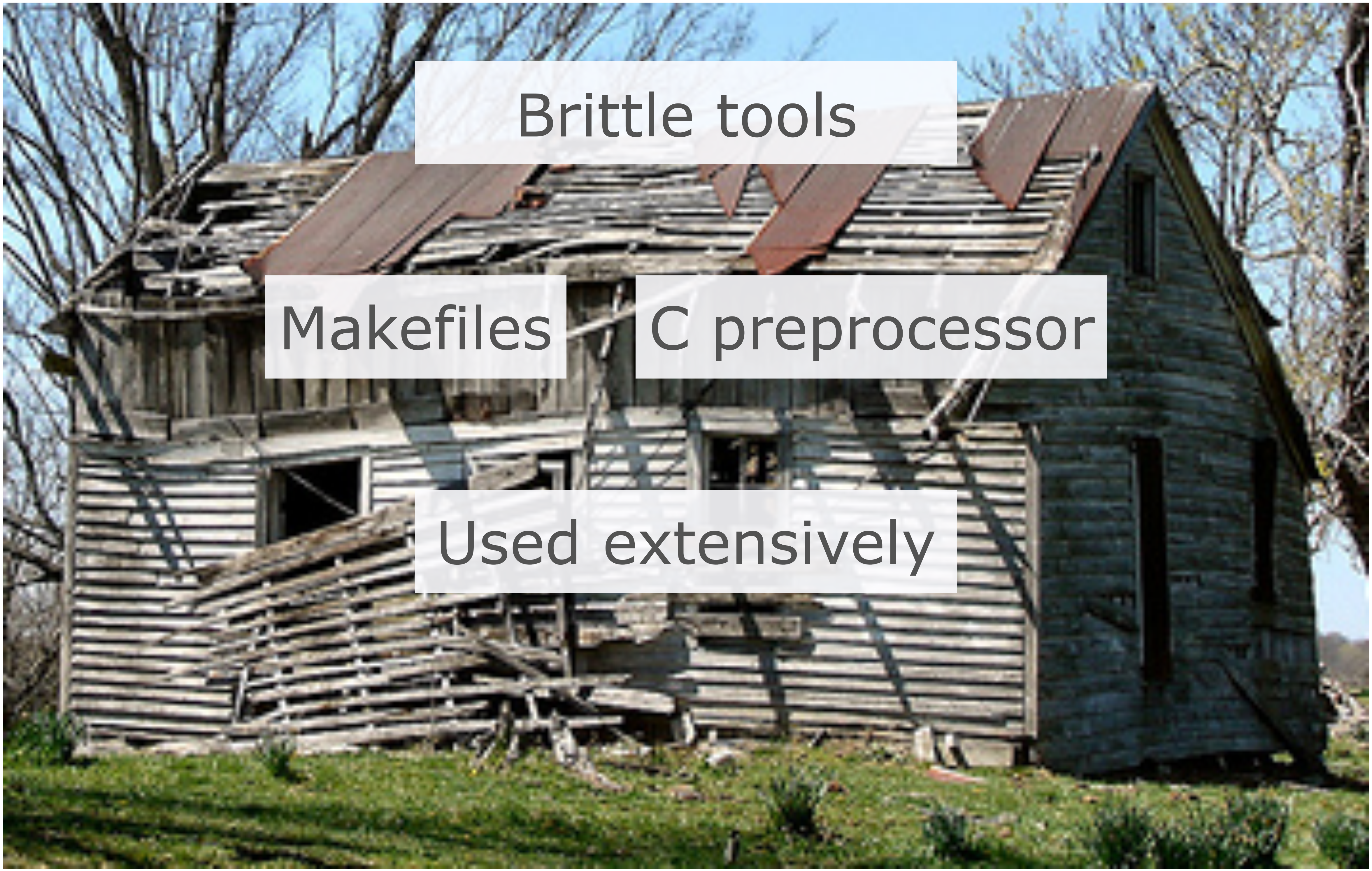
Linux Variability

Linux Variability

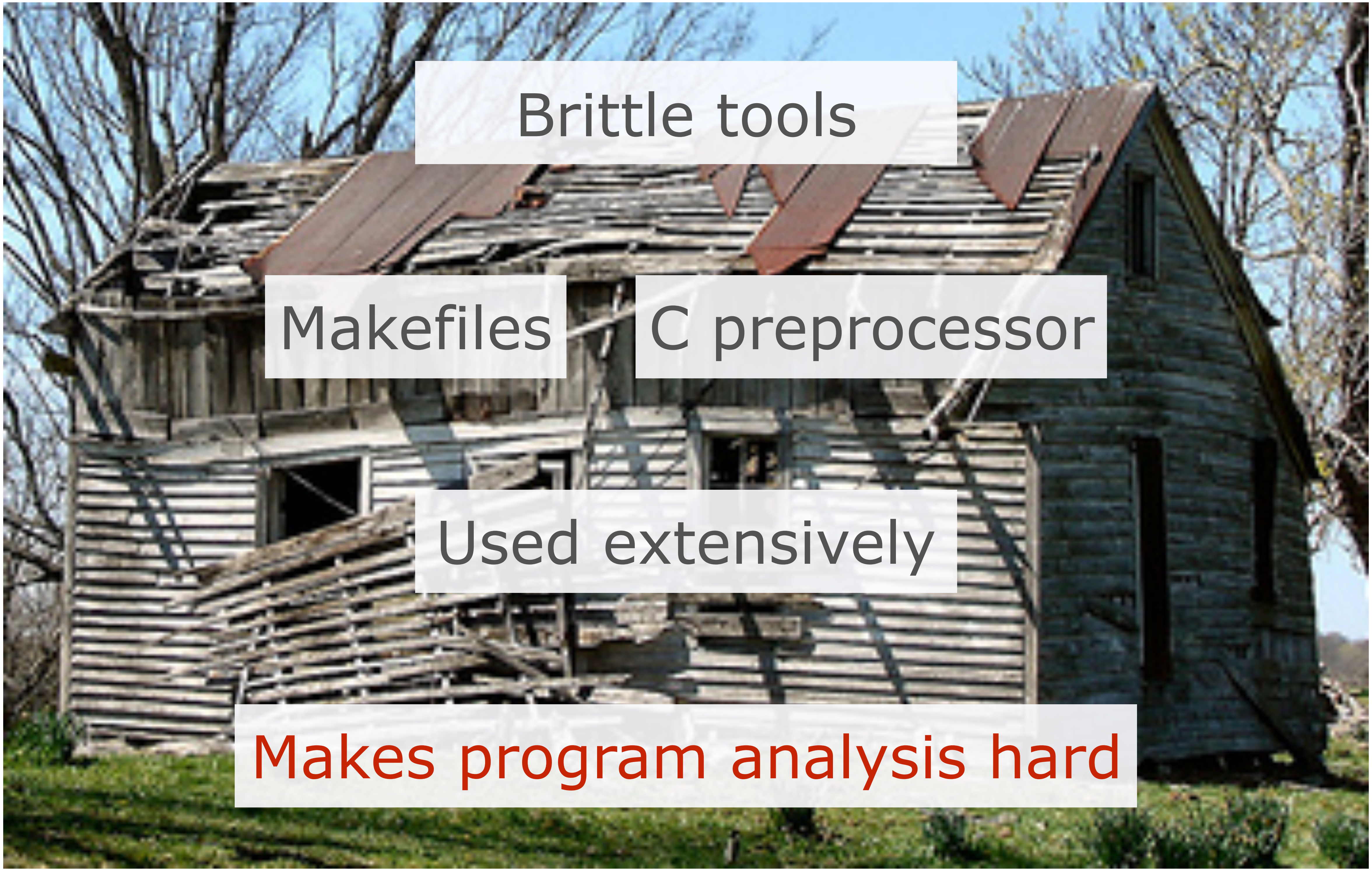
Linux Variability



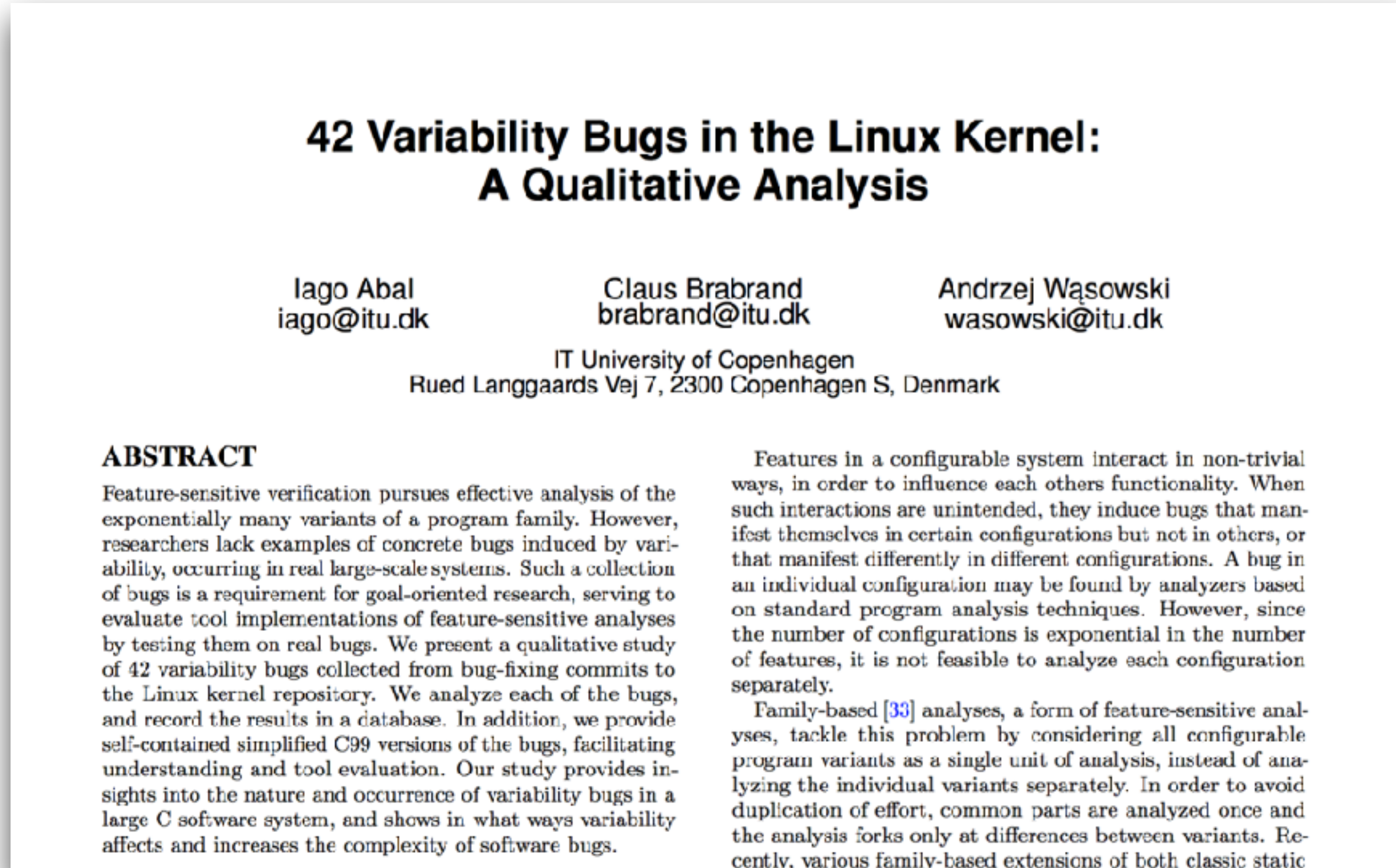
Linux Variability



Linux Variability



Real-world variability bugs happen



42 Variability Bugs in the Linux Kernel: A Qualitative Analysis

Iago Abal
iago@itu.dk

Claus Brabrand
brabrand@itu.dk

Andrzej Wąsowski
wasowski@itu.dk

IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

ABSTRACT

Feature-sensitive verification pursues effective analysis of the exponentially many variants of a program family. However, researchers lack examples of concrete bugs induced by variability, occurring in real large-scale systems. Such a collection of bugs is a requirement for goal-oriented research, serving to evaluate tool implementations of feature-sensitive analyses by testing them on real bugs. We present a qualitative study of 42 variability bugs collected from bug-fixing commits to the Linux kernel repository. We analyze each of the bugs, and record the results in a database. In addition, we provide self-contained simplified C99 versions of the bugs, facilitating understanding and tool evaluation. Our study provides insights into the nature and occurrence of variability bugs in a large C software system, and shows in what ways variability affects and increases the complexity of software bugs.

Features in a configurable system interact in non-trivial ways, in order to influence each others functionality. When such interactions are unintended, they induce bugs that manifest themselves in certain configurations but not in others, or that manifest differently in different configurations. A bug in an individual configuration may be found by analyzers based on standard program analysis techniques. However, since the number of configurations is exponential in the number of features, it is not feasible to analyze each configuration separately.

Family-based [33] analyses, a form of feature-sensitive analyses, tackle this problem by considering all configurable program variants as a single unit of analysis, instead of analyzing the individual variants separately. In order to avoid duplication of effort, common parts are analyzed once and the analysis forks only at differences between variants. Recently, various family-based extensions of both classic static

Real-world variability bugs happen

Existing bug-finders mostly punt on variability

42 Variability Bugs in the Linux Kernel: A Qualitative Analysis

Iago Abal
iago@itu.dk

Claus Brabrand
brabrand@itu.dk

Andrzej Wąsowski
wasowski@itu.dk

IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

ABSTRACT

Feature-sensitive verification pursues effective analysis of the exponentially many variants of a program family. However, researchers lack examples of concrete bugs induced by variability, occurring in real large-scale systems. Such a collection of bugs is a requirement for goal-oriented research, serving to evaluate tool implementations of feature-sensitive analyses by testing them on real bugs. We present a qualitative study of 42 variability bugs collected from bug-fixing commits to the Linux kernel repository. We analyze each of the bugs, and record the results in a database. In addition, we provide self-contained simplified C99 versions of the bugs, facilitating understanding and tool evaluation. Our study provides insights into the nature and occurrence of variability bugs in a large C software system, and shows in what ways variability affects and increases the complexity of software bugs.

Features in a configurable system interact in non-trivial ways, in order to influence each others functionality. When such interactions are unintended, they induce bugs that manifest themselves in certain configurations but not in others, or that manifest differently in different configurations. A bug in an individual configuration may be found by analyzers based on standard program analysis techniques. However, since the number of configurations is exponential in the number of features, it is not feasible to analyze each configuration separately.

Family-based [33] analyses, a form of feature-sensitive analyses, tackle this problem by considering all configurable program variants as a single unit of analysis, instead of analyzing the individual variants separately. In order to avoid duplication of effort, common parts are analyzed once and the analysis forks only at differences between variants. Recently, various family-based extensions of both classic static

Real-world variability bugs happen

Existing bug-finders mostly punt on variability

Other tools: code browsers, verification, refactoring, …

Kmax: Analyzing the Linux Build System

NYU CS Technical Report TR2015-976

ABS

Large-
But su
build s
variabi
awaren
the fan
produc
ploy in
pilation
tools su
toring t
of the
system
inform
make e
and ho
variabi
empiri

SuperC: Parsing All of C by Taming the Preprocessor

Paul Gazzillo Robert Grimm

New York University

{gazzillo,rgrimm}@cs.nyu.edu

Abstract

C tools, such as source browsers, bug finders, and automated refactorings, need to process two languages: C itself and the preprocessor. The latter improves expressivity through file includes, macros, and static conditionals. But it operates only on tokens, making it hard to even parse both languages. This paper presents a complete, performant solution to this problem. First, a configuration-preserving preprocessor resolves includes and macros yet leaves static conditionals intact, thus preserving a program's variability. To ensure completeness, we analyze all interactions between preprocessor features and identify techniques for correctly handling them. Second, a configuration-preserving parser generates a well-formed AST with static choice nodes for conditionals. It forks new subparsers when encountering static conditionals and merges them again after the conditionals. To ensure performance, we present a simple algorithm for table-driven Fork-Merge LR parsing and four novel optimizations. We demonstrate the effectiveness of our ap-

to C constructs and operates only on individual tokens. Real-world C code reflects both points: preprocessor usage is widespread and often violates C syntax [14].

Existing C tools punt on the full complexity of processing both languages. They either process one configuration at a time (e.g., the Cxref source browser [8], the Astrée bug finder [9], and Xcode refactorings [10]), rely on a single, maximal configuration (e.g., the Coverity bug finder [6]), or build on incomplete heuristics (e.g., the LXR source browser [20] and Eclipse refactorings [21]). Processing one configuration at a time is infeasible for large programs such as Linux, which has over 10,000 configuration variables [38]. Maximal configurations cover only part of the source code, mainly due to static conditionals with more than one branch. For example, Linux' `allyesconfig` enables less than 80% of the code blocks contained in conditionals [37]. And heuristic algorithms prevent programmers from utilizing the full expressivity of C and its preprocessor. Most research focused on parsing the two

Kmax: Analyzing the Linux Build System

NYU CS Technical Report TR2015-976

SuperC: Parsing All of C by Taming the Preprocessor

Paul Gazzillo Robert Grimm

{gazzillo,rgrimm}@cs.nyu.edu

New algorithms
Preprocessing and parsing
all configurations of C files

Abstract

C tools, such as source browse... torings, need to process two la... sor. The latter improves expressivity through file includes, macros, and static conditionals. But it operates only on tokens, making it hard to even parse both languages. This paper presents a complete, performant solution to this problem. First, a configuration-preserving preprocessor resolves includes and macros yet leaves static conditionals intact, thus preserving a program's variability. To ensure completeness, we analyze all interactions between preprocessor features and identify techniques for correctly handling them. Second, a configuration-preserving parser generates a well-formed AST with static choice nodes for conditionals. It forks new subparsers when encountering static conditionals and merges them again after the conditionals. To ensure performance, we present a simple algorithm for table-driven Fork-Merge LR parsing and four novel optimizations. We demonstrate the effectiveness of our ap-

...dividual tokens. Real-world C code ...r usage is widespread and often violates C syntax [14].

Existing C tools punt on the full complexity of processing both languages. They either process one configuration at a time (e.g., the Cxref source browser [8], the Astrée bug finder [9], and Xcode refactorings [10]), rely on a single, maximal configuration (e.g., the Coverity bug finder [6]), or build on incomplete heuristics (e.g., the LXR source browser [20] and Eclipse refactorings [21]). Processing one configuration at a time is infeasible for large programs such as Linux, which has over 10,000 configuration variables [38]. Maximal configurations cover only part of the source code, mainly due to static conditionals with more than one branch. For example, Linux' `allyesconfig` enables less than 80% of the code blocks contained in conditionals [37]. And heuristic algorithms prevent programmers from utilizing the full expressivity of C and its preprocessor. Most research focused on parsing the two

Kmax: Analyzing the Linux Build System

NYU CS Technical Report TR2015-976

New program analysis
algorithm for Makefiles

SuperC: Parsing All of C by Taming the Preprocessor

Paul Gazzillo Robert Grimm
New York University
{gazzillo,rgrimm}@cs.nyu.edu

New algorithms
Preprocessing and parsing
all configurations of C files

Abstract

C tools, such as source browsers, bug finders, and automated refactorings, need to process two languages: C itself and the preprocessor. The latter improves expressivity through file includes, macros, and static conditionals. But it operates only on tokens, making it hard to even parse both languages. This paper presents a complete, performant solution to this problem. First, a configuration-preserving preprocessor resolves includes and macros yet leaves static conditionals intact, thus preserving a program's variability. To ensure completeness, we analyze all interactions between preprocessor features and identify techniques for correctly handling them. Second, a configuration-preserving parser generates a well-formed AST with static choice nodes for conditionals. It forks new subparsers when encountering static conditionals and merges them again after the conditionals. To ensure performance, we present a simple algorithm for table-driven Fork-Merge LR parsing and four novel optimizations. We demonstrate the effectiveness of our ap-

to C constructs and operate only on individual tokens. Real-world C code reflects both points: preprocessor usage is widespread and often violates C syntax [14].

Existing C tools punt on the full complexity of processing both languages. They either process one configuration at a time (e.g., the Cxref source browser [8], the Astrée bug finder [9], and Xcode refactorings [10]), rely on a single, maximal configuration (e.g., the Coverity bug finder [6]), or build on incomplete heuristics (e.g., the LXR source browser [20] and Eclipse refactorings [21]). Processing one configuration at a time is infeasible for large programs such as Linux, which has over 10,000 configuration variables [38]. Maximal configurations cover only part of the source code, mainly due to static conditionals with more than one branch. For example, Linux' `allyesconfig` enables less than 80% of the code blocks contained in conditionals [37]. And heuristic algorithms prevent programmers from utilizing the full expressivity of C and its preprocessor. Most research focused on parsing the two

Main Idea

- Process all configurations, i.e., combinations of features
- Localize variability to avoid combinatorial explosion
- Exploit sharing between configurations

C Preprocessor Constructs

Object-like macros

Function-like macros

Macro definitions

Static conditionals

Conditional expressions

Includes

Stringification

Token-pasting

C … nstructs

```
#ifdef CONFIG_64BIT
#  define BITS_PER_LONG 64
#else
#  define BITS_PER_LONG 32
#endif
```

Function-like macros

Macro definitions

Static conditionals

Conditional expressions

Includes

Stringification

Token-pasting

C nstructs

```
#ifdef CONFIG_64BIT
#  define BITS_PER_LONG 64
#else
#  define BITS_PER_LONG 32
#endif
```

Function-like macros

Macro definitions

Static

```
__le ## 32
➜ __le32
```

Conditional expressions

Includes

Stringification

Token-pasting

Conditionals Need Hoisting

```
__le ## BITS_PER_LONG
```

Conditio … Hoisting

Macro expands to conditional

```
__le ## BITS_PER_LONG
```

```
__le ##
 #ifdef CONFIG_64BIT
   64
 #else
   32
 #endif
```

Conditio Hoisting

Macro expands to conditional

```
le ## BITS_PER_LONG
```

One operator: Two operations

```
__le ##
 #ifdef CONFIG_64BIT
   64
 #else
   32
 #endif
```

Conditio… Hoisting

Macro expands to conditional

```
le ## BITS_PER_LONG
```

One operator: Two operations

Hoist conditional around token-paste

```
__le ##
 #ifdef CONFIG_64BIT
   64
 #else
   32
 #endif
```

```
#ifdef CONFIG_64BIT
  __le ## 64
#else
  __le ## 32
#endif
```

Parsing All Configurations

- *Forks* subparsers at conditionals
- *Merges* subparsers in the same state after conditionals
    - Joins AST subtrees with *static choice nodes*
    - Preserves mutually exclusive configurations

Fork-Merge Parsing in Action

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
  if (imajor(inode) == 10)
    i = 31;
  else
#endif
    i = iminor(inode) – 32;
if (i >= MOUSEDEV_MINORS) ...
```

(1) Fork subparsers on conditional

Merge Parsing in Action

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
  if (imajor(inode) == 10)
    i = 31;
  else
#endif
    i = iminor(inode) - 32;
if (i >= MOUSEDEV_MINORS) ...
```

(1) Fork subparsers on conditional

…erge Parsi…on

(2) Parse the *entire* if-then-else

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
  if (imajor(inode) == 10)
    i = 31;
  else
#endif
    i = iminor(inode) - 32;
if (i >= MOUSEDEV_MINORS) ...
```

If-Then-Else

F…erge Parsi…on

(1) Fork subparsers on conditional

(2) Parse the *entire* if-then-else

(3) Parse *just* the assignment

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
  if (imajor(inode) == 10)
    i = 31;

#endif
    i = iminor(inode) - 32;
if (i >= MOUSEDEV_MINORS) ...
```

If-Then-Else

Assignment

...erge Parsi... on

(1) Fork subparsers on conditional

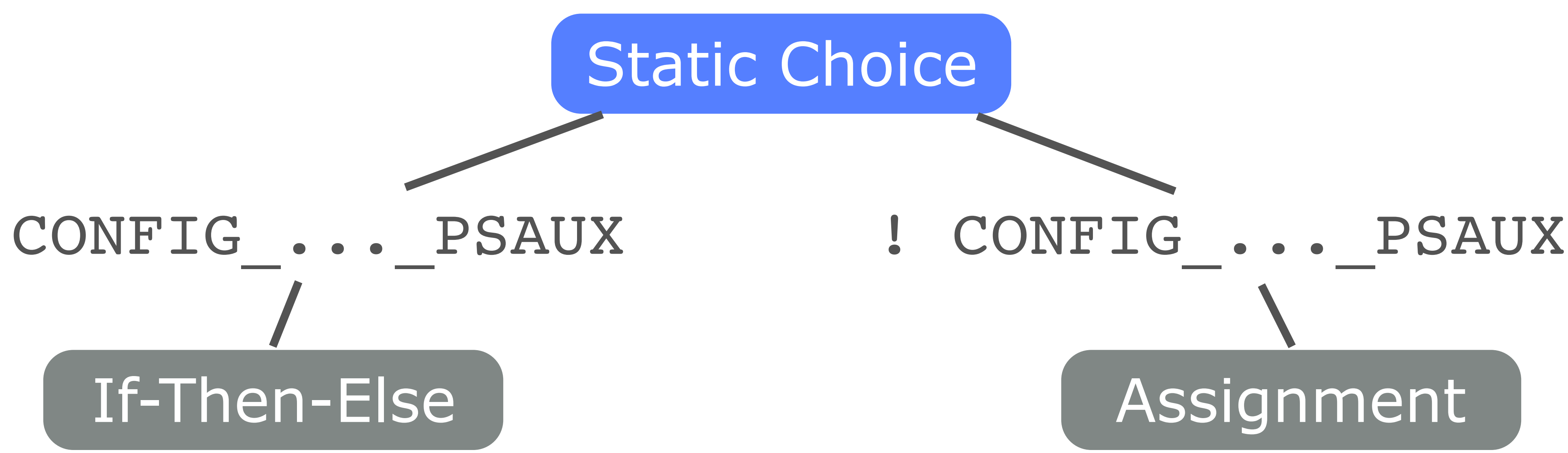
(2) Parse the *entire* if-then-else

(3) Parse *just* the assignment

(4) Merge and create the static choice node

```
#ifdef CONF..._INPUT_MOUSE..._PSAUX
  ... (imajor(inode) == 10)
  ... 31;

#e..dif
    i = i...nor(inode) ... ;
if (i >= MOUSEDEV_MINORS) ...
```



Results

- For full algorithms and results, see
  - Gazzillo and Grimm, PLDI 2012
  - Gazzillo, NYU Technical Report 2015
- SuperC
  - Parse entire Linux kernel, all configurations, pretty quickly
  - No more than 40 subparsers at any given time
- KMax
  - Collect all C file names and their configurations for Linux
  - Finds dead C files

What’s Next?

A Classification and Survey of Analysis Strategies for Software Product Lines

THOMAS THÜM, University of Magdeburg, Germany
SVEN APEL, University of Passau, Germany
CHRISTIAN KÄSTNER, Carnegie Mellon University, USA
INA SCHAEFER, University of Braunschweig, Germany
GUNTER SAAKE, University of Magdeburg, Germany

Software-product-line engineering has gained considerable momentum in the recent years, both in industry and in academia. A software product line is a family of software products that share a common set of features. Software product lines challenge traditional analysis techniques, such as type checking, model checking, and theorem proving, in their quest of ensuring correctness and reliability of software. Simply creating and analyzing all products of a product line is usually not feasible, due to the potentially exponential number of valid feature combinations. Recently, researchers began to develop analysis techniques that take the distinguishing properties of software product lines into account, for example, by checking feature-related code in isolation or by exploiting variability information during analysis. The emerging field of product-line analyses is both broad and diverse, so it is difficult for researchers and practitioners to understand their similarities and differences. We propose a classification of product-line analyses to enable systematic research and application. Based on our insights with classifying and comparing a corpus of 123 research articles, we develop a research agenda to guide future research on product-line analyses.

What’s Next?

Abstract interpretation

Alias analysis

Path-sensitivity

Context-sensitivity

Shape analysis

Interprocedural analyses

Side-channel attack freedom

Verification

Termination provers

What About Better Languages?

ASTEC: A New Approach to Refactoring C

Bill McCloskey Eric Brewer
Computer Science Division, EECS
University of California, Berkeley

ABSTR
The C lang
ticularly fo
upon macro
are difficult
that analyze
port for the
difficulty re
placement m
tant importa
many of the
converts exi
ASTEC-awa
naturally.
ASTEC's
bility. We p

A Variability-Aware Module System

Christian Kästner Klaus Ostermann Sebastian Erdweg
Philipps University Marburg, Germany

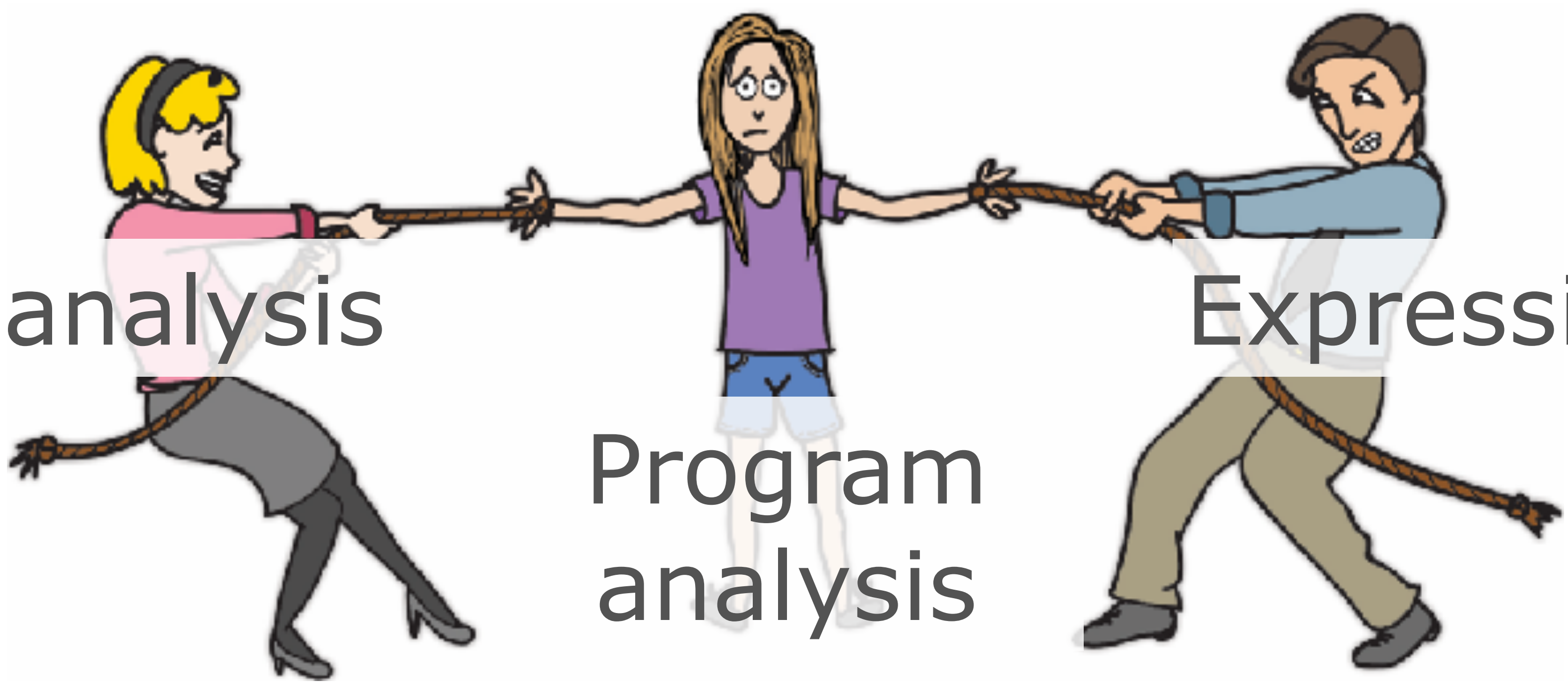
Abstract
Module systems
software developm
in software prod
different combinat
a dominant decom
variability-aware m
variability *inside* a
can be considered
in isolation. Varia
module system b
global variability
provides a path t

Effective Analysis of C Programs by Rewriting Variability

Alexandru F. Iosif-Lazar[a], Jean Melo[a], Aleksandar S. Dimovski[a], Claus Brabrand[a], and Andrzej Wąsowski[a]
a IT University of Copenhagen, Denmark

Abstract Context. Variability-intensive programs (program families) appear in many application areas
for many reasons today. Different family members, called variants, are derived by switching statically co
urable options (features) on and off, while reuse of the common code is maximized.
Inquiry. Verification of program families is challenging since the number of variants is exponential in
number of features. Existing single-program analysis and verification tools cannot be applied directly to
gram families, and designing and implementing the corresponding variability-aware versions is tedious
laborious.
Approach. In this work, we propose a range of variability-related transformations for translating program

Replacing the Preprocessor

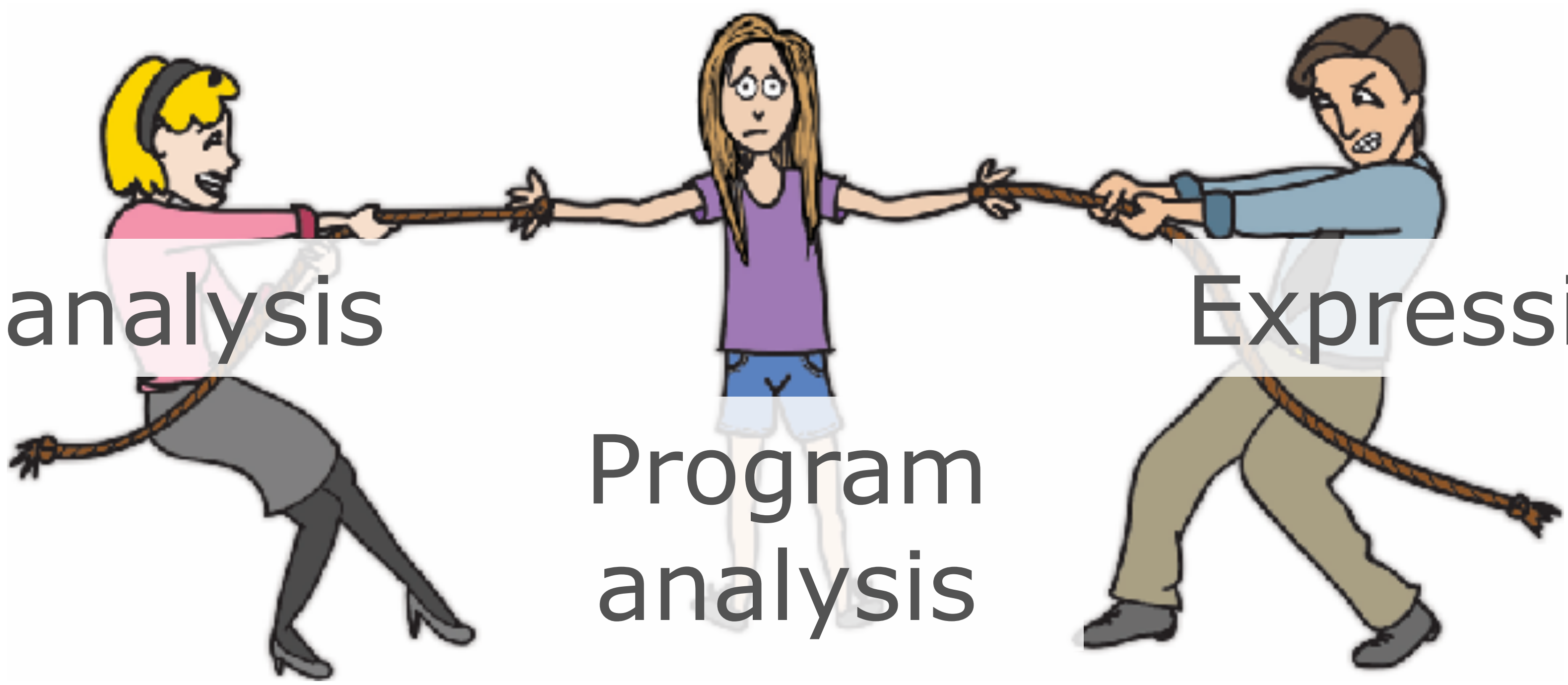


Ease of analysis

Expressivity

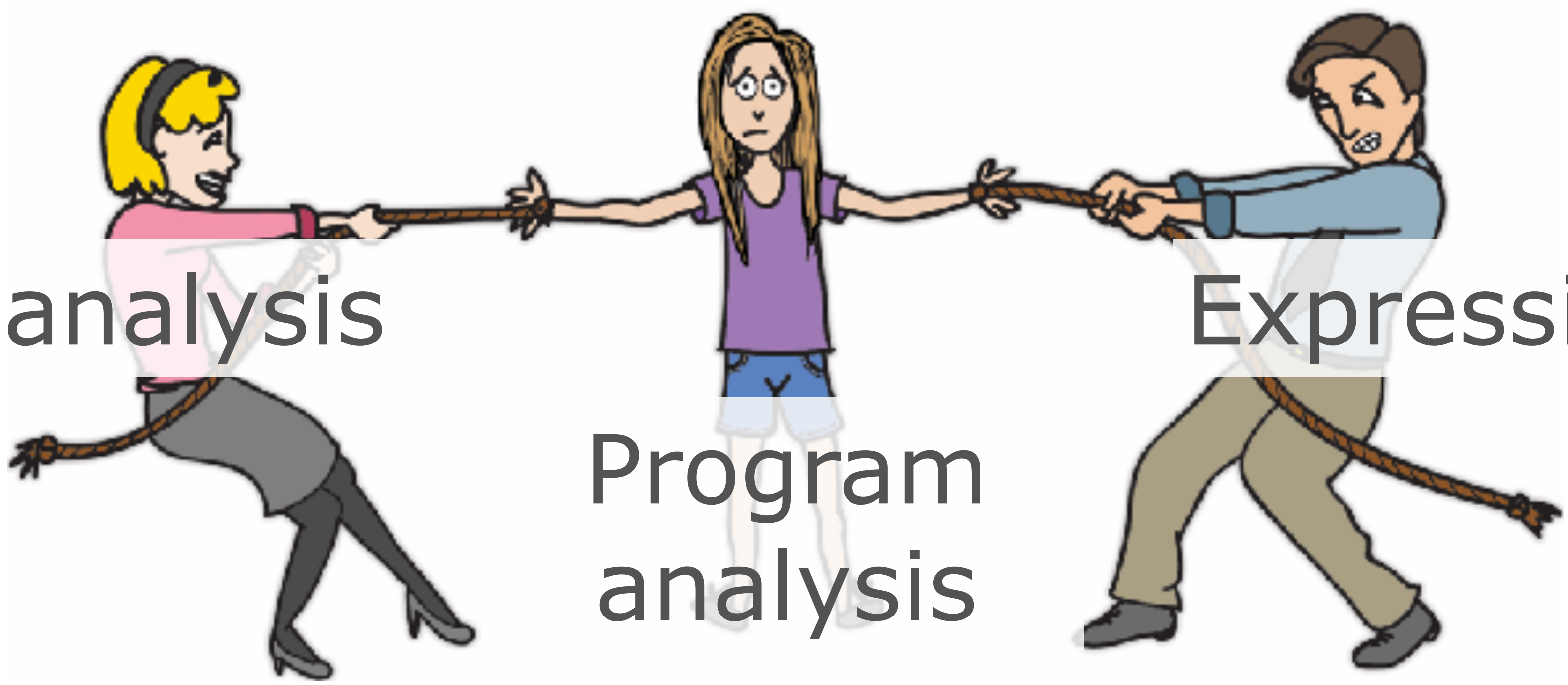
Program analysis

Replacing the Preprocessor



Not mutually exclusive

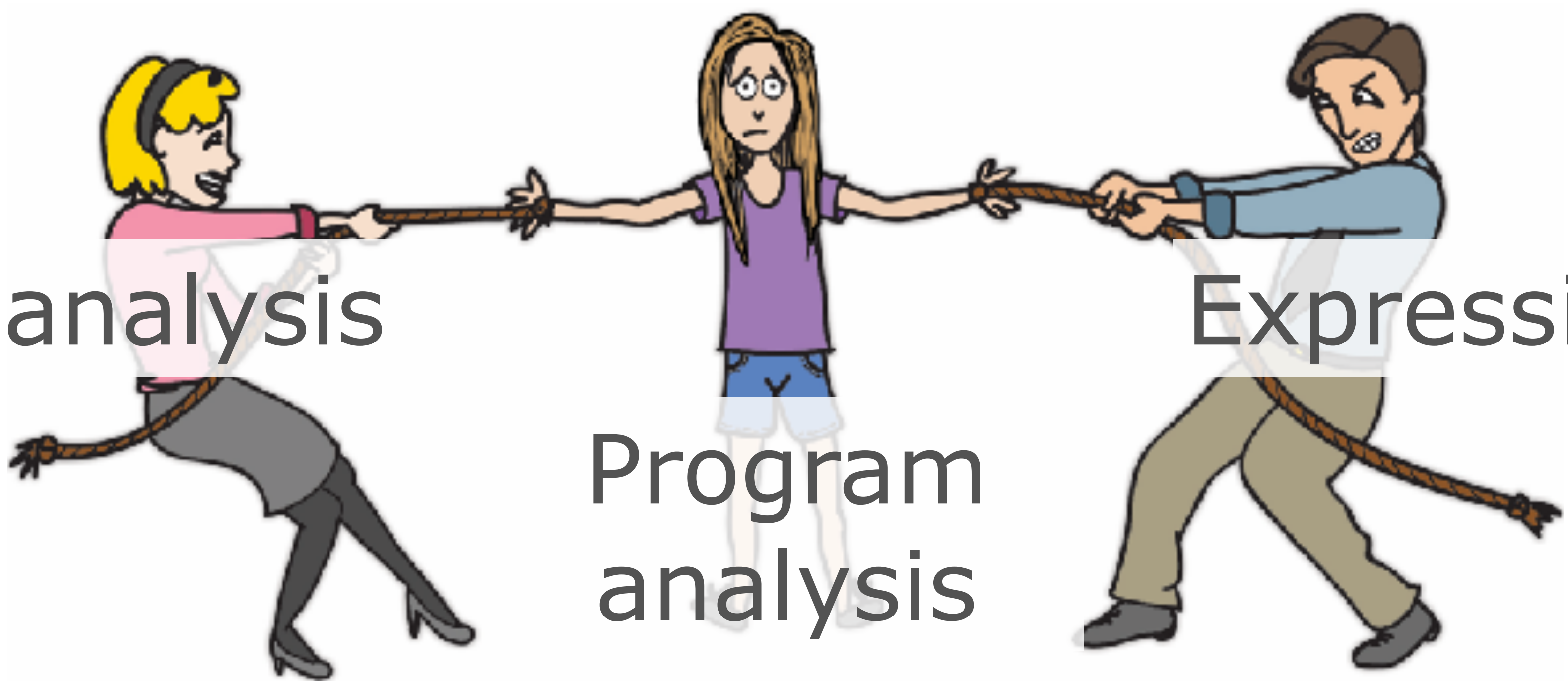
Replacing the Preprocessor



Not mutually exclusive

Developer inertia vs fundamental limitations

Replacing the Preprocessor



Ease of analysis

Expressivity

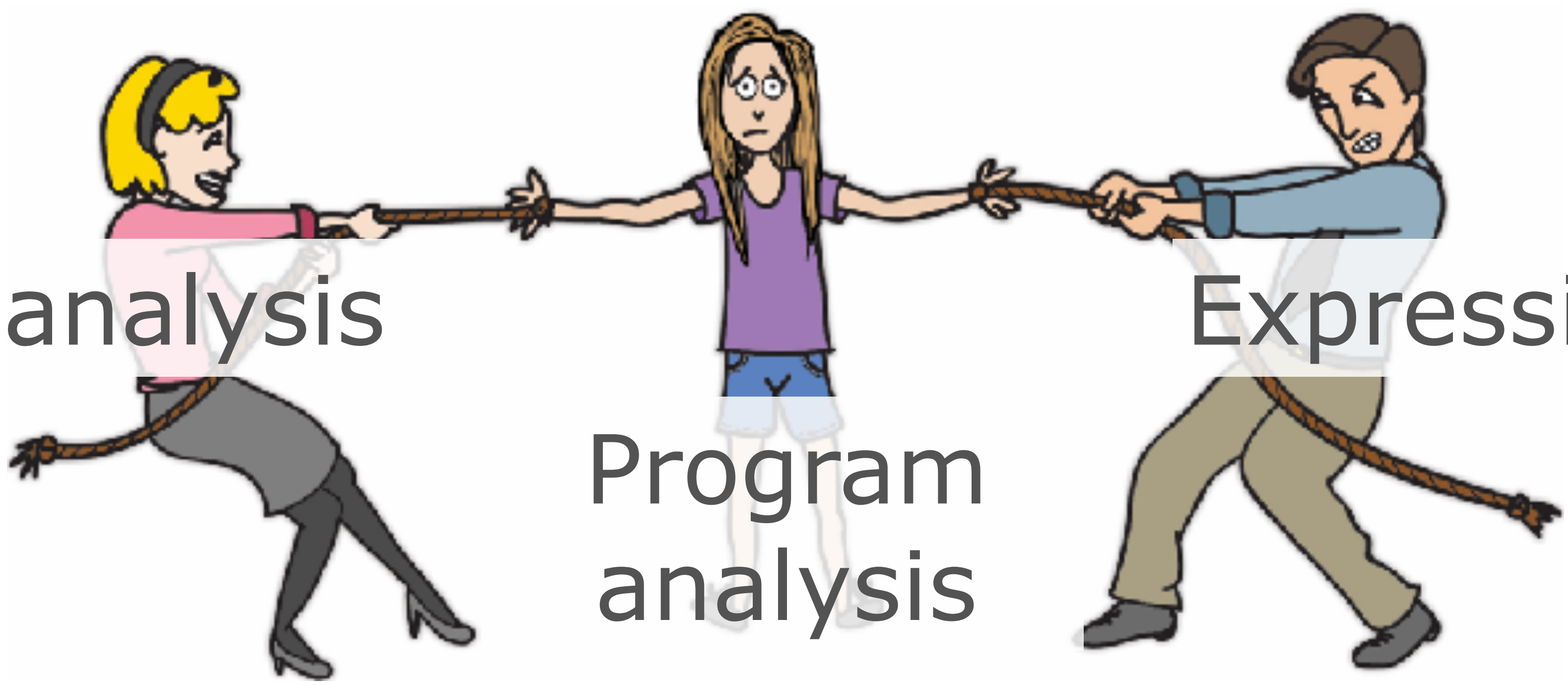
Program analysis

Not mutually exclusive

Developer inertia vs fundamental limitations

Human studies

Replacing the Preprocessor



Not mutually exclusive

Developer inertia vs fundamental limitations

Human studies

Adoption involves non-technical problems

Web: paulgazzillo.com

Twitter: @paul_gazzillo

Tools: https://github.com/paulgazz/xtc

src/xtc/lang/cpp (SuperC)

src/xtc/lang/cpp/kmax (Kmax)