Tom Was Right: Integration is Hard

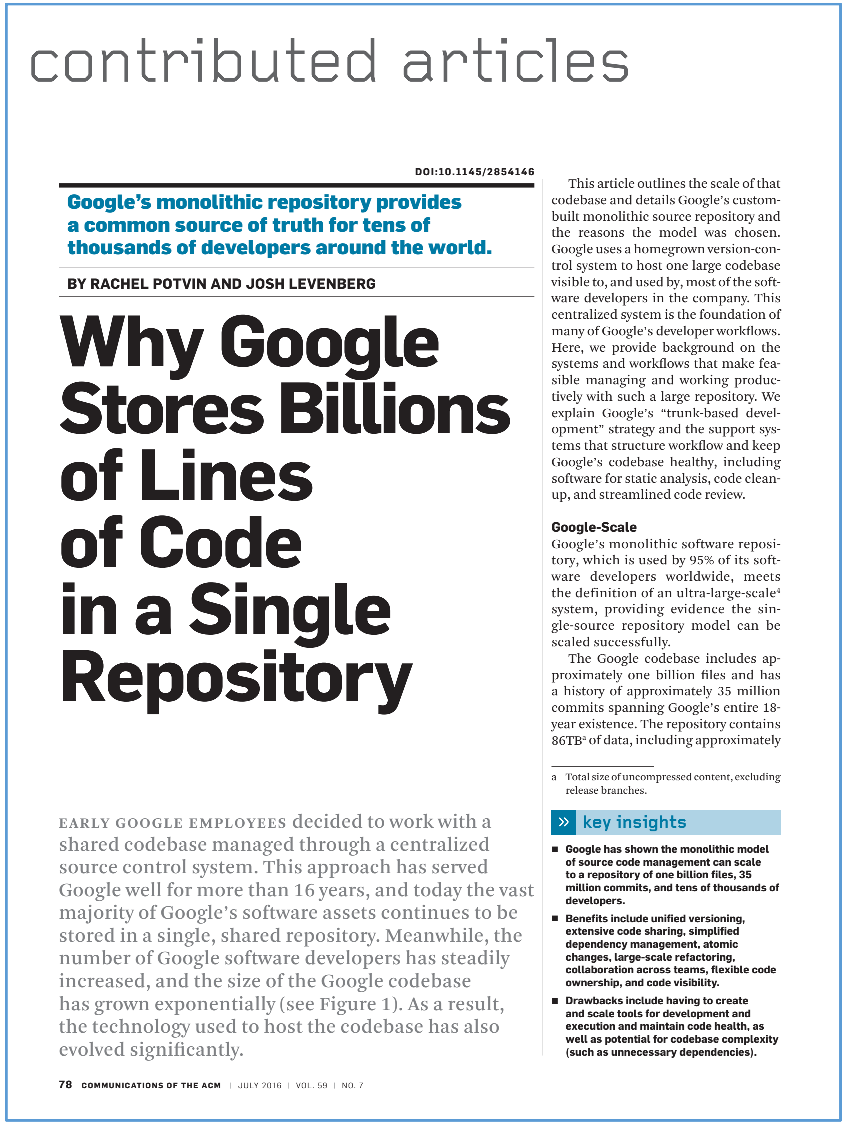
James Larus

EPFL

Reps at 60

September 11, 2016

Reading this Paper



contributed articles

DOI:10.1145/2854146

Google's monolithic repository provides a common source of truth for tens of thousands of developers around the world.

BY RACHEL POTVIN AND JOSH LEVENBERG

Why Google Stores Billions of Lines of Code in a Single Repository

EARLY GOOGLE EMPLOYEES decided to work with a shared codebase managed through a centralized source control system. This approach has served Google well for more than 16 years, and today the vast majority of Google's software assets continues to be stored in a single, shared repository. Meanwhile, the number of Google software developers has steadily increased, and the size of the Google codebase has grown exponentially (see Figure 1). As a result, the technology used to host the codebase has also evolved significantly.

This article outlines the scale of that codebase and details Google's custom-built monolithic source repository and the reasons the model was chosen. Google uses a homegrown version-control system to host one large codebase visible to, and used by, most of the software developers in the company. This centralized system is the foundation of many of Google's developer workflows. Here, we provide background on the systems and workflows that make feasible managing and working productively with such a large repository. We explain Google's "trunk-based development" strategy and the support systems that structure workflow and keep Google's codebase healthy, including software for static analysis, code cleanup, and streamlined code review.

Google-Scale

Google's monolithic software repository, which is used by 95% of its software developers worldwide, meets the definition of an ultra-large-scale[4] system, providing evidence the single-source repository model can be scaled successfully.

The Google codebase includes approximately one billion files and has a history of approximately 35 million commits spanning Google's entire 18-year existence. The repository contains 86TB[a] of data, including approximately

[a] Total size of uncompressed content, excluding release branches.

» key insights

- **Google has shown the monolithic model of source code management can scale to a repository of one billion files, 35 million commits, and tens of thousands of developers.**
- **Benefits include unified versioning, extensive code sharing, simplified dependency management, atomic changes, large-scale refactoring, collaboration across teams, flexible code ownership, and code visibility.**
- **Drawbacks include having to create and scale tools for development and execution and maintain code health, as well as potential for codebase complexity (such as unnecessary dependencies).**

78 COMMUNICATIONS OF THE ACM | JULY 2016 | VOL. 59 | NO. 7

R. Potvin and J. Levenberg, "Why Google Stores Billions of Lines of Code in a Single Repository,"
CACM, vol. 59, no. 7, pp. 78–87, June 2016.

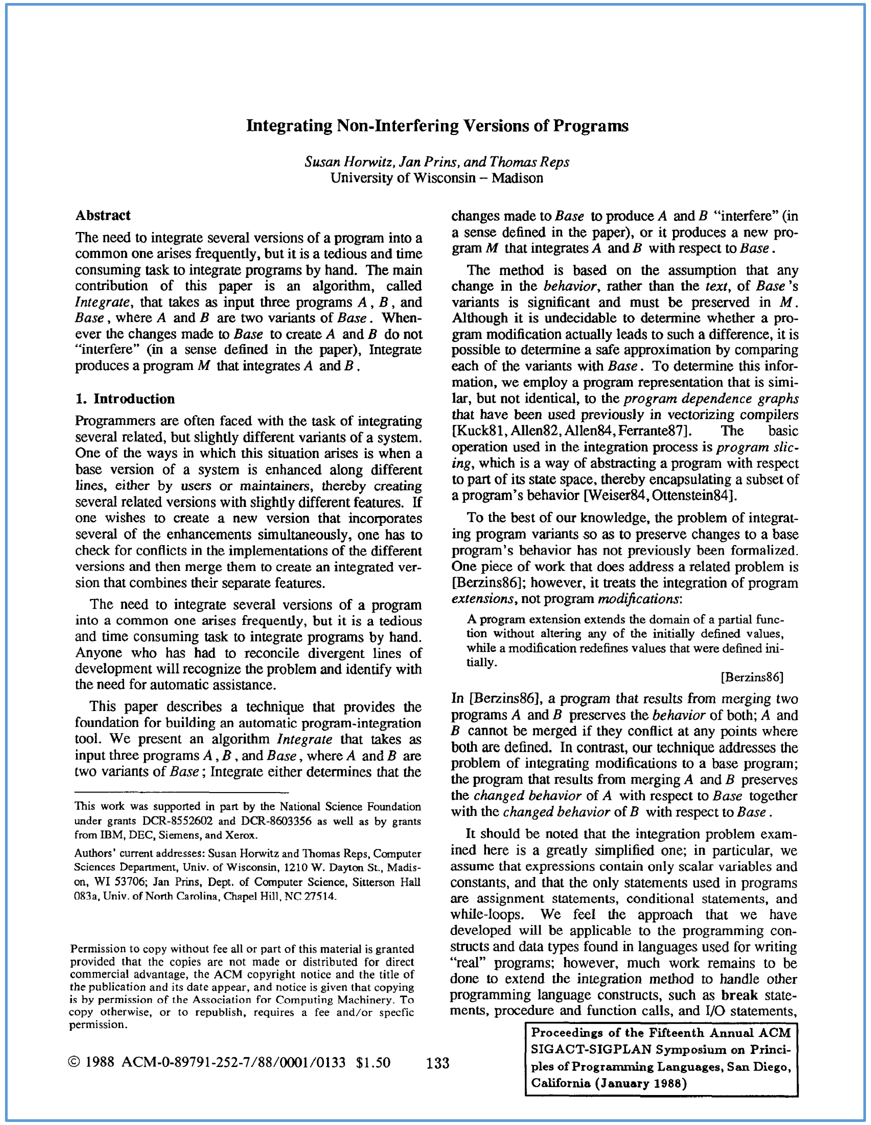
I Encountered These Sentences

Google practices trunk-based development on top of the Piper source repository. The vast majority of Piper users work at the “head,” or most recent, version of a single copy of the code called “trunk” or “mainline.”

…

Trunk-based development is beneficial in part because it avoids the painful merges that often occur when it is time to reconcile long-lived branches.

I Couldn't Help Thinking of This Paper



Integrating Non-Interfering Versions of Programs

Susan Horwitz, Jan Prins, and Thomas Reps
University of Wisconsin – Madison

Abstract

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to integrate programs by hand. The main contribution of this paper is an algorithm, called *Integrate*, that takes as input three programs *A* , *B* , and *Base* , where *A* and *B* are two variants of *Base* . Whenever the changes made to *Base* to create *A* and *B* do not "interfere" (in a sense defined in the paper), Integrate produces a program *M* that integrates *A* and *B* .

1. Introduction

Programmers are often faced with the task of integrating several related, but slightly different variants of a system. One of the ways in which this situation arises is when a base version of a system is enhanced along different lines, either by users or maintainers, thereby creating several related versions with slightly different features. If one wishes to create a new version that incorporates several of the enhancements simultaneously, one has to check for conflicts in the implementations of the different versions and then merge them to create an integrated version that combines their separate features.

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to integrate programs by hand. Anyone who has had to reconcile divergent lines of development will recognize the problem and identify with the need for automatic assistance.

This paper describes a technique that provides the foundation for building an automatic program-integration tool. We present an algorithm *Integrate* that takes as input three programs *A* , *B* , and *Base* , where *A* and *B* are two variants of *Base* ; Integrate either determines that the changes made to *Base* to produce *A* and *B* "interfere" (in a sense defined in the paper), or it produces a new program *M* that integrates *A* and *B* with respect to *Base* .

The method is based on the assumption that any change in the *behavior*, rather than the *text*, of *Base* 's variants is significant and must be preserved in *M* . Although it is undecidable to determine whether a program modification actually leads to such a difference, it is possible to determine a safe approximation by comparing each of the variants with *Base* . To determine this information, we employ a program representation that is similar, but not identical, to the *program dependence graphs* that have been used previously in vectorizing compilers [Kuck81, Allen82, Allen84, Ferrante87]. The basic operation used in the integration process is *program slicing*, which is a way of abstracting a program with respect to part of its state space, thereby encapsulating a subset of a program's behavior [Weiser84, Ottenstein84].

To the best of our knowledge, the problem of integrating program variants so as to preserve changes to a base program's behavior has not previously been formalized. One piece of work that does address a related problem is [Berzins86]; however, it treats the integration of program *extensions*, not program *modifications*:

> A program extension extends the domain of a partial function without altering any of the initially defined values, while a modification redefines values that were defined initially.
>
> [Berzins86]

In [Berzins86], a program that results from merging two programs *A* and *B* preserves the *behavior* of both; *A* and *B* cannot be merged if they conflict at any points where both are defined. In contrast, our technique addresses the problem of integrating modifications to a base program; the program that results from merging *A* and *B* preserves the *changed behavior* of *A* with respect to *Base* together with the *changed behavior* of *B* with respect to *Base* .

It should be noted that the integration problem examined here is a greatly simplified one; in particular, we assume that expressions contain only scalar variables and constants, and that the only statements used in programs are assignment statements, conditional statements, and while-loops. We feel the approach that we have developed will be applicable to the programming constructs and data types found in languages used for writing "real" programs; however, much work remains to be done to extend the integration method to handle other programming language constructs, such as **break** statements, procedure and function calls, and I/O statements,

This work was supported in part by the National Science Foundation under grants DCR-8552602 and DCR-8603356 as well as by grants from IBM, DEC, Siemens, and Xerox.

Authors' current addresses: Susan Horwitz and Thomas Reps, Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706; Jan Prins, Dept. of Computer Science, Sitterson Hall 083a, Univ. of North Carolina, Chapel Hill, NC 27514.

Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988)

S. Horwitz, J. Prins, and T. Reps, "Integrating Non-Interfering Versions of Programs," *Fifteenth ACM Symposium on Principles of Programming Languages*, 1988, pp. 133–145.

Which Starts…

Programmers are often faced with the task of integrating several related, but slightly different variants of a system. One of the ways in which this situation arises is when a base version of a system is enhanced along different lines, either by users or maintainers, thereby creating several related versions with slightly different features.

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to integrate programs by hand. Anyone who has had to reconcile divergent lines of development will recognize the problem and identify with the need for automatic assistance.

First Step in a Larger Project

Program slicing, differencing, merging, chopping, “etc.”

A Bit of History

Tom and Susan started at UW in 1985 as professors

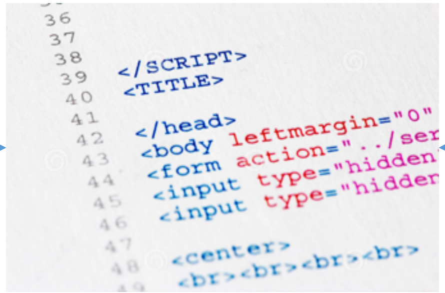
I joined UW in 1989 as a professor

Program slicing was Tom and Susan's big research project

I left UW in 1997 to join MSR

Program Integration

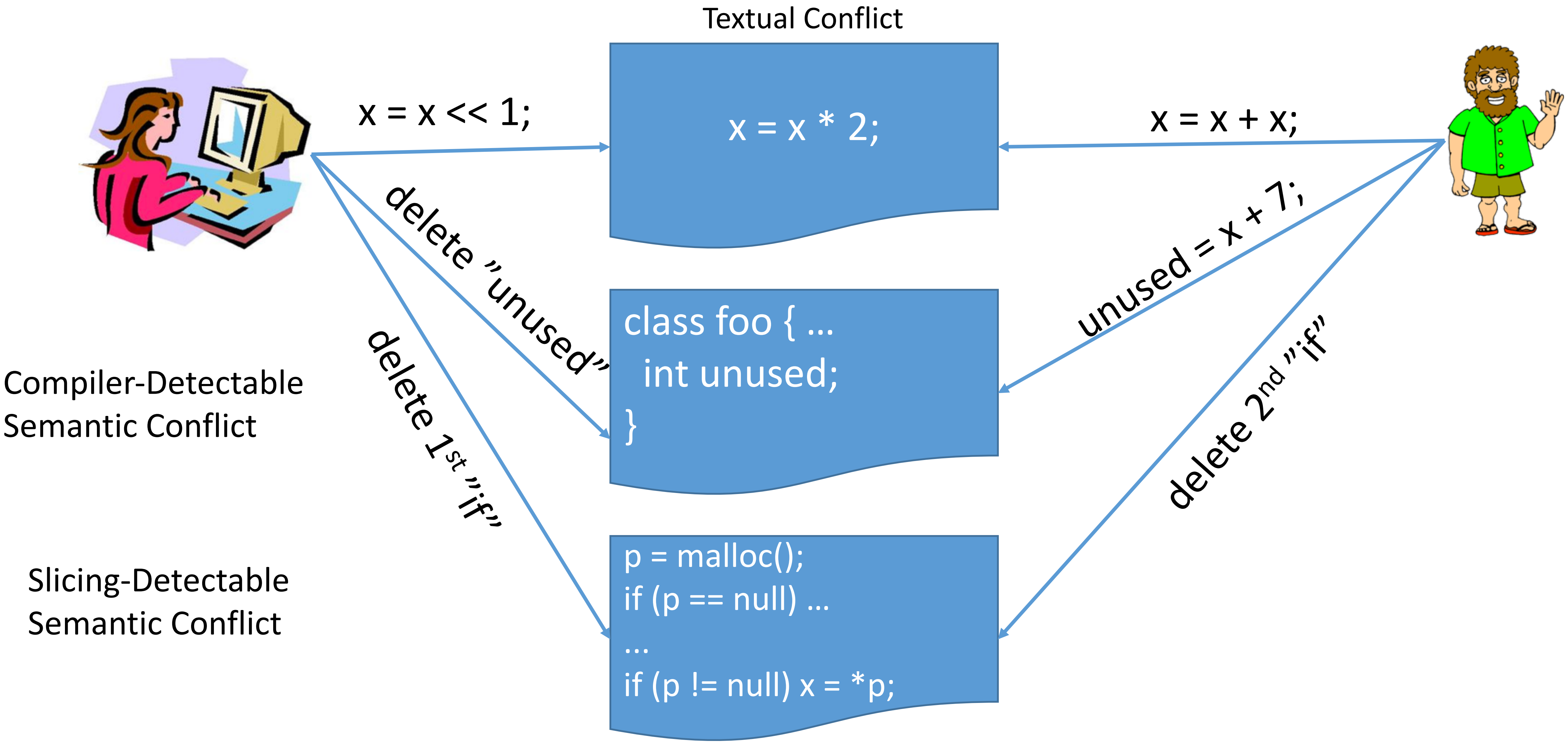
Alice and Bob make concurrent changes to code



```
</SCRIPT>
<TITLE>

</head>
<body leftmargin="0"
<form action="../ser
<input type="hidden
<input type="hidden

<center>
<br><br><br><br>
```

?

What Could Go Wrong?



Textual Conflict

x = x << 1;

```
x = x * 2;
```

x = x + x;

Compiler-Detectable Semantic Conflict

delete "unused"

```
class foo { …
  int unused;
}
```

unused = x + 7;

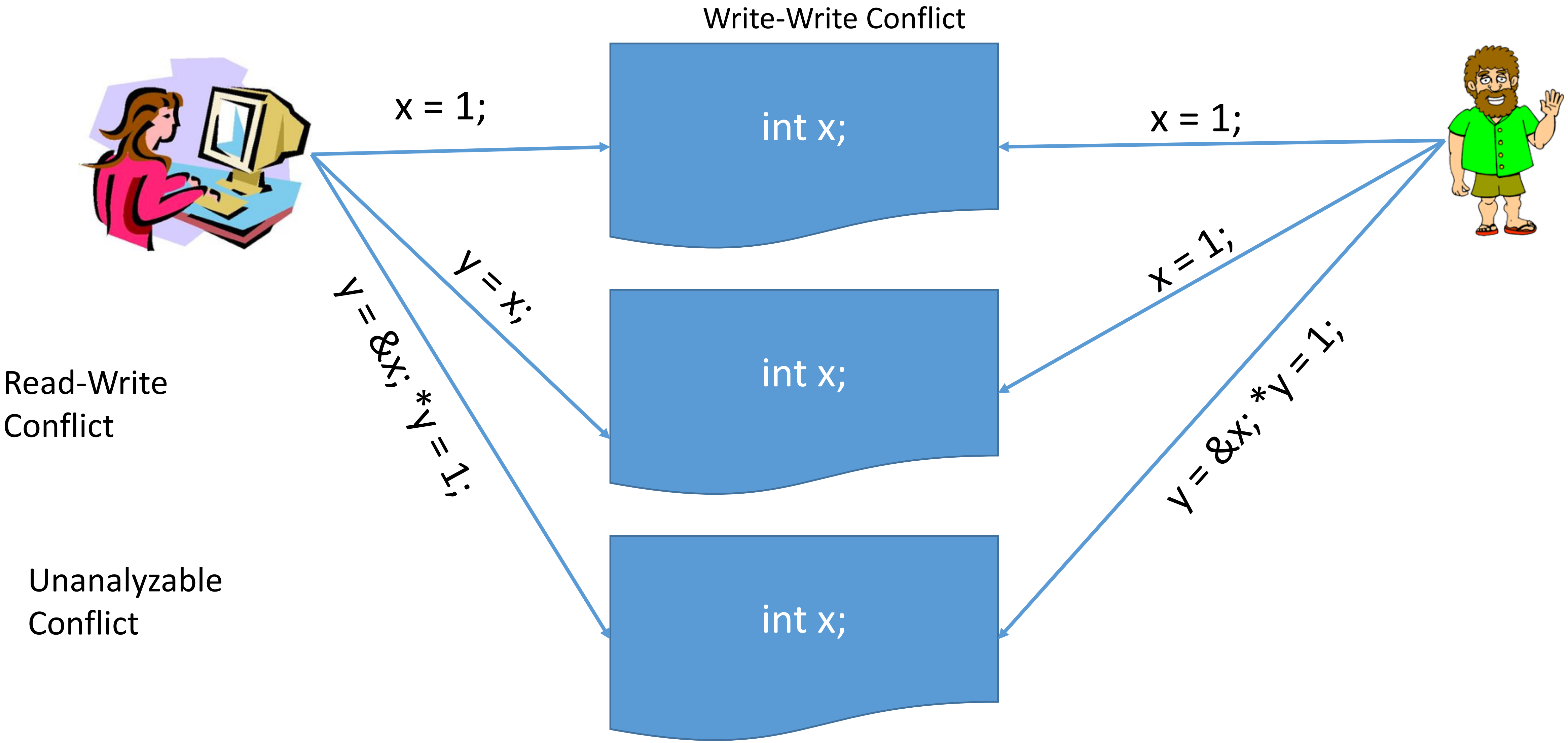
Slicing-Detectable Semantic Conflict

delete 1st "if"

```
p = malloc();
if (p == null) …
…
if (p != null) x = *p;
```

delete 2nd "if"

Analogous to Concurrent Update Problem



Write-Write Conflict

x = 1;

int x;

x = 1;

Read-Write Conflict

y = x;

int x;

x = 1;

Unanalyzable Conflict

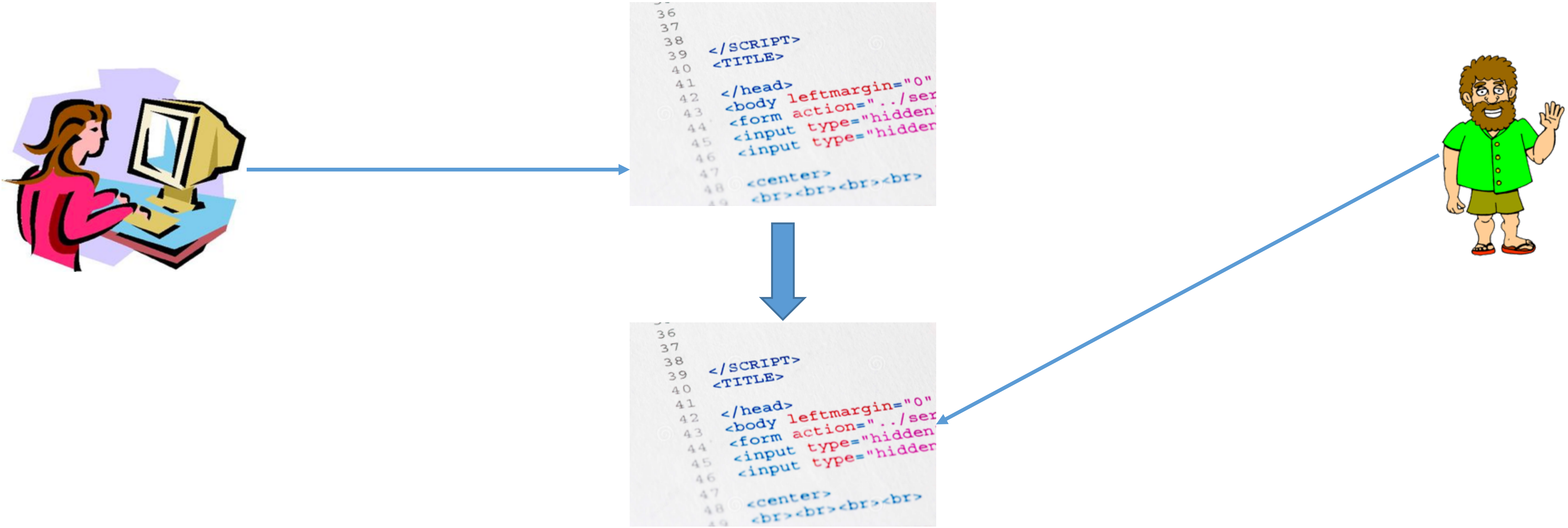
y = &x; *y = 1;

int x;

y = &x; *y = 1;

What To Do About This?

- Concurrency control
  - Use synchronization (lock or people) to serialize access



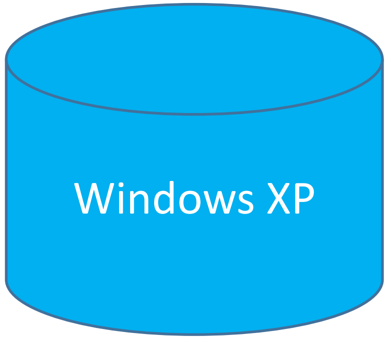
```
</SCRIPT>
<TITLE>

</head>
<body leftmargin="0"
<form action="../ser
<input type="hidden
<input type="hidder

<center>
<br><br><br><br>
```

```
</SCRIPT>
<TITLE>

</head>
<body leftmargin="0"
<form action="../ser
<input type="hidden
<input type="hidder

<center>
<br><br><br><br>
```

Microsoft, before Windows XP

SLM

(internal version of RCS)



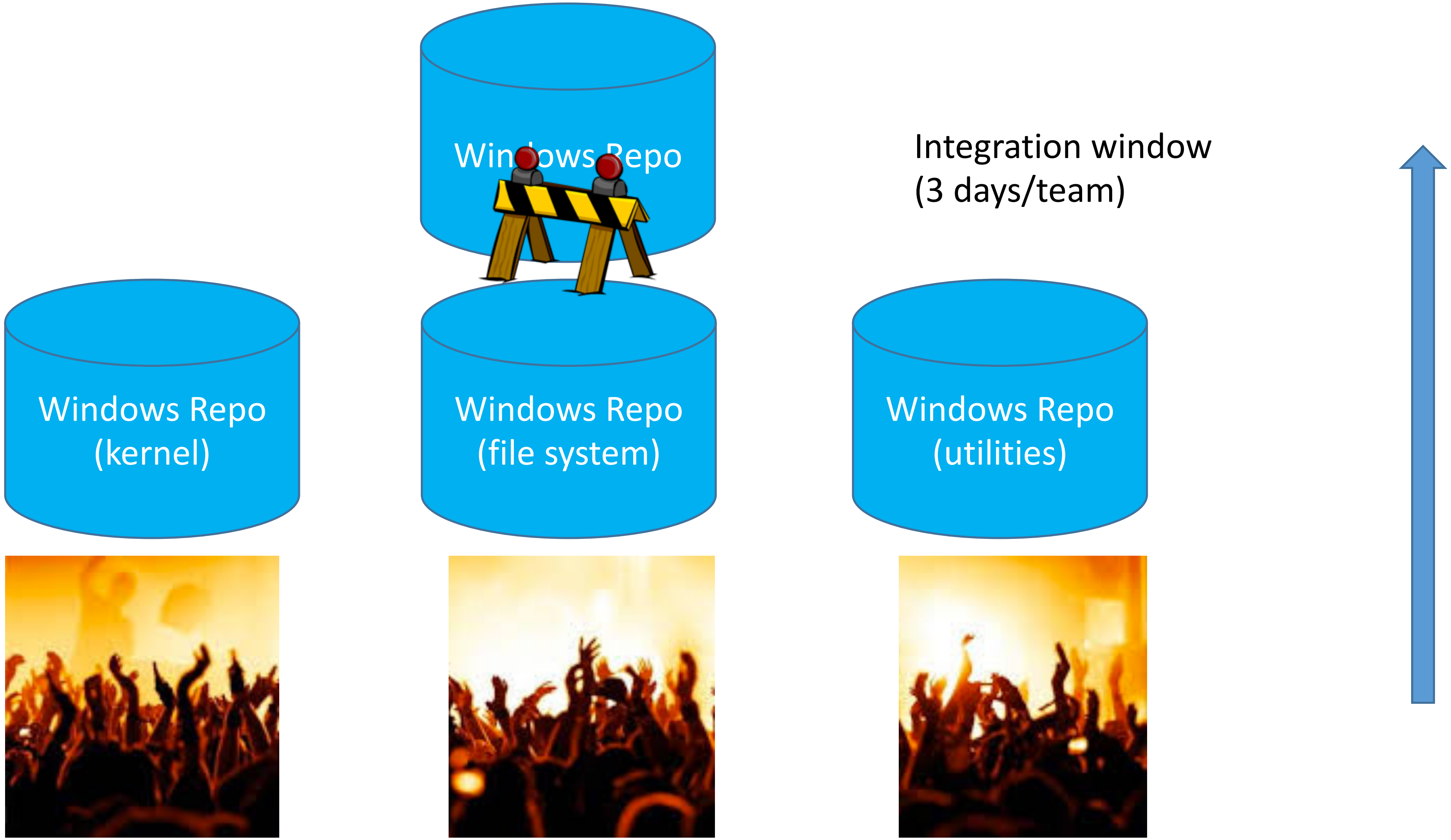
Windows XP

- RCS
  - “Revision Control System” (Walter Tichy)
  - State of the art ~1980s
  - Single-file locking
    - No atomic commits
  - Branching unusable

Microsoft, after Windows XP

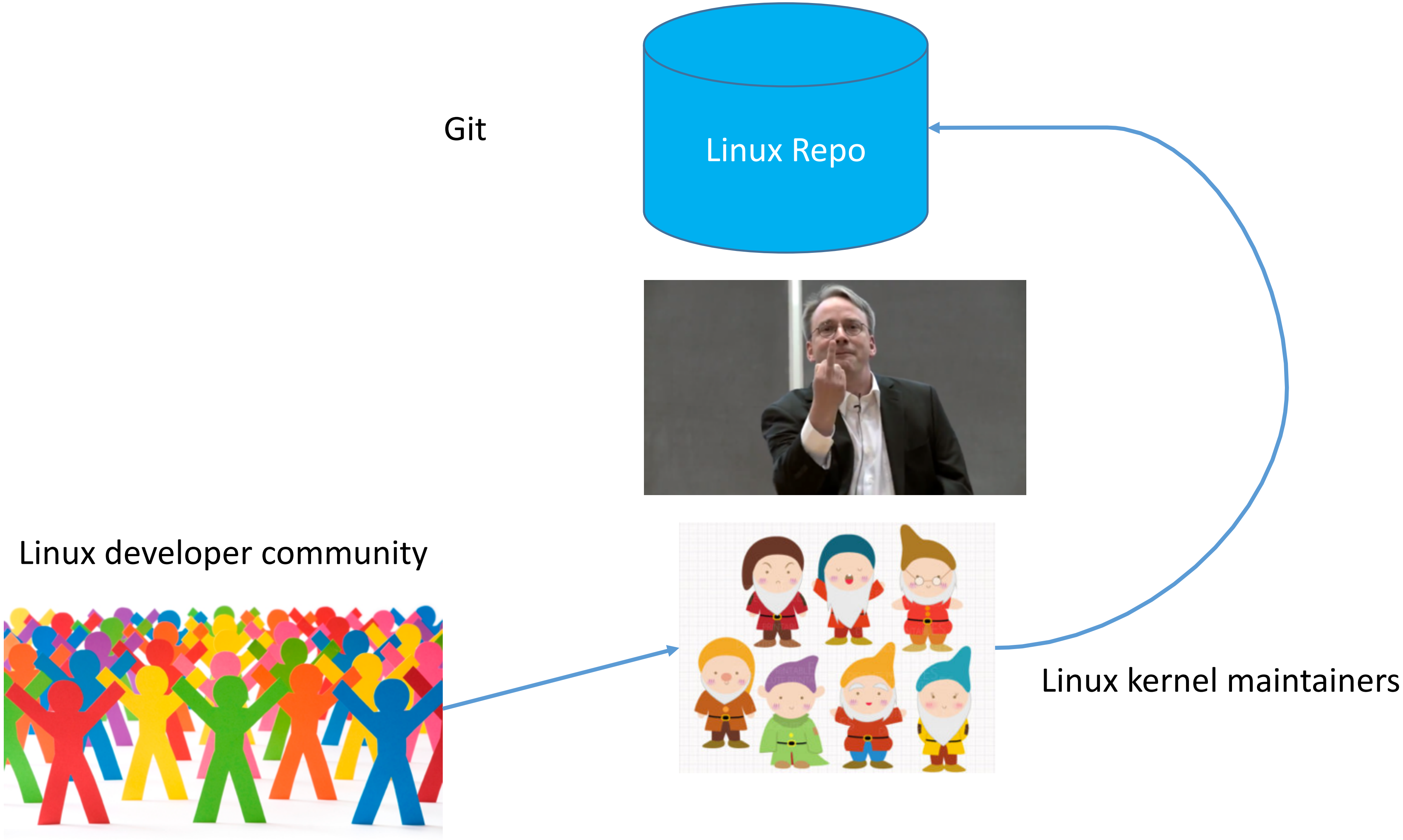
SourceDepot

Microsoft version of Perforce



Integration window
(3 days/team)

Linux



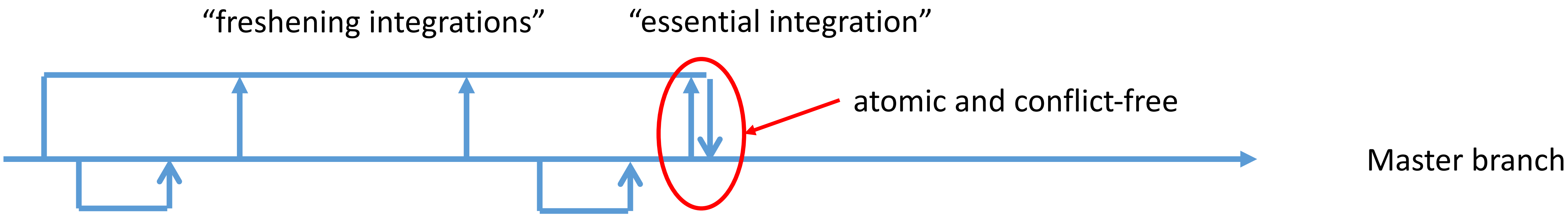
Git

Linux Repo

Linux developer community

Linux kernel maintainers

Modern (Git) Practice



Aside: closer to transactional memory (optimistic concurrency) than locking

How Do We Know Integration is Correct?

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTER-NOON!
S. Adams E-mail: SCOTTADAMS@AOL.COM
11/13 © 1995 United Feature Syndicate, Inc.(NYC)

Yes, software testing

Program Slicing

- Not used in practice
- Does not come up in discussions in SE

Testing ~= Semantic Analysis?

- When you can’t perform static analysis (too slow, too imprecise), then test
- In practice, testing is still the primary way developers find bugs and ensure code quality
    - Not the only way

What does this say about SW?
What does it say about Tom’s research?

1980s -> 2010s: Enormous Progress

1980s

- lint and compiler warnings were primary development tools
- Interprocedural analysis was not well understood or widely used
- Pointer analysis not well understood
- PDGs just published
- SAT was the canonical NP-complete problem
- 8MB was a large memory and 300MB was a large disk

2010s

- Large number of open source and commercial tools
  - MS: SLAM, Dafny, Z3, Pex, Slayer, …
  - Even some viable companies
- Program analysis well grounded and understood
  - Thanks Tom!
- Main memory in GB (heading to TB), disk in PB (heading to EB)
- Development tools still not well understood or widely used

Why Are Bugs Not Found By Tools?

- Legacy languages
- Legacy software
- Non-zero false positive / false negative rates
- Developers' inability to understand / write specifications
- Diversity of bugs
- Fixing existing code is a fool's errand

- Not because of immaturity of program analysis

Tom’s Role

- Tom’s contributions to program analysis are broad and fundamental
    - Interprocedural analysis
    - Pointer and shape analysis
    - Multi-threaded program analysis
    - Path problems
    - Model checking
    - …
- Without Tom’s research, the field of program analysis would be on a far shakier and less rigorous basis

Aside: Hope on the Horizon

- Verified software
  - CompCert (Inria), seL4 (UNSW), IronFleet (MSR)
- Co-design and verification

The ability to change and rearrange code in discussion with the design team (to predict performance impact) was an important factor in the verification team's productivity.

seL4: Formal Verification of an OS Kernel

Gerwin Klein[1,2], Kevin Elphinstone[1,2], Gernot Heiser[1,2,3]
June Andronick[1,2], David Cock[1], Philip Derrin[1*], Dhammika Elkaduwe[1,2‡], Kai Engelhardt[1,2]
Rafal Kolanski[1,2], Michael Norrish[1,4], Thomas Sewell[1], Harvey Tuch[1,2†], Simon Winwood[1,2]
[1] NICTA, [2] UNSW, [3] Open Kernel Labs, [4] ANU
ertos@nicta.com.au

ABSTRACT

Complete formal verification is the only known way to guarantee that a system is free of programming errors.

We present our experience in performing the formal, machine-checked verification of the seL4 microkernel from an abstract specification down to its C implementation. We assume correctness of compiler, assembly code, and hardware, and we used a unique design approach that fuses formal and operating systems techniques. To our knowledge, this is the first formal proof of functional correctness of a complete, general-purpose operating-system kernel. Functional correctness means here that the implementation always strictly follows our high-level abstract specification of kernel behaviour. This encompasses traditional design and implementation safety properties such as the kernel will never crash, and it will never perform an unsafe operation. It also proves much more: we can predict precisely how the kernel will behave in every possible situation.

seL4, a third-generation microkernel of L4 provenance, comprises 8,700 lines of C code and 600 lines of assembler. Its performance is comparable to other high-performance L4 kernels.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification, Design

Keywords

Isabelle/HOL, L4, microkernel, seL4

1. INTRODUCTION

The security and reliability of a computer system can only be as good as that of the underlying operating system (OS) kernel. The kernel, defined as the part of the system executing in the most privileged mode of the processor, has unlimited hardware access. Therefore, any fault in the kernel's implementation has the potential to undermine the correct operation of the rest of the system.

General wisdom has it that bugs in any sizeable code base are inevitable. As a consequence, when security or reliability is paramount, the usual approach is to reduce the amount of privileged code, in order to minimise the exposure to bugs. This is a primary motivation behind security kernels and separation kernels [38, 54], the MILS approach [4], microkernels [1, 12, 35, 45, 57, 71] and isolation kernels [69], the use of small hypervisors as a minimal trust base [16, 26, 56, 59], as well as systems that require the use of type-safe languages for all code except some "dirty" core [7,23]. Similarly, the Common Criteria [66] at the strictest evaluation level requires the system under evaluation to have a "simple" design.

With truly small kernels it becomes possible to take security and robustness further, to the point where it is possible to *guarantee* the absence of bugs [22, 36, 56, 64]. This can be achieved by *formal, machine-checked verification*, providing mathematical proof that the kernel implementation is consistent with its specification and free from programmer-induced implementation defects.

We present seL4, a member of the L4 [46] microkernel family, designed to provide this ultimate degree of assurance of functional correctness by machine-assisted and machine-checked formal proof. We have shown the correctness of a very detailed, low-level design of seL4 and we have formally verified its C implementation. We assume the correctness of the compiler, assembly code, boot code, management of caches, and the hardware; we prove everything else.

Specifically, seL4 achieves the following:

- it is suitable for real-life use, and able to achieve performance that is comparable with the best-performing microkernels;
- its behaviour is precisely formally specified at an abstract level;
- its formal design is used to prove desirable properties, including termination and execution safety;
- its implementation is formally proven to satisfy the specification; and

[*] Philip Derrin is now at Open Kernel Labs.
[†] Harvey Tuch is now at VMware.
[‡] Dhammika Elkaduwe is now at University of Peradeniya

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

G. Klein, et al., "SeL4: Formal Verification of an OS Kernel," *ACM SIGOPS 22nd SOSPiples*, 2009, pp. 207–220.

Conclusion

- Tom’s contributions to program analysis are broad and fundamental
- We aren’t there yet
  - But, we are a lot further along because Tom’s insights and contributions

- Tom’s many other virtues
  - Leader and role model in his field
  - Extraordinarily well-written papers
  - Excellent advisor with many talented and successful students and postdocs
  - Generous colleague

But

- Are you really 60?