Proving termination, and beyond

Byron Cook

Microsoft Research &

University College London

Talk to Byron after his talk - Lib 2612 3:15

Outline

- 10 years work on termination
- More broadly about me, my career, jobs, other things I work on
- Some notes on Evergreen & me
- Questions
  - though please interrupt when things are unclear!

Automatic formal verification

- View artifact of interest as a mathematical system:
  - Software
  - Hardware
  - Biological system
  - Railway switching
  - …….
- Automatically prove desired properties using mathematics and logic
- Status:
  - Active area of research in the 70s,
  - Dead in the 80s-90s,
  - Renaissance since 2000

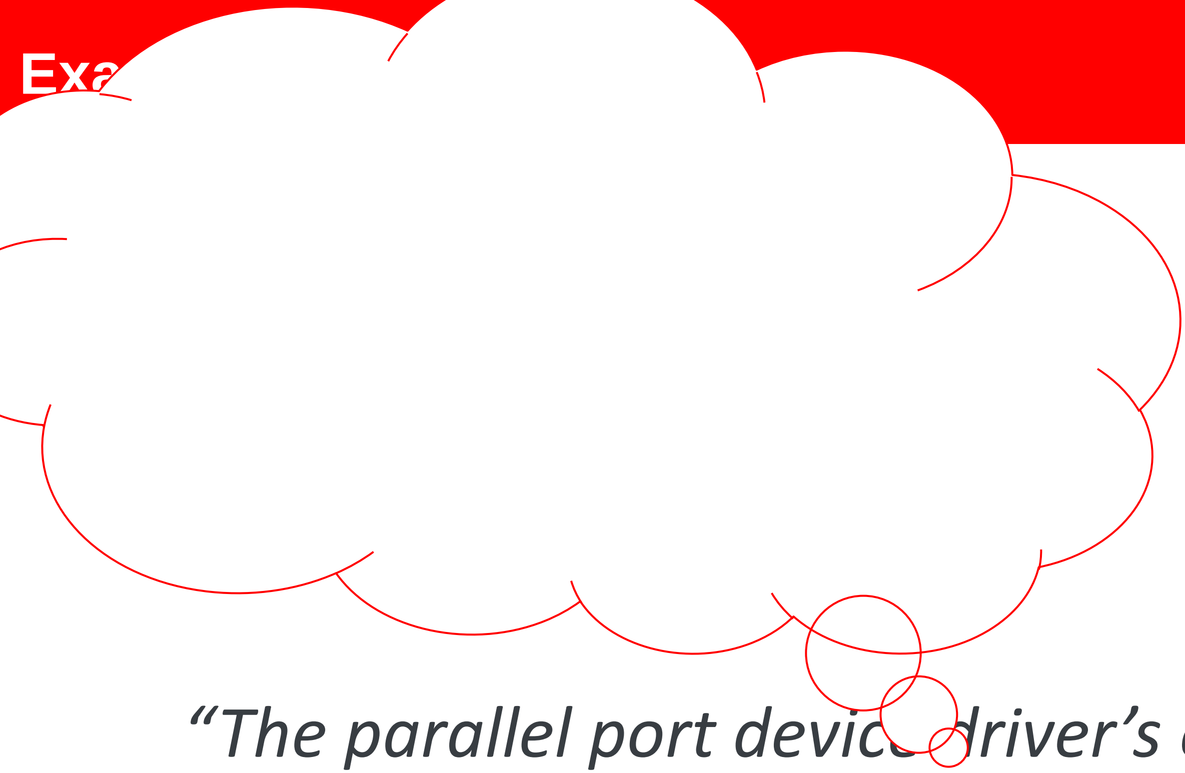
Example property

“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”

Example property

“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”

Exa

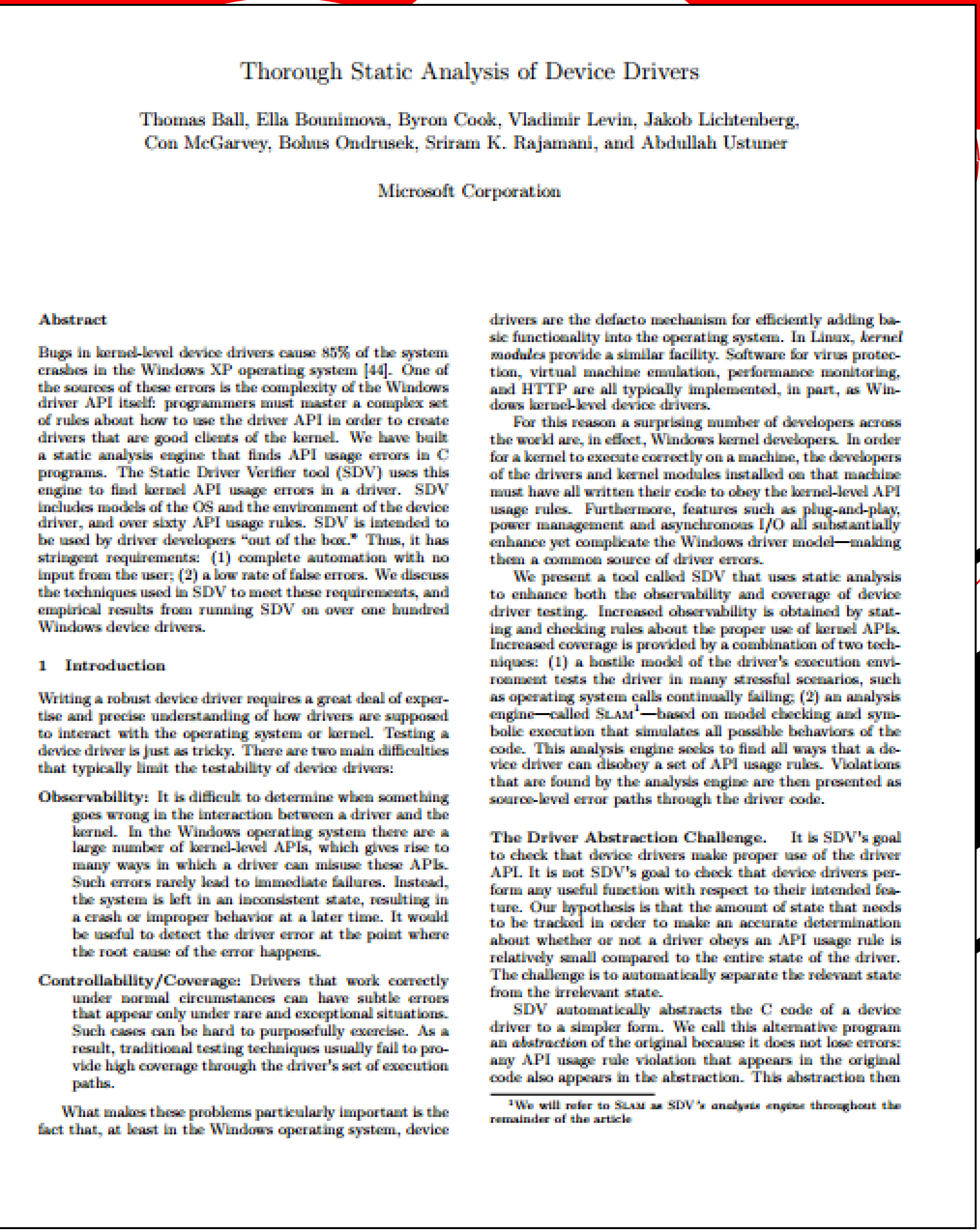


“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”

Exa

“The parallel port device driver’s event-handling routine only calls KeReleaseSpinLock() when IRQL=PASSIVE”

SLAM



Thorough Static Analysis of Device Drivers

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg,
Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner

Microsoft Corporation

Abstract

Bugs in kernel-level device drivers cause 85% of the system crashes in the Windows XP operating system [44]. One of the sources of these errors is the complexity of the Windows driver API itself: programmers must master a complex set of rules about how to use the driver API in order to create drivers that are good clients of the kernel. We have built a static analysis engine that finds API usage errors in C programs. The Static Driver Verifier tool (SDV) uses this engine to find kernel API usage errors in a driver. SDV includes models of the OS and the environment of the device driver, and over sixty API usage rules. SDV is intended to be used by driver developers "out of the box." Thus, it has stringent requirements: (1) complete automation with no input from the user; (2) a low rate of false errors. We discuss the techniques used in SDV to meet these requirements, and empirical results from running SDV on over one hundred Windows device drivers.

1 Introduction

Writing a robust device driver requires a great deal of expertise and precise understanding of how drivers are supposed to interact with the operating system or kernel. Testing a device driver is just as tricky. There are two main difficulties that typically limit the testability of device drivers:

Observability: It is difficult to determine when something goes wrong in the interaction between a driver and the kernel. In the Windows operating system there are a large number of kernel-level APIs, which gives rise to many ways in which a driver can misuse these APIs. Such errors rarely lead to immediate failures. Instead, the system is left in an inconsistent state, resulting in a crash or improper behavior at a later time. It would be useful to detect the driver error at the point where the root cause of the error happens.

Controllability/Coverage: Drivers that work correctly under normal circumstances can have subtle errors that appear only under rare and exceptional situations. Such cases can be hard to purposefully exercise. As a result, traditional testing techniques usually fail to provide high coverage through the driver's set of execution paths.

What makes these problems particularly important is the fact that, at least in the Windows operating system, device drivers are the defacto mechanism for efficiently adding basic functionality into the operating system. In Linux, *kernel modules* provide a similar facility. Software for virus protection, virtual machine emulation, performance monitoring, and HTTP are all typically implemented, in part, as Windows kernel-level device drivers.

For this reason a surprising number of developers across the world are, in effect, Windows kernel developers. In order for a kernel to execute correctly on a machine, the developers of the drivers and kernel modules installed on that machine must have all written their code to obey the kernel-level API usage rules. Furthermore, features such as plug-and-play, power management and asynchronous I/O all substantially enhance yet complicate the Windows driver model—making them a common source of driver errors.

We present a tool called SDV that uses static analysis to enhance both the observability and coverage of device driver testing. Increased observability is obtained by stating and checking rules about the proper use of kernel APIs. Increased coverage is provided by a combination of two techniques: (1) a hostile model of the driver's execution environment tests the driver in many stressful scenarios, such as operating system calls continually failing; (2) an analysis engine—called SLAM[1]—based on model checking and symbolic execution that simulates all possible behaviors of the code. This analysis engine seeks to find all ways that a device driver can disobey a set of API usage rules. Violations that are found by the analysis engine are then presented as source-level error paths through the driver code.

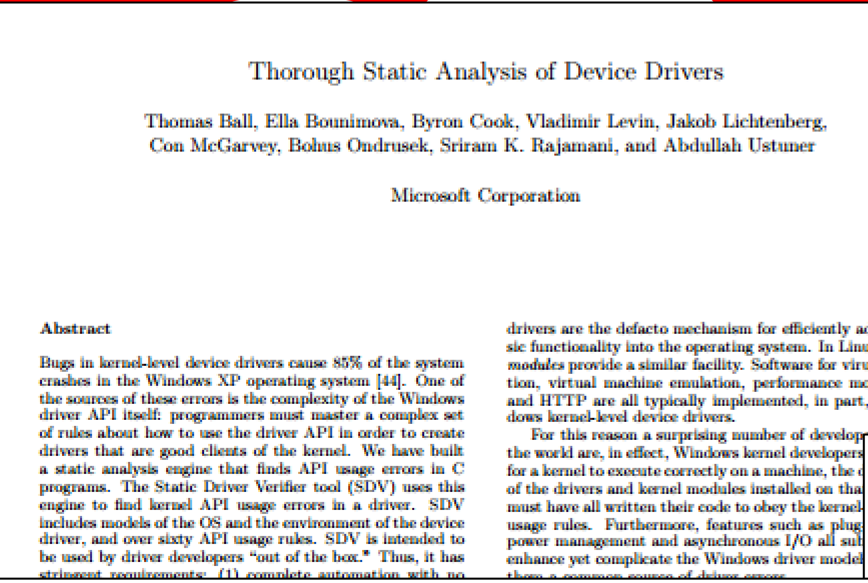
The Driver Abstraction Challenge. It is SDV's goal to check that device drivers make proper use of the driver API. It is not SDV's goal to check that device drivers perform any useful function with respect to their intended feature. Our hypothesis is that the amount of state that needs to be tracked in order to make an accurate determination about whether or not a driver obeys an API usage rule is relatively small compared to the entire state of the driver. The challenge is to automatically separate the relevant state from the irrelevant state.

SDV automatically abstracts the C code of a device driver to a simpler form. We call this alternative program an *abstraction* of the original because it does not lose errors: any API usage rule violation that appears in the original code also appears in the abstraction. This abstraction then

[1] We will refer to SLAM as SDV's analysis engine throughout the remainder of the article

AM

r's event-handling
pinLock() when



Thorough Static Analysis of Device Drivers

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg,
Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner

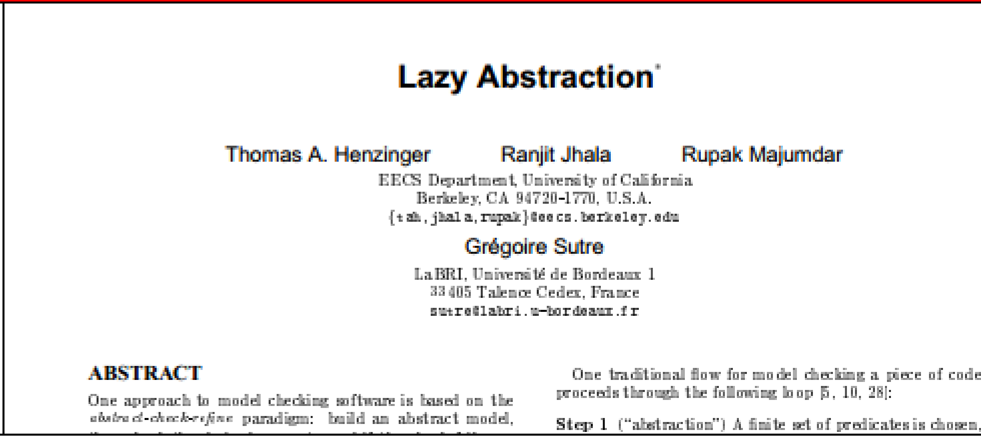
Microsoft Corporation

Abstract

Bugs in kernel-level device drivers cause 85% of the system crashes in the Windows XP operating system [44]. One of the sources of these errors is the complexity of the Windows driver API itself: programmers must master a complex set of rules about how to use the driver API in order to create drivers that are good clients of the kernel. We have built a static analysis engine that finds API usage errors in C programs. The Static Driver Verifier tool (SDV) uses this engine to find kernel API usage errors in a driver. SDV includes models of the OS and the environment of the device driver, and over sixty API usage rules. SDV is intended to be used by driver developers "out of the box." Thus, it has stringent requirements: (1) complete automation with no

drivers are the defacto mechanism for efficiently adding basic functionality into the operating system. In Linux, *kernel modules* provide a similar facility. Software for virus protection, virtual machine emulation, performance monitoring, and HTTP are all typically implemented, in part, as Windows kernel-level device drivers.

For this reason a surprising number of develop
the world are, in effect, Windows kernel developers
for a kernel to execute correctly on a machine, the
of the drivers and kernel modules installed on tha
must have all written their code to obey the kernel
usage rules. Furthermore, features such as plug
power management and asynchronous I/O all sub
enhance yet complicate the Windows driver model
there a common source of driver errors.



Lazy Abstraction*

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar
EECS Department, University of California
Berkeley, CA 94720-1770, U.S.A.
{tah, jhala, rupak}@eecs.berkeley.edu

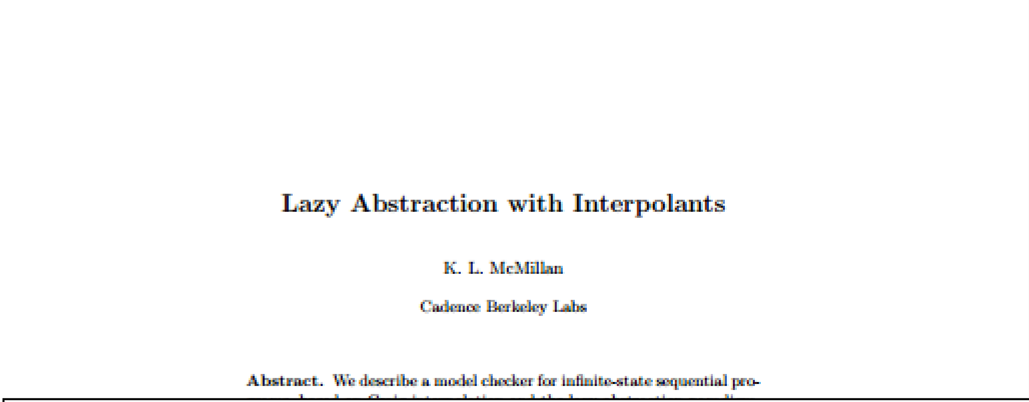
Grégoire Sutre
LaBRI, Université de Bordeaux 1
33405 Talence Cedex, France
sutre@labri.u-bordeaux.fr

ABSTRACT

One approach to model checking software is based on the *abstract-check-refine* paradigm: build an abstract model,

One traditional flow for model checking a piece of code proceeds through the following loop [5, 10, 28]:

Step 1 ("abstraction") A finite set of predicates is chosen,

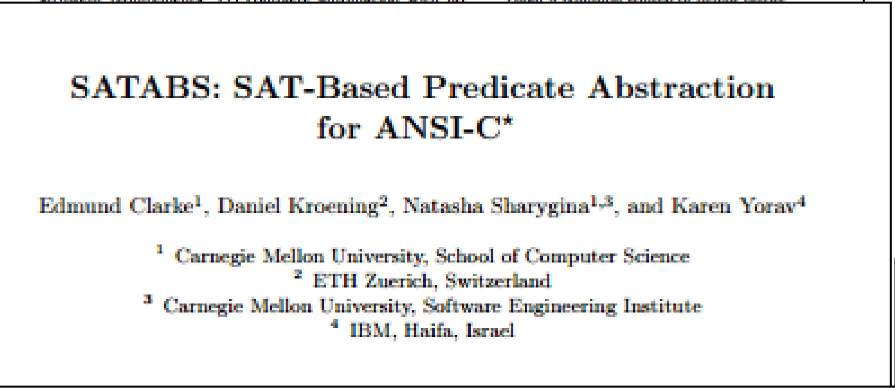


Lazy Abstraction with Interpolants

K. L. McMillan

Cadence Berkeley Labs

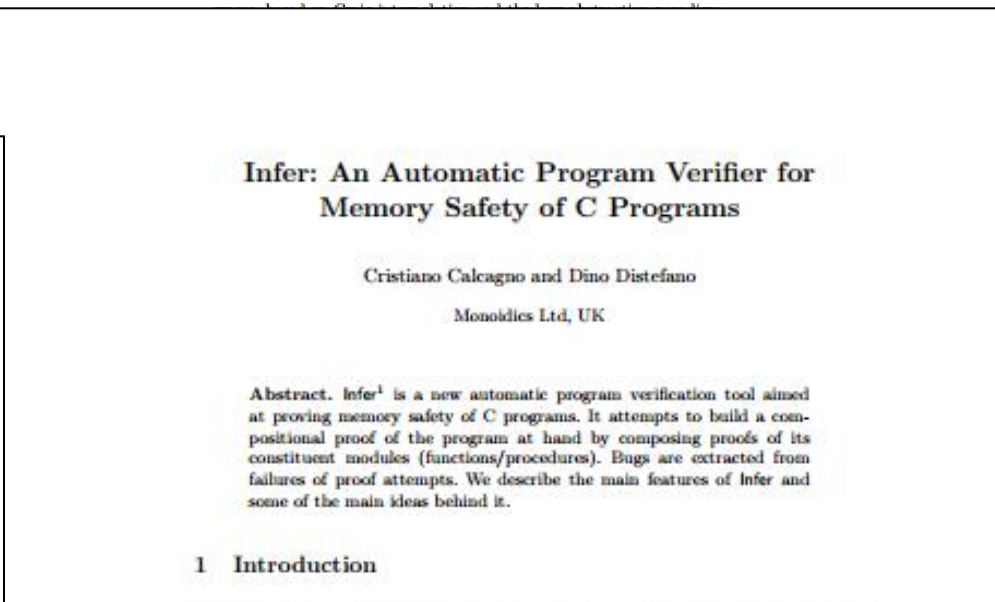
Abstract. We describe a model checker for infinite-state sequential pro-



SATABS: SAT-Based Predicate Abstraction for ANSI-C*

Edmund Clarke[1], Daniel Kroening[2], Natasha Sharygina[1,3], and Karen Yorav[4]

[1] Carnegie Mellon University, School of Computer Science
[2] ETH Zuerich, Switzerland
[3] Carnegie Mellon University, Software Engineering Institute
[4] IBM, Haifa, Israel



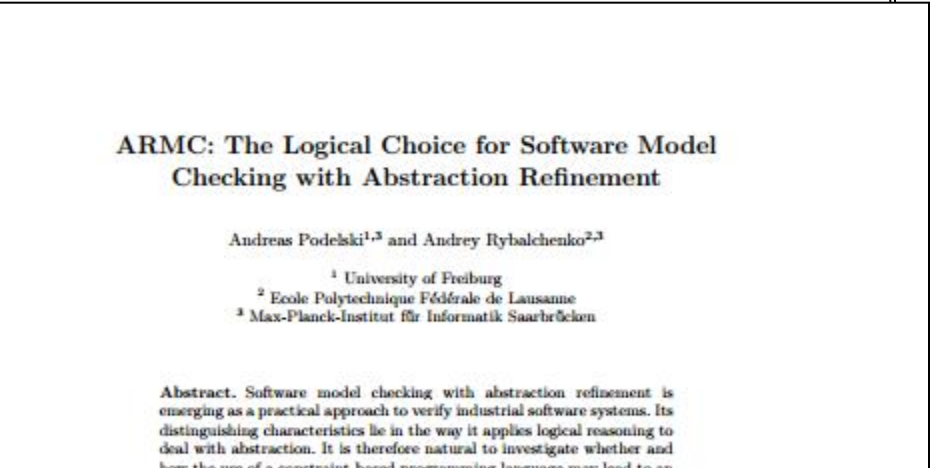
Infer: An Automatic Program Verifier for Memory Safety of C Programs

Cristiano Calcagno and Dino Distefano

Monoidics Ltd, UK

Abstract. Infer[1] is a new automatic program verification tool aimed at proving memory safety of C programs. It attempts to build a compositional proof of the program at hand by composing proofs of its constituent modules (functions/procedures). Bugs are extracted from failures of proof attempts. We describe the main features of Infer and some of the main ideas behind it.

1 Introduction

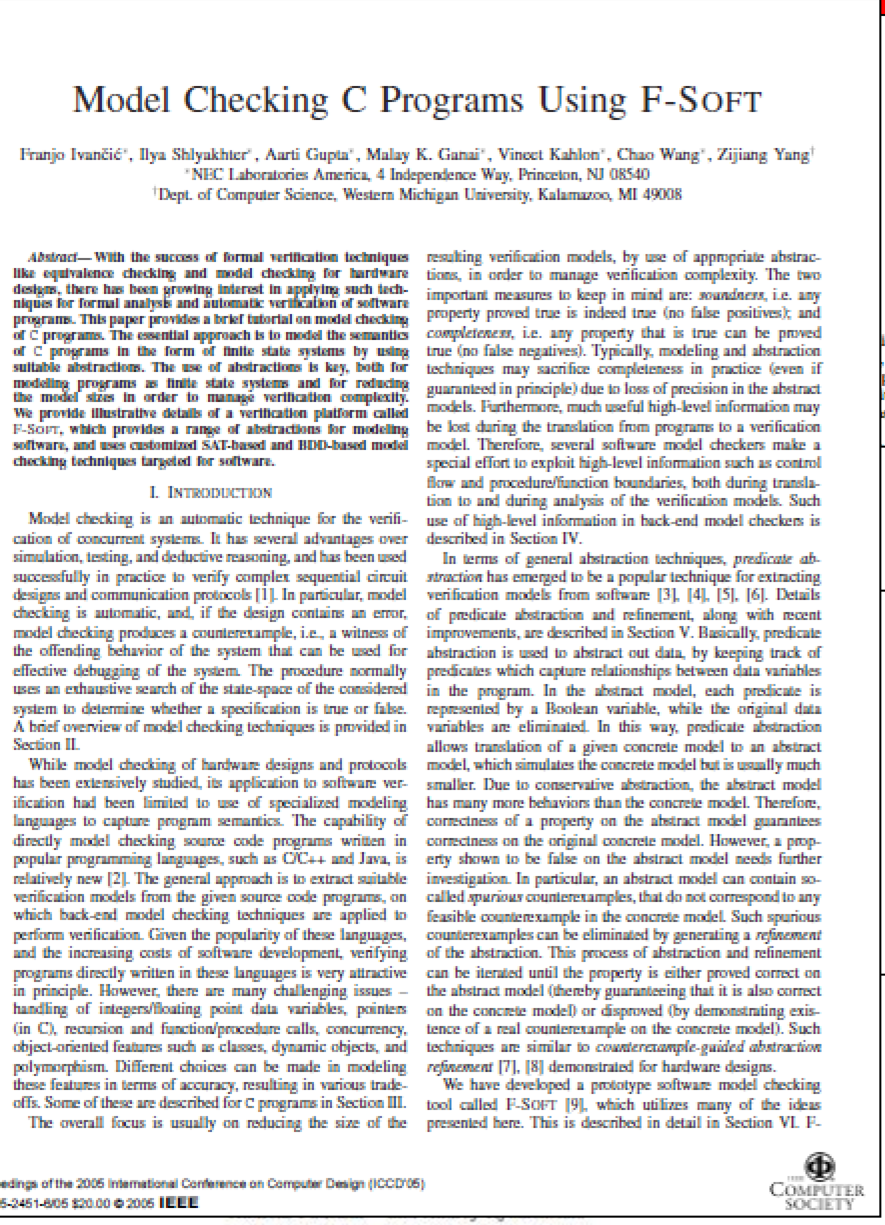


ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement

Andreas Podelski[1,3] and Andrey Rybalchenko[2,3]

[1] University of Freiburg
[2] Ecole Polytechnique Fédérale de Lausanne
[3] Max-Planck-Institut für Informatik Saarbrücken

Abstract. Software model checking with abstraction refinement is emerging as a practical approach to verify industrial software systems. Its distinguishing characteristics lie in the way it applies logical reasoning to deal with abstraction. It is therefore natural to investigate whether and



Model Checking C Programs Using F-SOFT

Franjo Ivančić[*], Ilya Shlyakhter[*], Aarti Gupta[*], Malay K. Ganai[*], Vineet Kahlon[*], Chao Wang[*], Zijiang Yang[†]
[*]NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540
[†]Dept. of Computer Science, Western Michigan University, Kalamazoo, MI 49008

***Abstract*— With the success of formal verification techniques like equivalence checking and model checking for hardware designs, there has been growing interest in applying such techniques for formal analysis and automatic verification of software programs. This paper provides a brief tutorial on model checking of C programs. The essential approach is to model the semantics of C programs in the form of finite state systems by using suitable abstractions. The use of abstractions is key, both for modeling programs as finite state systems and for reducing the model sizes in order to manage verification complexity. We provide illustrative details of a verification platform called F-SOFT, which provides a range of abstractions for modeling software, and uses customized SAT-based and BDD-based model checking techniques targeted for software.**

I. INTRODUCTION

Model checking is an automatic technique for the verification of concurrent systems. It has several advantages over simulation, testing, and deductive reasoning, and has been used successfully in practice to verify complex sequential circuit designs and communication protocols [1]. In particular, model checking is automatic, and, if the design contains an error, model checking produces a counterexample, i.e., a witness of the offending behavior of the system that can be used for effective debugging of the system. The procedure normally uses an exhaustive search of the state-space of the considered system to determine whether a specification is true or false. A brief overview of model checking techniques is provided in Section II.

While model checking of hardware designs and protocols has been extensively studied, its application to software verification had been limited to use of specialized modeling languages to capture program semantics. The capability of directly model checking source code programs written in popular programming languages, such as C/C++ and Java, is relatively new [2]. The general approach is to extract suitable verification models from the given source code programs, on which back-end model checking techniques are applied to perform verification. Given the popularity of these languages, and the increasing costs of software development, verifying programs directly written in these languages is very attractive in principle. However, there are many challenging issues – handling of integers/floating point data variables, pointers (in C), recursion and function/procedure calls, concurrency, object-oriented features such as classes, dynamic objects, and polymorphism. Different choices can be made in modeling these features in terms of accuracy, resulting in various trade-offs. Some of these are described for C programs in Section III.

The overall focus is usually on reducing the size of the resulting verification models, by use of appropriate abstractions, in order to manage verification complexity. The two important measures to keep in mind are: *soundness*, i.e. any property proved true is indeed true (no false positives); and *completeness*, i.e. any property that is true can be proved true (no false negatives). Typically, modeling and abstraction techniques may sacrifice completeness in practice (even if guaranteed in principle) due to loss of precision in the abstract models. Furthermore, much useful high-level information may be lost during the translation from programs to a verification model. Therefore, several software model checkers make a special effort to exploit high-level information such as control flow and procedure/function boundaries, both during translation to and during analysis of the verification models. Such use of high-level information in back-end model checkers is described in Section IV.

In terms of general abstraction techniques, *predicate abstraction* has emerged to be a popular technique for extracting verification models from software [3], [4], [5], [6]. Details of predicate abstraction and refinement, along with recent improvements, are described in Section V. Basically, predicate abstraction is used to abstract out data, by keeping track of predicates which capture relationships between data variables in the program. In the abstract model, each predicate is represented by a Boolean variable, while the original data variables are eliminated. In this way, predicate abstraction allows translation of a given concrete model to an abstract model, which simulates the concrete model but is usually much smaller. Due to conservative abstraction, the abstract model has many more behaviors than the concrete model. Therefore, correctness of a property on the abstract model guarantees correctness on the original concrete model. However, a property shown to be false on the abstract model needs further investigation. In particular, an abstract model can contain so-called *spurious* counterexamples, that do not correspond to any feasible counterexample in the concrete model. Such spurious counterexamples can be eliminated by generating a *refinement* of the abstraction. This process of abstraction and refinement can be iterated until the property is either proved correct on the abstract model (thereby guaranteeing that it is also correct on the concrete model) or disproved (by demonstrating existence of a real counterexample on the concrete model). Such techniques are similar to *counterexample-guided abstraction refinement* [7], [8] demonstrated for hardware designs.

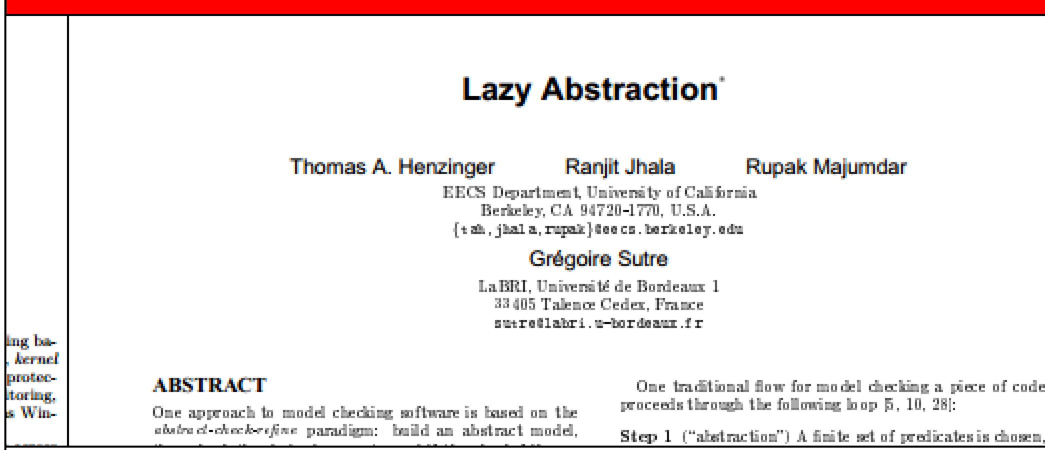
We have developed a prototype software model checking tool called F-SOFT [9], which utilizes many of the ideas presented here. This is described in detail in Section VI. F-

edings of the 2005 International Conference on Computer Design (ICCD'05)
5-2451-6/05 $20.00 © 2005 IEEE

COMPUTER SOCIETY

[1] University of Freiburg
[2] Ecole Polytechnique Fédérale de Lausanne
[3] Max-Planck-Institut für Informatik Saarbrücken

Abstract. Software model checking with abstraction refinement is emerging as a practical approach to verify industrial software systems. Its distinguishing characteristics lie in the way it applies logical reasoning to deal with abstraction. It is therefore natural to investigate whether and



ing ba-
, *kernel*
protec-
toring,
s Win-

Lazy Abstraction[*]

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar
EECS Department, University of California
Berkeley, CA 94720-1770, U.S.A.
{tah,jhala,rupak}@eecs.berkeley.edu

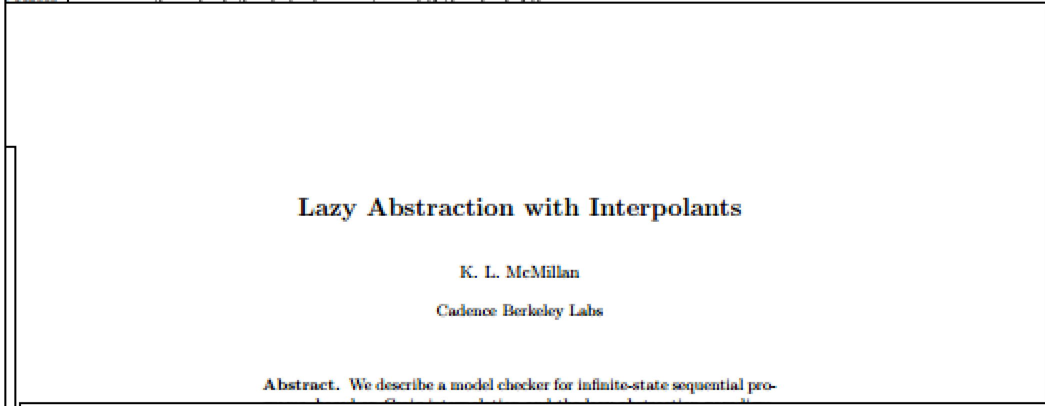
Grégoire Sutre
LaBRI, Université de Bordeaux 1
33 405 Talence Cedex, France
sutre@labri.u-bordeaux.fr

ABSTRACT

One approach to model checking software is based on the *abstract-check-refine* paradigm: build an abstract model,

One traditional flow for model checking a piece of code proceeds through the following loop [5, 10, 28]:

Step 1 ("abstraction") A finite set of predicates is chosen,

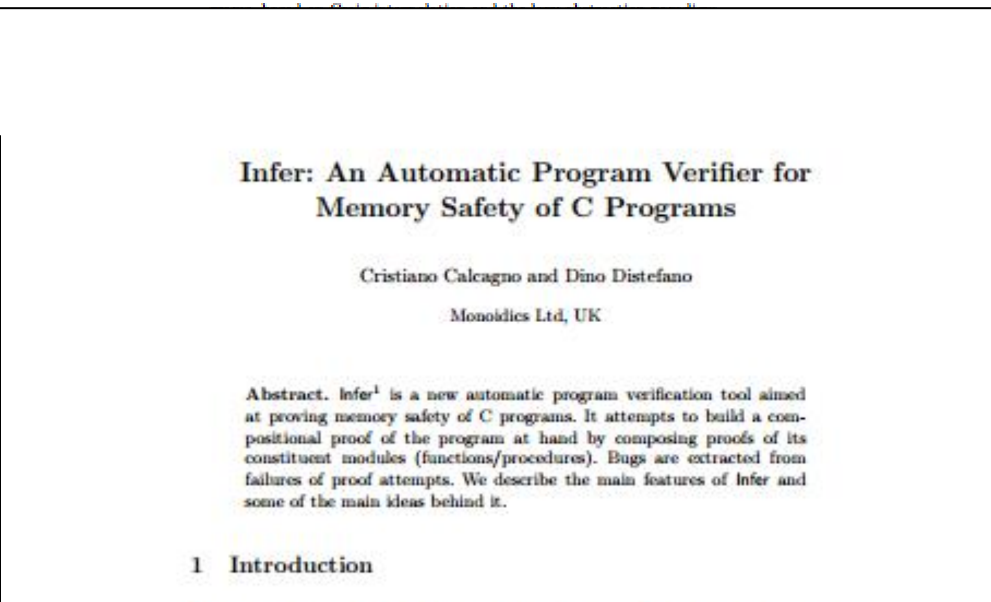


Lazy Abstraction with Interpolants

K. L. McMillan

Cadence Berkeley Labs

Abstract. We describe a model checker for infinite-state sequential pro-



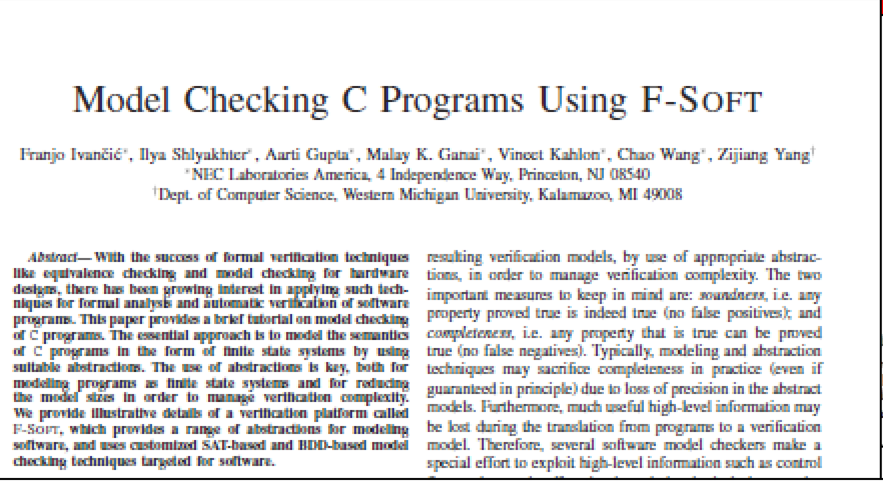
Infer: An Automatic Program Verifier for Memory Safety of C Programs

Cristiano Calcagno and Dino Distefano

Monoidics Ltd, UK

Abstract. Infer[1] is a new automatic program verification tool aimed at proving memory safety of C programs. It attempts to build a compositional proof of the program at hand by composing proofs of its constituent modules (functions/procedures). Bugs are extracted from failures of proof attempts. We describe the main features of Infer and some of the main ideas behind it.

1 Introduction

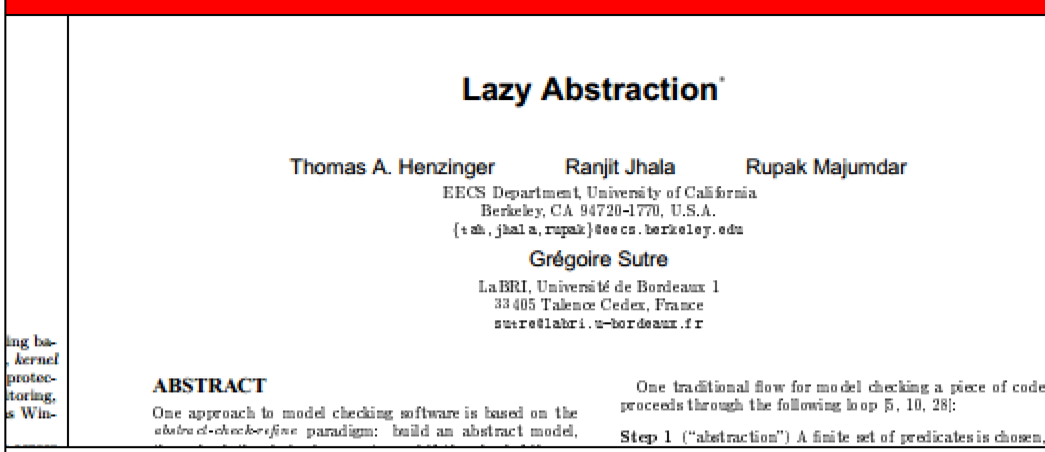


Model Checking C Programs Using F-Soft

Franjo Ivančić[*], Ilya Shlyakhter[*], Aarti Gupta[*], Malay K. Ganai[*], Vineet Kahlon[*], Chao Wang[*], Zijiang Yang[†]
[*]NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540
[†]Dept. of Computer Science, Western Michigan University, Kalamazoo, MI 49008

Abstract— With the success of formal verification techniques like equivalence checking and model checking for hardware designs, there has been growing interest in applying such techniques for formal analysis and automatic verification of software programs. This paper provides a brief tutorial on model checking of C programs. The essential approach is to model the semantics of C programs in the form of finite state systems by using suitable abstractions. The use of abstractions is key, both for modeling programs as finite state systems and for reducing the model sizes in order to manage verification complexity. We provide illustrative details of a verification platform called F-Soft, which provides a range of abstractions for modeling software, and uses customized SAT-based and BDD-based model checking techniques targeted for software.

resulting verification models, by use of appropriate abstractions, in order to manage verification complexity. The two important measures to keep in mind are: *soundness*, i.e. any property proved true is indeed true (no false positives); and *completeness*, i.e. any property that is true can be proved true (no false negatives). Typically, modeling and abstraction techniques may sacrifice completeness in practice (even if guaranteed in principle) due to loss of precision in the abstract models. Furthermore, much useful high-level information may be lost during the translation from programs to a verification model. Therefore, several software model checkers make a special effort to exploit high-level information such as control



ing ba-
, *kernel*
protec-
toring,
s Win-

Lazy Abstraction[*]

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar
EECS Department, University of California
Berkeley, CA 94720-1770, U.S.A.
{tah, jhala, rupak}@eecs.berkeley.edu

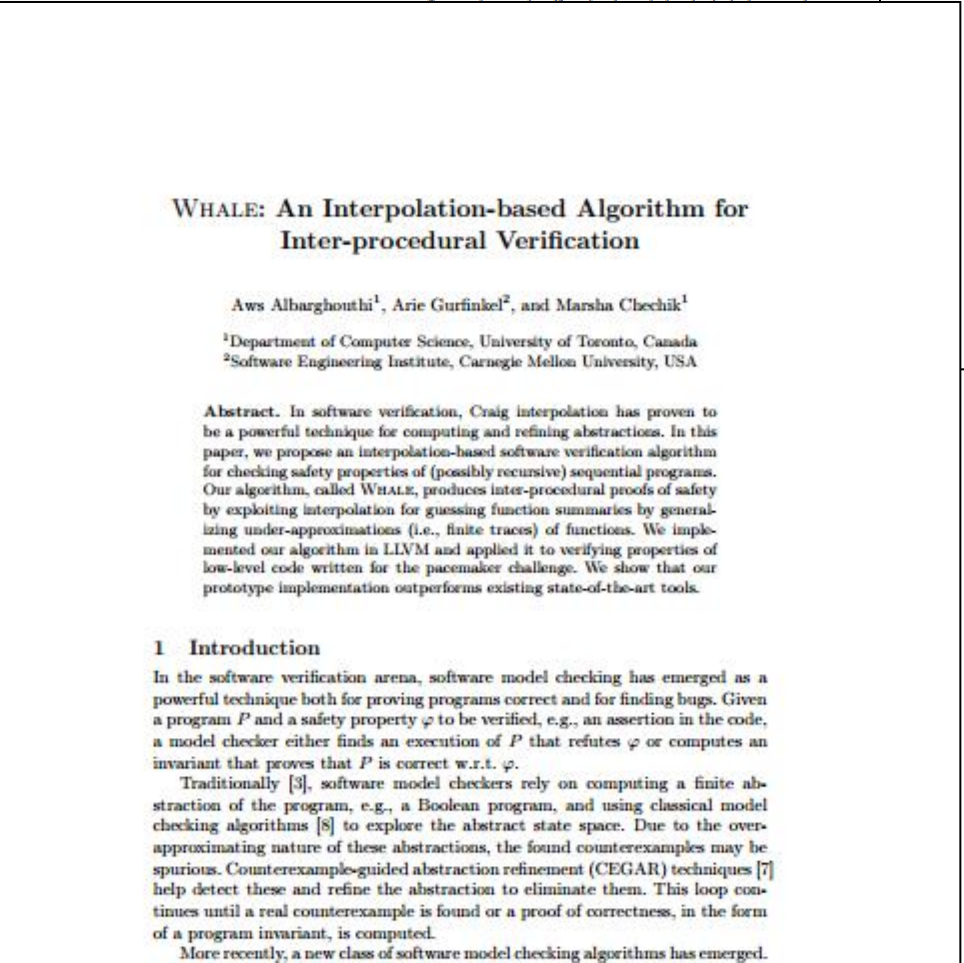
Grégoire Sutre
LaBRI, Université de Bordeaux 1
33405 Talence Cedex, France
sutre@labri.u-bordeaux.fr

ABSTRACT

One approach to model checking software is based on the *abstract-check-refine* paradigm: build an abstract model,

One traditional flow for model checking a piece of code proceeds through the following loop [5, 10, 28]:

Step 1 ("abstraction") A finite set of predicates is chosen,



Whale: An Interpolation-based Algorithm for Inter-procedural Verification

Aws Albarghouthi[1], Arie Gurfinkel[2], and Marsha Chechik[1]

[1]Department of Computer Science, University of Toronto, Canada
[2]Software Engineering Institute, Carnegie Mellon University, USA

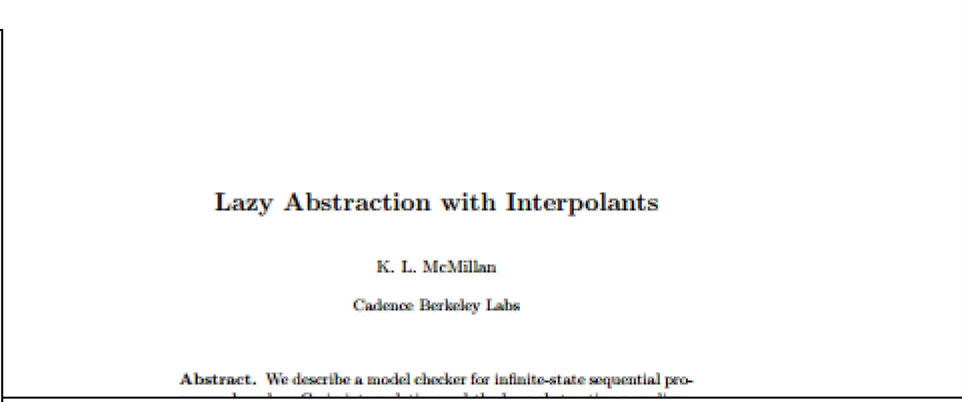
Abstract. In software verification, Craig interpolation has proven to be a powerful technique for computing and refining abstractions. In this paper, we propose an interpolation-based software verification algorithm for checking safety properties of (possibly recursive) sequential programs. Our algorithm, called Whale, produces inter-procedural proofs of safety by exploiting interpolation for guessing function summaries by generalizing under-approximations (i.e., finite traces) of functions. We implemented our algorithm in LLVM and applied it to verifying properties of low-level code written for the pacemaker challenge. We show that our prototype implementation outperforms existing state-of-the-art tools.

1 Introduction

In the software verification arena, software model checking has emerged as a powerful technique both for proving programs correct and for finding bugs. Given a program P and a safety property φ to be verified, e.g., an assertion in the code, a model checker either finds an execution of P that refutes φ or computes an invariant that proves that P is correct w.r.t. φ.

Traditionally [3], software model checkers rely on computing a finite abstraction of the program, e.g., a Boolean program, and using classical model checking algorithms [8] to explore the abstract state space. Due to the over-approximating nature of these abstractions, the found counterexamples may be spurious. Counterexample-guided abstraction refinement (CEGAR) techniques [7] help detect these and refine the abstraction to eliminate them. This loop continues until a real counterexample is found or a proof of correctness, in the form of a program invariant, is computed.

More recently, a new class of software model checking algorithms has emerged.

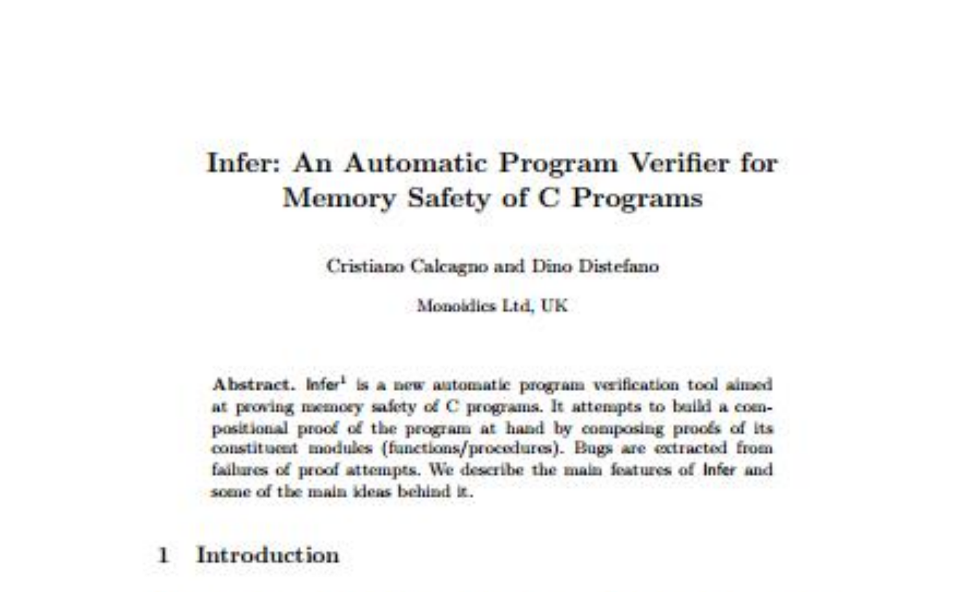


Lazy Abstraction with Interpolants

K. L. McMillan

Cadence Berkeley Labs

Abstract. We describe a model checker for infinite-state sequential pro-



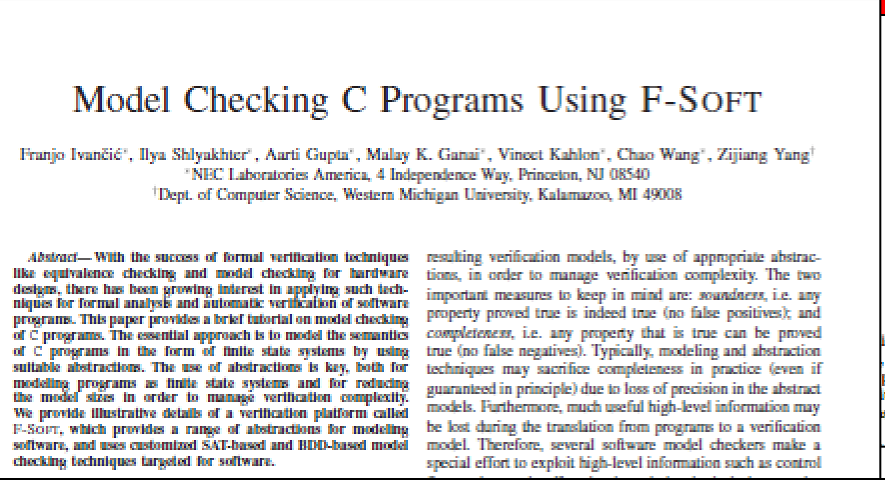
Infer: An Automatic Program Verifier for Memory Safety of C Programs

Cristiano Calcagno and Dino Distefano

Monoidics Ltd, UK

Abstract. Infer[1] is a new automatic program verification tool aimed at proving memory safety of C programs. It attempts to build a compositional proof of the program at hand by composing proofs of its constituent modules (functions/procedures). Bugs are extracted from failures of proof attempts. We describe the main features of Infer and some of the main ideas behind it.

1 Introduction



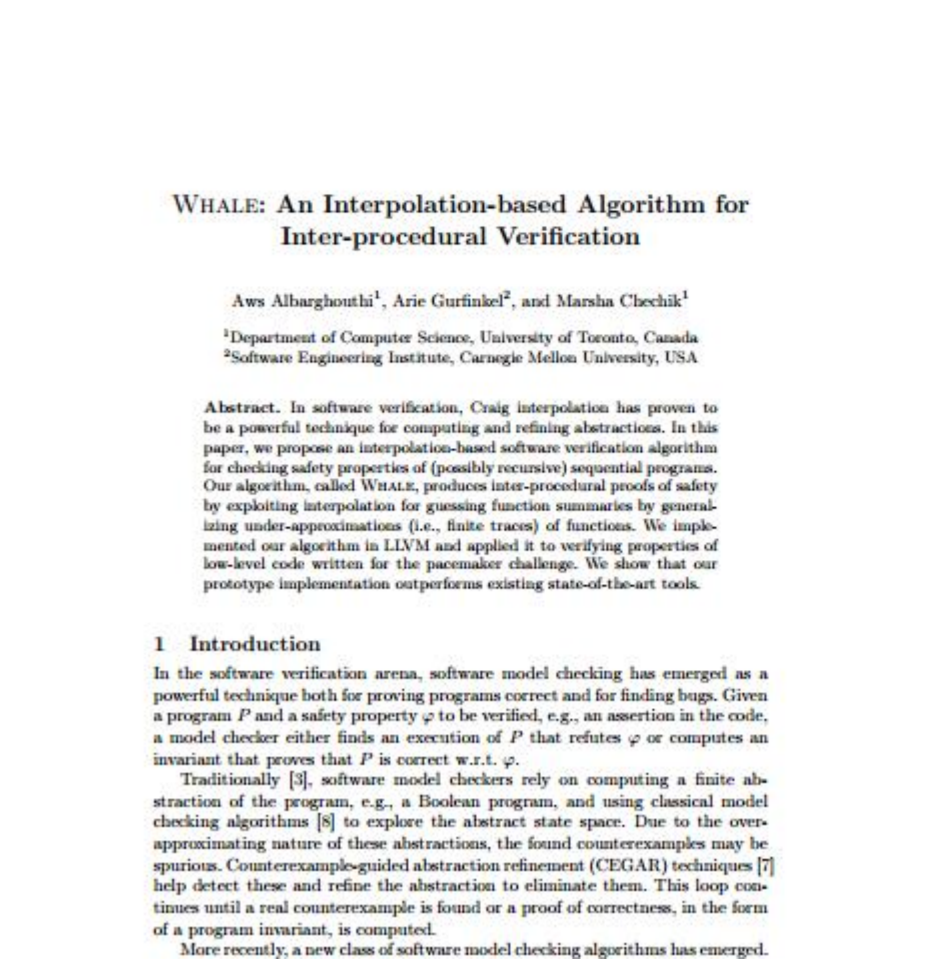
Model Checking C Programs Using F-SOFT

Franjo Ivančić*, Ilya Shlyakhter*, Aarti Gupta*, Malay K. Ganai*, Vineet Kahlon*, Chao Wang*, Zijiang Yang†
*NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540
†Dept. of Computer Science, Western Michigan University, Kalamazoo, MI 49008

Abstract— With the success of formal verification techniques like equivalence checking and model checking for hardware designs, there has been growing interest in applying such techniques for formal analysis and automatic verification of software programs. This paper provides a brief tutorial on model checking of C programs. The essential approach is to model the semantics of C programs in the form of finite state systems by using suitable abstractions. The use of abstractions is key, both for modeling programs as finite state systems and for reducing the model sizes in order to manage verification complexity. We provide illustrative details of a verification platform called F-SOFT, which provides a range of abstractions for modeling software, and uses customized SAT-based and BDD-based model checking techniques targeted for software.

resulting verification models, by use of appropriate abstractions, in order to manage verification complexity. The two important measures to keep in mind are: *soundness*, i.e. any property proved true is indeed true (no false positives); and *completeness*, i.e. any property that is true can be proved true (no false negatives). Typically, modeling and abstraction techniques may sacrifice completeness in practice (even if guaranteed in principle) due to loss of precision in the abstract models. Furthermore, much useful high-level information may be lost during the translation from programs to a verification model. Therefore, several software model checkers make a special effort to exploit high-level information such as control

ing ba
, kerne
protec
toring
s Win
e of code
is chosen,



WHALE: An Interpolation-based Algorithm for Inter-procedural Verification

Aws Albarghouthi[1], Arie Gurfinkel[2], and Marsha Chechik[1]

[1]Department of Computer Science, University of Toronto, Canada
[2]Software Engineering Institute, Carnegie Mellon University, USA

Abstract. In software verification, Craig interpolation has proven to be a powerful technique for computing and refining abstractions. In this paper, we propose an interpolation-based software verification algorithm for checking safety properties of (possibly recursive) sequential programs. Our algorithm, called WHALE, produces inter-procedural proofs of safety by exploiting interpolation for guessing function summaries by generalizing under-approximations (i.e., finite traces) of functions. We implemented our algorithm in LLVM and applied it to verifying properties of low-level code written for the pacemaker challenge. We show that our prototype implementation outperforms existing state-of-the-art tools.

1 Introduction

In the software verification arena, software model checking has emerged as a powerful technique both for proving programs correct and for finding bugs. Given a program P and a safety property φ to be verified, e.g., an assertion in the code, a model checker either finds an execution of P that refutes φ or computes an invariant that proves that P is correct w.r.t. φ.

Traditionally [3], software model checkers rely on computing a finite abstraction of the program, e.g., a Boolean program, and using classical model checking algorithms [8] to explore the abstract state space. Due to the overapproximating nature of these abstractions, the found counterexamples may be spurious. Counterexample-guided abstraction refinement (CEGAR) techniques [7] help detect these and refine the abstraction to eliminate them. This loop continues until a real counterexample is found or a proof of correctness, in the form of a program invariant, is computed.

More recently, a new class of software model checking algorithms has emerged.

Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic*

Kamil Dudka, Petr Peringer, and Tomáš Vojnar

FIT, Brno University of Technology, Czech Republic

Abstract. Predator is a new open source tool for verification of sequential C programs with dynamic linked data structures. The tool is based on separation logic with inductive predicates although it uses a graph description of heaps. Predator currently handles various forms of lists, including singly-linked as well as doubly-linked lists that may be circular, hierarchically nested and that may have various additional pointer links. Predator is implemented as a `gcc` plug-in and it is capable of handling lists in the form they appear in real system code, especially the Linux kernel, including a limited support of pointer arithmetic. Collaboration on further development of Predator is welcome.

1 Introduction

In this paper, we present a new tool called *Predator* for fully automatic verification of sequential C programs with dynamic linked data structures. In particular, Predator can currently handle various complex kinds of *singly-linked as well as doubly-linked lists* that may be circular, shared, hierarchically nested, and that can have various additional pointers (head/tail pointers, data pointers, etc.). Predator implicitly checks for absence of generic errors, such as null dereferences, double deletion, memory leakage, etc. It can also print out a symbolic representation of the shapes of the memory structures arising in a program. Finally, users can, of course, use Predator to check custom properties about the data structures being used in their code by writing (directly in C) tester programs exercising these structures.

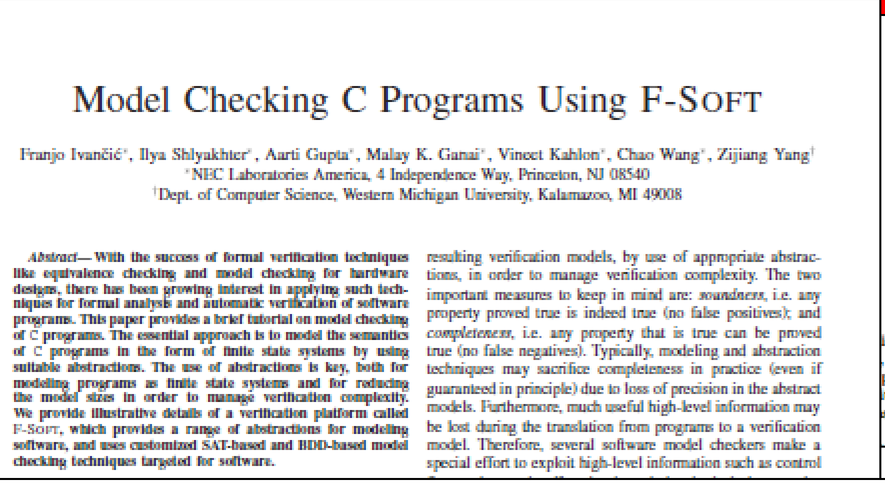
Predator is based on *separation logic* with *higher-order inductive predicates*. It is inspired by the works [2,9,10] and the very influential tool called Space Invader[1] (or simply Invader). However, compared to Invader, the heap representation in Predator is not based on lists of separation logic formulae, but rather a *graph representation* of these formulae. The algorithms handling the symbolic heap representation (in particular, the abstraction and join operators based on detecting occurrences of heap structures that can be described by inductive predicates) have been newly designed.

Compared to Invader that contains a partial support of doubly-linked lists only, Predator supports them equally well as singly-linked lists. Predator also contains a special support for list segments of length 0 or 1 that are common in practice [9] and that may cause problems to Invader (as we illustrate further on).

* This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the BUT FIT project FIT-S-11-1. An extended version of the paper is available as the technical report [6].

[1] `http://www.eastlondonmassive.org/East_London_Massive/Invader_Home.html`

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 372–378, 2011.
© Springer-Verlag Berlin Heidelberg 2011



Model Checking C Programs Using F-Soft

Franjo Ivančić*, Ilya Shlyakhter*, Aarti Gupta*, Malay K. Ganai*, Vineet Kahlon*, Chao Wang*, Zijiang Yang†
*NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540
†Dept. of Computer Science, Western Michigan University, Kalamazoo, MI 49008

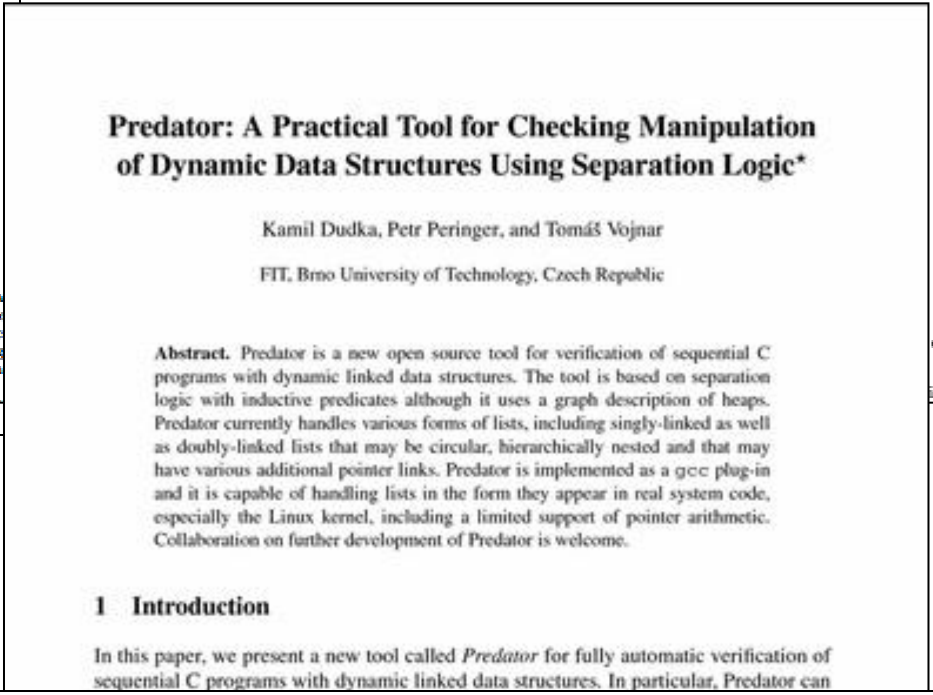
Abstract— With the success of formal verification techniques like equivalence checking and model checking for hardware designs, there has been growing interest in applying such techniques for formal analysis and automatic verification of software programs. This paper provides a brief tutorial on model checking of C programs. The essential approach is to model the semantics of C programs in the form of finite state systems by using suitable abstractions. The use of abstractions is key, both for modeling programs as finite state systems and for reducing the model sizes in order to manage verification complexity. We provide illustrative details of a verification platform called F-Soft, which provides a range of abstractions for modeling software, and uses customized SAT-based and BDD-based model checking techniques targeted for software.

resulting verification models, by use of appropriate abstractions, in order to manage verification complexity. The two important measures to keep in mind are: *soundness*, i.e. any property proved true is indeed true (no false positives); and *completeness*, i.e. any property that is true can be proved true (no false negatives). Typically, modeling and abstraction techniques may sacrifice completeness in practice (even if guaranteed in principle) due to loss of precision in the abstract models. Furthermore, much useful high-level information may be lost during the translation from programs to a verification model. Therefore, several software model checkers make a special effort to exploit high-level information such as control

ing ba
, *kerne*
protec
itoring
s Win

e of code

is chosen,



Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic*

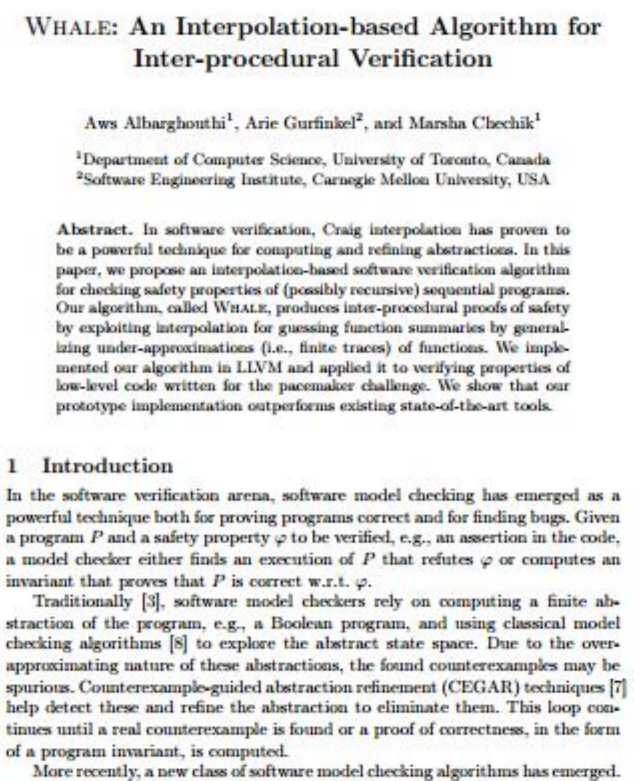
Kamil Dudka, Petr Peringer, and Tomáš Vojnar

FIT, Brno University of Technology, Czech Republic

Abstract. Predator is a new open source tool for verification of sequential C programs with dynamic linked data structures. The tool is based on separation logic with inductive predicates although it uses a graph description of heaps. Predator currently handles various forms of lists, including singly-linked as well as doubly-linked lists that may be circular, hierarchically nested and that may have various additional pointer links. Predator is implemented as a gcc plug-in and it is capable of handling lists in the form they appear in real system code, especially the Linux kernel, including a limited support of pointer arithmetic. Collaboration on further development of Predator is welcome.

1 Introduction

In this paper, we present a new tool called *Predator* for fully automatic verification of sequential C programs with dynamic linked data structures. In particular, Predator can



Whale: An Interpolation-based Algorithm for Inter-procedural Verification

Aws Albarghouthi[1], Arie Gurfinkel[2], and Marsha Chechik[1]

[1]Department of Computer Science, University of Toronto, Canada
[2]Software Engineering Institute, Carnegie Mellon University, USA

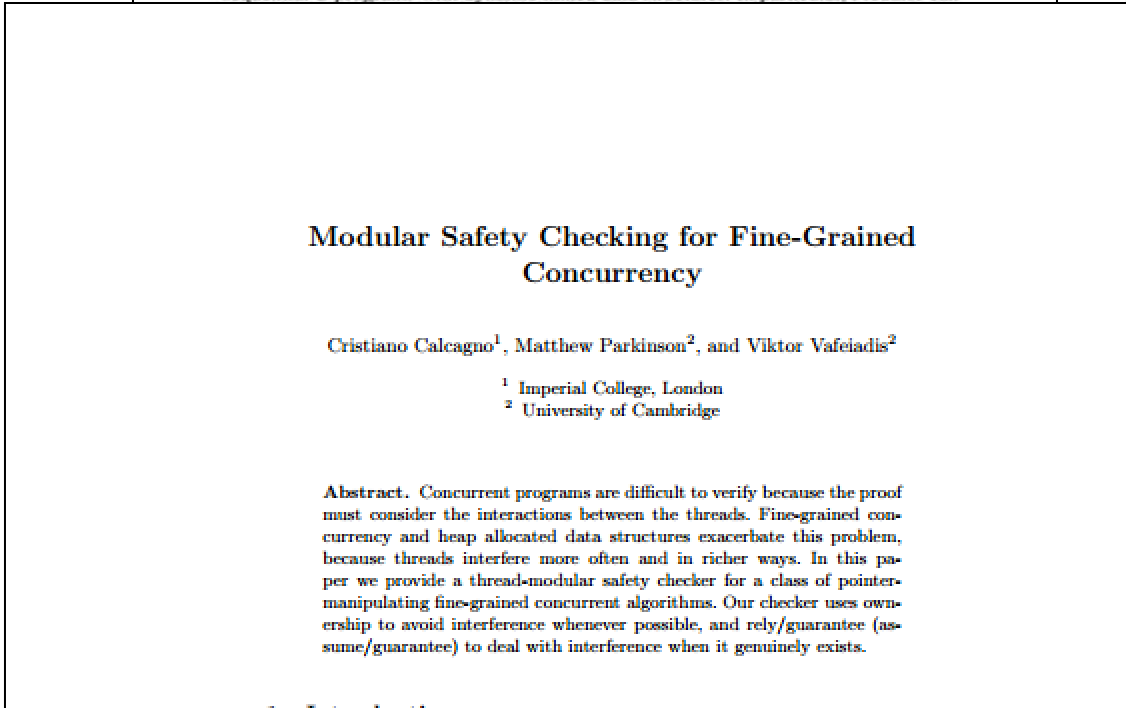
Abstract. In software verification, Craig interpolation has proven to be a powerful technique for computing and refining abstractions. In this paper, we propose an interpolation-based software verification algorithm for checking safety properties of (possibly recursive) sequential programs. Our algorithm, called Whale, produces inter-procedural proofs of safety by exploiting interpolation for guessing function summaries by generalizing under-approximations (i.e., finite traces) of functions. We implemented our algorithm in LLVM and applied it to verifying properties of low-level code written for the pacemaker challenge. We show that our prototype implementation outperforms existing state-of-the-art tools.

1 Introduction

In the software verification arena, software model checking has emerged as a powerful technique both for proving programs correct and for finding bugs. Given a program P and a safety property φ to be verified, e.g., an assertion in the code, a model checker either finds an execution of P that refutes φ or computes an invariant that proves that P is correct w.r.t. φ.

Traditionally [3], software model checkers rely on computing a finite abstraction of the program, e.g., a Boolean program, and using classical model checking algorithms [8] to explore the abstract state space. Due to the over-approximating nature of these abstractions, the found counterexamples may be spurious. Counterexample-guided abstraction refinement (CEGAR) techniques [7] help detect these and refine the abstraction to eliminate them. This loop continues until a real counterexample is found or a proof of correctness, in the form of a program invariant, is computed.

More recently, a new class of software model checking algorithms has emerged.



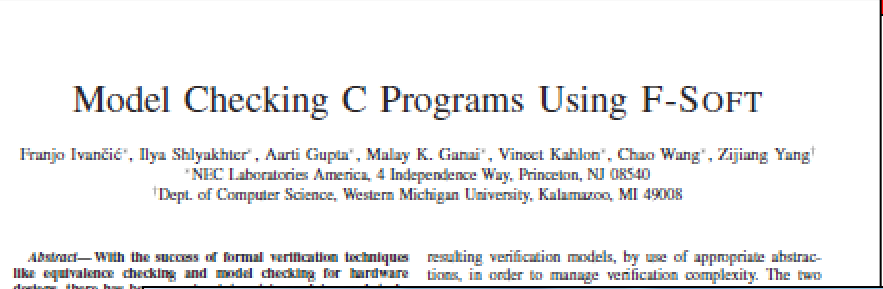
Modular Safety Checking for Fine-Grained Concurrency

Cristiano Calcagno[1], Matthew Parkinson[2], and Viktor Vafeiadis[2]

[1] Imperial College, London
[2] University of Cambridge

Abstract. Concurrent programs are difficult to verify because the proof must consider the interactions between the threads. Fine-grained concurrency and heap allocated data structures exacerbate this problem, because threads interfere more often and in richer ways. In this paper we provide a thread-modular safety checker for a class of pointer-manipulating fine-grained concurrent algorithms. Our checker uses ownership to avoid interference whenever possible, and rely/guarantee (assume/guarantee) to deal with interference when it genuinely exists.

1 Introduction



Model Checking C Programs Using F-Soft

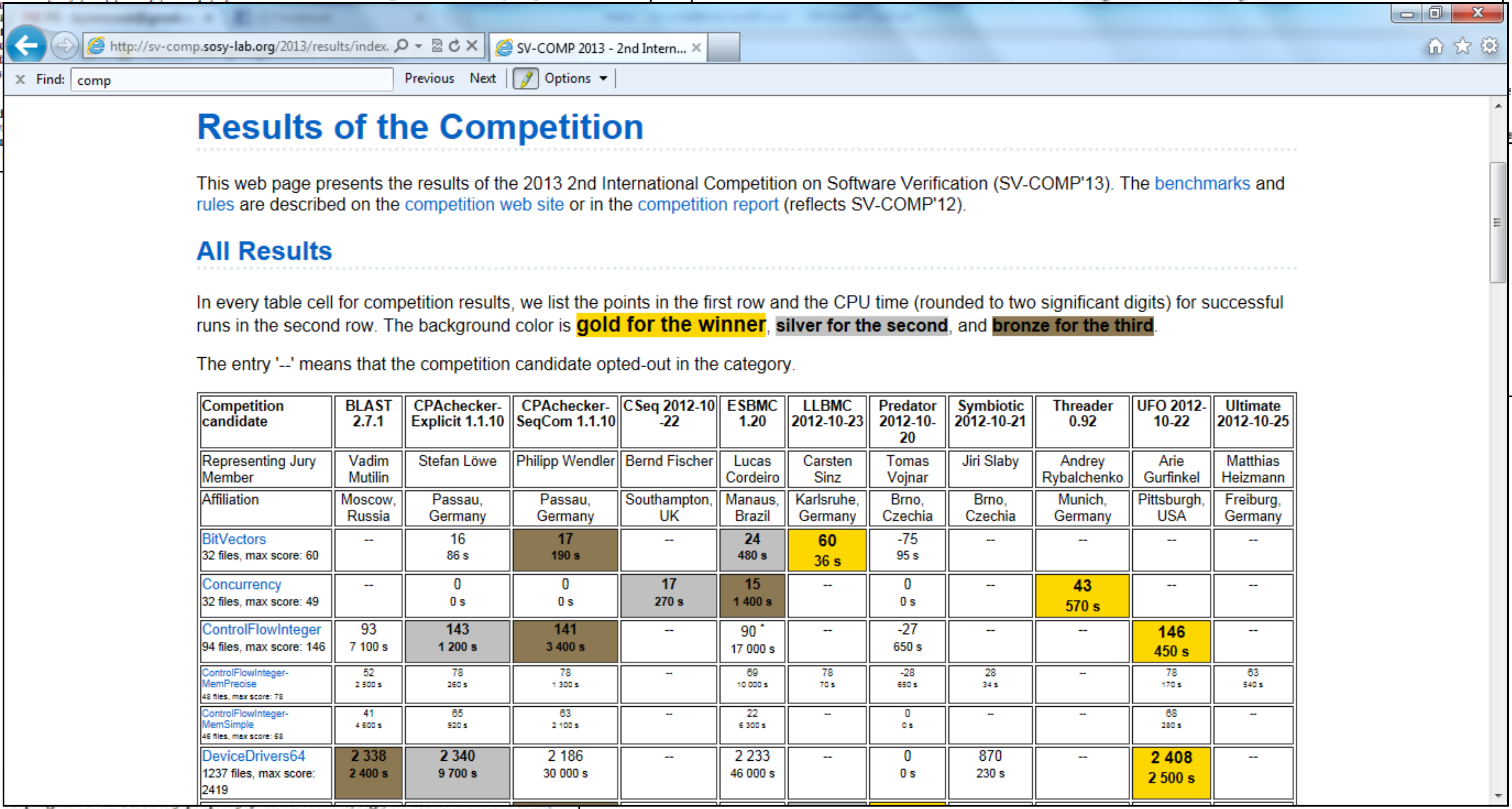
Franjo Ivančić*, Ilya Shlyakhter*, Aarti Gupta*, Malay K. Ganai*, Vineet Kahlon*, Chao Wang*, Zijiang Yang†
*NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540
†Dept. of Computer Science, Western Michigan University, Kalamazoo, MI 49008

Abstract— With the success of formal verification techniques like equivalence checking and model checking for hardware designs, there has be

resulting verification models, by use of appropriate abstractions, in order to manage verification complexity. The two

Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic*

Kamil Dudka, Petr Peringer, and Tomáš Vojnar



http://sv-comp.sosy-lab.org/2013/results/index.

Find: comp

Results of the Competition

This web page presents the results of the 2013 2nd International Competition on Software Verification (SV-COMP'13). The benchmarks and rules are described on the competition web site or in the competition report (reflects SV-COMP'12).

All Results

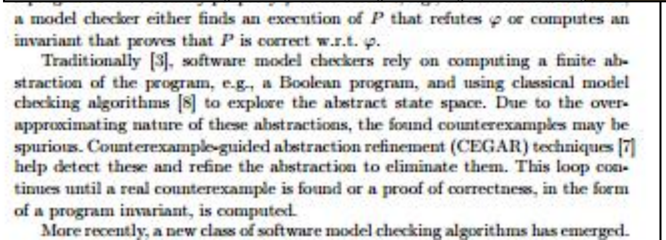
In every table cell for competition results, we list the points in the first row and the CPU time (rounded to two significant digits) for successful runs in the second row. The background color is **gold for the winner**, **silver for the second**, and **bronze for the third**.

The entry '--' means that the competition candidate opted-out in the category.

Competition candidate	BLAST 2.7.1	CPAchecker-Explicit 1.1.10	CPAchecker-SeqCom 1.1.10	CSeq 2012-10-22	ESBMC 1.20	LLBMC 2012-10-23	Predator 2012-10-20	Symbiotic 2012-10-21	Threader 0.92	UFO 2012-10-22	Ultimate 2012-10-25
Representing Jury Member	Vadim Mutilin	Stefan Löwe	Philipp Wendler	Bernd Fischer	Lucas Cordeiro	Carsten Sinz	Tomas Vojnar	Jiri Slaby	Andrey Rybalchenko	Arie Gurfinkel	Matthias Heizmann
Affiliation	Moscow, Russia	Passau, Germany	Passau, Germany	Southampton, UK	Manaus, Brazil	Karlsruhe, Germany	Brno, Czechia	Brno, Czechia	Munich, Germany	Pittsburgh, USA	Freiburg, Germany
BitVectors 32 files, max score: 60	--	16 86 s	17 190 s	--	24 480 s	60 36 s	-75 95 s	--	--	--	--
Concurrency 32 files, max score: 49	--	0 0 s	0 0 s	17 270 s	15 1 400 s	--	0 0 s	--	43 570 s	--	--
ControlFlowInteger 94 files, max score: 146	93 7 100 s	143 1 200 s	141 3 400 s	--	90 * 17 000 s	--	-27 650 s	--	--	146 450 s	--
ControlFlowInteger-MemPrecise 48 files, max score: 78	52 2 500 s	78 260 s	78 1 300 s	--	69 10 000 s	78 70 s	-28 650 s	28 34 s	--	78 170 s	63 540 s
ControlFlowInteger-MemSimple 46 files, max score: 68	41 4 600 s	65 920 s	63 2 100 s	--	22 6 300 s	--	0 0 s	--	--	68 280 s	--
DeviceDrivers64 1237 files, max score: 2419	2 338 2 400 s	2 340 9 700 s	2 186 30 000 s	--	2 233 46 000 s	--	0 0 s	870 230 s	--	2 408 2 500 s	--

of code

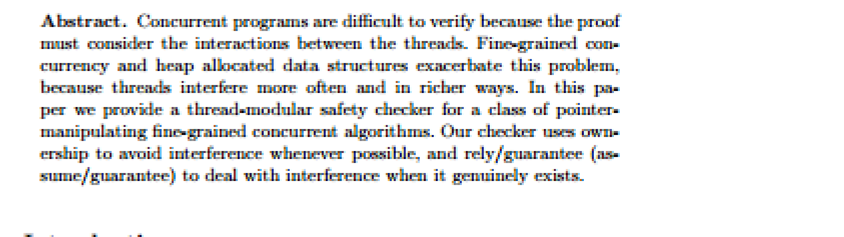
s chosen,



a model checker either finds an execution of P that refutes φ or computes an invariant that proves that P is correct w.r.t. φ.

Traditionally [3], software model checkers rely on computing a finite abstraction of the program, e.g., a Boolean program, and using classical model checking algorithms [8] to explore the abstract state space. Due to the over-approximating nature of these abstractions, the found counterexamples may be spurious. Counterexample-guided abstraction refinement (CEGAR) techniques [7] help detect these and refine the abstraction to eliminate them. This loop continues until a real counterexample is found or a proof of correctness, in the form of a program invariant, is computed.

More recently, a new class of software model checking algorithms has emerged.



Abstract. Concurrent programs are difficult to verify because the proof must consider the interactions between the threads. Fine-grained concurrency and heap allocated data structures exacerbate this problem, because threads interfere more often and in richer ways. In this paper we provide a thread-modular safety checker for a class of pointer-manipulating fine-grained concurrent algorithms. Our checker uses ownership to avoid interference whenever possible, and rely/guarantee (assume/guarantee) to deal with interference when it genuinely exists.

Example property

“The mouse device driver’s event-handling routine always eventually terminates”

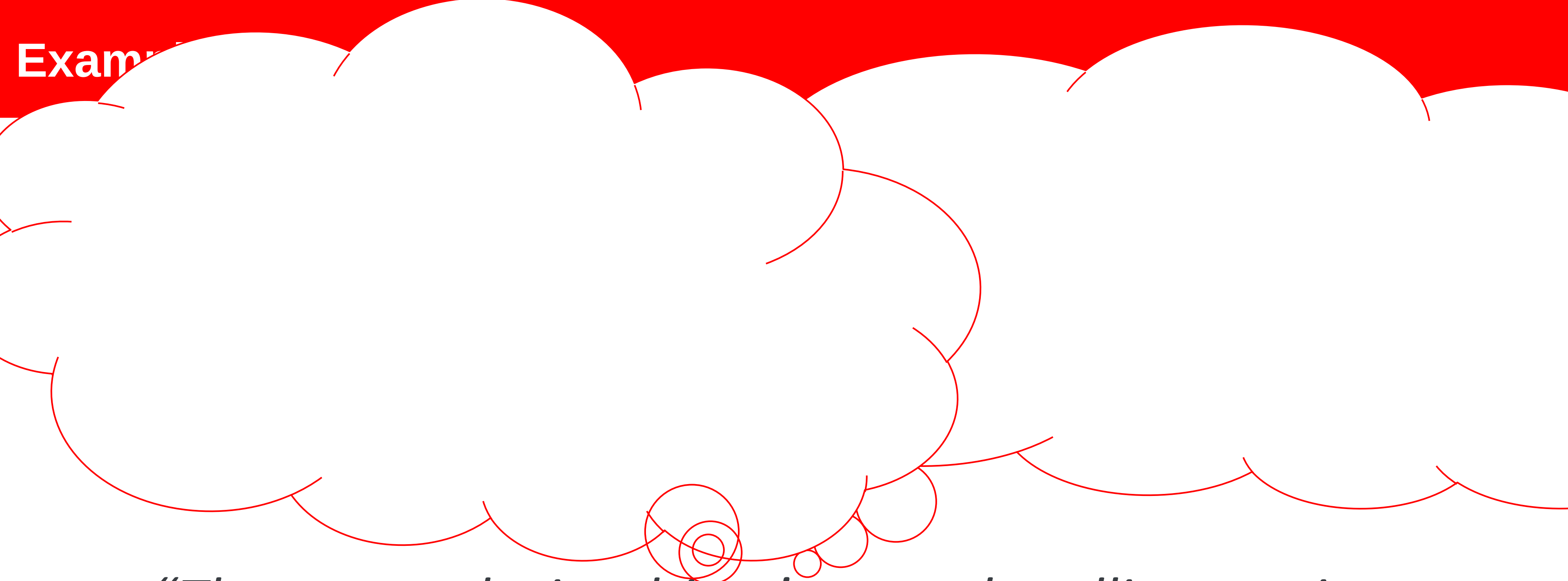
Example property

“The mouse device driver’s event-handling routine always eventually terminates”

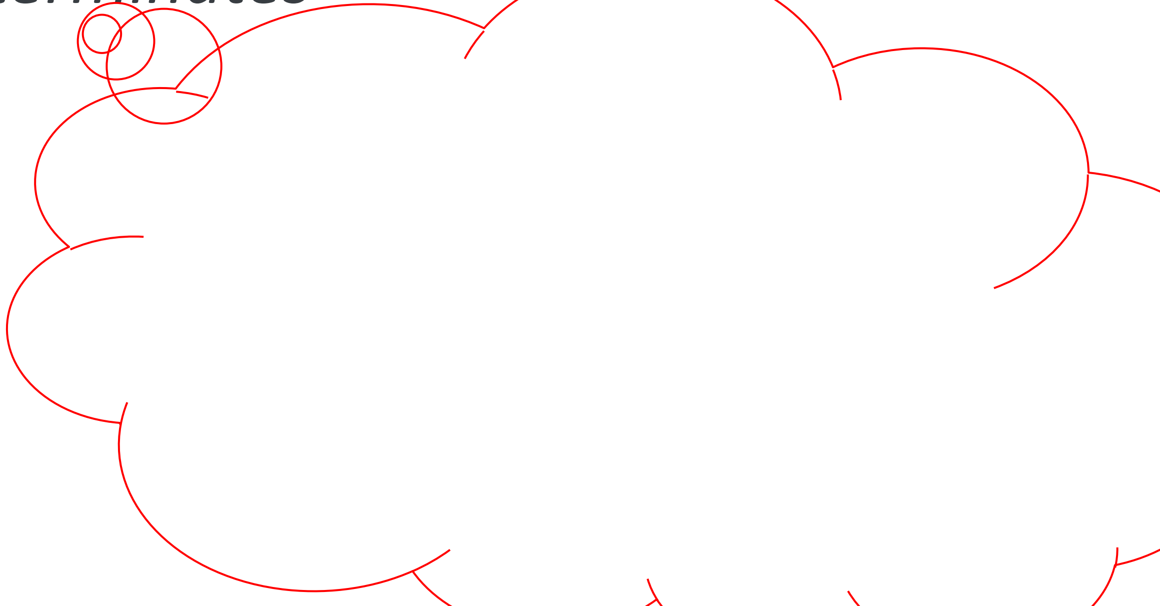
Example property

“The mouse device driver’s event-handling routine always eventually terminates”

Exam



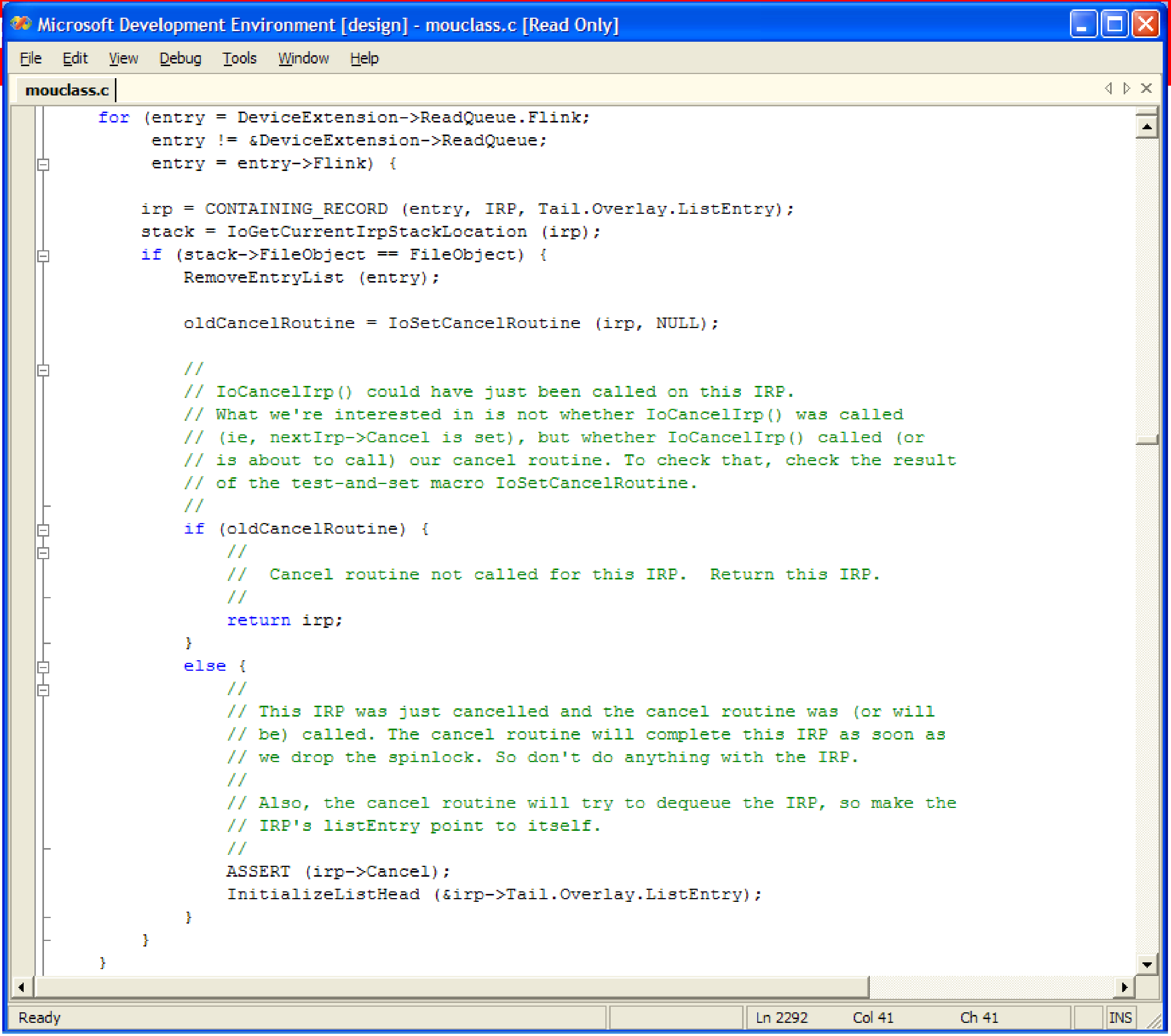
“The mouse device driver’s event-handling routine always eventually terminates”



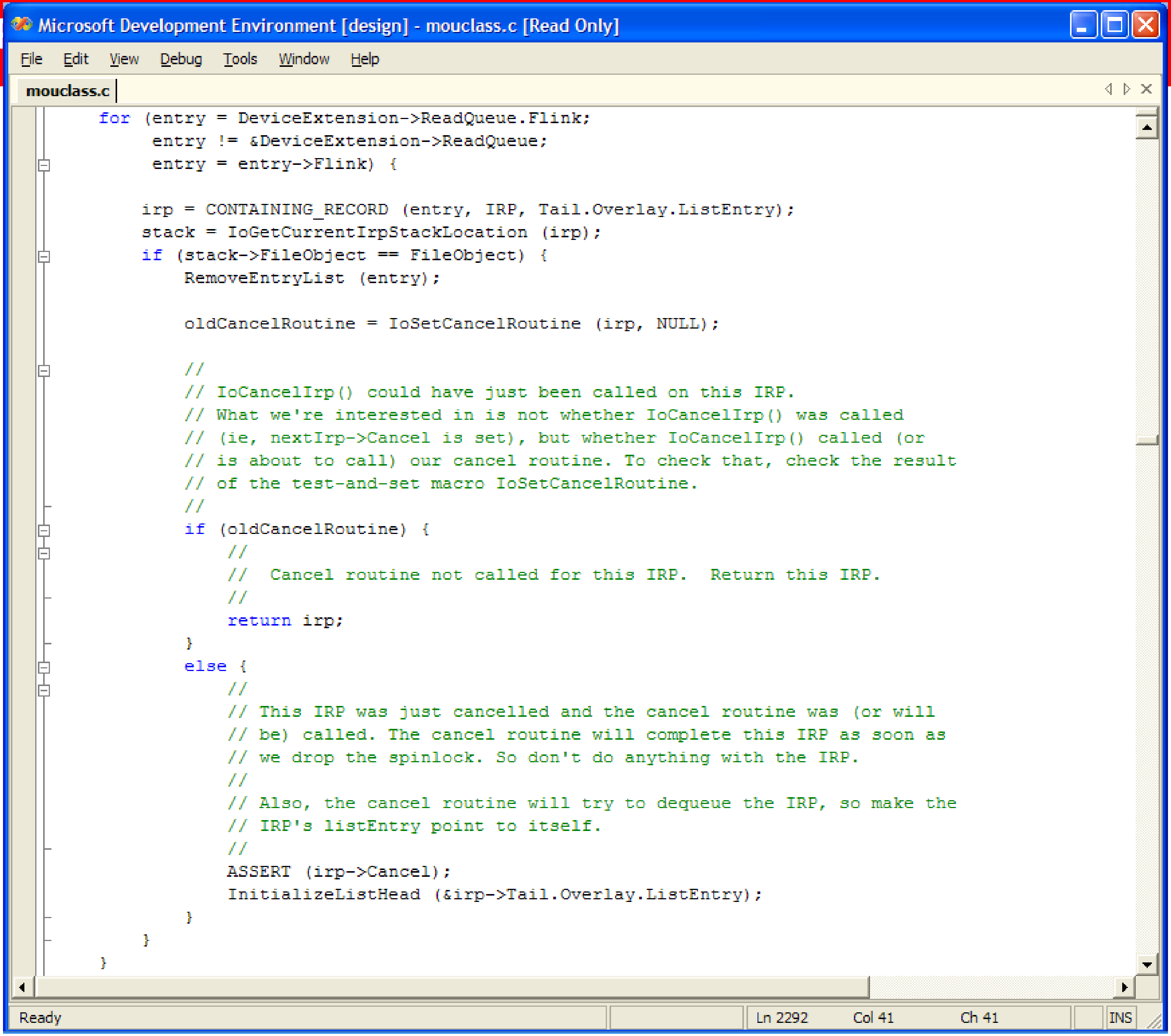
Exam

“The mouse device driver’s event handling routine always eventually terminates”

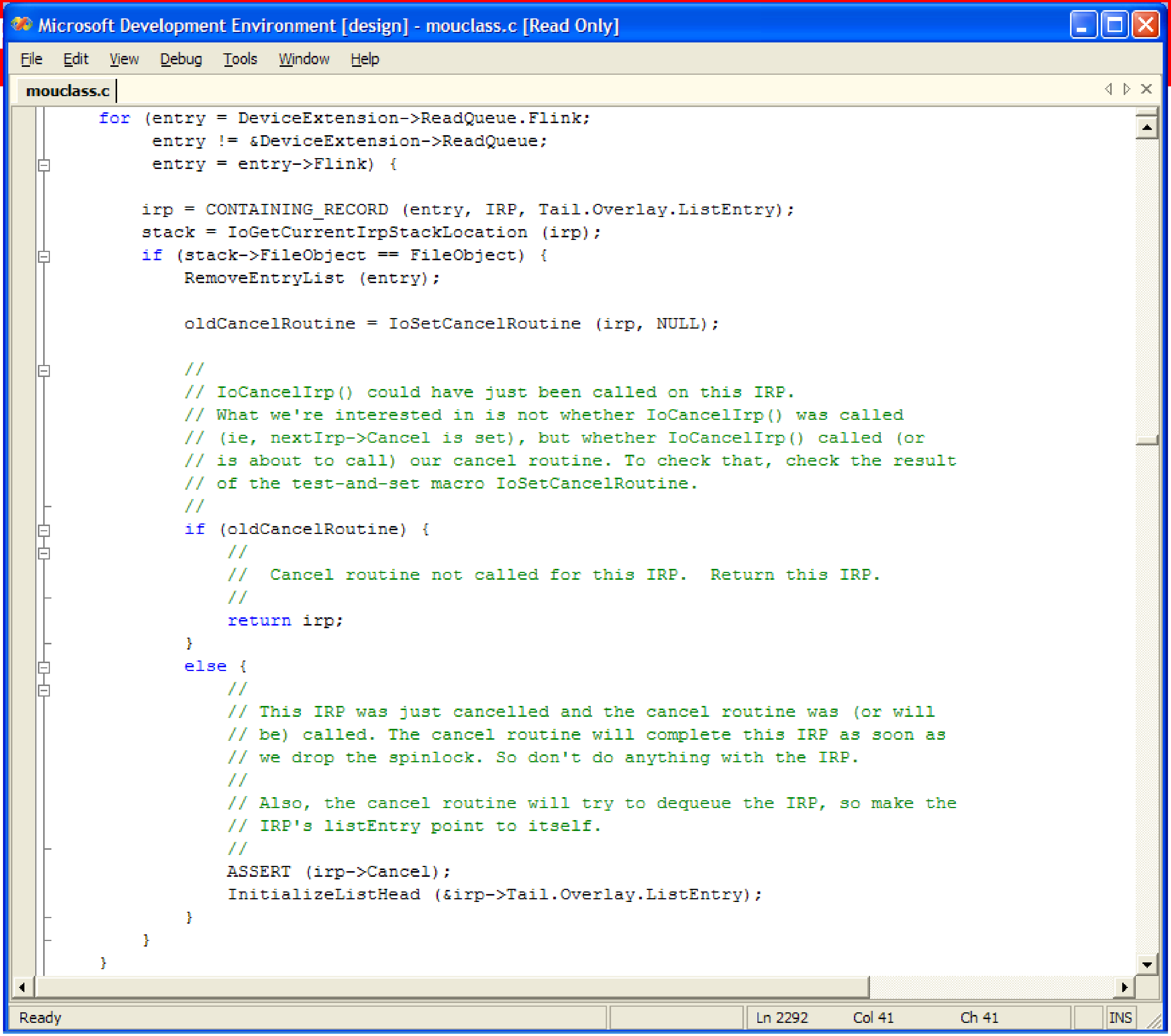
TERMINATOR



```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp() was called
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```



```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this IRP.
            // What we're interested in is not whether IoCancelIrp() was called
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```


Microsoft Development Environment [design] - mouclass.c [Read Only]
File
Edit
View
Debug
Tools
Window
Help
mouclass.c

```c
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp() was called
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Outline

- Introduction
- Termination basics
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

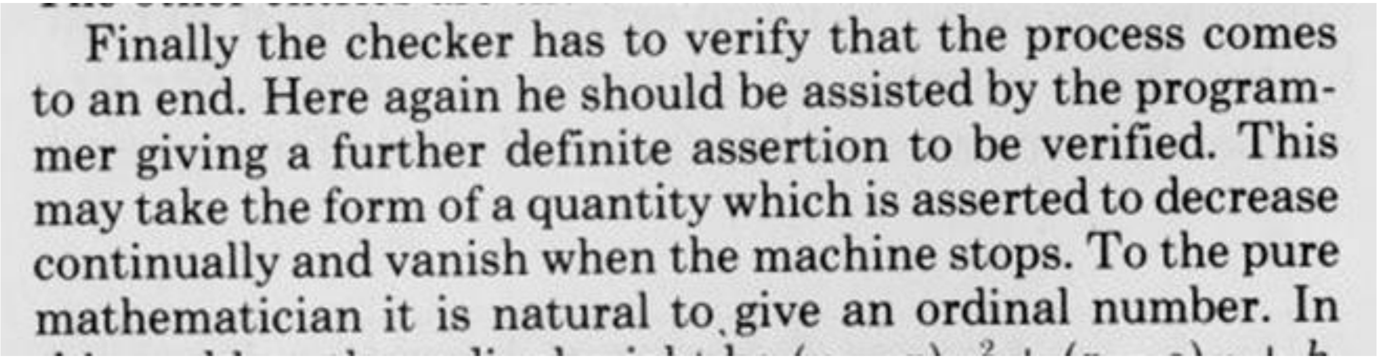
Outline

- Introduction
- Termination basics
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Proving termination

- Traditional termination proving method originally proposed by the forefathers of computing

- *E.g.* Turing, “Checking a large routine”, 1949

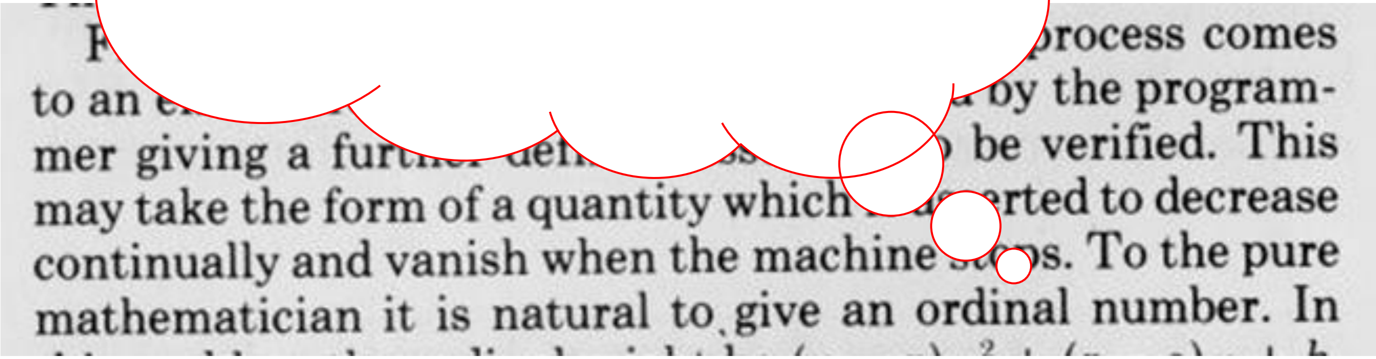


Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In

Proving termination

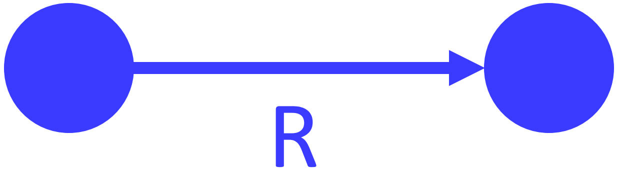
- Traditional termin proving method originally propose f computing

- *E.* ’, 1949

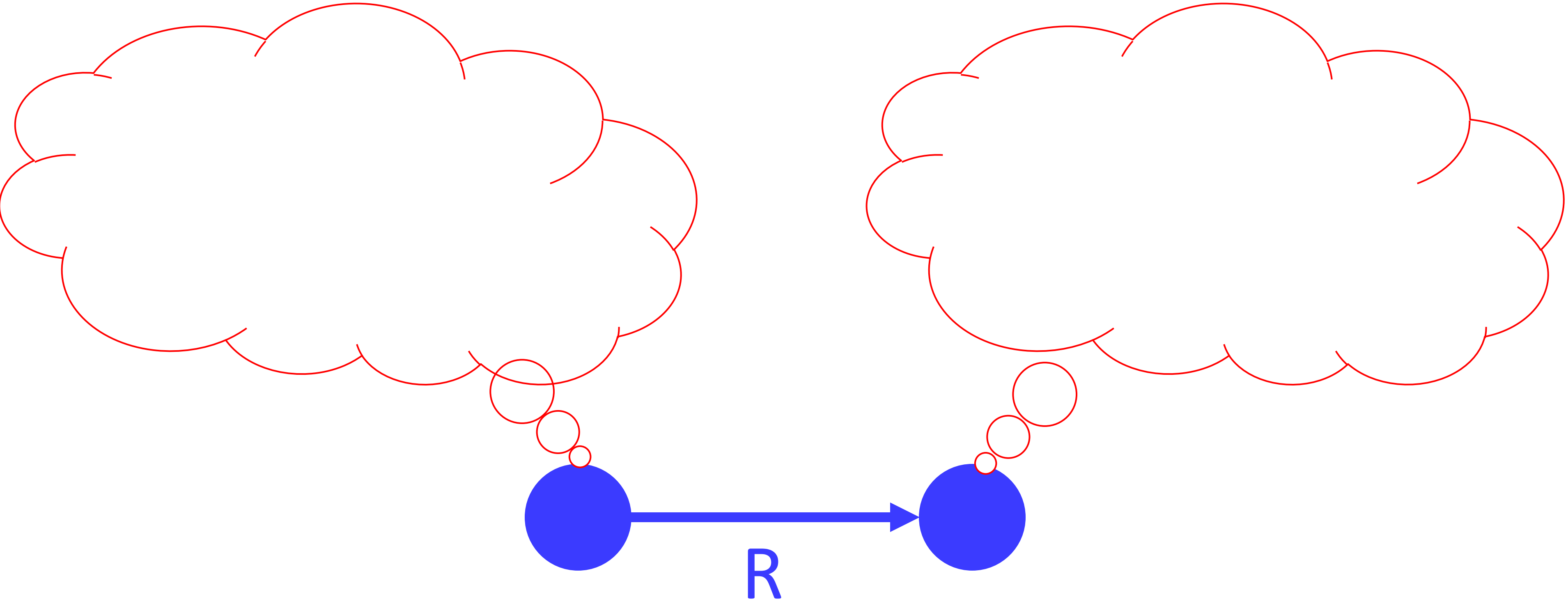


F process comes
to an e by the program-
mer giving a furt defi be verified. This
may take the form of a quantity which erted to decrease
continually and vanish when the machine ps. To the pure
mathematician it is natural to give an ordinal number. In

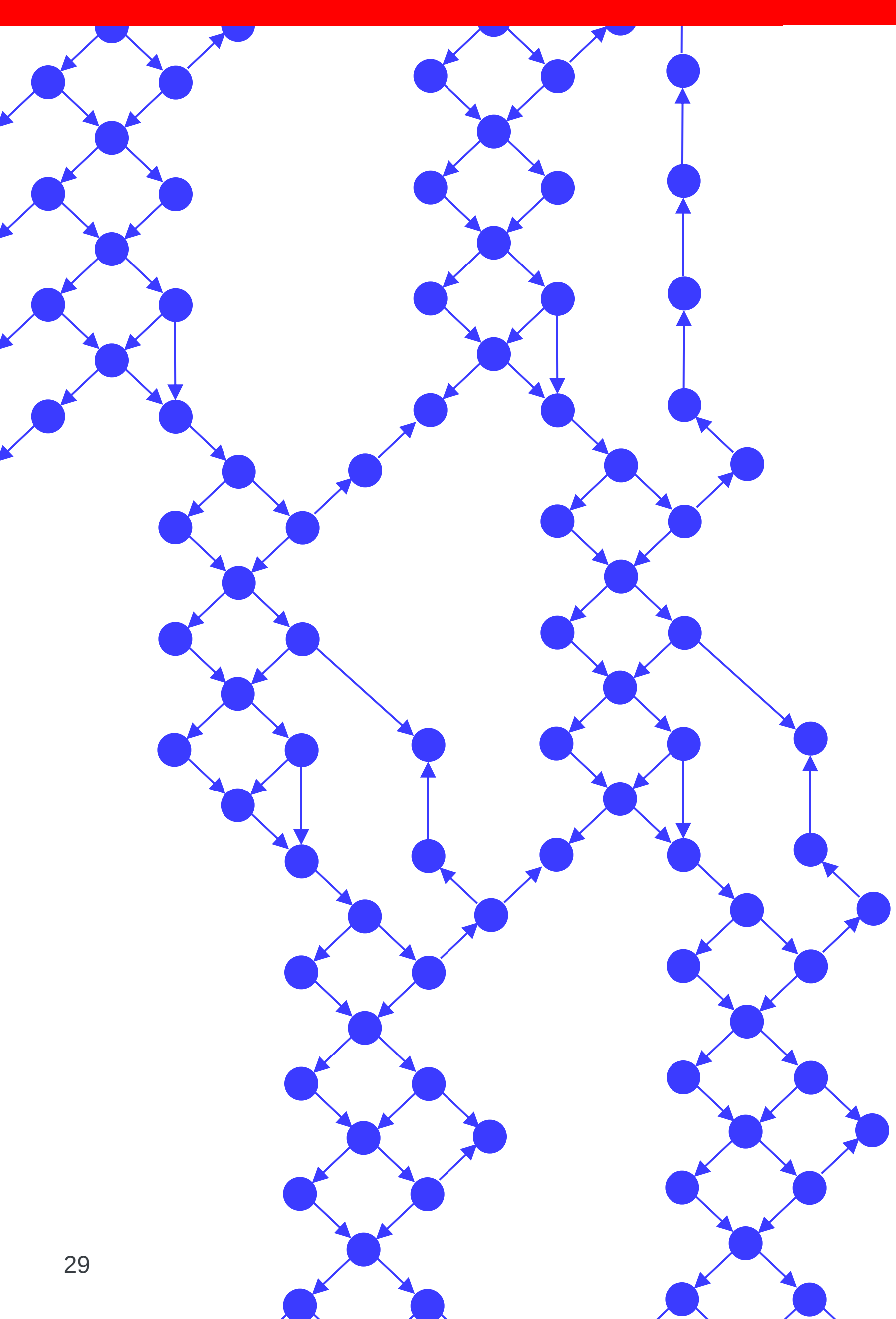
Proving termination



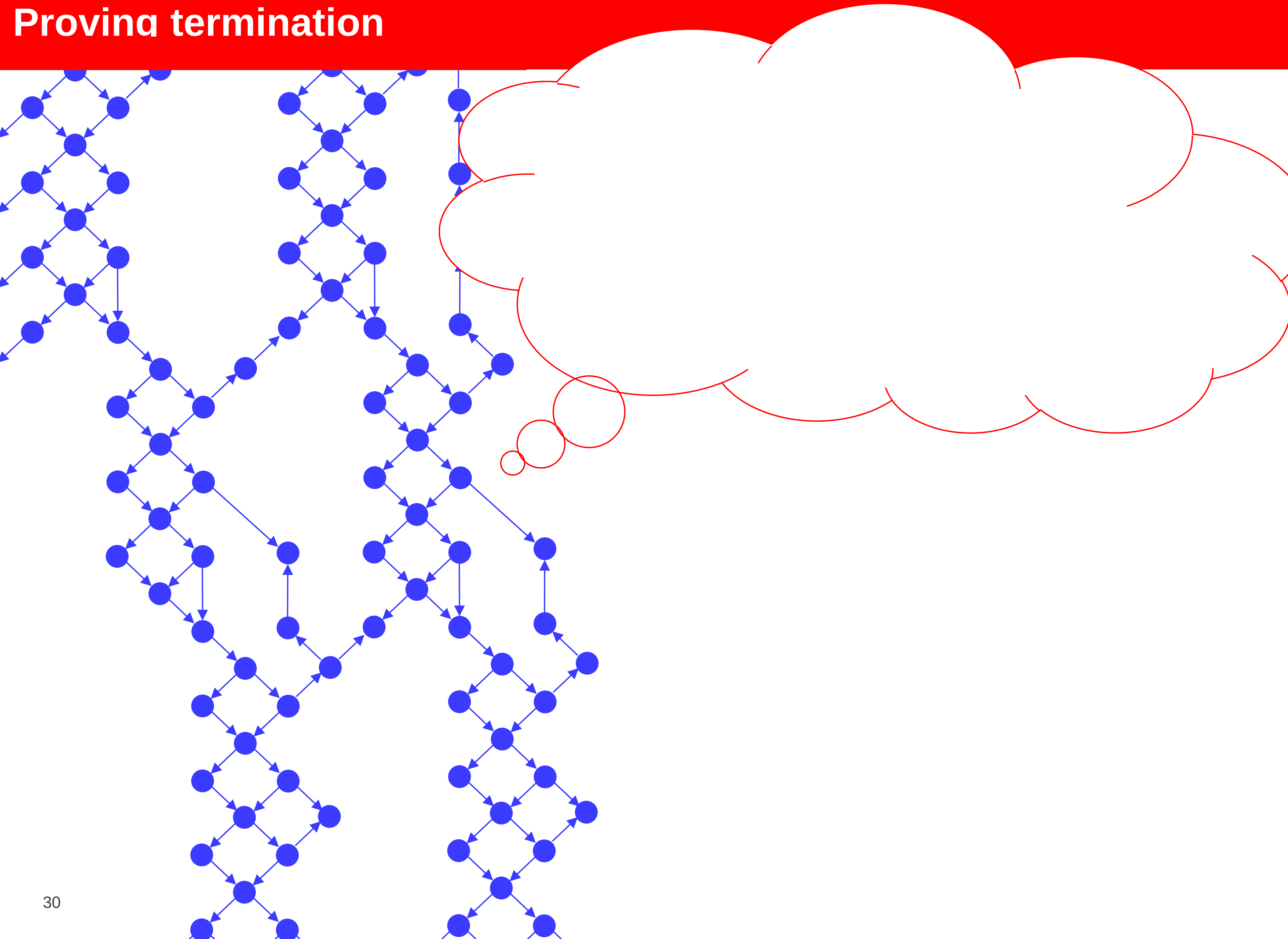
Proving termination



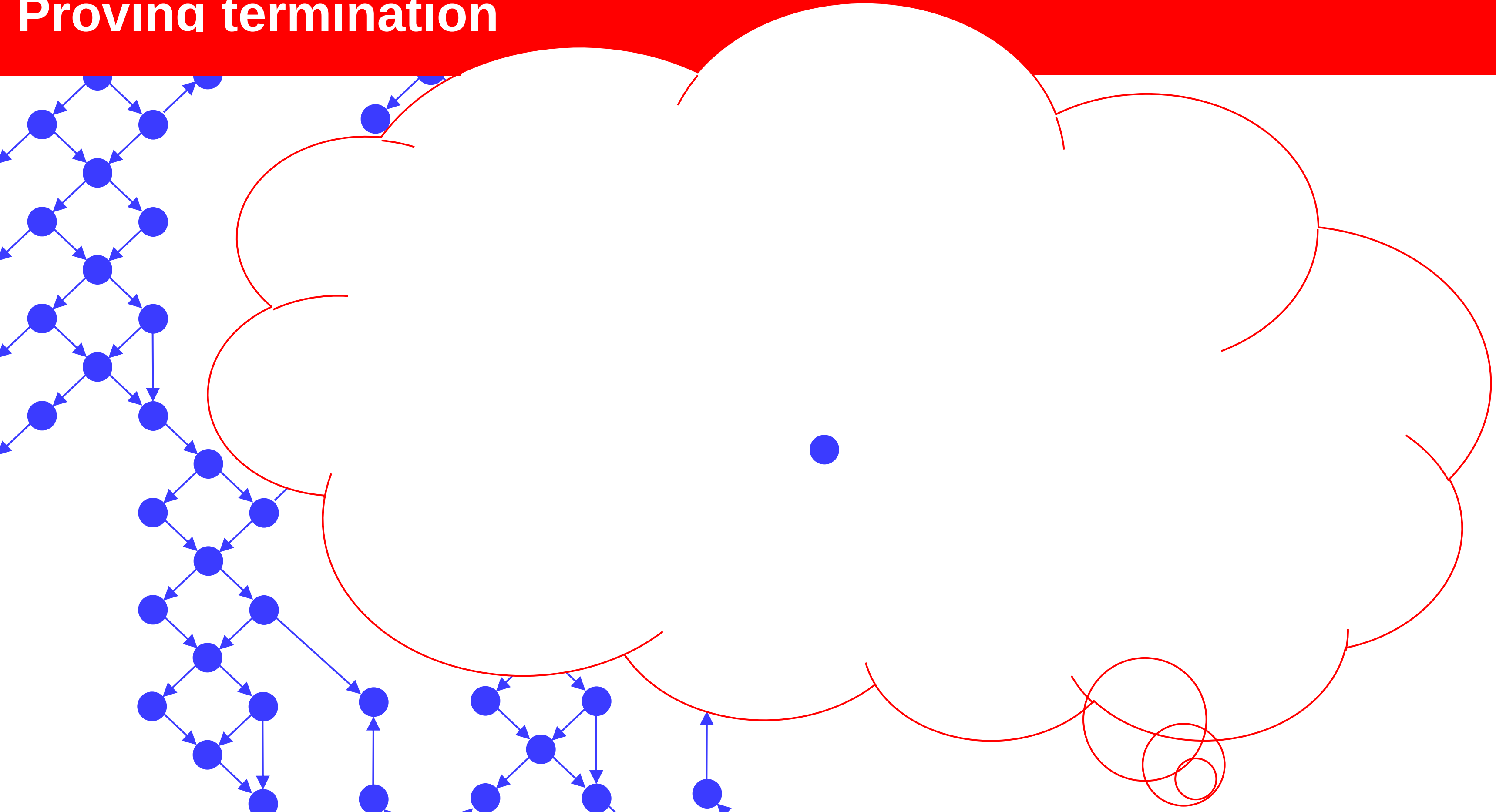
Proving termination

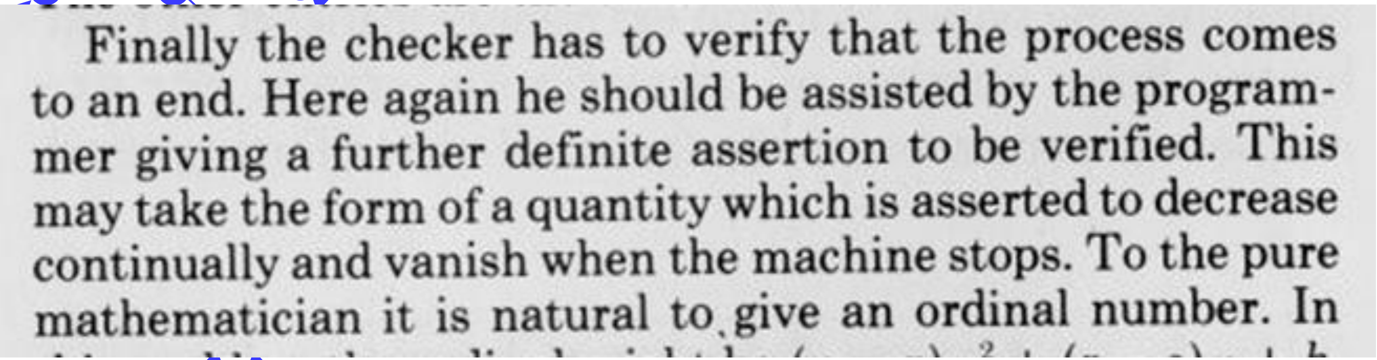


Proving termination



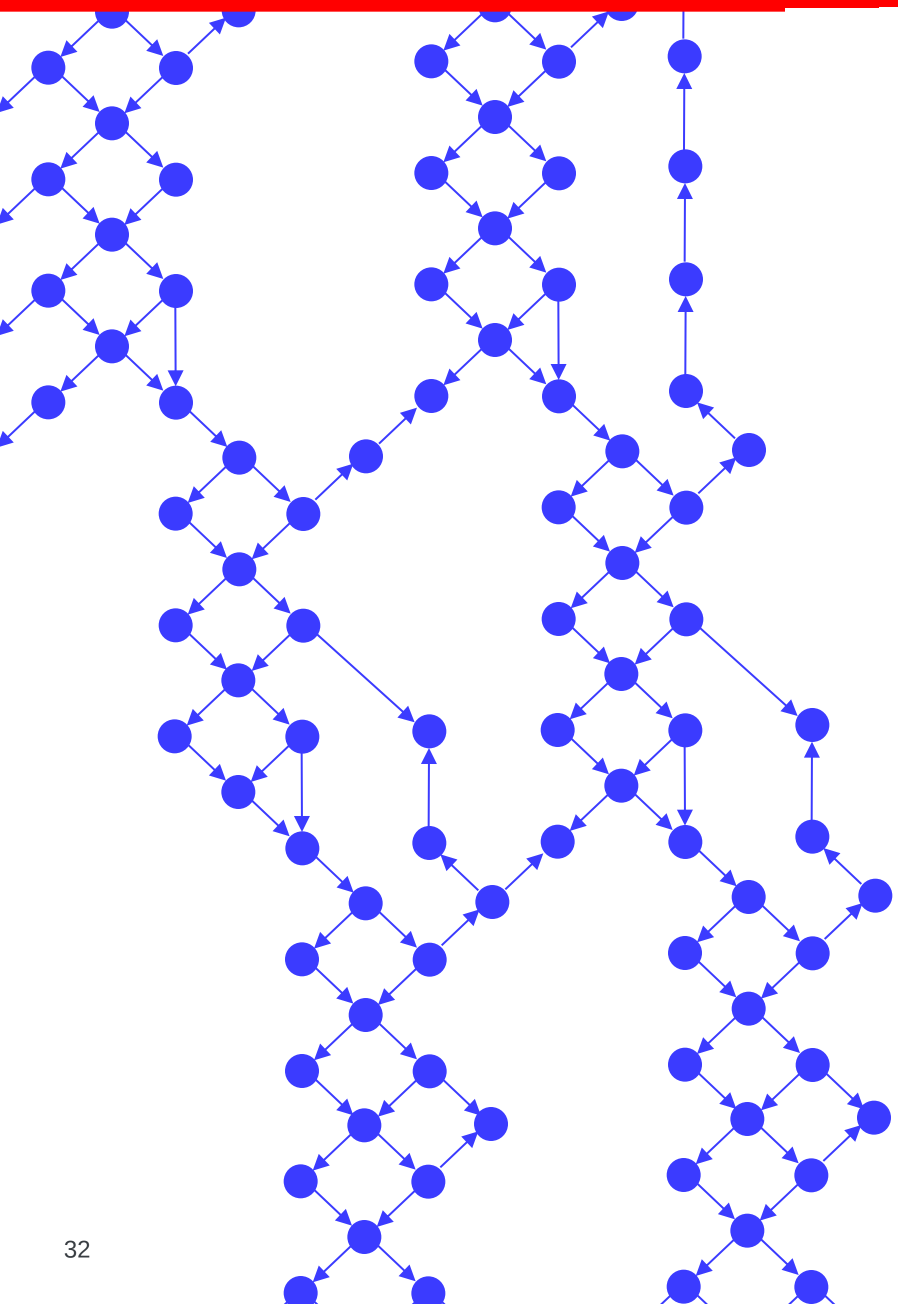
Proving termination

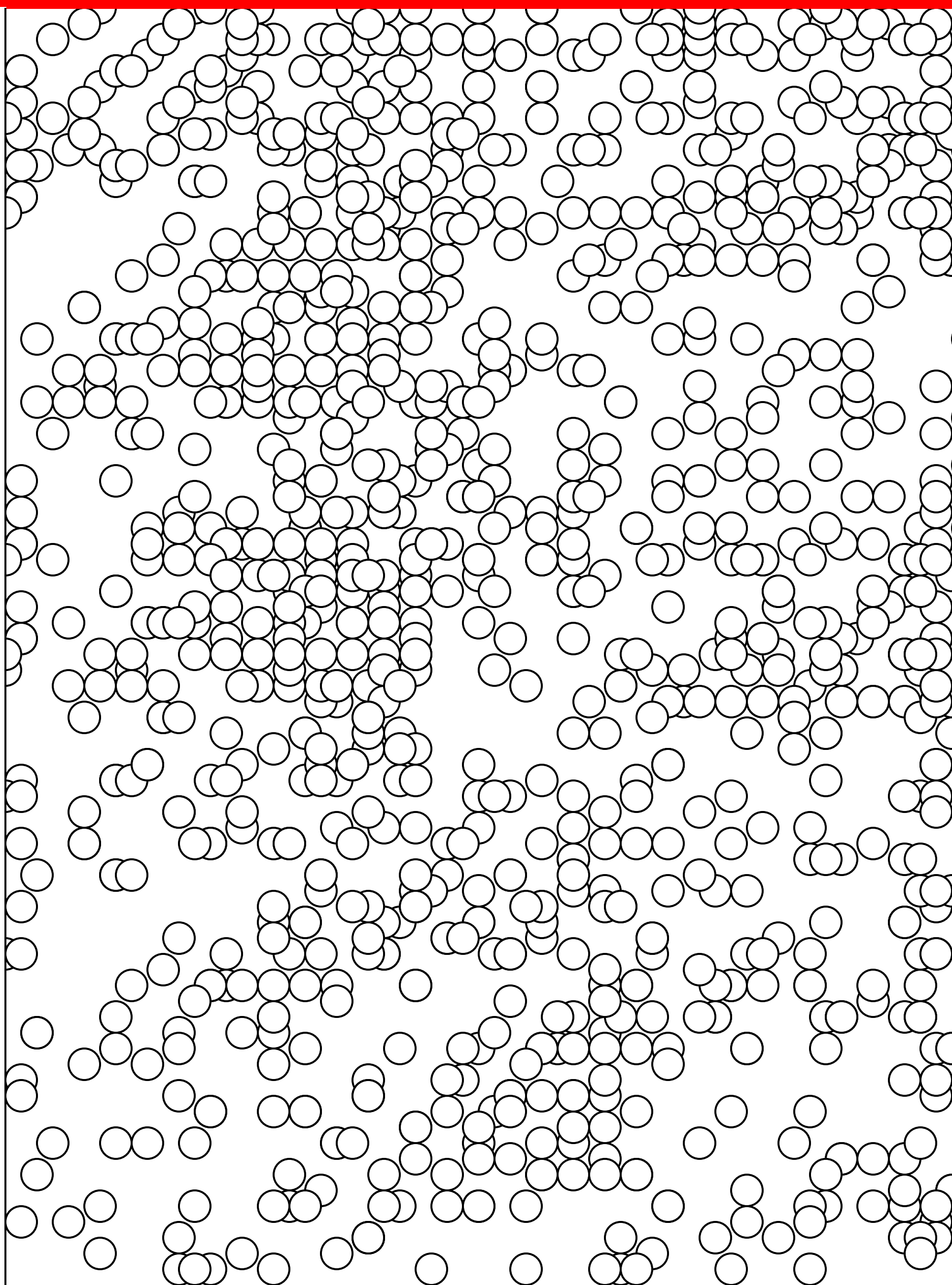




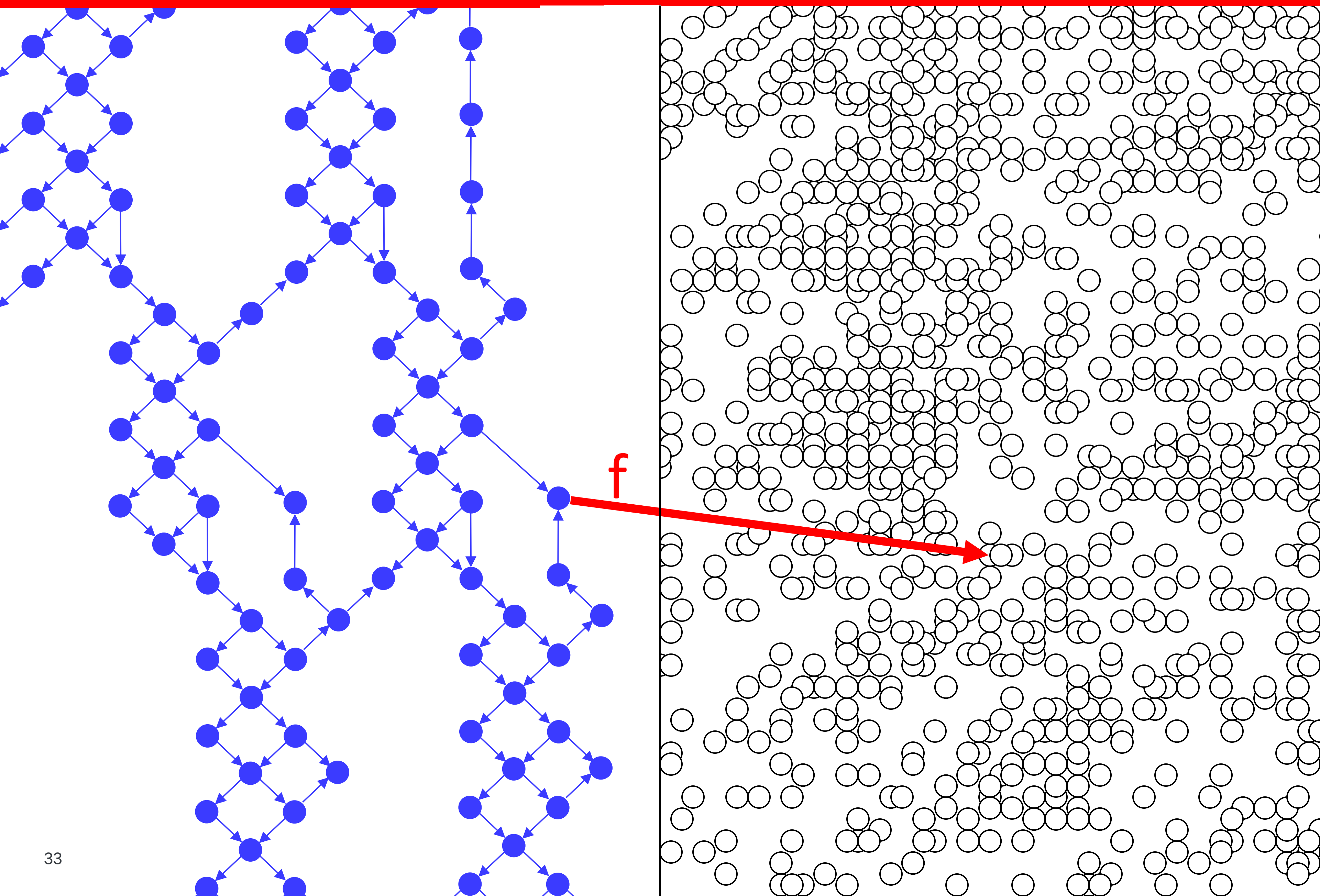
Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In

Proving termination

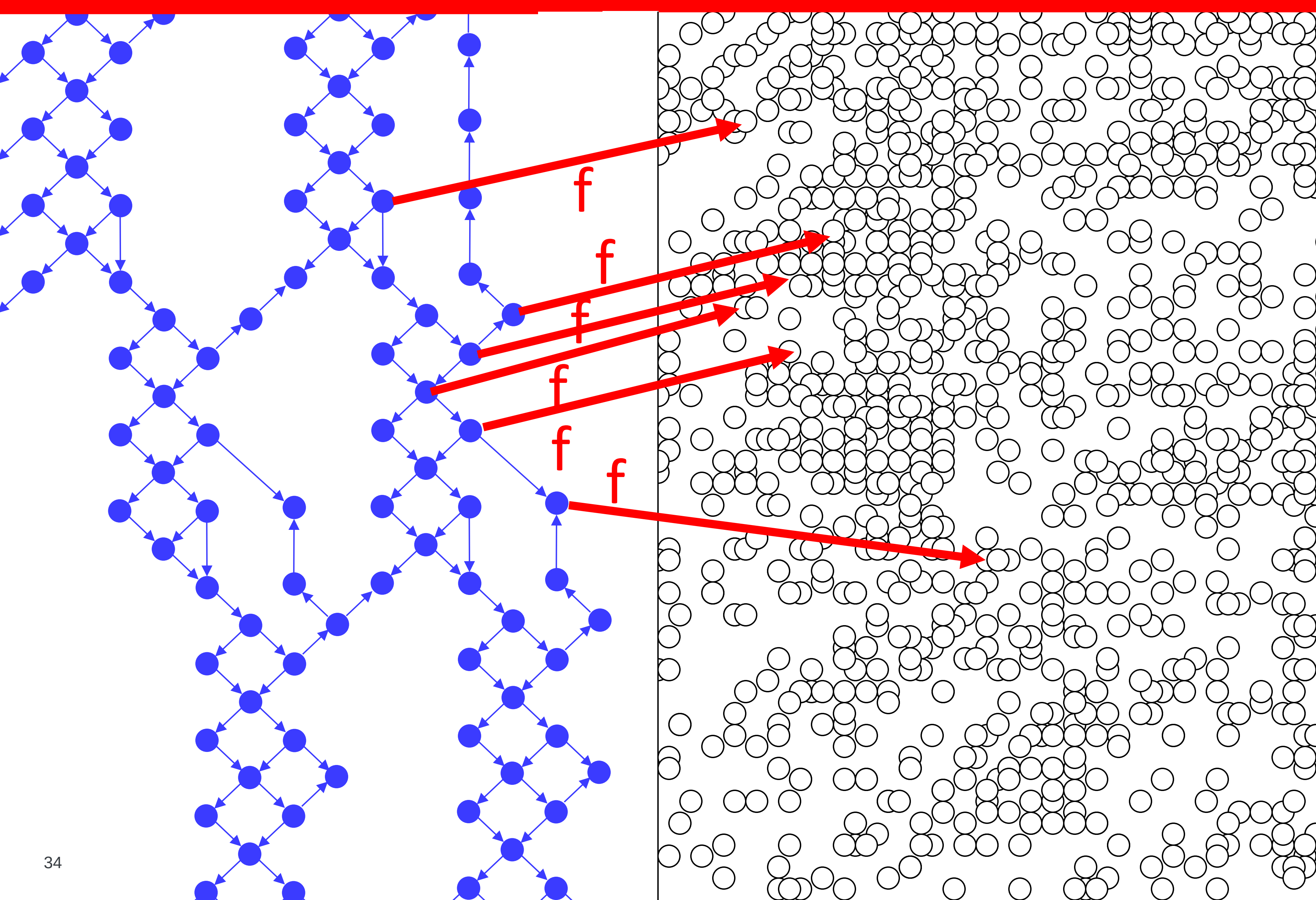




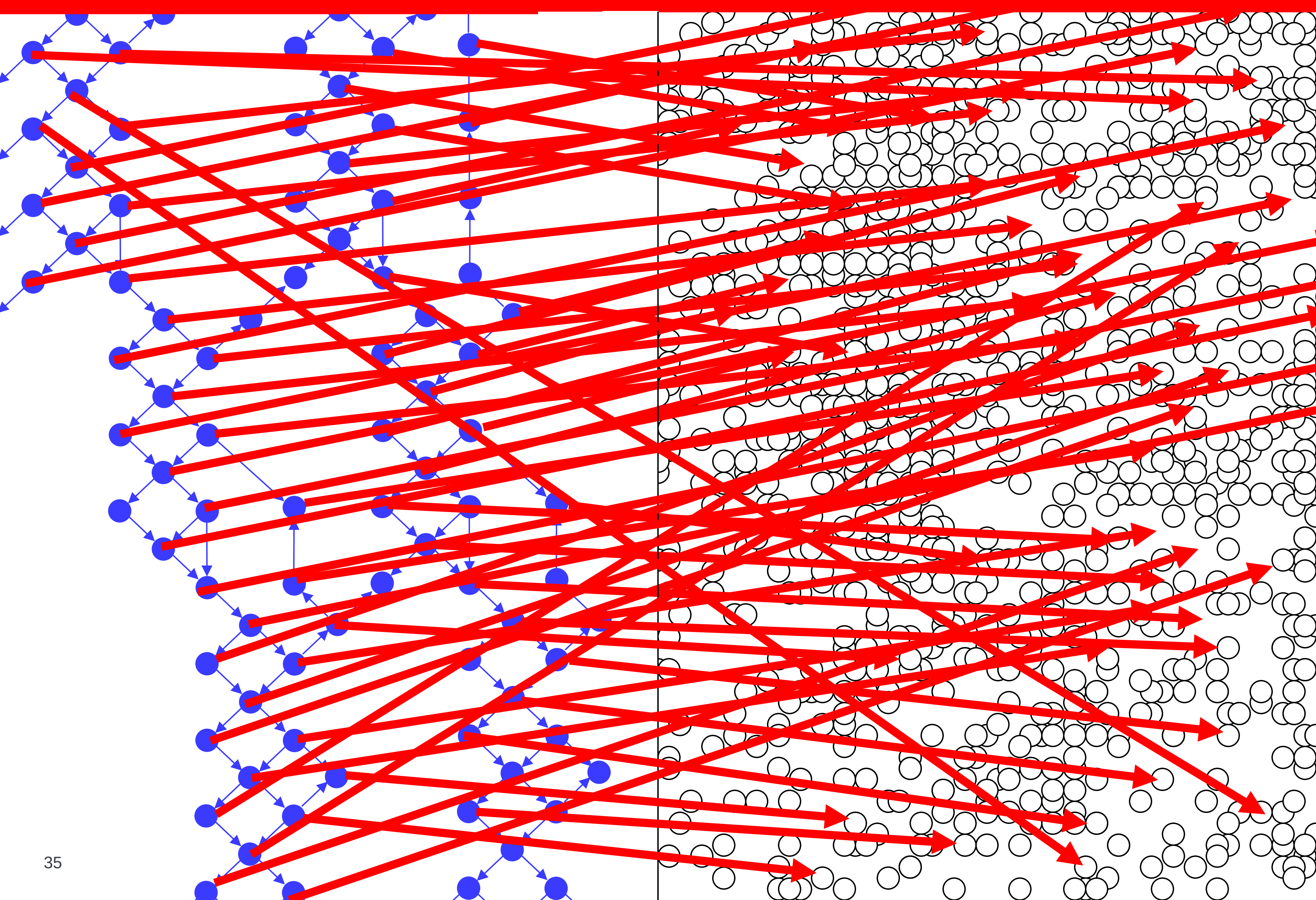
Proving termination



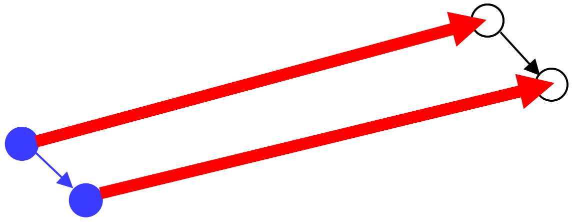
Proving termination



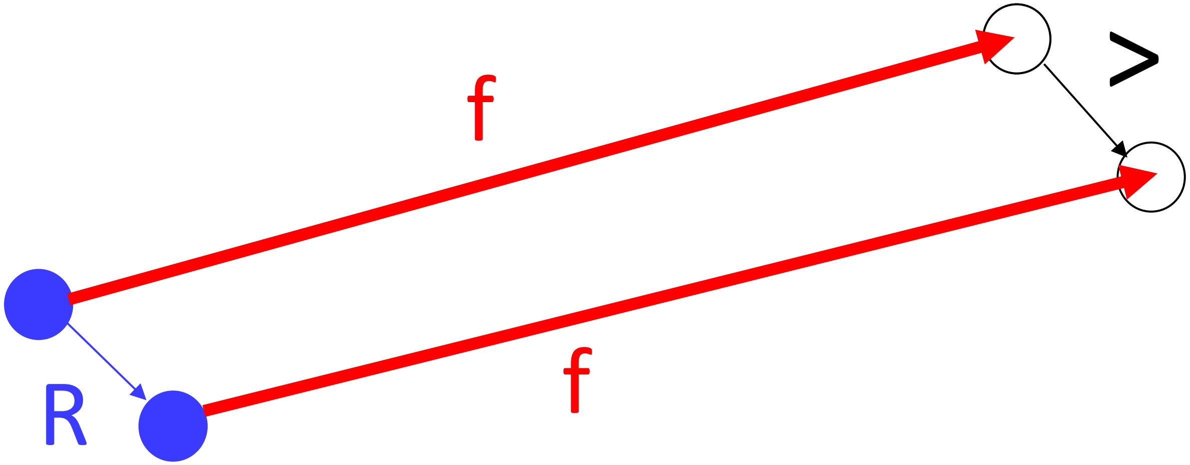
Proving termination



Proving termination

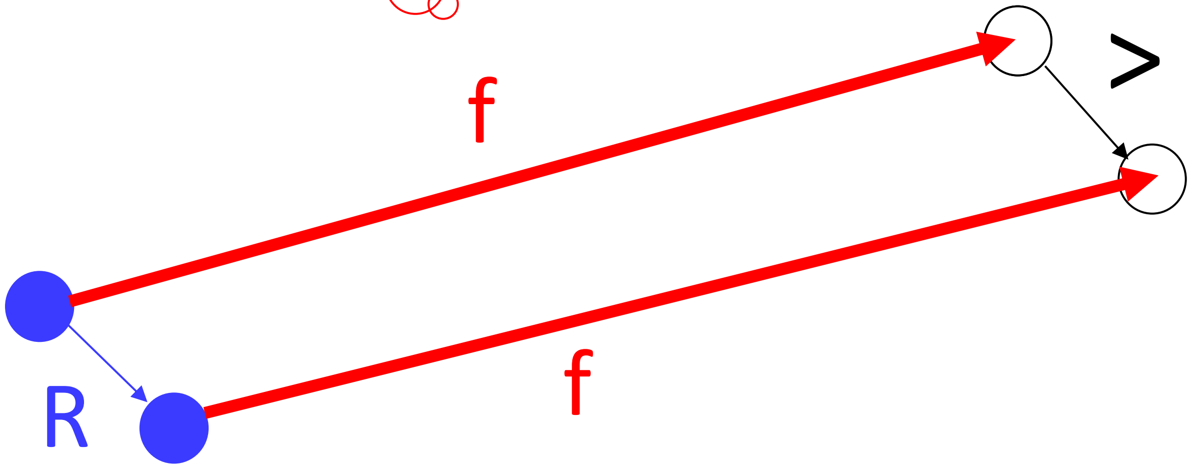


Proving termination



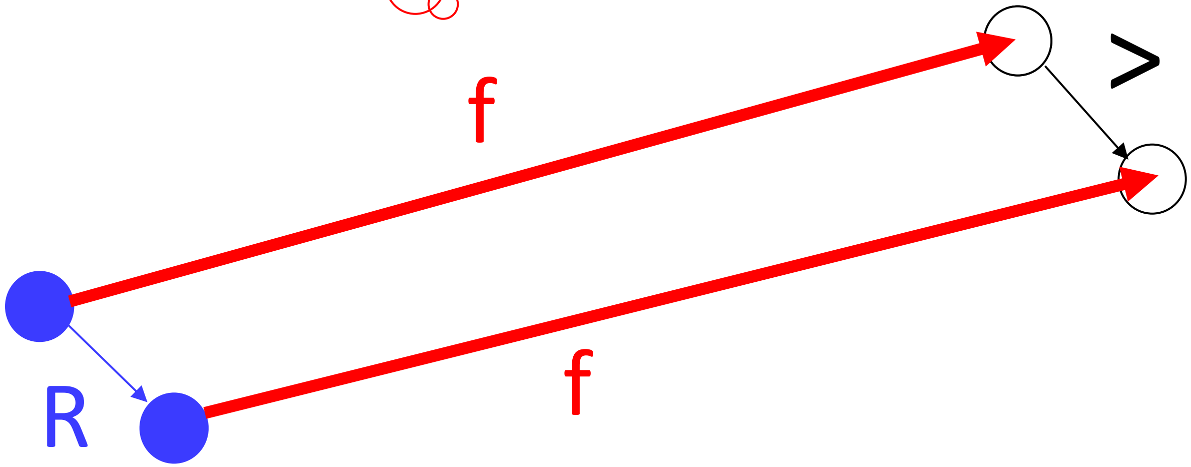
Pr

$$R \subseteq \unrhd_f$$



Pr

$$R \subseteq \unrhd_f$$



$$\unrhd_f \triangleq \{(s, t) \mid f(s) > f(t)\}$$

Pr

$$R \subseteq \trianglerighteq_f$$

Outline

- Introduction
- Termination basics
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

Outline

- Introduction
- Termination basics
- New advances for program termination proving
  - Proving termination argument validity
  - Finding termination arguments
- Conclusion

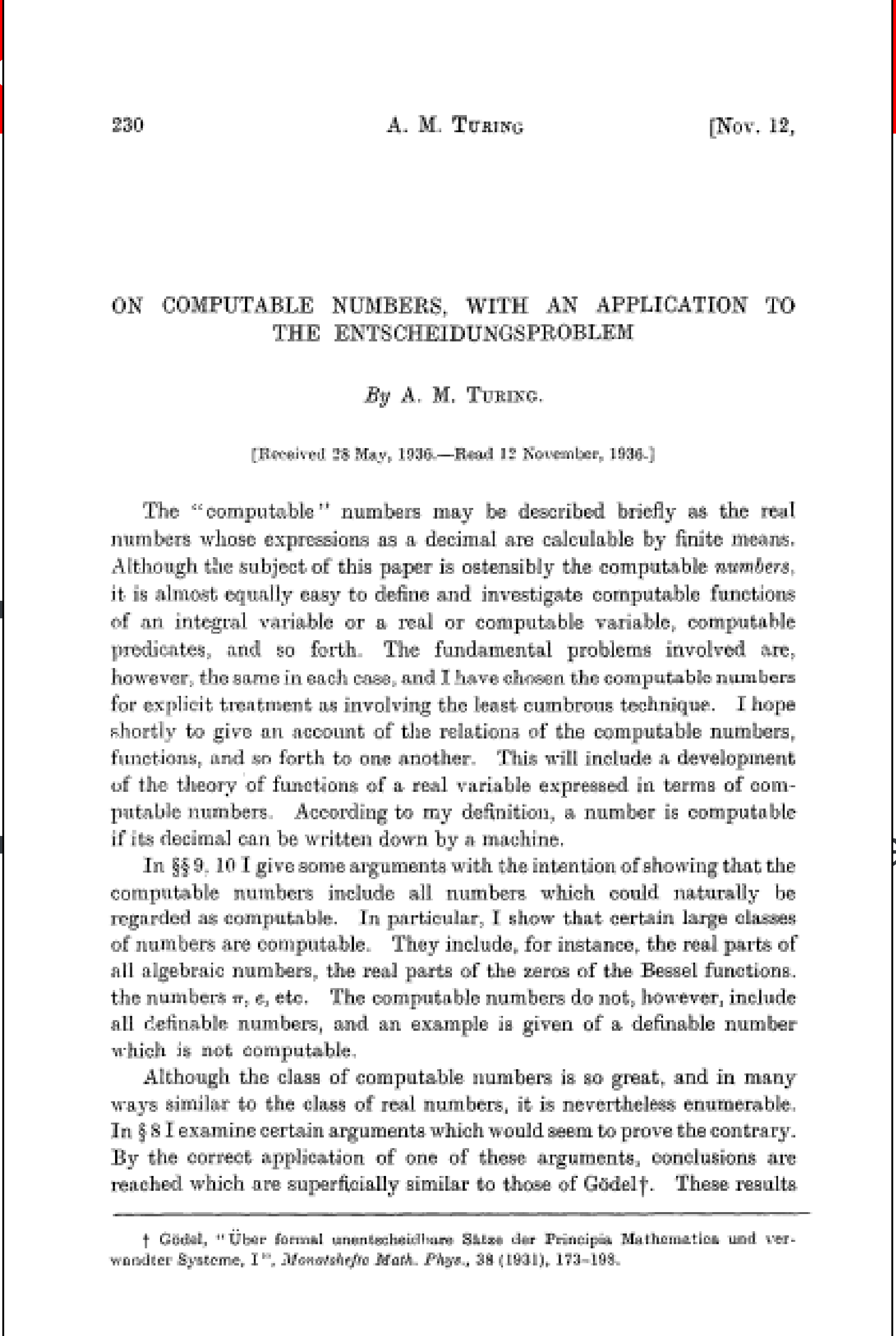
Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

Aut

→



230 A. M. TURING [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

rd in practice (and

ard in practice (and

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

- Finding an f such that $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

Automating the search for proofs

- Transition relations must be computed

$$R = U \cap [(U^*(I) \times U^*(I))]$$

- Technically, computing $U^*(I)$ is undeciable, so we must find a sound over-approximation using available techniques:

$$U^*(I) \subseteq Q$$

- Q represents an infinite set of states, but has a compact expression

Automating the search for proofs

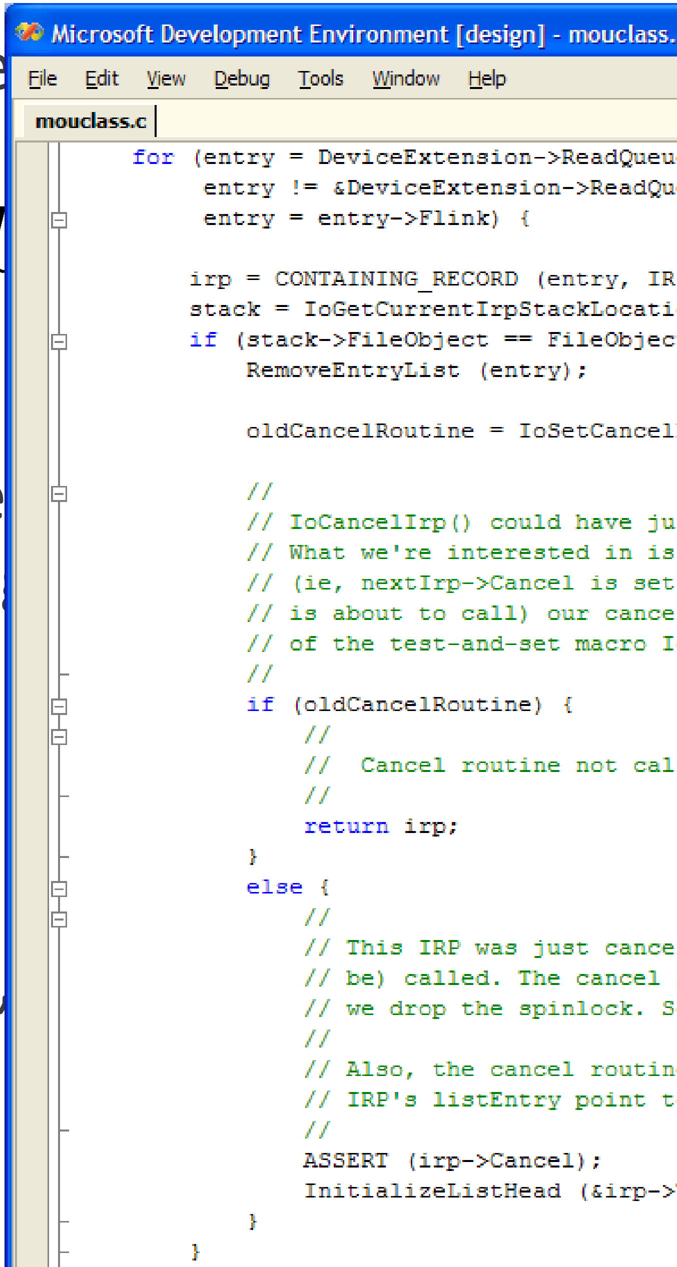
- Transition relations must be compute

$$R = U \cap [(U^*(I) \times U$$

- Technically, computing $U^*(I)$ is unde find a sound over-approximation usin techniques:

$$U^*(I) \subseteq Q$$

- Q represents an infinite set of states, expression


Microsoft Development Environment [design] - mouclass.
File Edit View Debug Tools Window Help
mouclass.c

```
    for (entry = DeviceExtension->ReadQueu
         entry != &DeviceExtension->ReadQu
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IR
        stack = IoGetCurrentIrpStackLocati
        if (stack->FileObject == FileObjec
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancel

            //
            // IoCancelIrp() could have ju
            // What we're interested in is
            // (ie, nextIrp->Cancel is set
            // is about to call) our cance
            // of the test-and-set macro I
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not cal
                //
                return irp;
            }
            else {
                //
                // This IRP was just cance
                // be) called. The cancel
                // we drop the spinlock. S
                //
                // Also, the cancel routin
                // IRP's listEntry point t
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->
            }
        }
```

Automating the search for proofs

- Transition relations must be computed

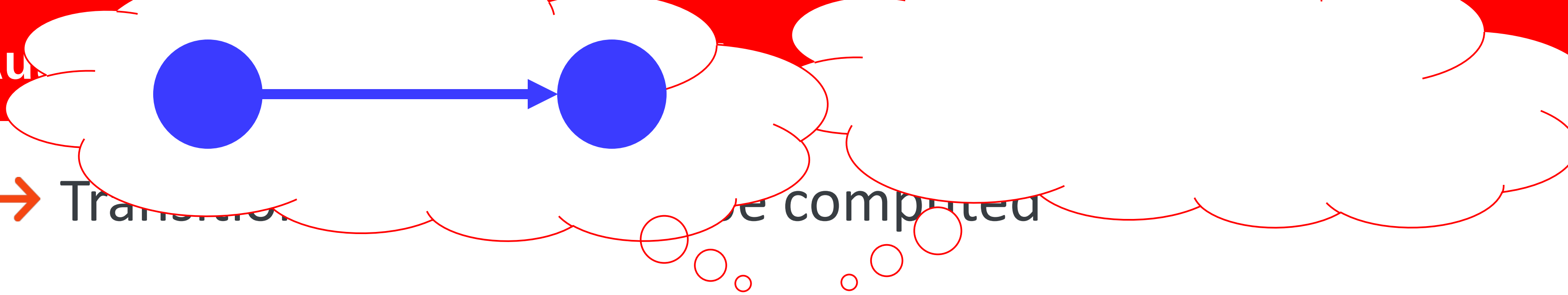
$$R = U \cap [(U^*(I) \times U^*(I))]$$

- Technically, computing $U^*(I)$ is undeciable, so we must find a sound over-approximation using available techniques:

$$U^*(I) \subseteq Q$$

- Q represents an infinite set of states, but has a compact expression

Au



→ Transitio be computed

$$R = U \cap [(U^*(I) \times U^*(I))]$$

→ Technically, computing $U^*(I)$ is undeciable, so we must find a sound over-approximation using available techniques:

$$U^*(I) \subseteq Q$$

→ Q represents an infinite set of states, but has a compact expression

Automating the search for proofs

- We use an over-approximation of the transition relation

$$R' = U \cap [Q \times Q]$$

- Since $R \subseteq R'$, we can prove termination by showing

$$R' \subseteq \trianglerighteq_f$$

- Meaning: there might be unrealistic transitions that we have to worry about

Automating the search for proofs

- → In practice, its extremely hard to find the right overapproximation Q
- → Luckily: recent breakthroughs in safety proving now make this possible.
- → In fact: the checking the validity of a termination argument can be directly encoded as a safety property
- → Tools like **SLAM** can be used to prove validity

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

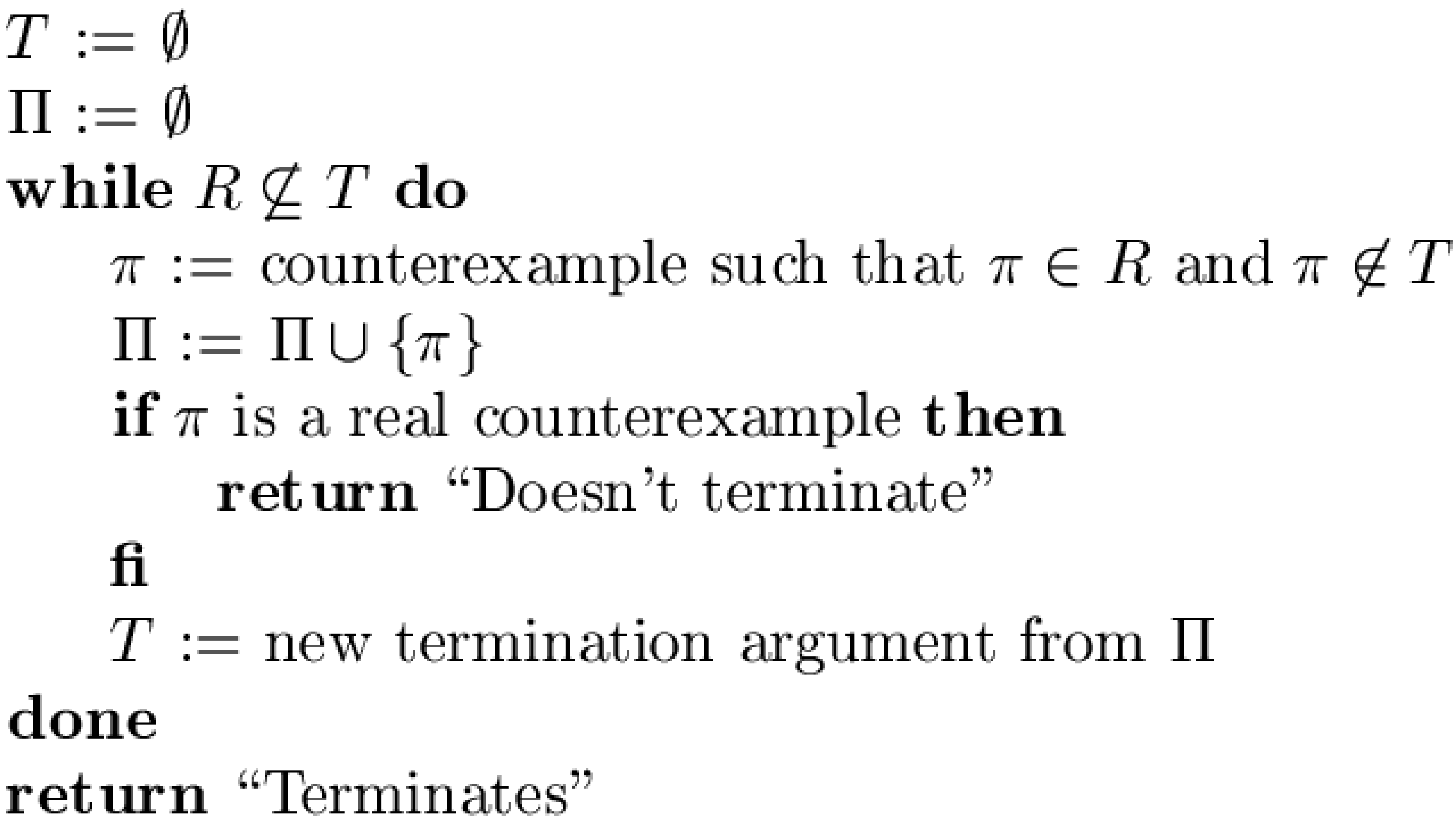
Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \unrhd_f$ is hard in practice (and undecidable in theory)

- Finding an f such that $R \subseteq \unrhd_f$ is hard in practice (and undecidable in theory)

Refinement-based termination proving



```
$T := \emptyset$
$\Pi := \emptyset$
while $R \not\subseteq T$ do
    $\pi :=$ counterexample such that $\pi \in R$ and $\pi \notin T$
    $\Pi := \Pi \cup \{\pi\}$
    if $\pi$ is a real counterexample then
        return "Doesn't terminate"
    fi
    $T :=$ new termination argument from $\Pi$
done
return "Terminates"
```

Refinement-based termination proving

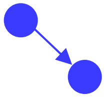
$$R \subseteq \emptyset$$

Refinement-based termination proving

$$R \overset{\times}{\subseteq} \emptyset$$

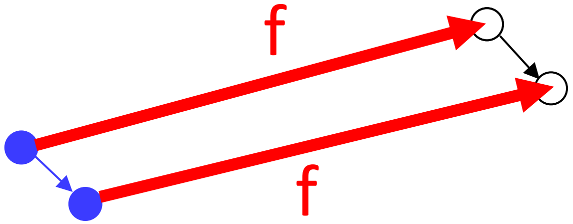
Refinement-based termination proving

$R \subseteq \emptyset$ ✗



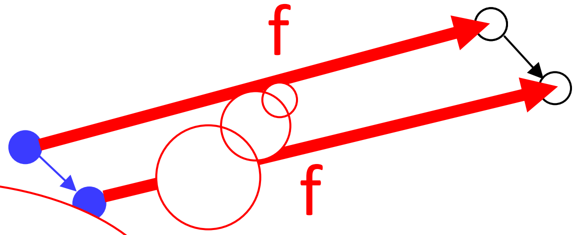
Refinement-based termination proving

$R \subseteq \emptyset$ ✗



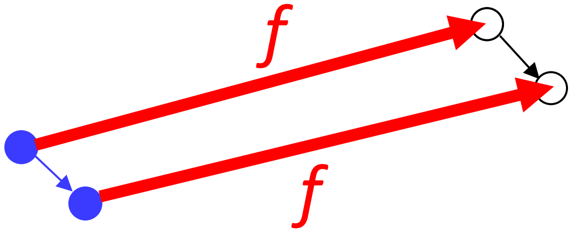
Refinement-based termination proving

$R \cancel{\subseteq} \emptyset$



Refinement-based termination proving

$$R \cancel{\subseteq} \emptyset$$

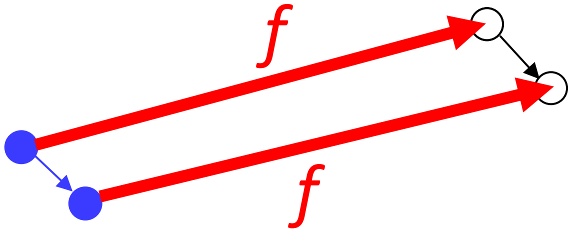


Refinement-based termination proving

$$R \cancel{\subseteq} \emptyset$$

$$\downarrow$$

$$R \subseteq \trianglerighteq_f$$

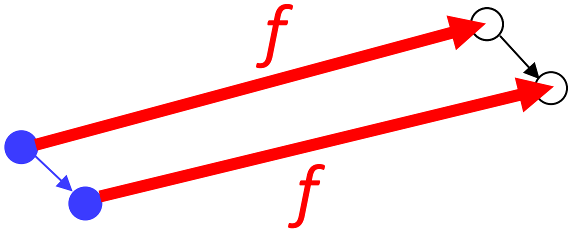


Refinement-based termination proving

$$R \overset{✗}{\subseteq} \emptyset$$

$$\downarrow$$

$$R \overset{✗}{\subseteq} \trianglerighteq_f$$

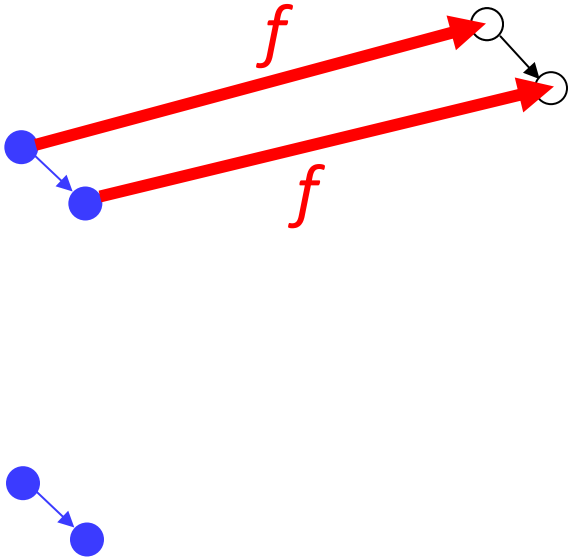


Refinement-based termination proving

R ✗ $\subseteq \emptyset$

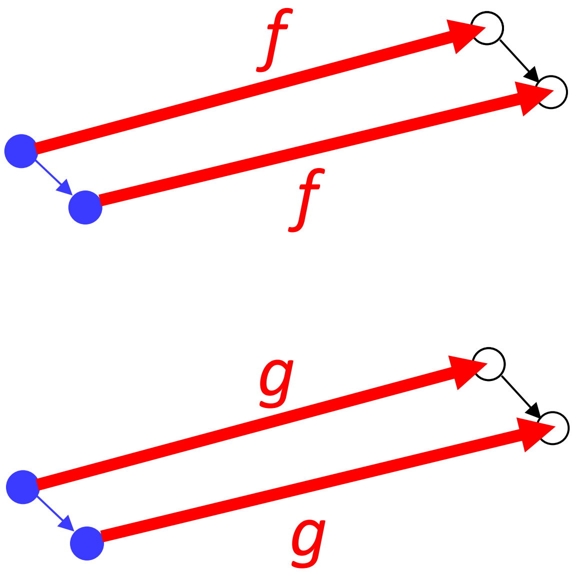
↓

R ✗ $\subseteq \unrhd_f$



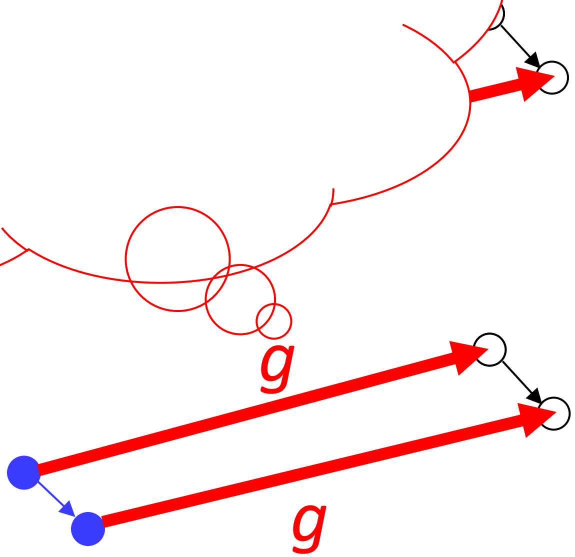
Refinement-based termination proving

$$\begin{tikzcd} R \cancel{\subseteq} \emptyset \arrow[d] \\ R \cancel{\subseteq} \trianglerighteq_f \end{tikzcd}$$



↓

$$R \cancel{\subseteq} \unrhd_f$$

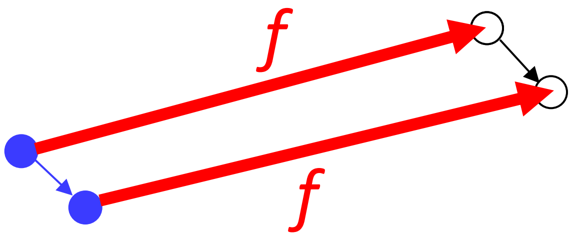


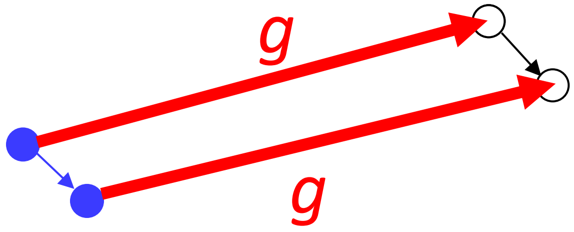
Refinement-based termination proving

$$R \subseteq \emptyset$$ ✗

$$\downarrow$$

$$R \subseteq \trianglerighteq_f$$ ✗





Refinement-based termination proving

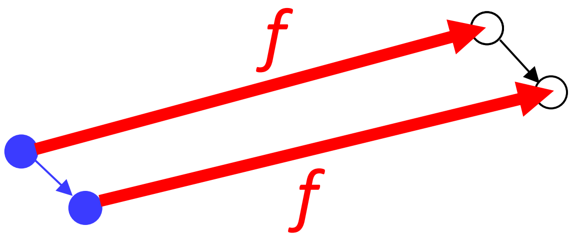
$$R \not\subseteq \emptyset$$

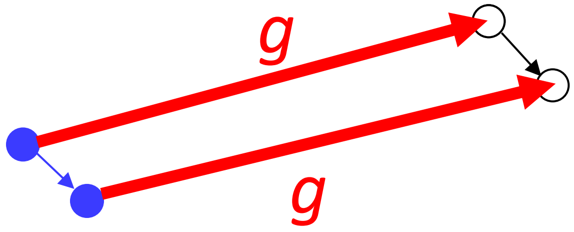
$$\downarrow$$

$$R \not\subseteq \trianglerighteq_f$$

$$\downarrow$$

$$R \subseteq \trianglerighteq_g$$





Refinement-based termination proving

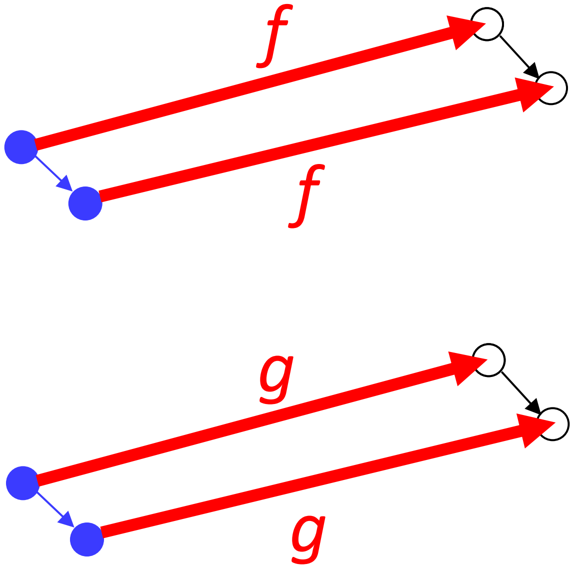
$R \subseteq \emptyset$ ✗

↓

$R \subseteq \unrhd_f$ ✗

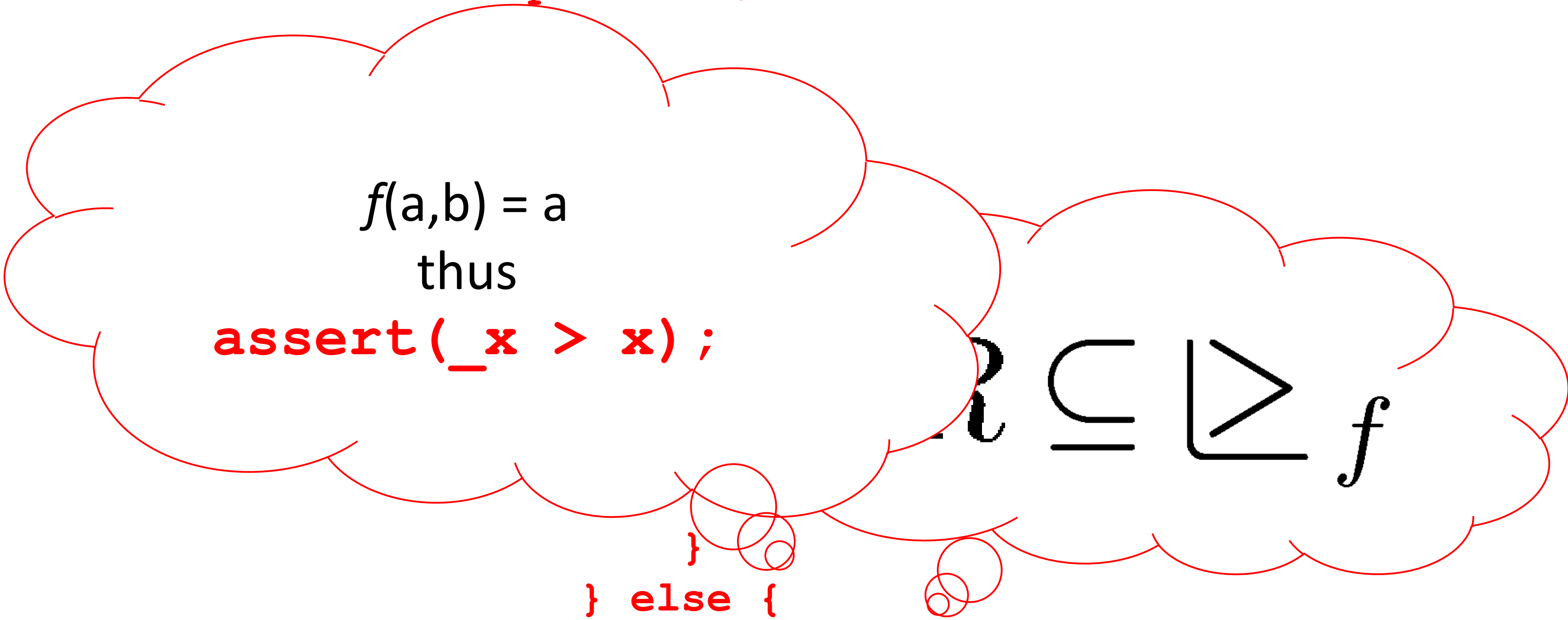
↓

$R \subseteq \unrhd_g$ ✓



Terminator

```
copied = 0;
```



f(a,b) = a

thus

```
assert(_x > x);
```

$R \subseteq \trianglerighteq_f$

```
            }
        } else {
            assert( f(_x,_y) > f(x,y) );
            exit();
        }
```

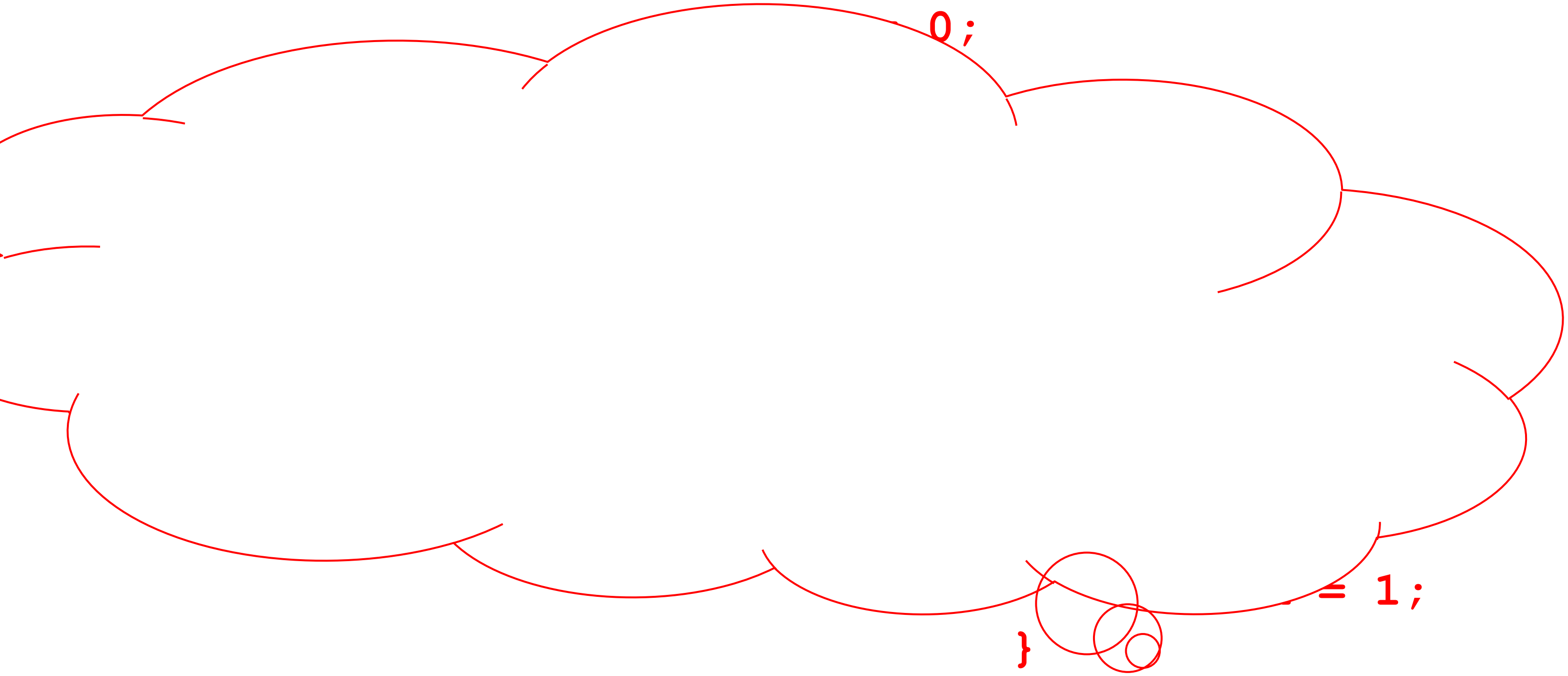
TERMINATOR reduction

```
if (!copied) {
    if (*) {
        _x = x;
        _y = y;
        copied = 1;
    }
} else {
    assert (f(_x,_y) > f(x,y));
}
```

Terminator

```
}
    }
  } else {
    assert((_x>x && _y>=y) || _y>y);
    exit();
  }
```

Terminator



```
 0;
       = 1;
  }
} else {
  assert(_x>x);
  exit();
}
```

Automating the search for proofs

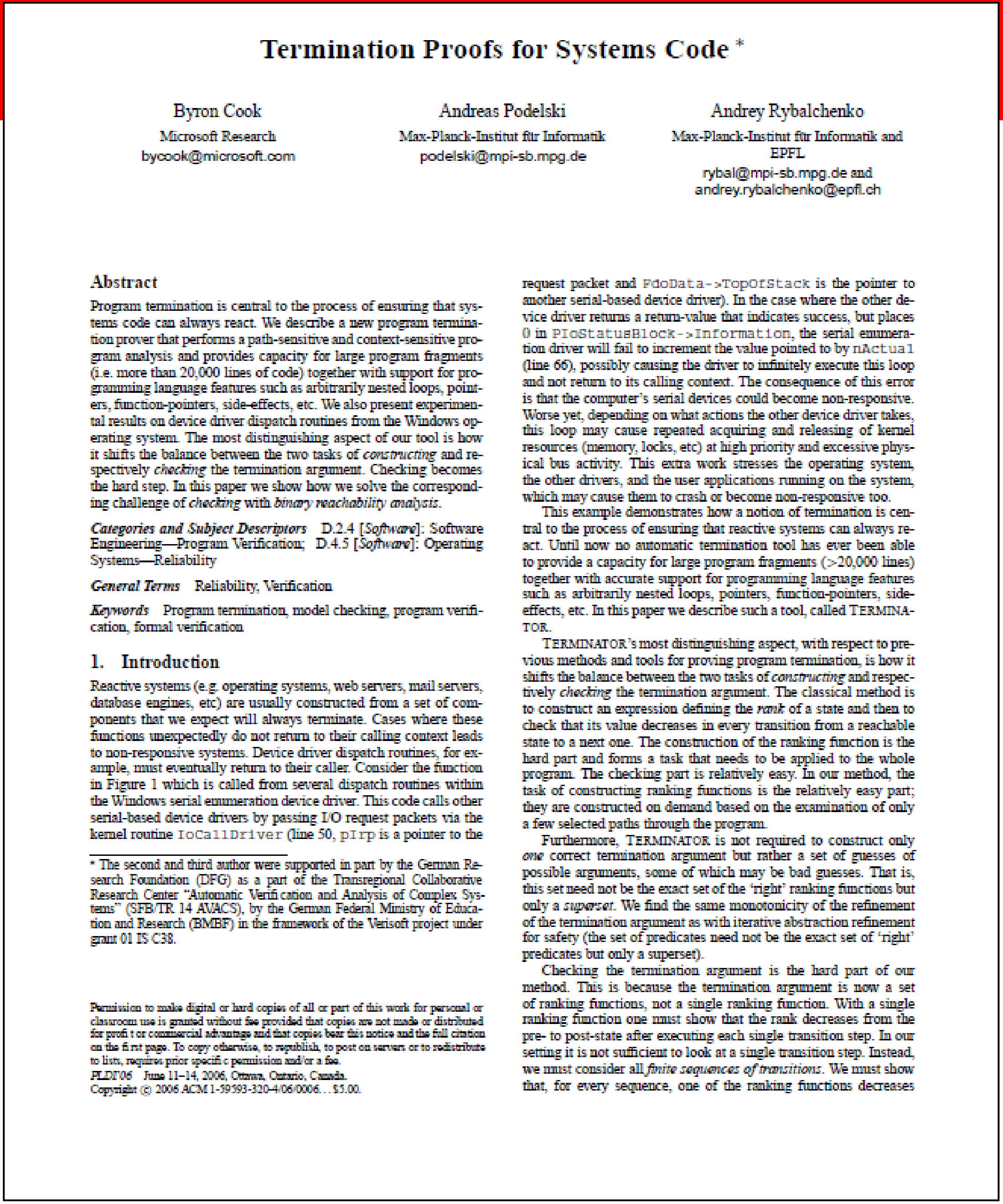
→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)

Automating the search for proofs

→ Difficulties:

- Proving the inclusion $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)
- Finding an f such that $R \subseteq \trianglerighteq_f$ is hard in practice (and undecidable in theory)



Termination Proofs for Systems Code [*]

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
Max-Planck-Institut für Informatik and
EPFL
rybal@mpi-sb.mpg.de and
andrey.rybalchenko@epfl.ch

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the request packet and `FdoData->TopOfStack` is the pointer to another serial-based device driver). In the case where the other device driver returns a return-value that indicates success, but places `0` in `PIoStatusBlock->Information`, the serial enumeration driver will fail to increment the value pointed to by `nActual` (line 66), possibly causing the driver to infinitely execute this loop and not return to its calling context. The consequence of this error is that the computer's serial devices could become non-responsive. Worse yet, depending on what actions the other device driver takes, this loop may cause repeated acquiring and releasing of kernel resources (memory, locks, etc) at high priority and excessive physical bus activity. This extra work stresses the operating system, the other drivers, and the user applications running on the system, which may cause them to crash or become non-responsive too.

This example demonstrates how a notion of termination is central to the process of ensuring that reactive systems can always react. Until now no automatic termination tool has ever been able to provide a capacity for large program fragments (>20,000 lines) together with accurate support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. In this paper we describe such a tool, called TERMINATOR.

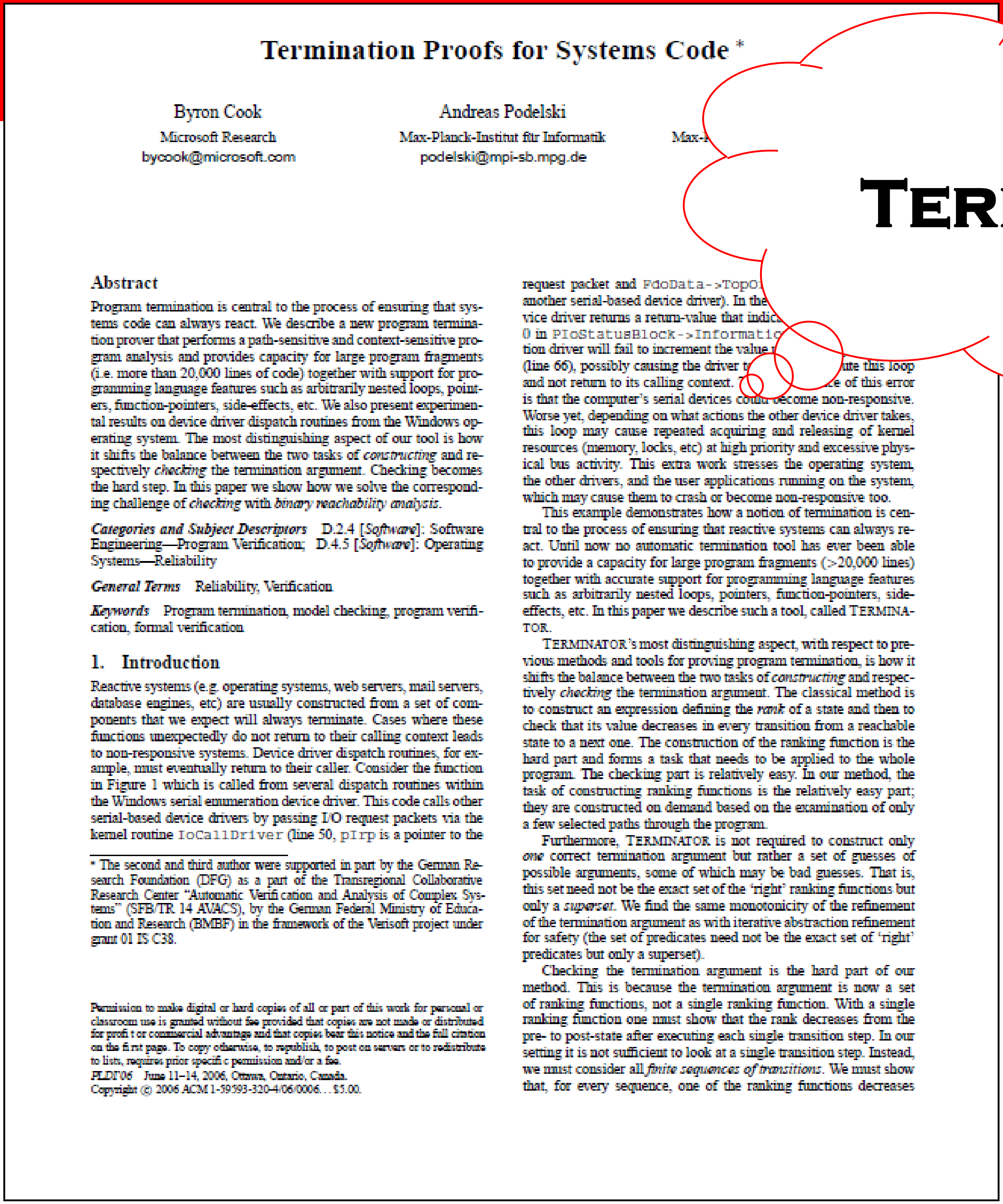
TERMINATOR's most distinguishing aspect, with respect to previous methods and tools for proving program termination, is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. The classical method is to construct an expression defining the *rank* of a state and then to check that its value decreases in every transition from a reachable state to a next one. The construction of the ranking function is the hard part and forms a task that needs to be applied to the whole program. The checking part is relatively easy. In our method, the task of constructing ranking functions is the relatively easy part; they are constructed on demand based on the examination of only a few selected paths through the program.

Furthermore, TERMINATOR is not required to construct only *one* correct termination argument but rather a set of guesses of possible arguments, some of which may be bad guesses. That is, this set need not be the exact set of the 'right' ranking functions but only a *superset*. We find the same monotonicity of the refinement of the termination argument as with iterative abstraction refinement for safety (the set of predicates need not be the exact set of 'right' predicates but only a superset).

Checking the termination argument is the hard part of our method. This is because the termination argument is now a set of ranking functions, not a single ranking function. With a single ranking function one must show that the rank decreases from the pre- to post-state after executing each single transition step. In our setting it is not sufficient to look at a single transition step. Instead, we must consider all *finite sequences of transitions*. We must show that, for every sequence, one of the ranking functions decreases

[*] The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.



Termination Proofs for Systems Code [*]

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Max-

TERMINATOR

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the request packet and `FdoData->TopO`

another serial-based device driver). In the

vice driver returns a return-value that indic

0 in `PIoStatusBlock->Informatio`

tion driver will fail to increment the value

(line 66), possibly causing the driver t ute this loop

and not return to its calling context. T ce of this error

is that the computer's serial devices could become non-responsive. Worse yet, depending on what actions the other device driver takes, this loop may cause repeated acquiring and releasing of kernel resources (memory, locks, etc) at high priority and excessive physical bus activity. This extra work stresses the operating system, the other drivers, and the user applications running on the system, which may cause them to crash or become non-responsive too.

This example demonstrates how a notion of termination is central to the process of ensuring that reactive systems can always react. Until now no automatic termination tool has ever been able to provide a capacity for large program fragments (>20,000 lines) together with accurate support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. In this paper we describe such a tool, called TERMINATOR.

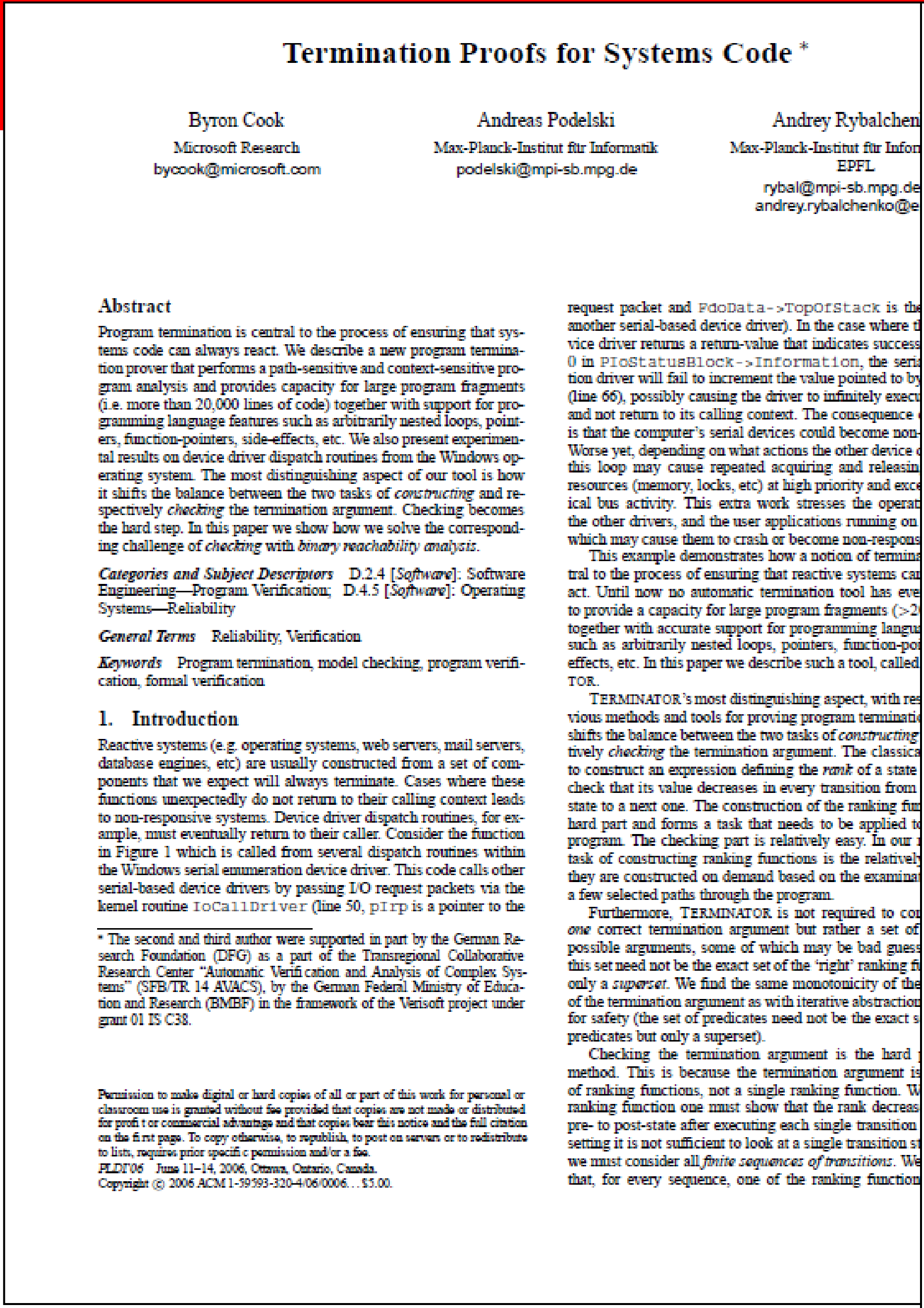
TERMINATOR's most distinguishing aspect, with respect to previous methods and tools for proving program termination, is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. The classical method is to construct an expression defining the *rank* of a state and then to check that its value decreases in every transition from a reachable state to a next one. The construction of the ranking function is the hard part and forms a task that needs to be applied to the whole program. The checking part is relatively easy. In our method, the task of constructing ranking functions is the relatively easy part; they are constructed on demand based on the examination of only a few selected paths through the program.

Furthermore, TERMINATOR is not required to construct only *one* correct termination argument but rather a set of guesses of possible arguments, some of which may be bad guesses. That is, this set need not be the exact set of the 'right' ranking functions but only a *superset*. We find the same monotonicity of the refinement of the termination argument as with iterative abstraction refinement for safety (the set of predicates need not be the exact set of 'right' predicates but only a superset).

Checking the termination argument is the hard part of our method. This is because the termination argument is now a set of ranking functions, not a single ranking function. With a single ranking function one must show that the rank decreases from the pre- to post-state after executing each single transition step. In our setting it is not sufficient to look at a single transition step. Instead, we must consider all *finite sequences of transitions*. We must show that, for every sequence, one of the ranking functions decreases

[*] The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andrey Rybalchen
Max-Planck-Institut für Infor
EPFL
rybal@mpi-sb.mpg.de
andrey.rybalchenko@e

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the
request packet and `FdoData->TopOfStack` is the
another serial-based device driver). In the case where t
vice driver returns a return-value that indicates success
0 in `PIoStatusBlock->Information`, the seria
tion driver will fail to increment the value pointed to by
(line 66), possibly causing the driver to infinitely execu
and not return to its calling context. The consequence
is that the computer's serial devices could become non-
Worse yet, depending on what actions the other device
this loop may cause repeated acquiring and releasin
resources (memory, locks, etc) at high priority and exce
ical bus activity. This extra work stresses the operat
the other drivers, and the user applications running on
which may cause them to crash or become non-respons

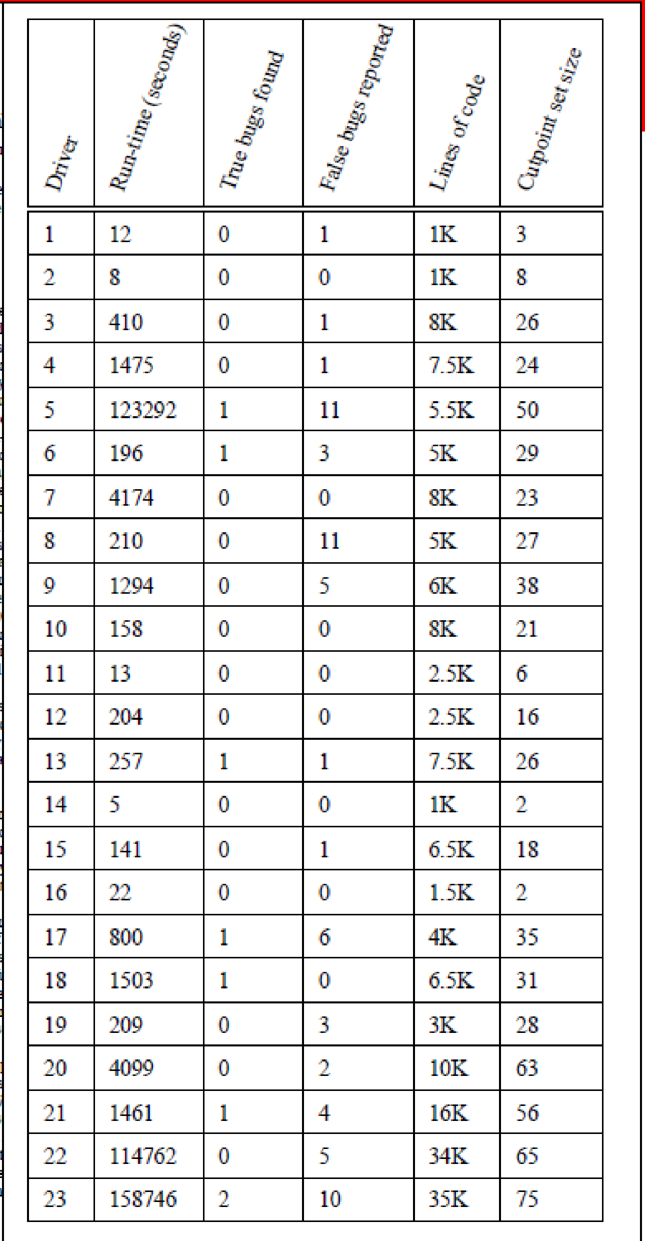
This example demonstrates how a notion of termina
tral to the process of ensuring that reactive systems ca
act. Until now no automatic termination tool has eve
to provide a capacity for large program fragments (>2
together with accurate support for programming langu
such as arbitrarily nested loops, pointers, function-poi
effects, etc. In this paper we describe such a tool, called
TOR.

TERMINATOR's most distinguishing aspect, with res
vious methods and tools for proving program terminati
shifts the balance between the two tasks of *constructing*
tively *checking* the termination argument. The classica
to construct an expression defining the *rank* of a state
check that its value decreases in every transition from
state to a next one. The construction of the ranking fu
hard part and forms a task that needs to be applied t
program. The checking part is relatively easy. In our
task of constructing ranking functions is the relatively
they are constructed on demand based on the examina
a few selected paths through the program.

Furthermore, TERMINATOR is not required to co
one correct termination argument but rather a set of
possible arguments, some of which may be bad guess
this set need not be the exact set of the 'right' ranking f
only a *superset*. We find the same monotonicity of the
of the termination argument as with iterative abstractio
for safety (the set of predicates need not be the exact s
predicates but only a superset).

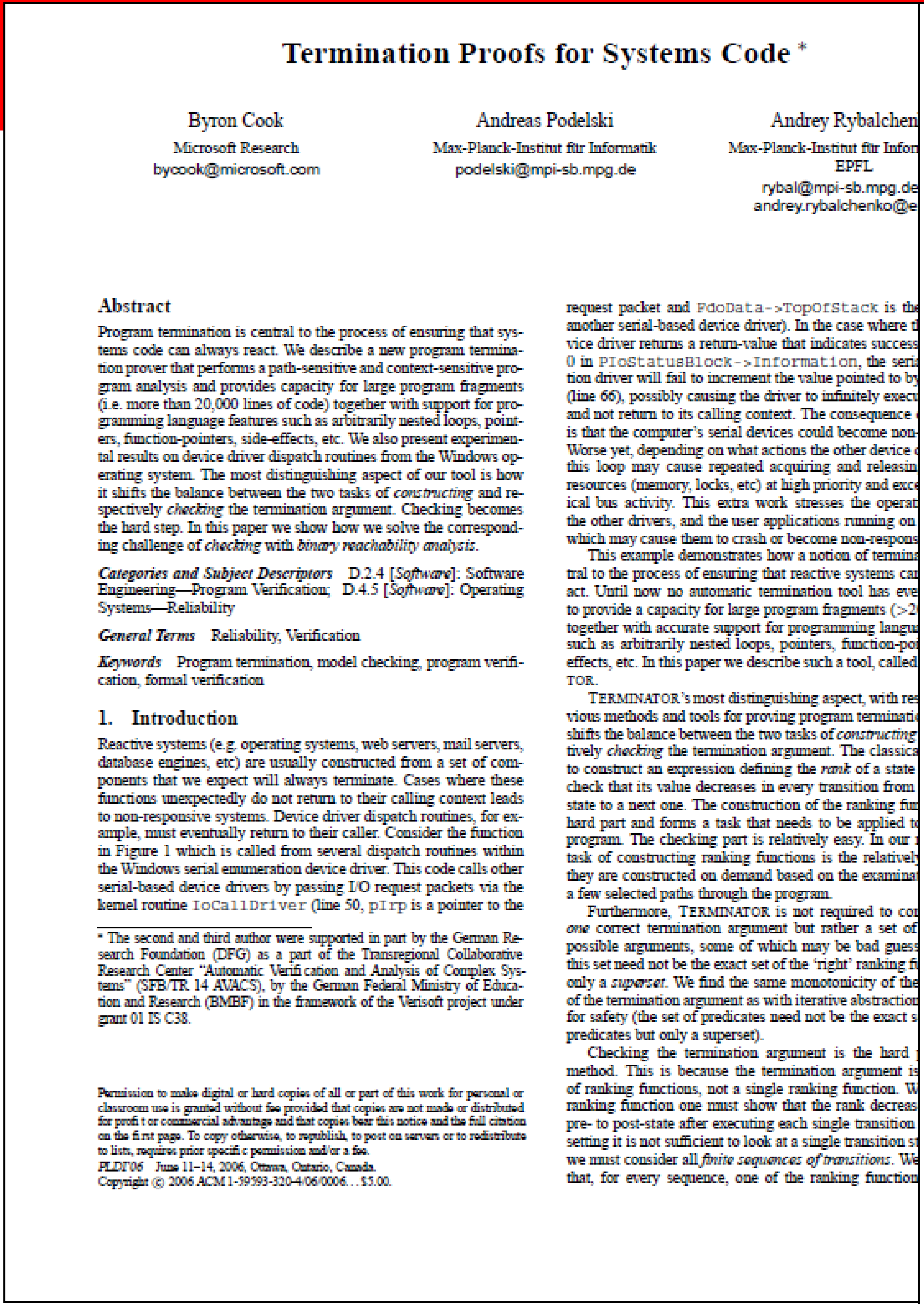
Checking the termination argument is the hard
method. This is because the termination argument i
of ranking functions, not a single ranking function. W
ranking function one must show that the rank decreas
pre- to post-state after executing each single transition
setting it is not sufficient to look at a single transition s
we must consider all *finite sequences of transitions*. We
that, for every sequence, one of the ranking function

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.



Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
1	12	0	1	1K	3
2	8	0	0	1K	8
3	410	0	1	8K	26
4	1475	0	1	7.5K	24
5	123292	1	11	5.5K	50
6	196	1	3	5K	29
7	4174	0	0	8K	23
8	210	0	11	5K	27
9	1294	0	5	6K	38
10	158	0	0	8K	21
11	13	0	0	2.5K	6
12	204	0	0	2.5K	16
13	257	1	1	7.5K	26
14	5	0	0	1K	2
15	141	0	1	6.5K	18
16	22	0	0	1.5K	2
17	800	1	6	4K	35
18	1503	1	0	6.5K	31
19	209	0	3	3K	28
20	4099	0	2	10K	63
21	1461	1	4	16K	56
22	114762	0	5	34K	65
23	158746	2	10	35K	75

Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andrey Rybalchen
Max-Planck-Institut für Infor
EPFL
rybal@mpi-sb.mpg.de
andrey.rybalchenko@e

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial emmeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the request packet and `FdoData->TopOfStack` is the
another serial-based device driver). In the case where t
vice driver returns a return-value that indicates success
0 in `PIoStatusBlock->Information`, the seria
tion driver will fail to increment the value pointed to by
(line 66), possibly causing the driver to infinitely execu
and not return to its calling context. The consequence
is that the computer's serial devices could become non-
Worse yet, depending on what actions the other device
this loop may cause repeated acquiring and releasin
resources (memory, locks, etc) at high priority and exce
ical bus activity. This extra work stresses the operat
the other drivers, and the user applications running on
which may cause them to crash or become non-respons

This example demonstrates how a notion of termina
tral to the process of ensuring that reactive systems ca
act. Until now no automatic termination tool has eve
to provide a capacity for large program fragments (>2
together with accurate support for programming langu
such as arbitrarily nested loops, pointers, function-po
effects, etc. In this paper we describe such a tool, called
TOR.

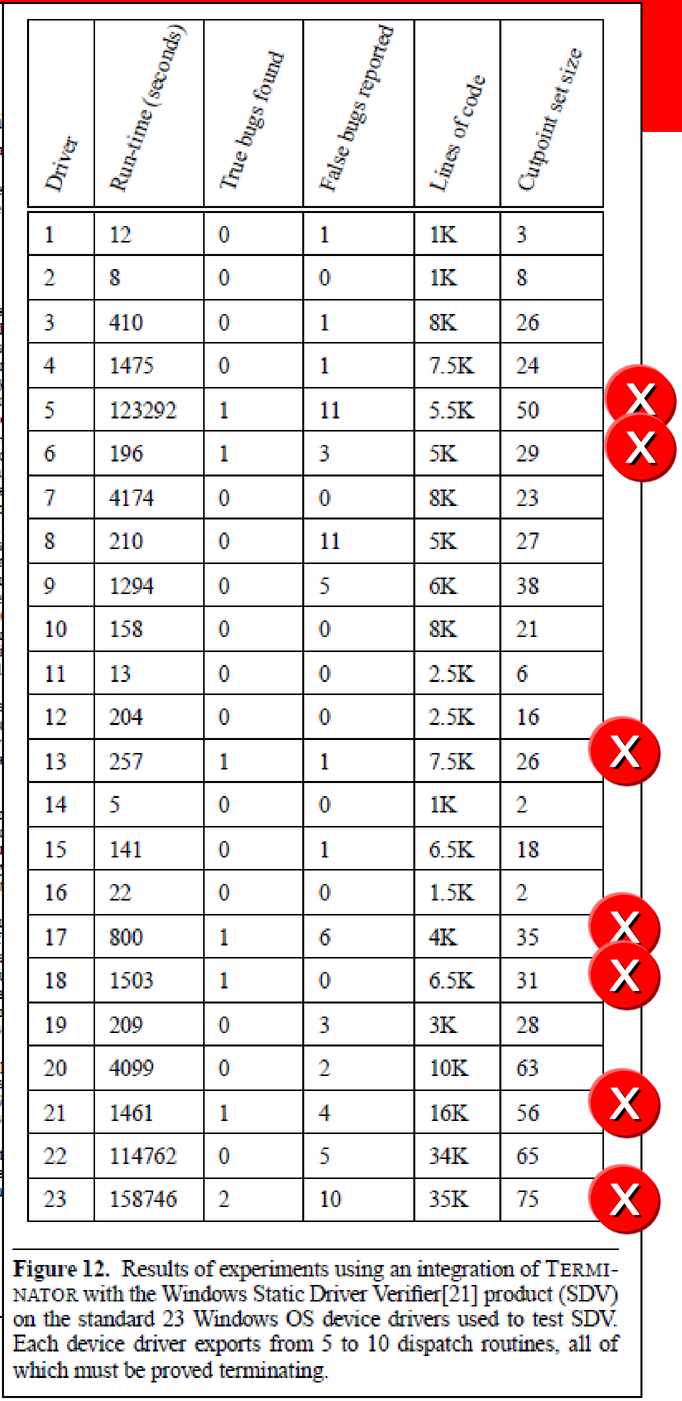
TERMINATOR's most distinguishing aspect, with res
vious methods and tools for proving program terminati
shifts the balance between the two tasks of *constructing*
tively *checking* the termination argument. The classica
to construct an expression defining the *rank* of a state
check that its value decreases in every transition from
state to a next one. The construction of the ranking fu
hard part and forms a task that needs to be applied t
program. The checking part is relatively easy. In our
task of constructing ranking functions is the relatively
they are constructed on demand based on the examina
a few selected paths through the program.

Furthermore, TERMINATOR is not required to co
one correct termination argument but rather a set of
possible arguments, some of which may be bad guess
this set need not be the exact set of the 'right' ranking f
only a *superset*. We find the same monotonicity of the
of the termination argument as with iterative abstractio
for safety (the set of predicates need not be the exact s
predicates but only a superset).

Checking the termination argument is the hard
method. This is because the termination argument is
of ranking functions, not a single ranking function. W
ranking function one must show that the rank decreas
pre- to post-state after executing each single transition
setting it is not sufficient to look at a single transition s
we must consider all *finite sequences of transitions*. We
that, for every sequence, one of the ranking function

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verifi cation and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

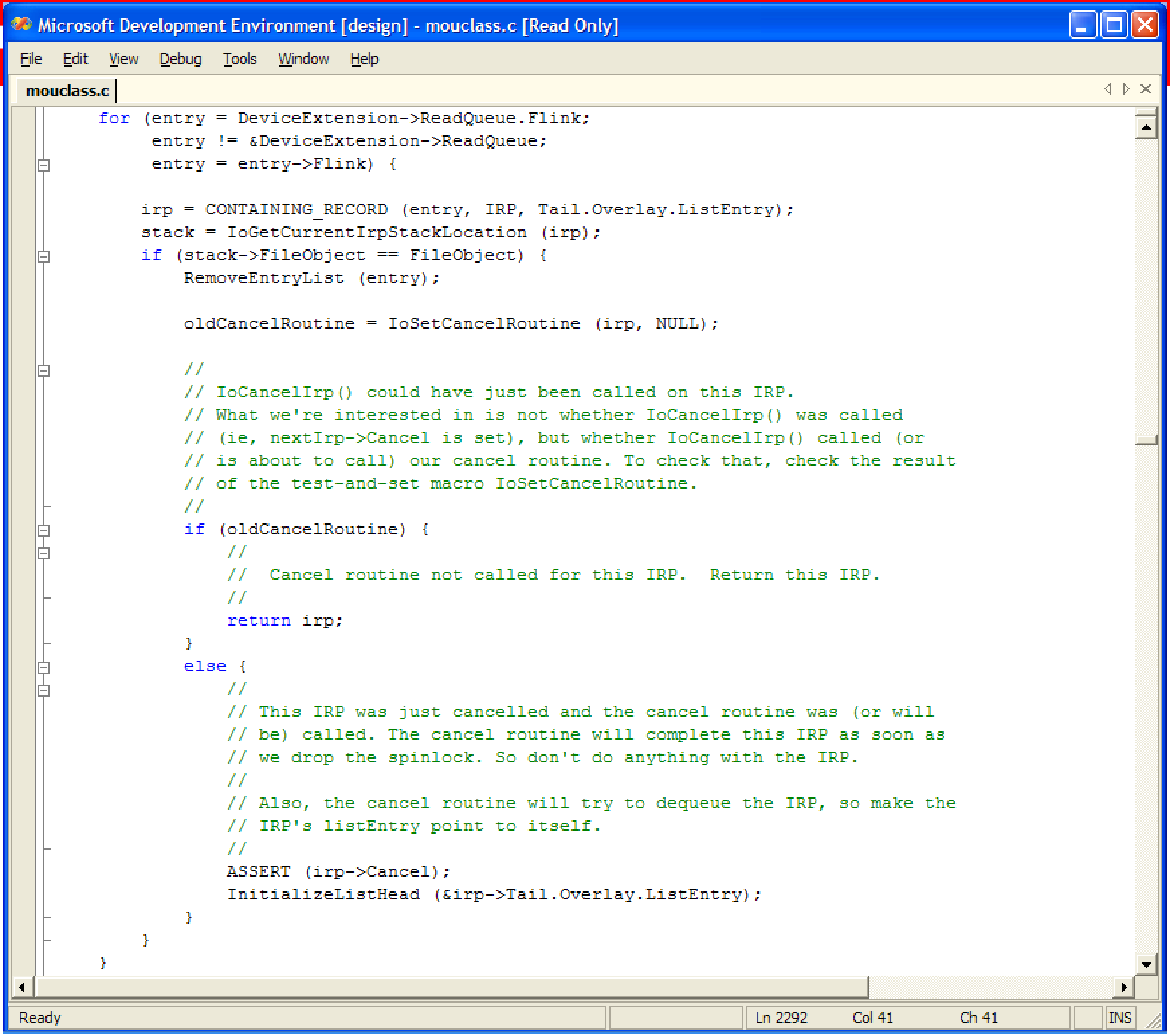
PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.



Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size	
1	12	0	1	1K	3	
2	8	0	0	1K	8	
3	410	0	1	8K	26	
4	1475	0	1	7.5K	24	
5	123292	1	11	5.5K	50	X
6	196	1	3	5K	29	X
7	4174	0	0	8K	23	
8	210	0	11	5K	27	
9	1294	0	5	6K	38	
10	158	0	0	8K	21	
11	13	0	0	2.5K	6	
12	204	0	0	2.5K	16	
13	257	1	1	7.5K	26	X
14	5	0	0	1K	2	
15	141	0	1	6.5K	18	
16	22	0	0	1.5K	2	
17	800	1	6	4K	35	X
18	1503	1	0	6.5K	31	X
19	209	0	3	3K	28	
20	4099	0	2	10K	63	
21	1461	1	4	16K	56	X
22	114762	0	5	34K	65	
23	158746	2	10	35K	75	X

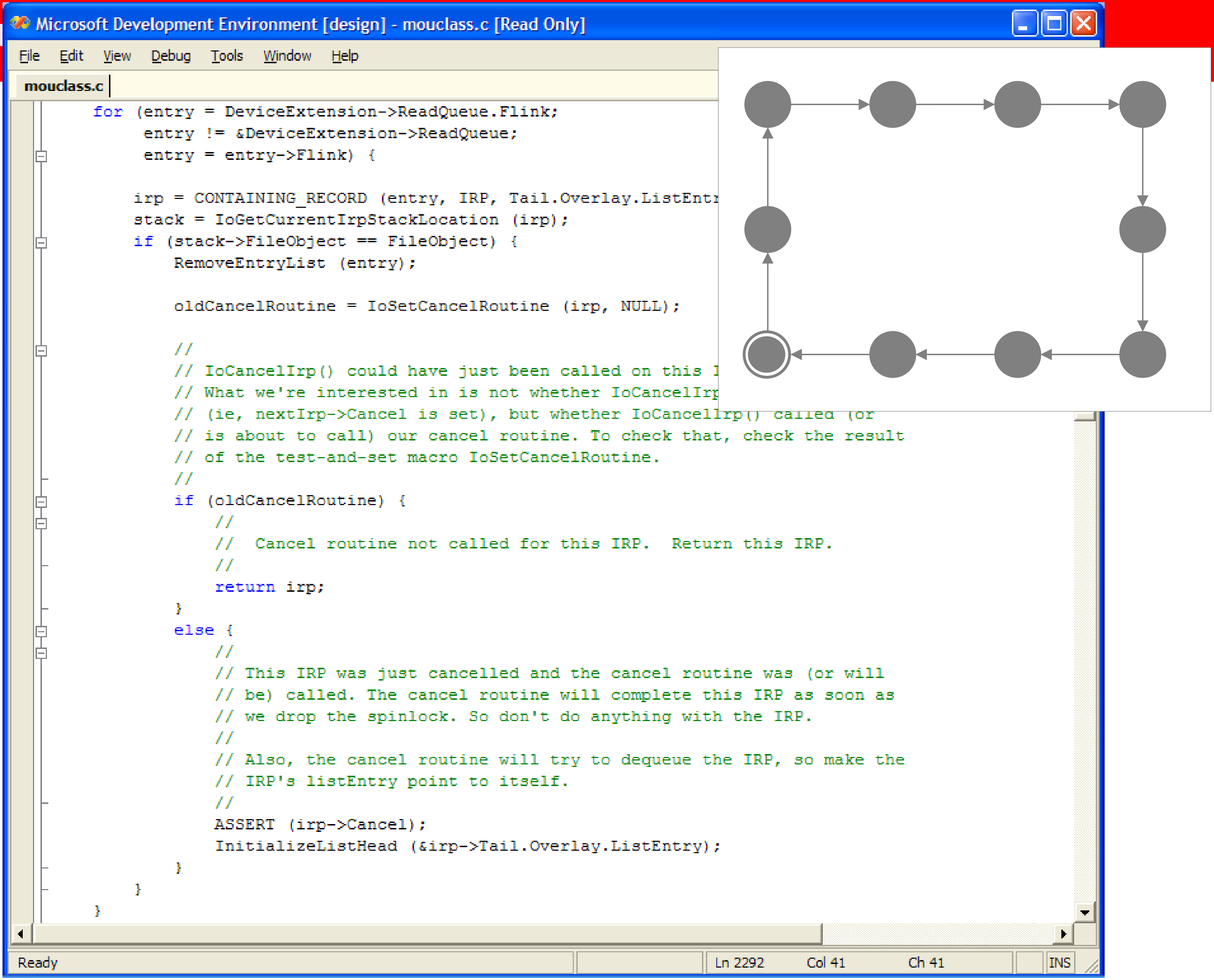
Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.

Misu


Microsoft Development Environment [design] - mouclass.c [Read Only]
File
Edit
View
Debug
Tools
Window
Help
mouclass.c

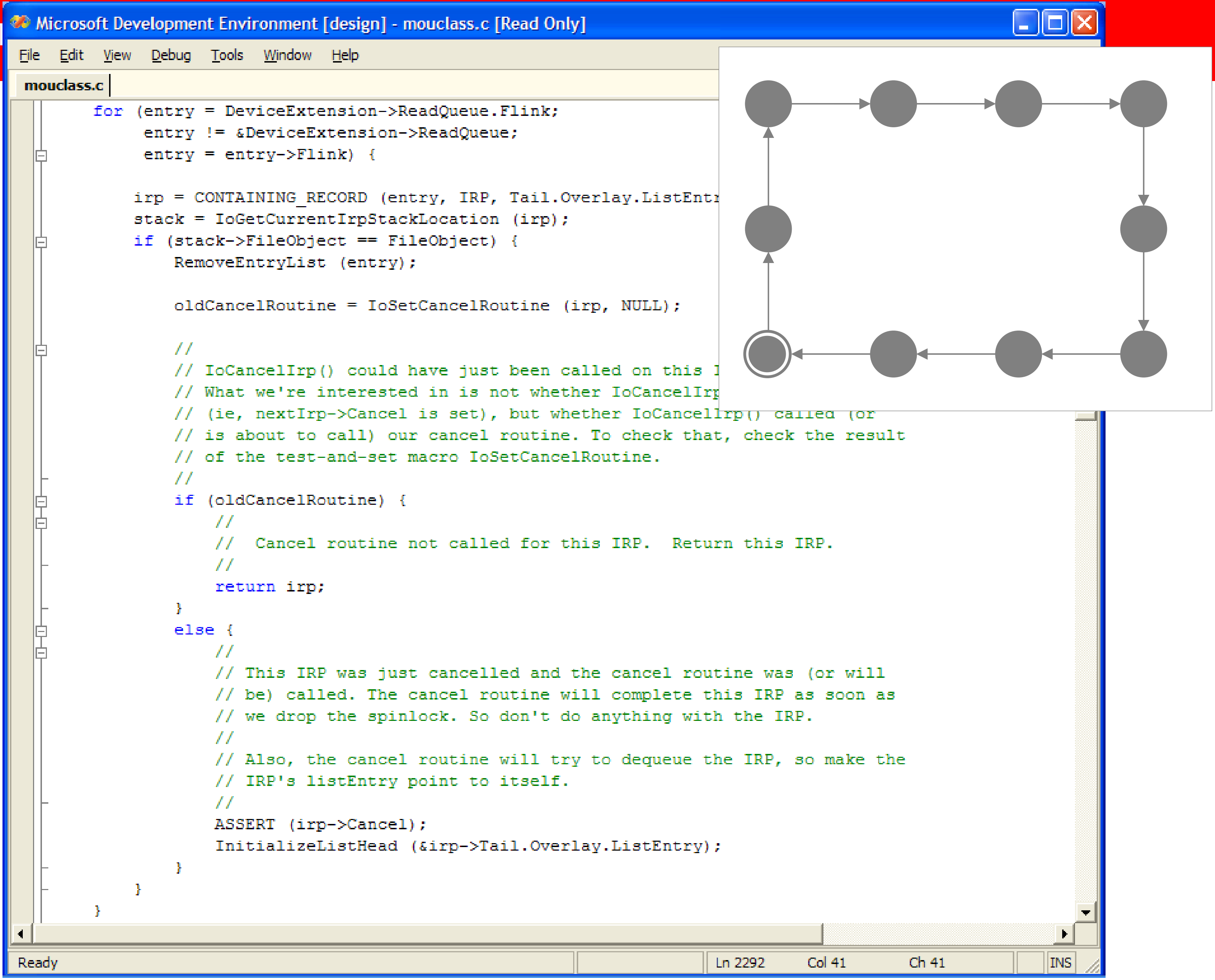
```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this IRP.
        // What we're interested in is not whether IoCancelIrp() was called
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Misu



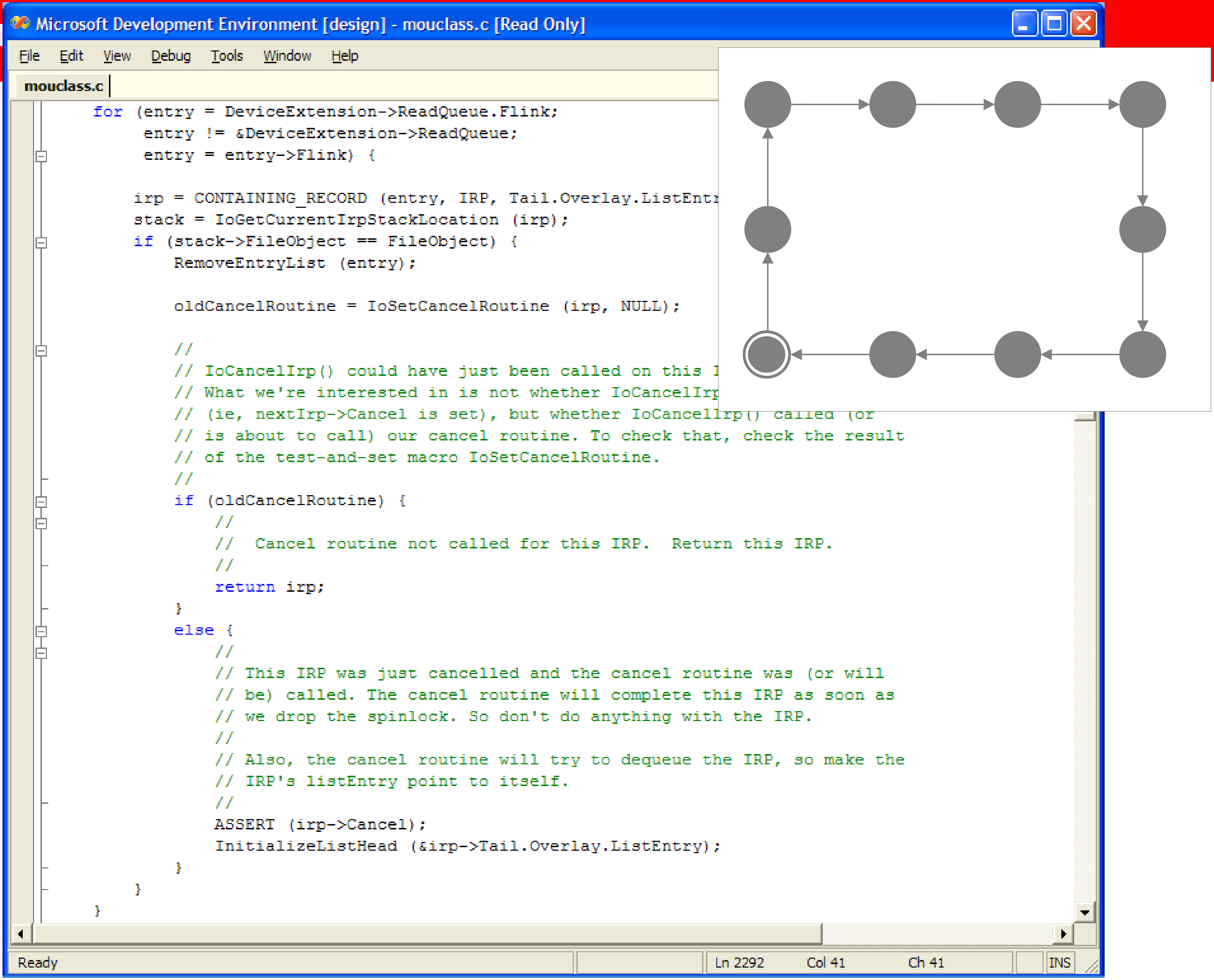
```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Misu



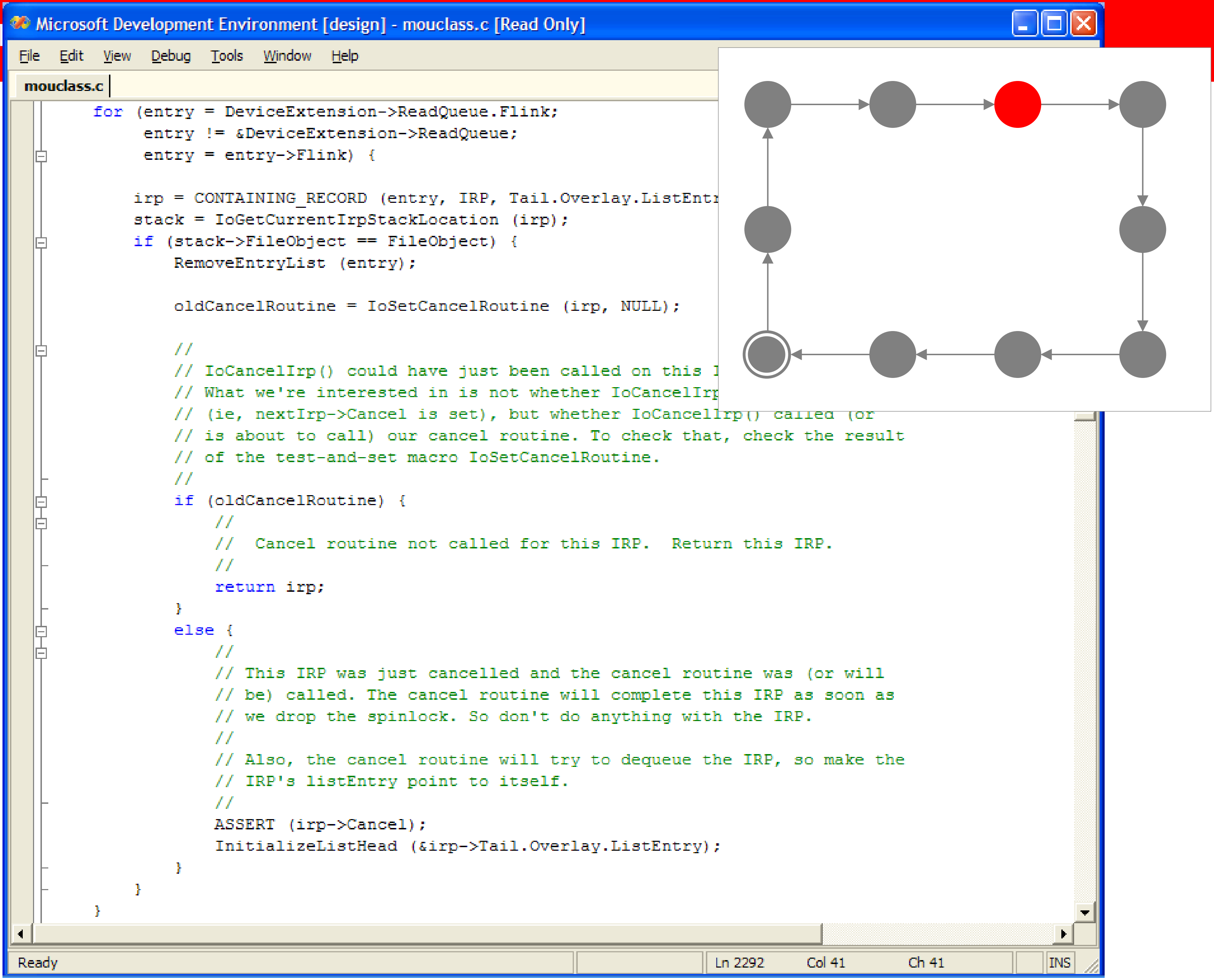
```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Misu



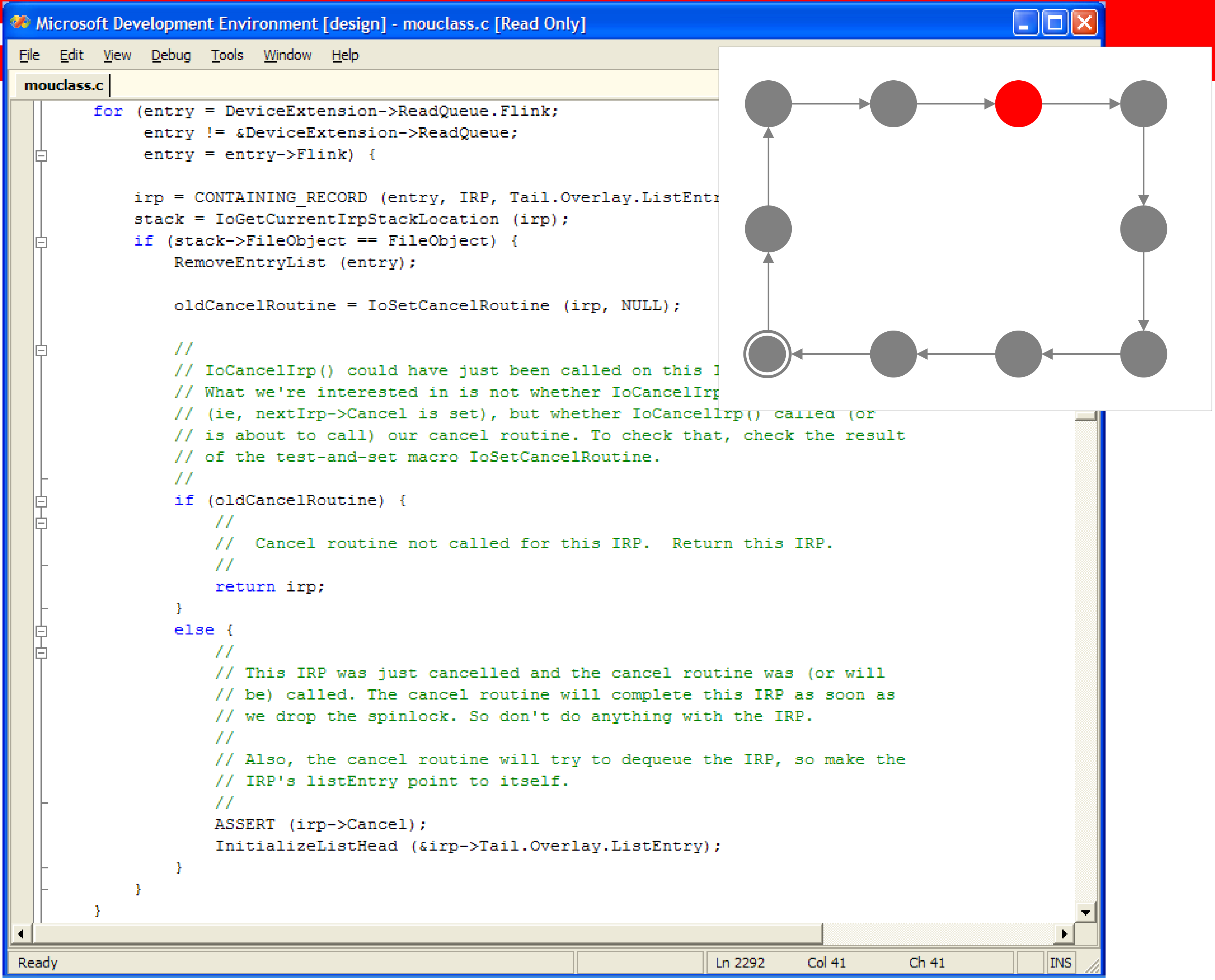
```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Misu



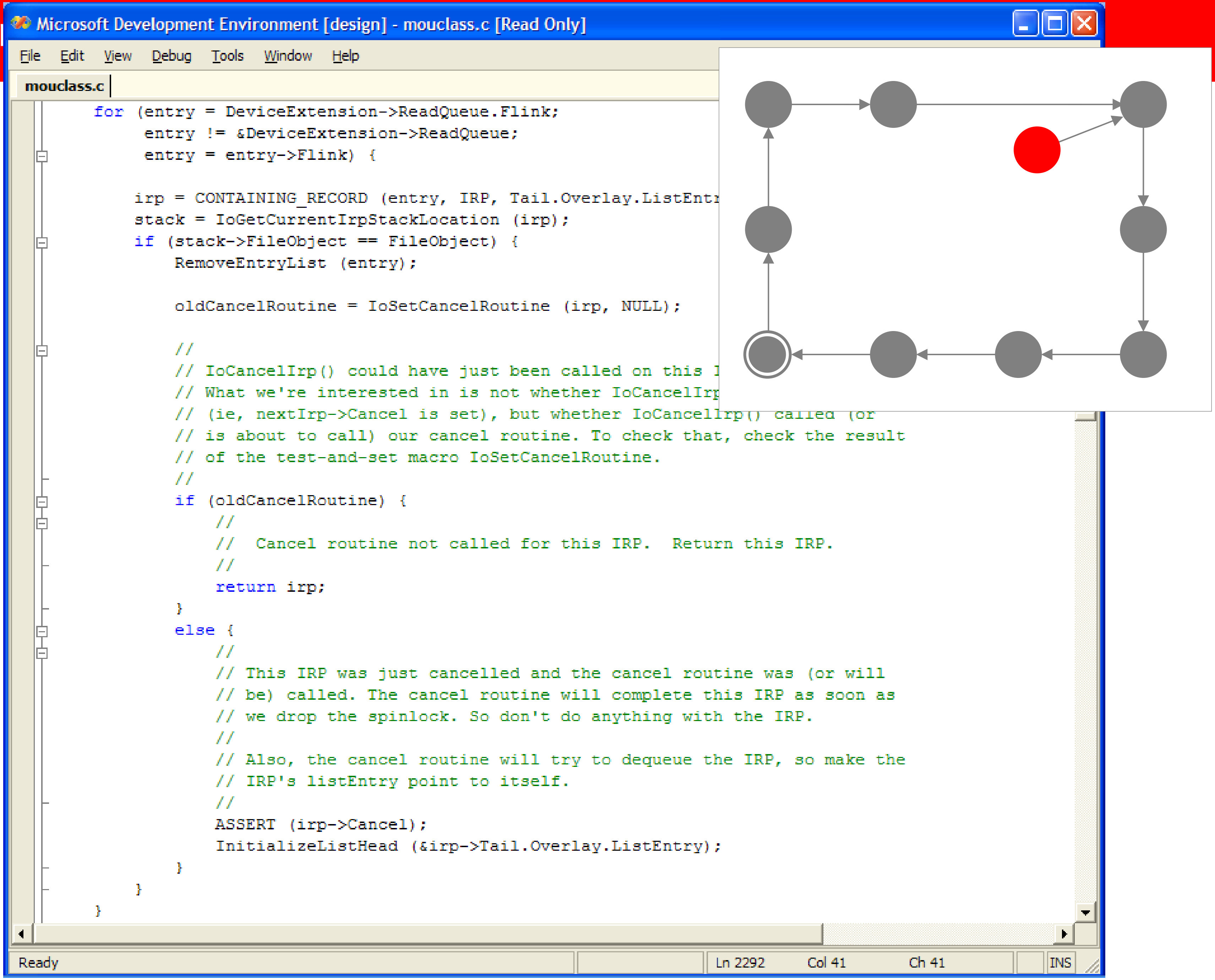
```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Misu



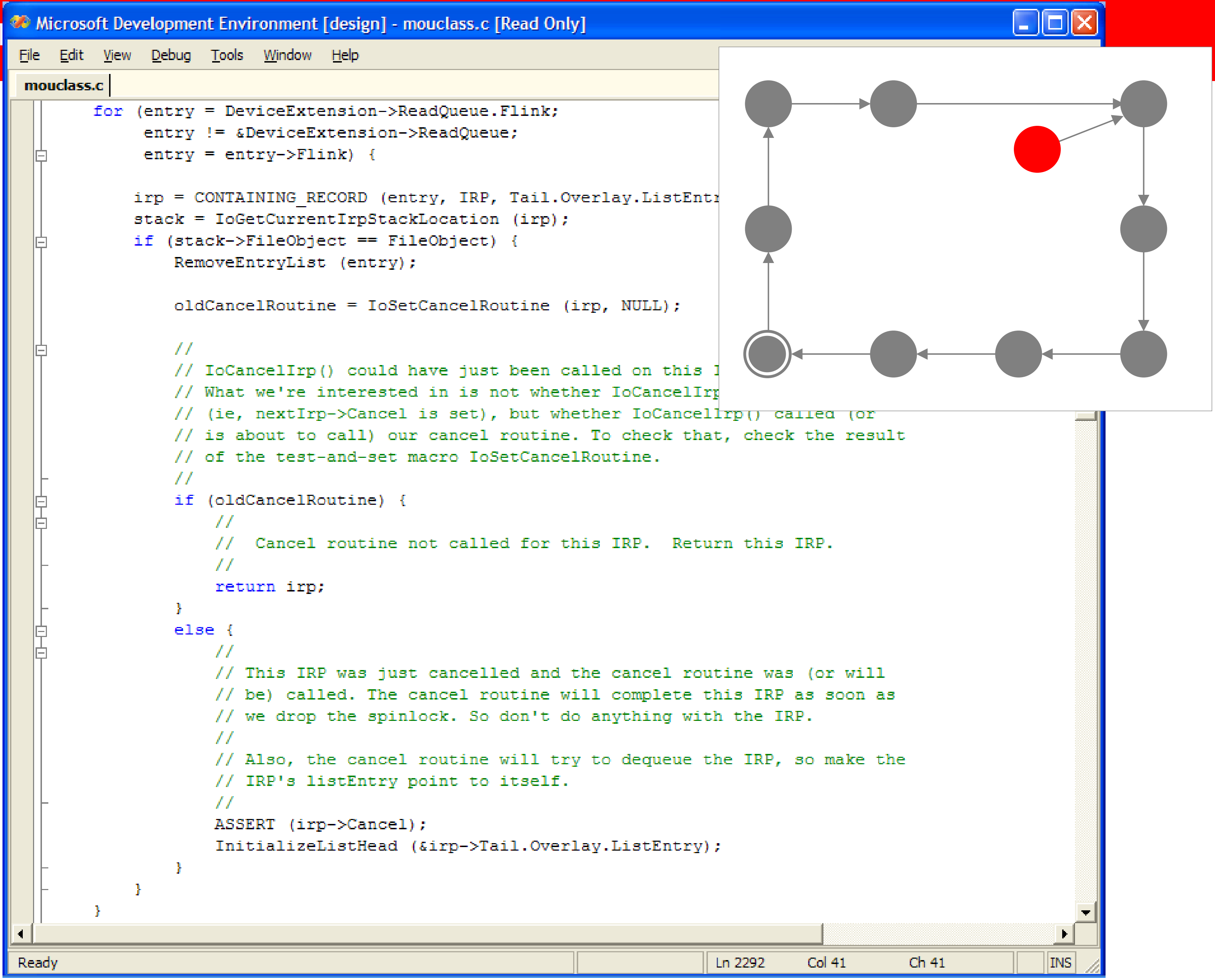
```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this I
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Misu



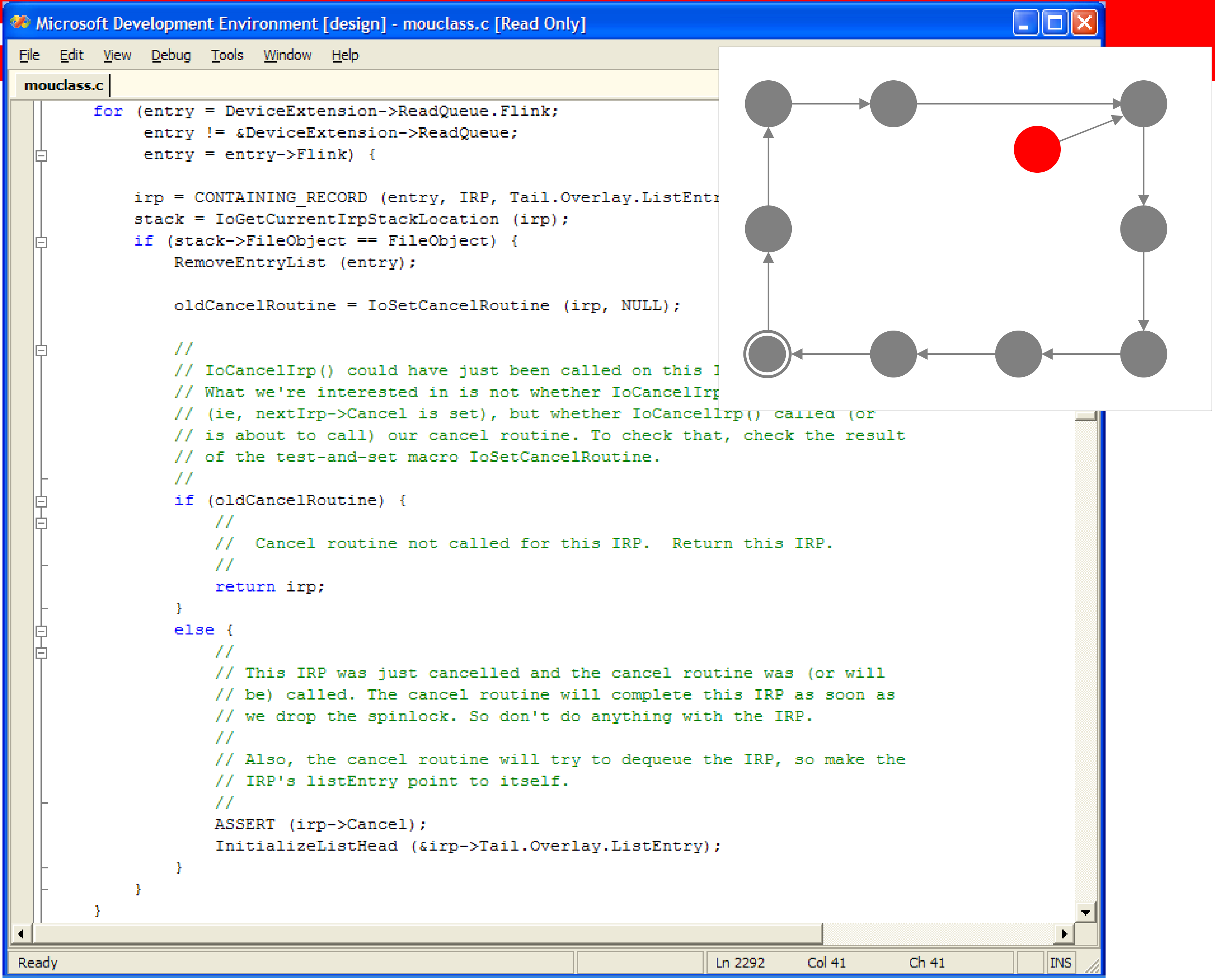
```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this I
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Misu



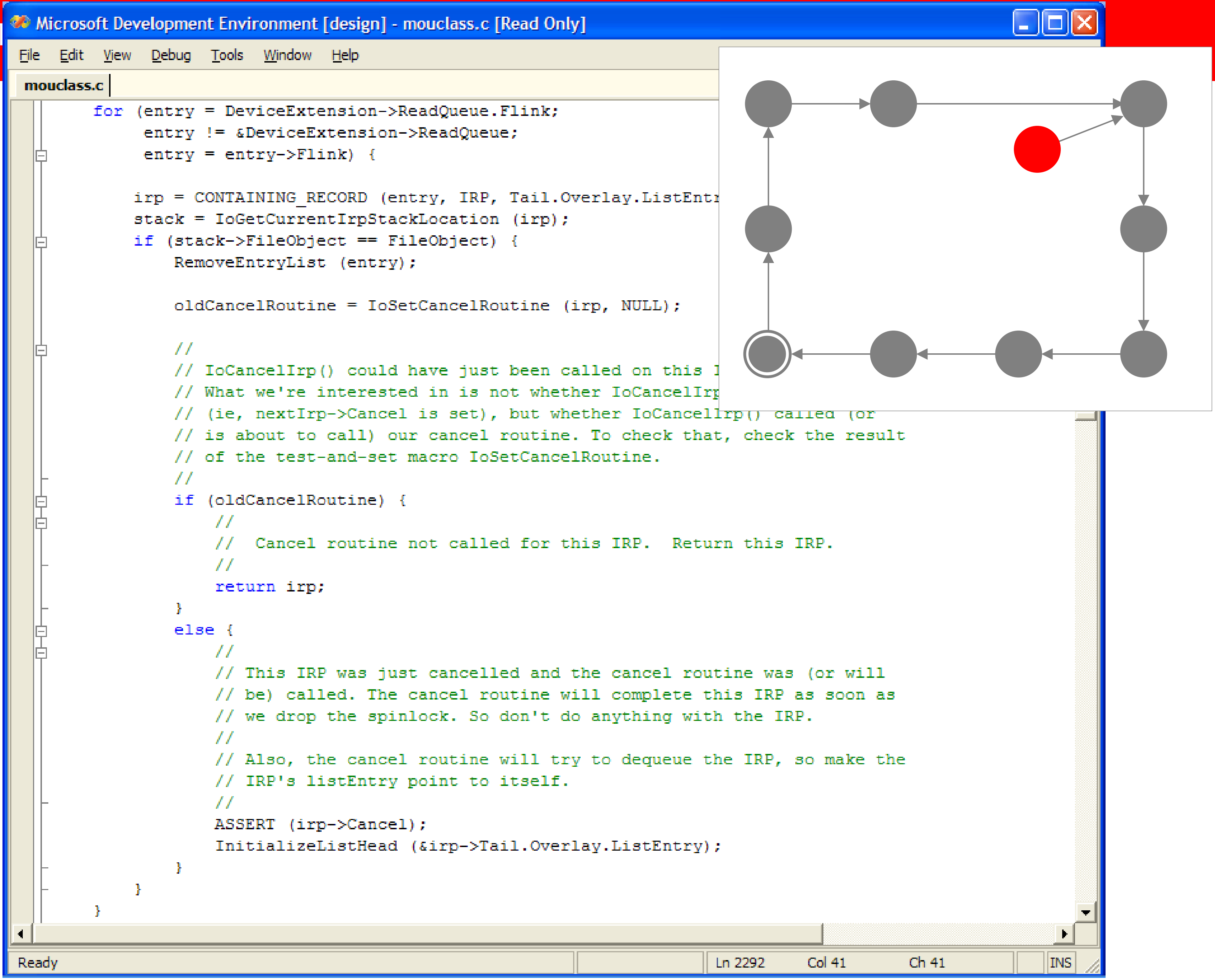
```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this I
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Misu



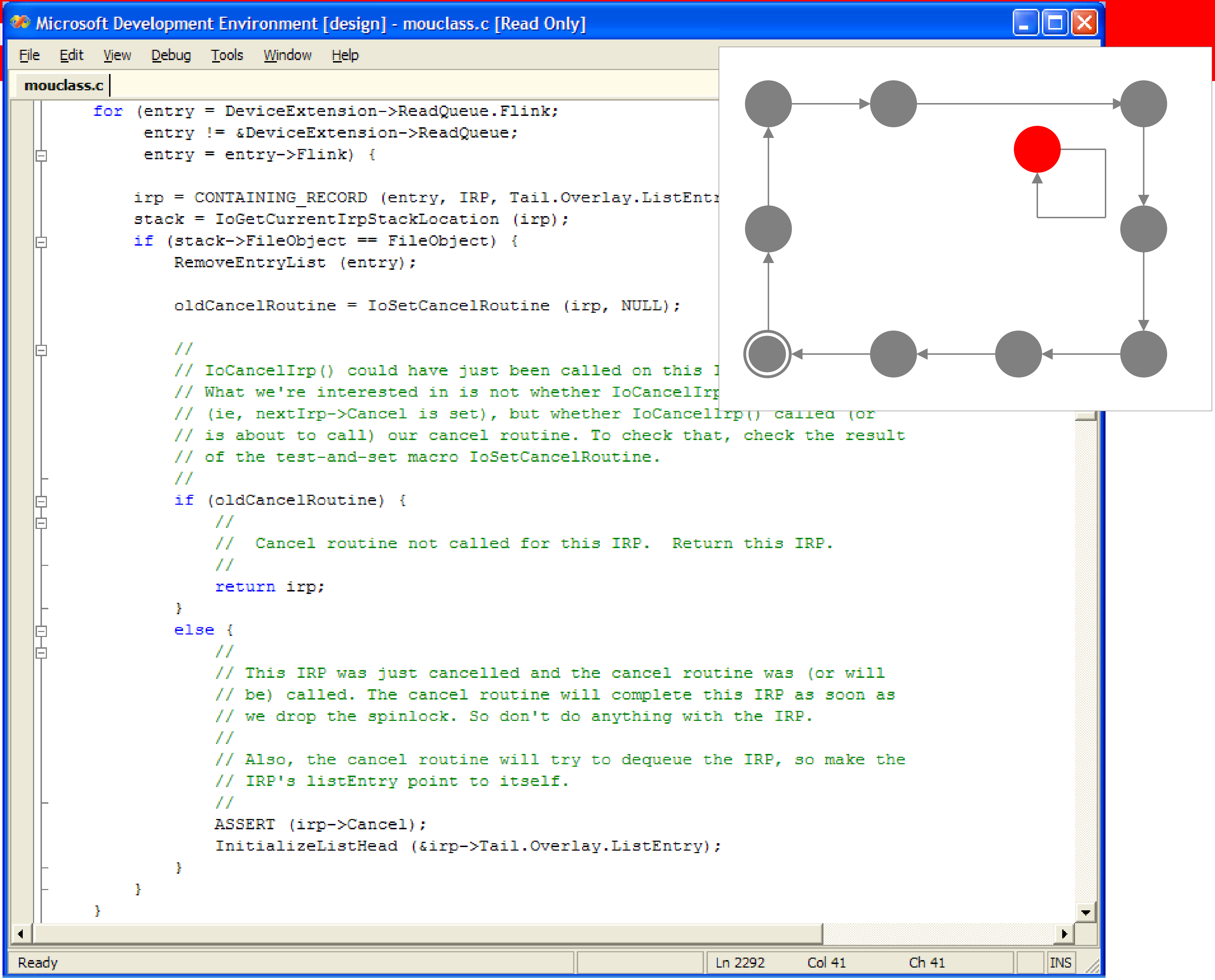
```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this I
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Misu



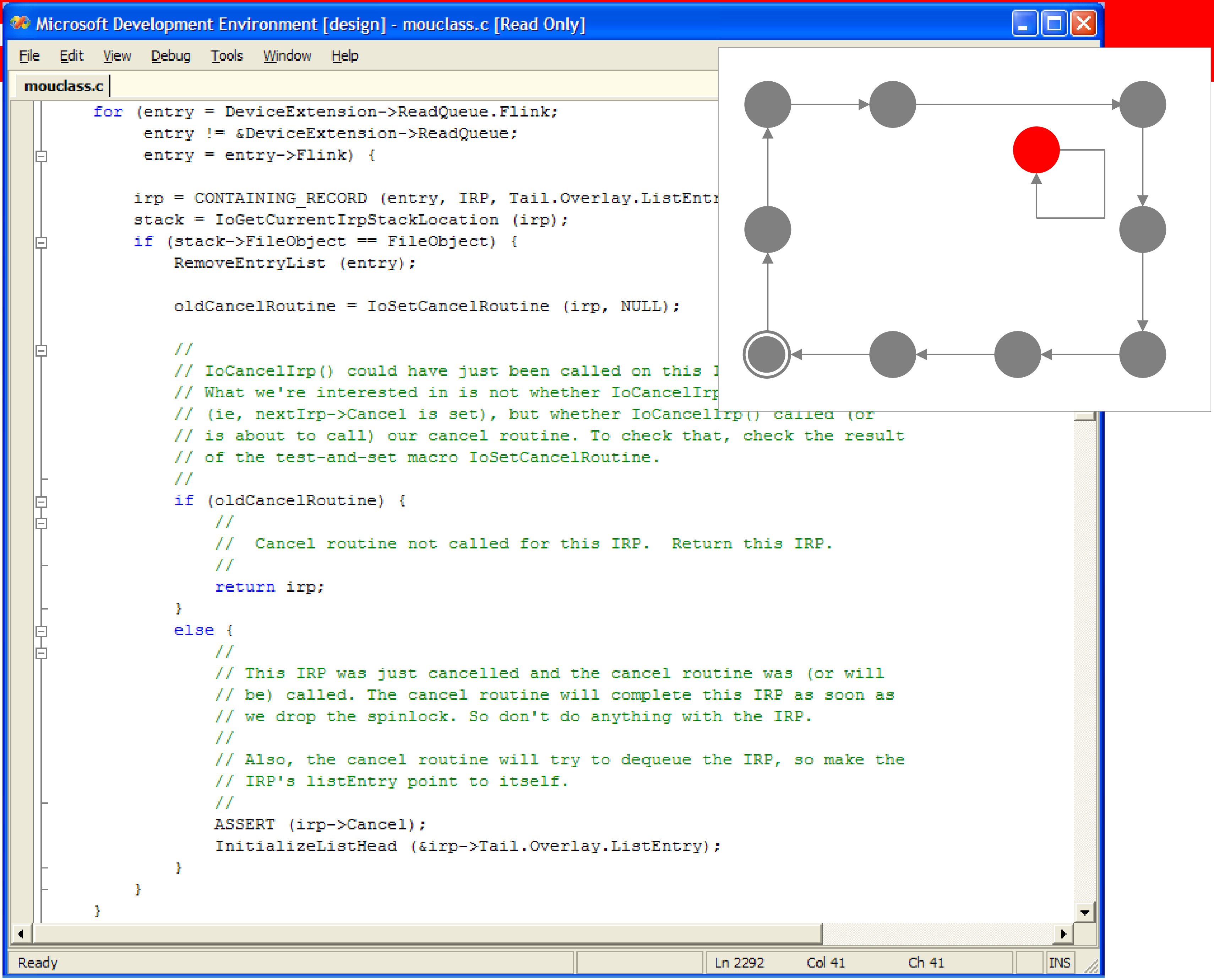
```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this I
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Misu



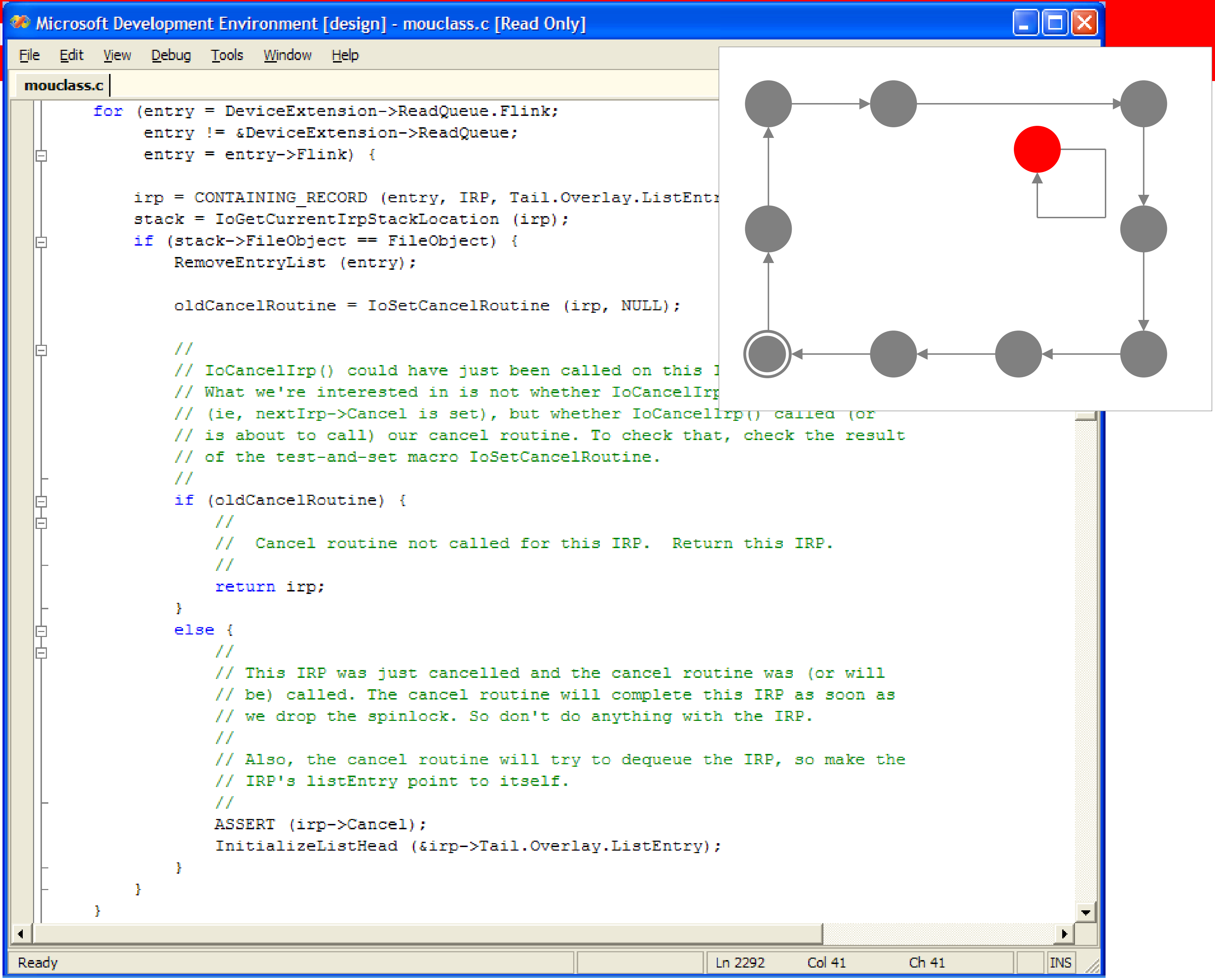
```
for (entry = DeviceExtension->ReadQueue.Flink;
     entry != &DeviceExtension->ReadQueue;
     entry = entry->Flink) {

    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
    stack = IoGetCurrentIrpStackLocation (irp);
    if (stack->FileObject == FileObject) {
        RemoveEntryList (entry);

        oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

        //
        // IoCancelIrp() could have just been called on this I
        // What we're interested in is not whether IoCancelIrp
        // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
        // is about to call) our cancel routine. To check that, check the result
        // of the test-and-set macro IoSetCancelRoutine.
        //
        if (oldCancelRoutine) {
            //
            //  Cancel routine not called for this IRP.  Return this IRP.
            //
            return irp;
        }
        else {
            //
            // This IRP was just cancelled and the cancel routine was (or will
            // be) called. The cancel routine will complete this IRP as soon as
            // we drop the spinlock. So don't do anything with the IRP.
            //
            // Also, the cancel routine will try to dequeue the IRP, so make the
            // IRP's listEntry point to itself.
            //
            ASSERT (irp->Cancel);
            InitializeListHead (&irp->Tail.Overlay.ListEntry);
        }
    }
}
```

Misu


Microsoft Development Environment [design] - mouclass.c [Read Only]
File
Edit
View
Debug
Tools
Window
Help
mouclass.c

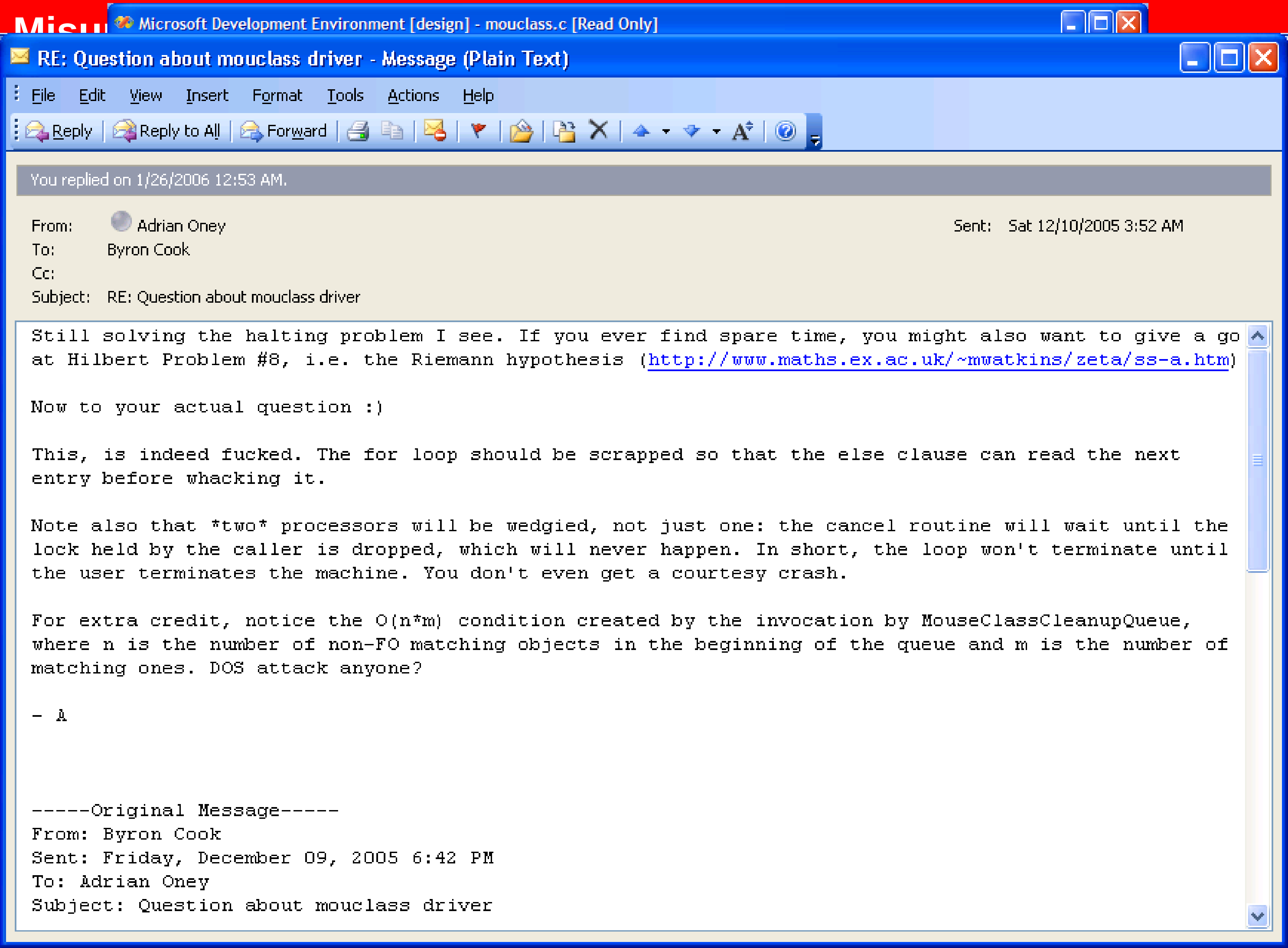
```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Misu



```
    for (entry = DeviceExtension->ReadQueue.Flink;
         entry != &DeviceExtension->ReadQueue;
         entry = entry->Flink) {

        irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntr
        stack = IoGetCurrentIrpStackLocation (irp);
        if (stack->FileObject == FileObject) {
            RemoveEntryList (entry);

            oldCancelRoutine = IoSetCancelRoutine (irp, NULL);

            //
            // IoCancelIrp() could have just been called on this I
            // What we're interested in is not whether IoCancelIrp
            // (ie, nextIrp->Cancel is set), but whether IoCancelIrp() called (or
            // is about to call) our cancel routine. To check that, check the result
            // of the test-and-set macro IoSetCancelRoutine.
            //
            if (oldCancelRoutine) {
                //
                //  Cancel routine not called for this IRP.  Return this IRP.
                //
                return irp;
            }
            else {
                //
                // This IRP was just cancelled and the cancel routine was (or will
                // be) called. The cancel routine will complete this IRP as soon as
                // we drop the spinlock. So don't do anything with the IRP.
                //
                // Also, the cancel routine will try to dequeue the IRP, so make the
                // IRP's listEntry point to itself.
                //
                ASSERT (irp->Cancel);
                InitializeListHead (&irp->Tail.Overlay.ListEntry);
            }
        }
    }
```

Misu

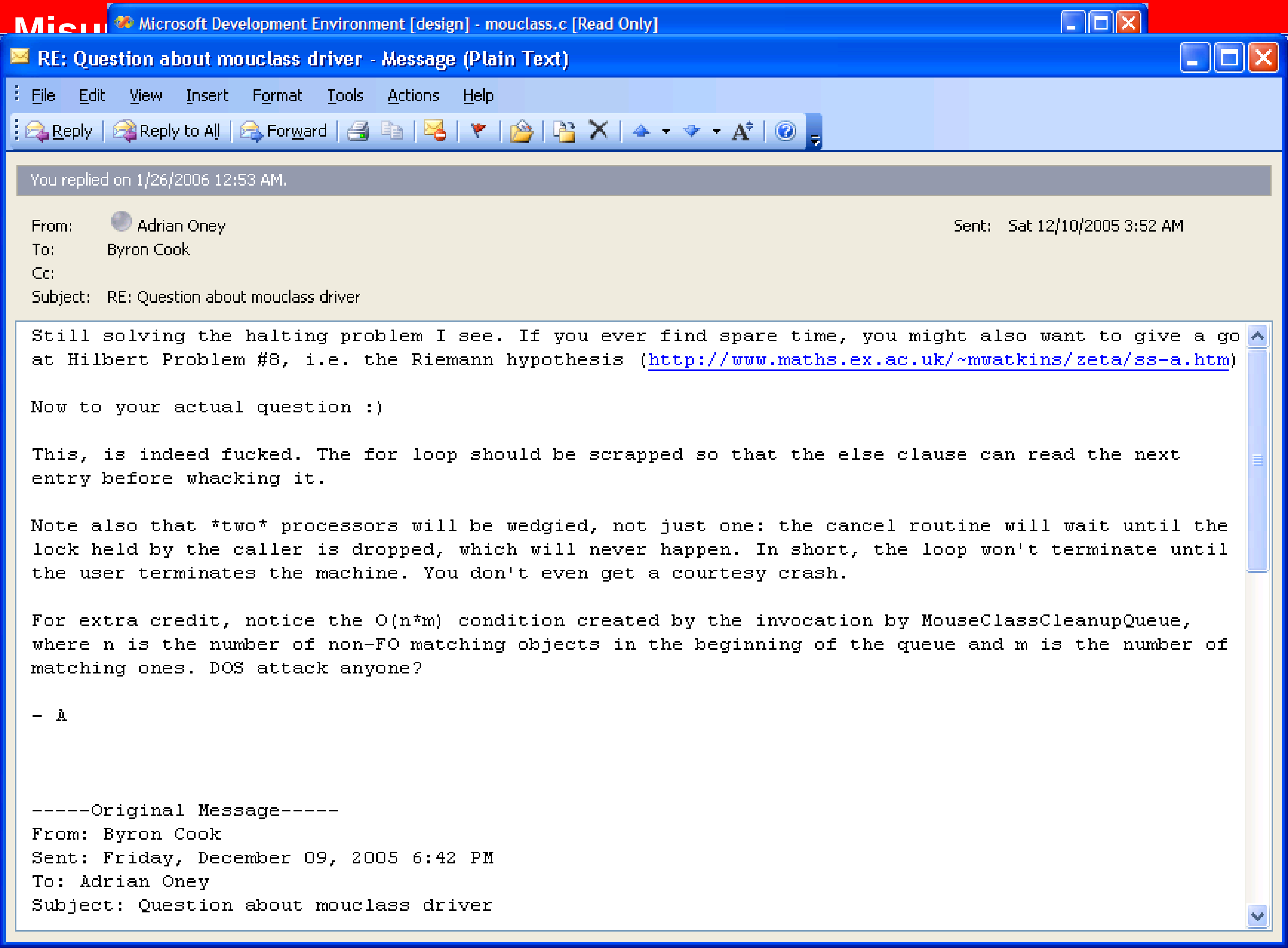

Microsoft Development Environment [design] - mouclass.c [Read Only]
RE: Question about mouclass driver - Message (Plain Text)
File Edit View Insert Format Tools Actions Help
Reply
Reply to All
Forward

You replied on 1/26/2006 12:53 AM.

From: Adrian Oney
Sent: Sat 12/10/2005 3:52 AM
To: Byron Cook
Cc:
Subject: RE: Question about mouclass driver

```
Still solving the halting problem I see. If you ever find spare time, you might also want to give a go
at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

This, is indeed fucked. The for loop should be scrapped so that the else clause can read the next
entry before whacking it.

Note also that *two* processors will be wedgied, not just one: the cancel routine will wait until the
lock held by the caller is dropped, which will never happen. In short, the loop won't terminate until
the user terminates the machine. You don't even get a courtesy crash.

For extra credit, notice the O(n*m) condition created by the invocation by MouseClassCleanupQueue,
where n is the number of non-FO matching objects in the beginning of the queue and m is the number of
matching ones. DOS attack anyone?

- A



-----Original Message-----
From: Byron Cook
Sent: Friday, December 09, 2005 6:42 PM
To: Adrian Oney
Subject: Question about mouclass driver
```

Misu


Microsoft Development Environment [design] - mouclass.c [Read Only]
RE: Question about mouclass driver - Message (Plain Text)
File
Edit
View
Insert
Format
Tools
Actions
Help
Reply
Reply to All
Forward

You replied on 1/26/2006 12:53 AM.

From: Adrian Oney
Sent: Sat 12/10/2005 3:52 AM
To: Byron Cook
Cc:
Subject: RE: Question about mouclass driver

Still solving the halting problem I see. If you ever find spare time, you might also want to give a go at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

This, is indeed fucked. The for loop should be scrapped so that the else clause can read the next entry before whacking it.

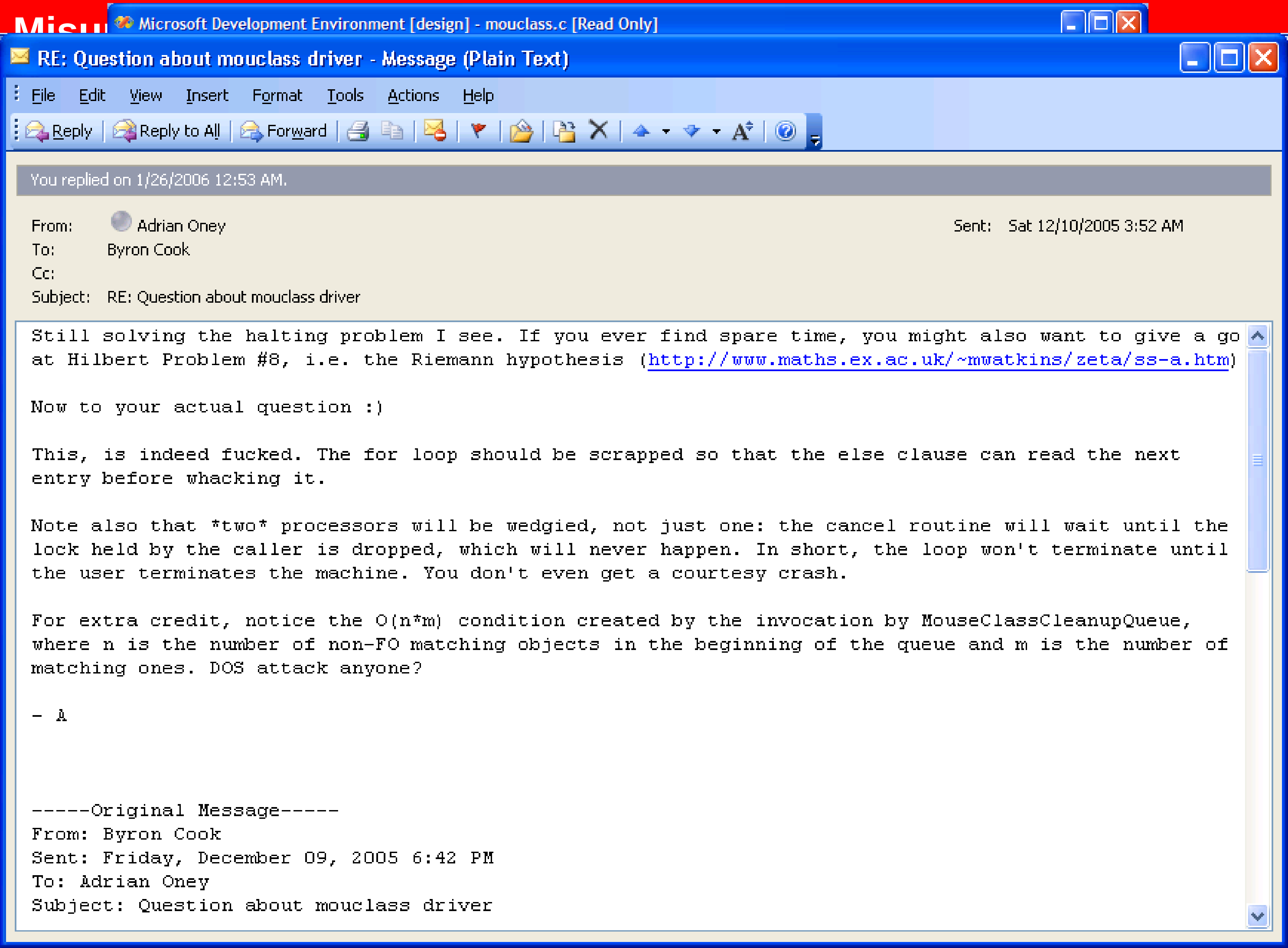
Note also that *two* processors will be wedgied, not just one: the cancel routine will wait until the lock held by the caller is dropped, which will never happen. In short, the loop won't terminate until the user terminates the machine. You don't even get a courtesy crash.

For extra credit, notice the O(n*m) condition created by the invocation by MouseClassCleanupQueue, where n is the number of non-FO matching objects in the beginning of the queue and m is the number of matching ones. DOS attack anyone?

- A

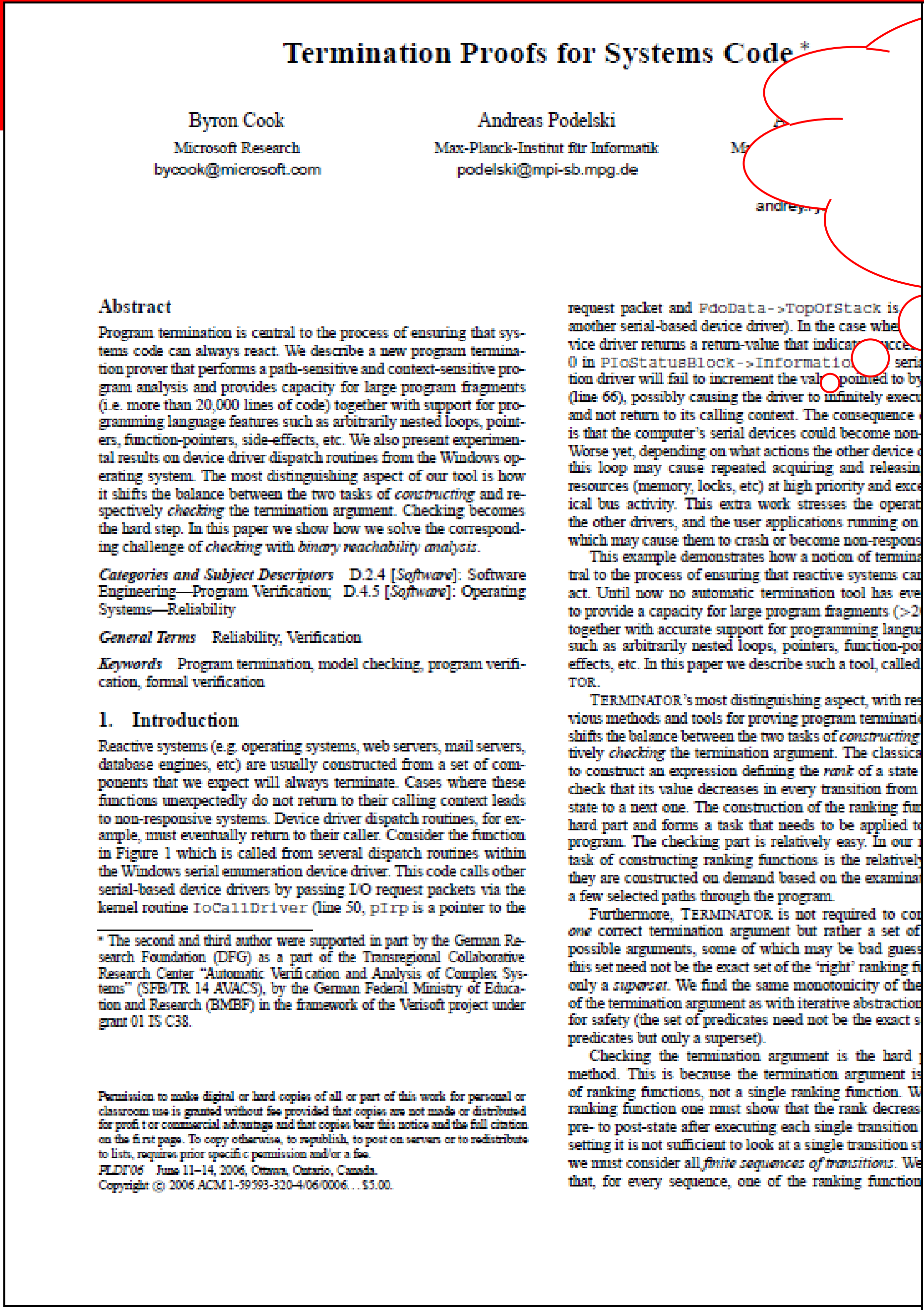
-----Original Message-----
From: Byron Cook
Sent: Friday, December 09, 2005 6:42 PM
To: Adrian Oney
Subject: Question about mouclass driver

Misu


Microsoft Development Environment [design] - mouclass.c [Read Only]
RE: Question about mouclass driver - Message (Plain Text)
File Edit View Insert Format Tools Actions Help
Reply
Reply to All
Forward
You replied on 1/26/2006 12:53 AM.

From: Adrian Oney
To: Byron Cook
Cc:
Subject: RE: Question about mouclass driver
Sent: Sat 12/10/2005 3:52 AM

```
Still solving the halting problem I see. If you ever find spare time, you might also want to give a go
at Hilbert Problem #8, i.e. the Riemann hypothesis (http://www.maths.ex.ac.uk/~mwatkins/zeta/ss-a.htm)

Now to your actual question :)

This, is indeed fucked. The for loop should be scrapped so that the else clause can read the next
entry before whacking it.

Note also that *two* processors will be wedgied, not just one: the cancel routine will wait until the
lock held by the caller is dropped, which will never happen. In short, the loop won't terminate until
the user terminates the machine. You don't even get a courtesy crash.

For extra credit, notice the O(n*m) condition created by the invocation by MouseClassCleanupQueue,
where n is the number of non-FO matching objects in the beginning of the queue and m is the number of
matching ones. DOS attack anyone?

- A



-----Original Message-----
From: Byron Cook
Sent: Friday, December 09, 2005 6:42 PM
To: Adrian Oney
Subject: Question about mouclass driver
```



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

A
M
andrey.

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the request packet and `FdoData->TopOfStack` is another serial-based device driver). In the case whe vice driver returns a return-value that indicat succe 0 in `PIoStatusBlock->Informatio` seria tion driver will fail to increment the val pointed to by (line 66), possibly causing the driver to infinitely execu and not return to its calling context. The consequence is that the computer's serial devices could become non- Worse yet, depending on what actions the other device this loop may cause repeated acquiring and releasin resources (memory, locks, etc) at high priority and exce ical bus activity. This extra work stresses the operat the other drivers, and the user applications running on which may cause them to crash or become non-respons

This example demonstrates how a notion of termina tral to the process of ensuring that reactive systems ca act. Until now no automatic termination tool has eve to provide a capacity for large program fragments (>2 together with accurate support for programming langu such as arbitrarily nested loops, pointers, function-poi effects, etc. In this paper we describe such a tool, called TOR.

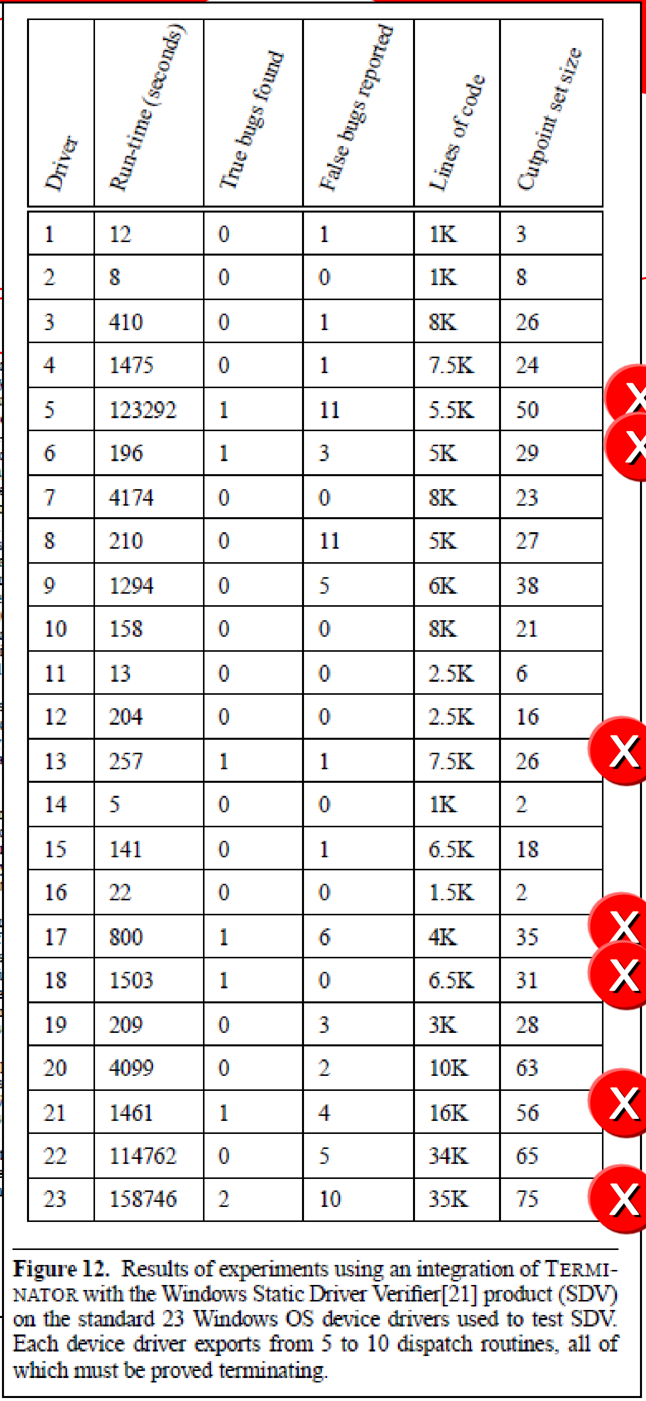
TERMINATOR's most distinguishing aspect, with res vious methods and tools for proving program terminati shifts the balance between the two tasks of *constructing* tively *checking* the termination argument. The classica to construct an expression defining the *rank* of a state check that its value decreases in every transition from state to a next one. The construction of the ranking fu hard part and forms a task that needs to be applied to program. The checking part is relatively easy. In our task of constructing ranking functions is the relatively they are constructed on demand based on the examinat a few selected paths through the program.

Furthermore, TERMINATOR is not required to co *one* correct termination argument but rather a set of possible arguments, some of which may be bad guess this set need not be the exact set of the 'right' ranking f only a *superset*. We find the same monotonicity of the of the termination argument as with iterative abstractio for safety (the set of predicates need not be the exact s predicates but only a superset).

Checking the termination argument is the hard method. This is because the termination argument is of ranking functions, not a single ranking function. W ranking function one must show that the rank decreas pre- to post-state after executing each single transition setting it is not sufficient to look at a single transition s we must consider all *finite sequences of transitions*. We that, for every sequence, one of the ranking function

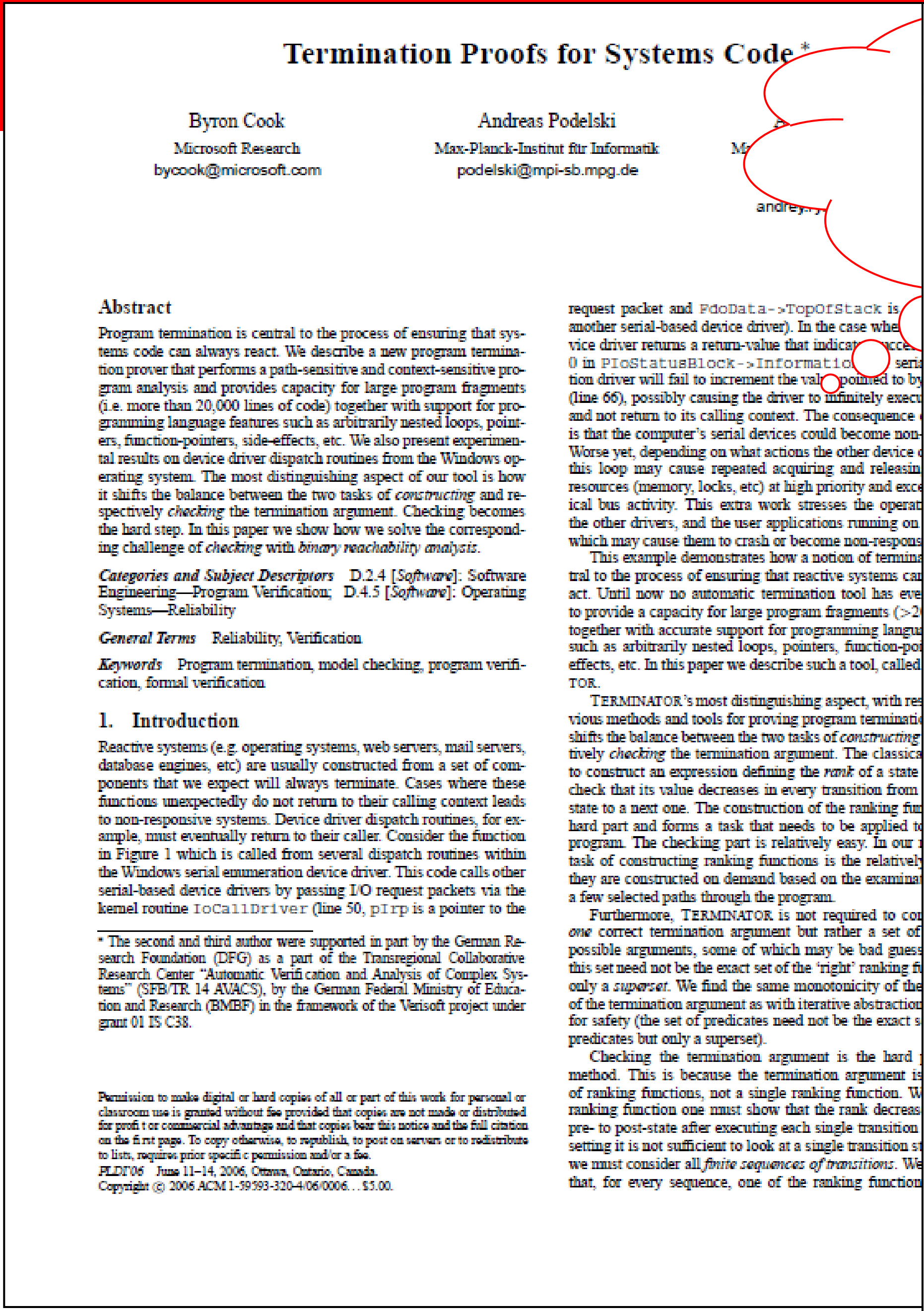
* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.



Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size	
1	12	0	1	1K	3	
2	8	0	0	1K	8	
3	410	0	1	8K	26	
4	1475	0	1	7.5K	24	
5	123292	1	11	5.5K	50	X
6	196	1	3	5K	29	X
7	4174	0	0	8K	23	
8	210	0	11	5K	27	
9	1294	0	5	6K	38	
10	158	0	0	8K	21	
11	13	0	0	2.5K	6	
12	204	0	0	2.5K	16	
13	257	1	1	7.5K	26	X
14	5	0	0	1K	2	
15	141	0	1	6.5K	18	
16	22	0	0	1.5K	2	
17	800	1	6	4K	35	X
18	1503	1	0	6.5K	31	X
19	209	0	3	3K	28	
20	4099	0	2	10K	63	
21	1461	1	4	16K	56	X
22	114762	0	5	34K	65	
23	158746	2	10	35K	75	X

Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

A
M
andrey.r

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine IoCallDriver (line 50, pIrp is a pointer to the request packet and FdoData->TopOfStack is
another serial-based device driver). In the case whe
vice driver returns a return-value that indicat ucce
0 in PIoStatusBlock->Informatio seria
tion driver will fail to increment the val pointed to by
(line 66), possibly causing the driver to infinitely execu
and not return to its calling context. The consequence
is that the computer's serial devices could become non-
Worse yet, depending on what actions the other device
this loop may cause repeated acquiring and releasin
resources (memory, locks, etc) at high priority and exce
ical bus activity. This extra work stresses the operat
the other drivers, and the user applications running on
which may cause them to crash or become non-respons

This example demonstrates how a notion of termina
tral to the process of ensuring that reactive systems ca
act. Until now no automatic termination tool has eve
to provide a capacity for large program fragments (>2
together with accurate support for programming langu
such as arbitrarily nested loops, pointers, function-poi
effects, etc. In this paper we describe such a tool, called
TOR.

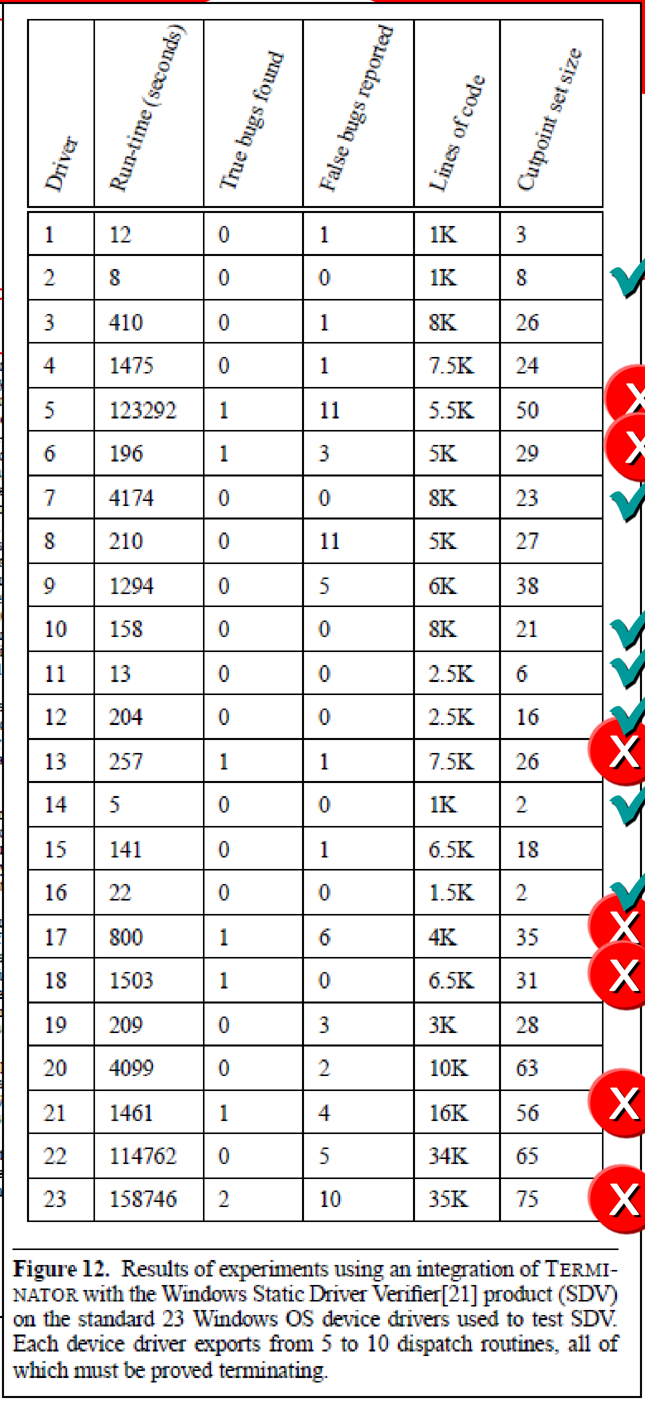
TERMINATOR's most distinguishing aspect, with res
vious methods and tools for proving program terminati
shifts the balance between the two tasks of *constructing*
tively *checking* the termination argument. The classica
to construct an expression defining the *rank* of a state
check that its value decreases in every transition from
state to a next one. The construction of the ranking fu
hard part and forms a task that needs to be applied t
program. The checking part is relatively easy. In our
task of constructing ranking functions is the relatively
they are constructed on demand based on the examina
a few selected paths through the program.

Furthermore, TERMINATOR is not required to co
one correct termination argument but rather a set of
possible arguments, some of which may be bad guess
this set need not be the exact set of the 'right' ranking f
only a *superset*. We find the same monotonicity of the
of the termination argument as with iterative abstractio
for safety (the set of predicates need not be the exact s
predicates but only a superset).

Checking the termination argument is the hard
method. This is because the termination argument is
of ranking functions, not a single ranking function. W
ranking function one must show that the rank decreas
pre- to post-state after executing each single transition
setting it is not sufficient to look at a single transition s
we must consider all *finite sequences of transitions*. We
that, for every sequence, one of the ranking function

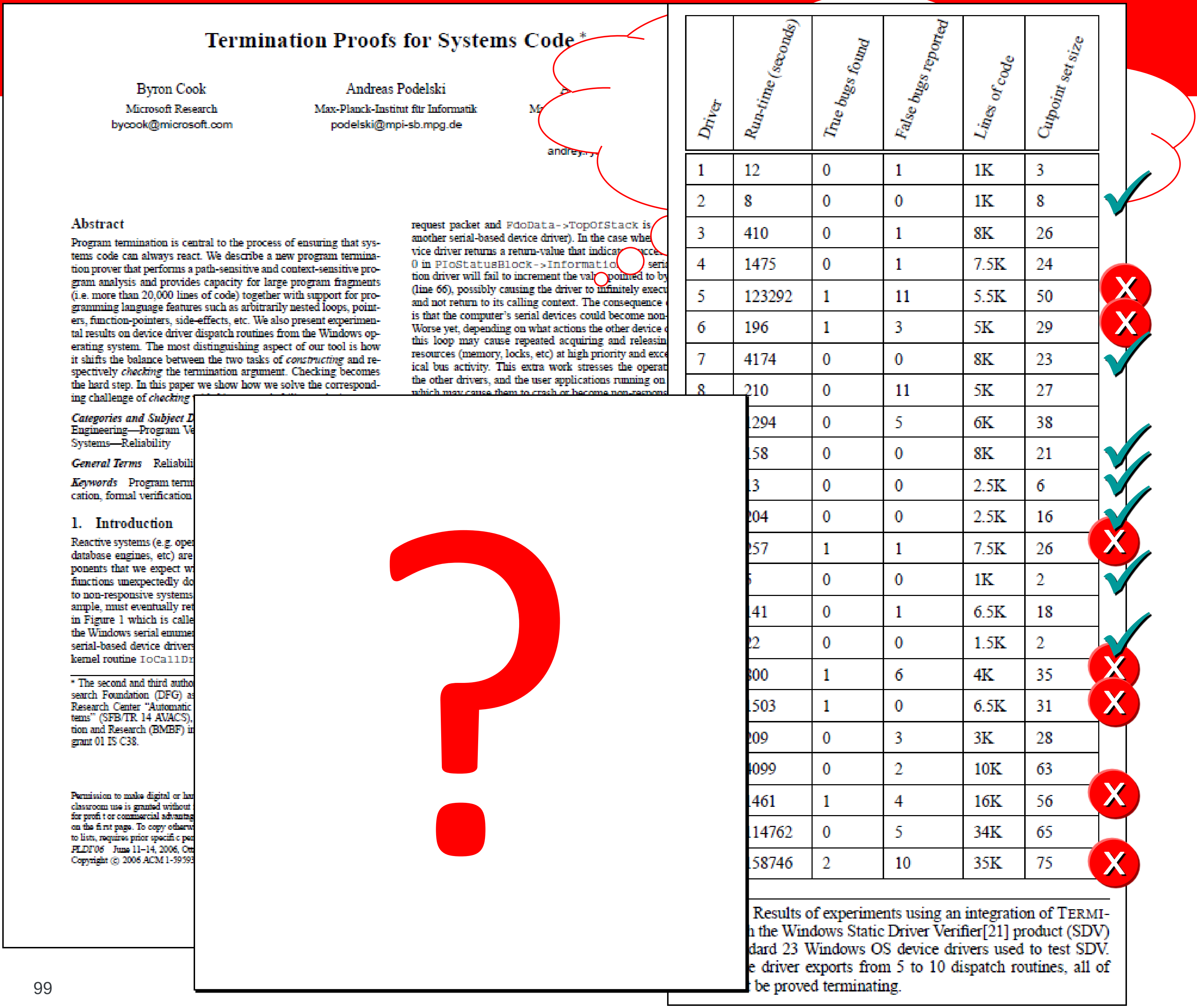
[*] The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.



Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size	
1	12	0	1	1K	3	
2	8	0	0	1K	8	✓
3	410	0	1	8K	26	
4	1475	0	1	7.5K	24	
5	123292	1	11	5.5K	50	X
6	196	1	3	5K	29	X
7	4174	0	0	8K	23	✓
8	210	0	11	5K	27	
9	1294	0	5	6K	38	
10	158	0	0	8K	21	✓
11	13	0	0	2.5K	6	✓
12	204	0	0	2.5K	16	✓
13	257	1	1	7.5K	26	X
14	5	0	0	1K	2	✓
15	141	0	1	6.5K	18	
16	22	0	0	1.5K	2	✓
17	800	1	6	4K	35	X
18	1503	1	0	6.5K	31	X
19	209	0	3	3K	28	
20	4099	0	2	10K	63	
21	1461	1	4	16K	56	X
22	114762	0	5	34K	65	
23	158746	2	10	35K	75	X

Figure 12. Results of experiments using an integration of TERMINATOR with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

A
M

andrey.r

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking*

Categories and Subject D
Engineering—Program Ve
Systems—Reliability

General Terms Reliabili

Keywords Program termi
cation, formal verification

1. Introduction

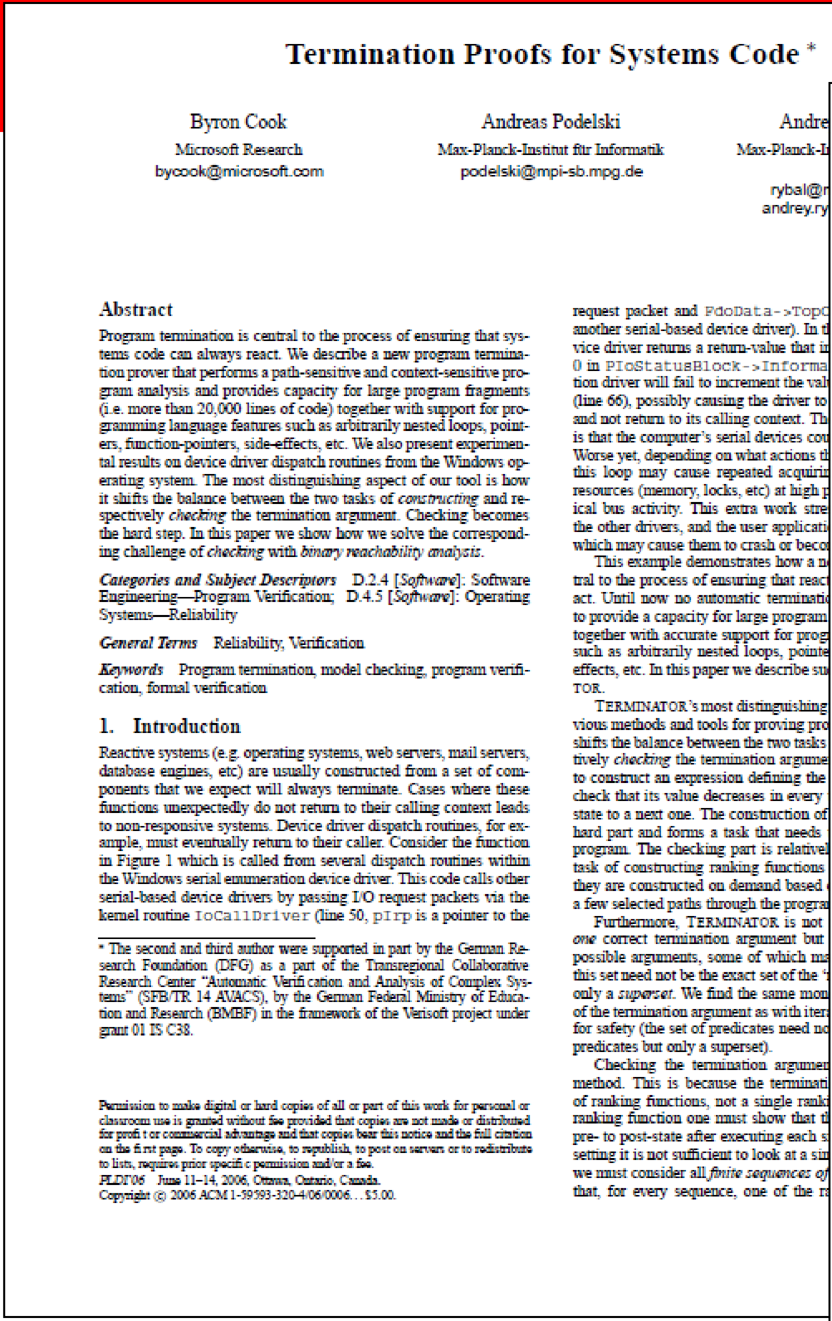
Reactive systems (e.g. oper
database engines, etc) are
ponents that we expect w
functions unexpectedly do
to non-responsive systems
ample, must eventually ret
in Figure 1 which is calle
the Windows serial enumer
serial-based device drivers
kernel routine `IoCallDr`

request packet and `FdoData->TopOfStack` is another serial-based device driver). In the case whe vice driver returns a return-value that indicat succe 0 in `PIoStatusBlock->Informatio` seria tion driver will fail to increment the val pointed to by (line 66), possibly causing the driver to infinitely execu and not return to its calling context. The consequence is that the computer's serial devices could become non- Worse yet, depending on what actions the other device this loop may cause repeated acquiring and releasin resources (memory, locks, etc) at high priority and exce ical bus activity. This extra work stresses the operat the other drivers, and the user applications running on which may cause them to crash or become non-respons

* The second and third autho
search Foundation (DFG) as
Research Center "Automatic
tems" (SFB/TR 14 AVACS),
tion and Research (BMBF) in
grant 01 IS C38.

Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size	
1	12	0	1	1K	3	
2	8	0	0	1K	8	✓
3	410	0	1	8K	26	
4	1475	0	1	7.5K	24	
5	123292	1	11	5.5K	50	X
6	196	1	3	5K	29	X
7	4174	0	0	8K	23	✓
8	210	0	11	5K	27	
	1294	0	5	6K	38	
	158	0	0	8K	21	✓
	13	0	0	2.5K	6	✓
	204	0	0	2.5K	16	✓
	257	1	1	7.5K	26	X
	5	0	0	1K	2	✓
	141	0	1	6.5K	18	
	22	0	0	1.5K	2	✓
	300	1	6	4K	35	X
	1503	1	0	6.5K	31	X
	209	0	3	3K	28	
	4099	0	2	10K	63	
	1461	1	4	16K	56	X
	114762	0	5	34K	65	
	158746	2	10	35K	75	X

Results of experiments using an integration of TERMI-
h the Windows Static Driver Verifier[21] product (SDV)
dard 23 Windows OS device drivers used to test SDV.
e driver exports from 5 to 10 dispatch routines, all of
t be proved terminating.



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I

rybal@r
andrey.ry

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the
request packet and `FdoData->TopO`
another serial-based device driver). In t
vice driver returns a return-value that i
0 in `PIoStatusBlock->Informa`
tion driver will fail to increment the val
(line 66), possibly causing the driver to
and not return to its calling context. Th
is that the computer's serial devices cou
Worse yet, depending on what actions th
this loop may cause repeated acquiri
resources (memory, locks, etc) at high p
ical bus activity. This extra work stre
the other drivers, and the user applicati
which may cause them to crash or beco

This example demonstrates how a n
tral to the process of ensuring that reac
act. Until now no automatic terminati
to provide a capacity for large program
together with accurate support for prog
such as arbitrarily nested loops, pointe
effects, etc. In this paper we describe su
TOR.

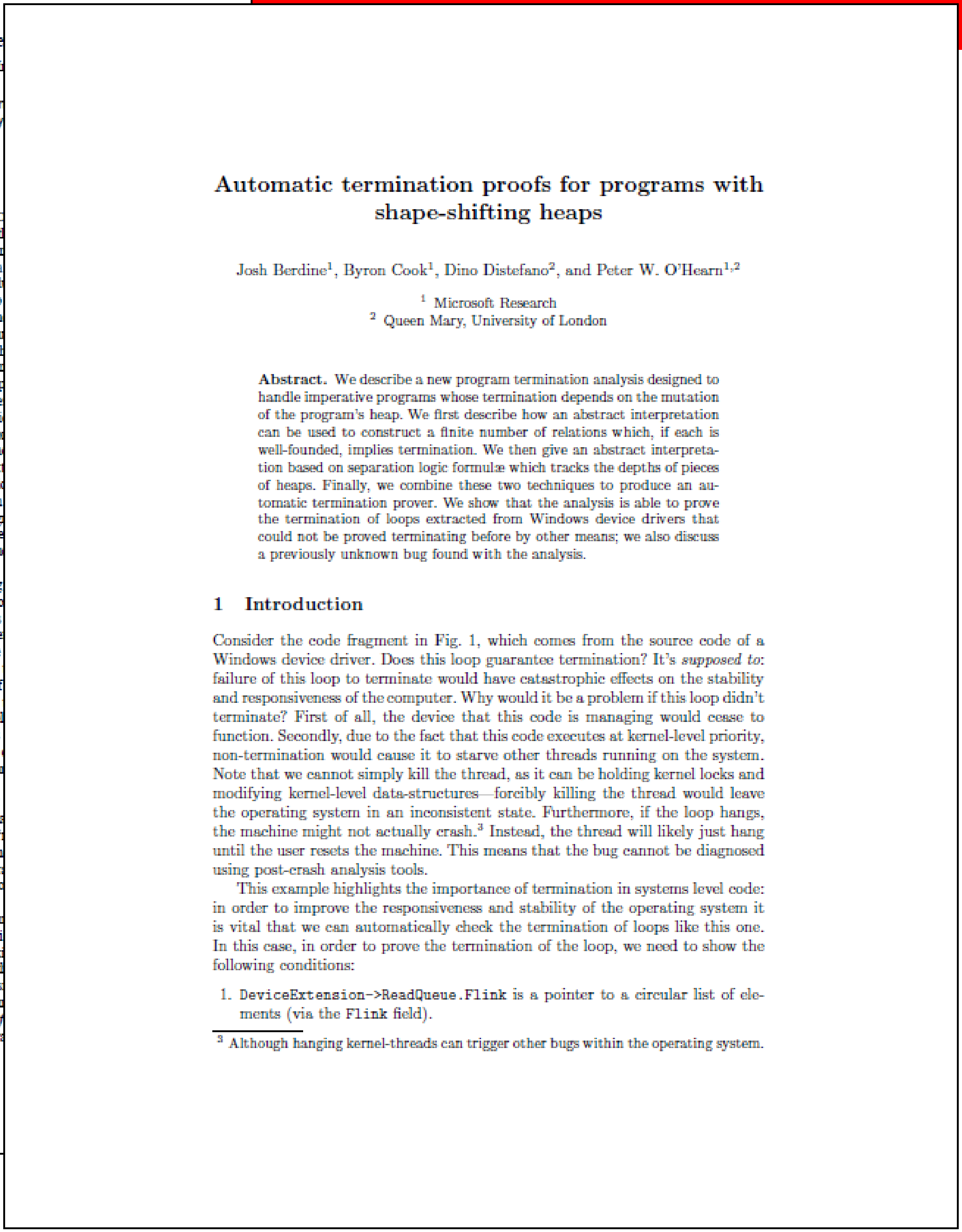
TERMINATOR's most distinguishing
vious methods and tools for proving pro
shifts the balance between the two tasks
tively *checking* the termination argume
to construct an expression defining the
check that its value decreases in every
state to a next one. The construction of
hard part and forms a task that needs
program. The checking part is relativel
task of constructing ranking functions
they are constructed on demand based
a few selected paths through the progra

Furthermore, TERMINATOR is not
one correct termination argument but
possible arguments, some of which m
this set need not be the exact set of the '
only a *superset*. We find the same mo
of the termination argument as with iter
for safety (the set of predicates need n
predicates but only a superset).

Checking the termination argume
method. This is because the terminati
of ranking functions, not a single rank
ranking function one must show that t
pre- to post-state after executing each s
setting it is not sufficient to look at a si
we must consider all *finite sequences o*
that, for every sequence, one of the r

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.



Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a Windows device driver. Does this loop guarantee termination? It's *supposed to*: failure of this loop to terminate would have catastrophic effects on the stability and responsiveness of the computer. Why would it be a problem if this loop didn't terminate? First of all, the device that this code is managing would cease to function. Secondly, due to the fact that this code executes at kernel-level priority, non-termination would cause it to starve other threads running on the system. Note that we cannot simply kill the thread, as it can be holding kernel locks and modifying kernel-level data-structures—forcibly killing the thread would leave the operating system in an inconsistent state. Furthermore, if the loop hangs, the machine might not actually crash.[3] Instead, the thread will likely just hang until the user resets the machine. This means that the bug cannot be diagnosed using post-crash analysis tools.

This example highlights the importance of termination in systems level code: in order to improve the responsiveness and stability of the operating system it is vital that we can automatically check the termination of loops like this one. In this case, in order to prove the termination of the loop, we need to show the following conditions:

1. `DeviceExtension->ReadQueue.Flink` is a pointer to a circular list of elements (via the `Flink` field).

[3] Although hanging kernel-threads can trigger other bugs within the operating system.

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I
rybal@r
andrey.ry

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a Windows device driver. Does this loop guarantee termination? It's *supposed to*: ailure of this loop to terminate would have catastrophic effects on the stability and responsiveness of the computer. Why would it be a problem if this loop didn't erminate? First of all, the device that this code is managing would cease to unction. Secondly, due to the fact that this code executes at kernel-level priority, on-termination would cause it to starve other threads running on the system. Note that we cannot simply kill the thread, as it can be holding kernel locks and nodifying kernel-level data-structures—forcibly killing the thread would leave he operating system in an inconsistent state. Furthermore, if the loop hangs, he machine might not actually crash.[3] Instead, the thread will likely just hang ntil the user resets the machine. This means that the bug cannot be diagnosed sing post-crash analysis tools.

This example highlights the importance of termination in systems level code: n order to improve the responsiveness and stability of the operating system it s vital that we can automatically check the termination of loops like this one. In this case, in order to prove the termination of the loop, we need to show the ollowing conditions:

1. `DeviceExtension->ReadQueue.Flink` is a pointer to a circular list of elements (via the `Flink` field).

[3] Although hanging kernel-threads can trigger other bugs within the operating system.

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Dino Distefano
Queen Mary, University of London
ddino@dcs.qmul.ac.uk

Peter O'Hearn
Queen Mary, University of London
ohearn@dcs.qmul.ac.uk

Abstract

An invariance assertion for a program location ℓ is a statement that always holds at ℓ during execution of the program. Program invariance analyses infer invariance assertions that can be useful when trying to prove safety properties. We use the term *variance assertion* to mean a statement that holds between any state at ℓ and any previous state that was also at ℓ. This paper is concerned with the development of analyses for variance assertions and their application to proving termination and liveness properties. We describe a method of constructing program variance analyses from invariance analyses. If we change the underlying invariance analysis, we get a different variance analysis. We describe several applications of the method, including variance analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variance analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today's state-of-the-art termination provers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Program Analysis, Liveness, Termination

1. Introduction

An *invariance analysis* takes in a program as input and infers a set of possibly disjunctive invariance assertions (*a.k.a.*, invariants) that is indexed by program locations. Each location ℓ in the program has an invariant that always holds during any execution at ℓ. These invariants can serve many purposes. They might be used directly to prove safety properties of programs. Or they might be used indirectly, for example, to aid the construction of abstract transition relations during symbolic software model checking [29]. If a desired safety property is not directly provable from a given invariant, the user (or algorithm calling the invariance analysis) might try to refine the abstraction. For example, if the tool is based on abstract interpretation they may choose to improve the abstraction by delaying the widening operation [28], using dynamic partitioning [33], employing a different abstract domain, etc.

The aim of this paper is to develop an analogous set of tools for program termination and liveness: we introduce a class of tools called *variance analyses* which infer assertions, called *variance assertions*, that hold between any state at a location ℓ and any previous state that was also at location ℓ. Note that a single variance assertion may itself be a disjunction. We present a generic method of constructing variance analyses from invariance analyses. For each invariance analysis, we can construct what we call its *induced variance analysis*.

This paper also introduces a condition on variance assertions called the *local termination predicate*. In this work, we show how the variance assertions inferred during our analysis can be used to establish local termination predicates. If this predicate can be established for each variance assertion inferred for a program, whole program termination has been proved; the correctness of this step relies on a result from [37] on *disjunctively well-founded overapproximations*. Analogously to invariance analysis, even if the induced variance analysis fails to prove whole program termination, it can still produce useful information. If the predicate can be established only for some subset of the variance assertions, this induces a different liveness property that holds of the program. Moreover, the information inferred can be used by other termination provers based on disjunctive well-foundedness, such as TERMINATOR [14]. If the underlying invariance analysis is based on abstract interpretation, the user or algorithm could use the same abstraction refinement techniques that are available for invariance analyses.

In this paper we illustrate the utility of our approach with three induced variance analyses. We construct a variance analysis for arithmetic programs based on the Octagon abstract domain [34]. The invariance analysis used as input to our algorithm is composed of a standard analysis based on Octagon, and a post-analysis phase that recovers some disjunctive information. This gives rise to a fast and yet surprisingly accurate termination prover. We similarly construct an induced variance analysis based on the domain of Polyhedra [23]. Finally, we show that an induced variance analysis based on the separation domain [24] is an improvement on a termination prover that was recently described in the literature [3]. These three

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I

rybal@
andrey.ry

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

ht try to
abstract
y delay-
ng [33],

of tools
of tools
variance
and any
variance
method
ses. For
induced

ssertions
ow how
used to
n be es-
n, whole
this step
ed over-
if the in-
nination,
be estab-
induces
Moreover,
provers
OR [14].
interpre-
n refine-

ith three
ysis for
ain [34].
omposed
is phase
to a fast
rly con-
Polyhe-
is based
mination
ese three

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a Windows device driver. Does this loop guarantee termination? It's *supposed to*: failure of this loop to terminate would have catastrophic effects on the stability and responsiveness of the computer. Why would it be a problem if this loop didn't terminate? First of all, the device that this code is managing would cease to function. Secondly, due to the fact that this code executes at kernel-level priority, non-termination would cause it to starve other threads running on the system. Note that we cannot simply kill the thread, as it can be holding kernel locks and modifying kernel-level data-structures—forcibly killing the thread would leave the operating system in an inconsistent state. Furthermore, if the loop hangs, the machine might not actually crash.[3] Instead, the thread will likely just hang until the user resets the machine. This means that the bug cannot be diagnosed using post-crash analysis tools.

This example highlights the importance of termination in systems level code: in order to improve the responsiveness and stability of the operating system it is vital that we can automatically check the termination of loops like this one. In this case, in order to prove the termination of the loop, we need to show the following conditions:

1. `DeviceExtension->ReadQueue.Flink` is a pointer to a circular list of elements (via the `Flink` field).

[3] Although hanging kernel-threads can trigger other bugs within the operating system.

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Zvonimir Rakamarić
Department of Computer Science, University of British Columbia
{babic,ajh,zrakamar}@cs.ubc.ca

Byron Cook
Microsoft Research
bycook@microsoft.com

Abstract

We describe a simple and efficient algorithm for proving the termination of a class of loops with nonlinear assignments to variables. The method is based on divergence testing for each variable in the cone-of-influence of the loop's termination condition. The analysis allows us to automatically prove the termination of loops that cannot be handled using previous techniques. The paper closes with experimental results using short examples drawn from industrial code.

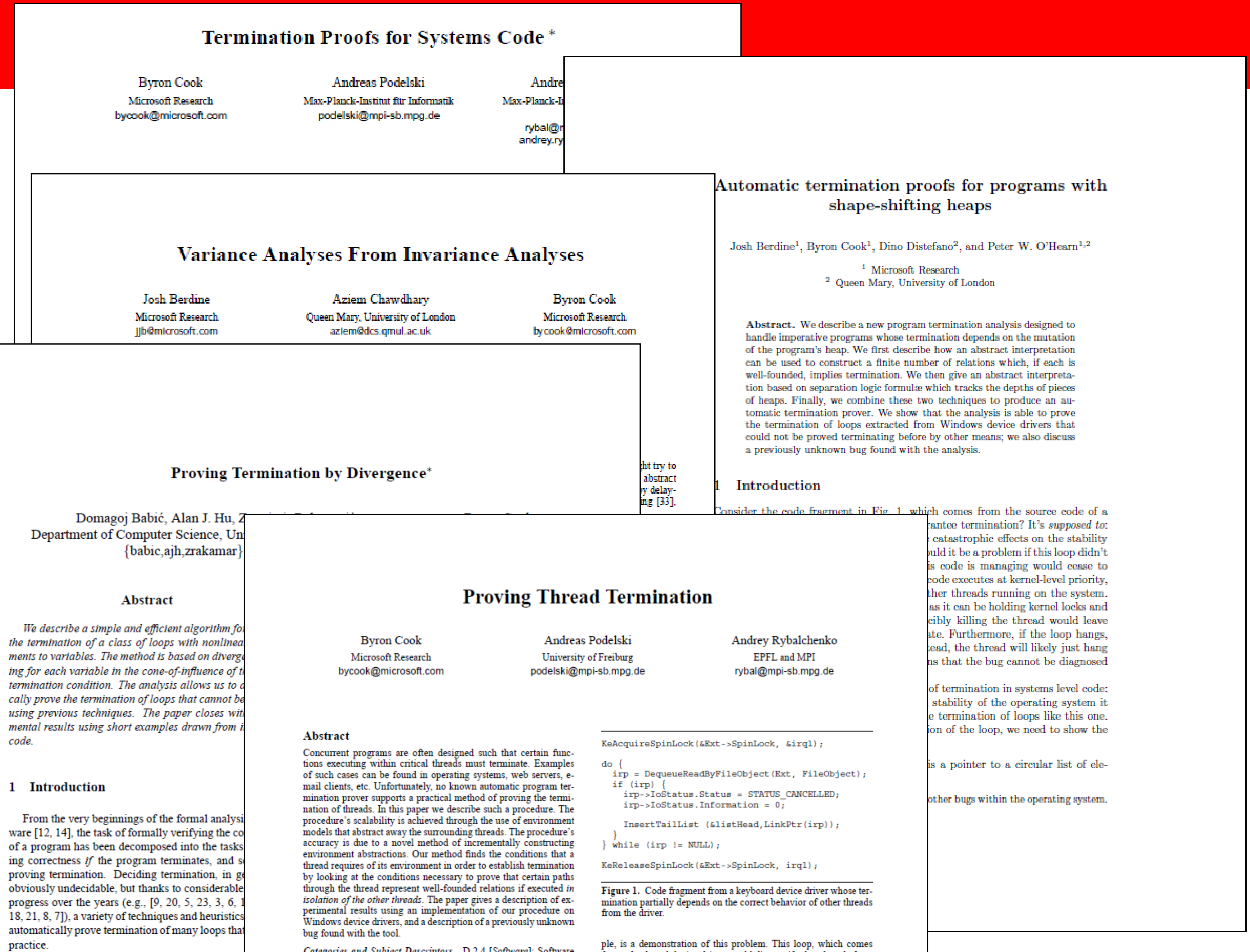
1 Introduction

From the very beginnings of the formal analysis of software [12, 14], the task of formally verifying the correctness of a program has been decomposed into the tasks of proving correctness *if* the program terminates, and separately proving termination. Deciding termination, in general, is obviously undecidable, but thanks to considerable research progress over the years (e.g., [9, 20, 5, 23, 3, 6, 13, 4, 16, 18, 21, 8, 7]), a variety of techniques and heuristics can now automatically prove termination of many loops that occur in practice.

```
 while (x < y) {
   x = pow(x,3) - 2*pow(x,2) - x + 2;
 }
}
```

This paper outlines a new proof procedure for cases of this sort. Using combination techniques described in [1] and [2], our intention for this proposed procedure is to be combined with the existing termination analysis techniques—making future termination provers a little less temperamental.

The proposed technique is based on divergence testing: the transition system of each program variable is independently examined for divergence to plus- or minus-infinity. The approach is limited to loops containing only polynomial update expressions with finite degree, allowing highly efficient computation of certain regions that guarantee divergence. Like all automated termination provers, the technique can't handle all loops. However, it is very fast, it is sound, and it can prove termination in cases that previously could not be handled or could be handled only by a much more expensive analysis. Our hope is that, in practice, this restricted analysis (and some extensions) will handle the termination of the majority of loops in which a nonlinear analysis is required. In our investigations, we have found that this simple type of nonlinear loop appears in industrial numerical computations and nonlinear digital fil-



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I

rybal@
andrey.ry

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a
rantee termination? It's *supposed to*:
catastrophic effects on the stability
uld it be a problem if this loop didn't
is code is managing would cease to
code executes at kernel-level priority,
ther threads running on the system.
as it can be holding kernel locks and
cibly killing the thread would leave
ate. Furthermore, if the loop hangs,
ead, the thread will likely just hang
ns that the bug cannot be diagnosed

of termination in systems level code:
stability of the operating system it
e termination of loops like this one.
ion of the loop, we need to show the

is a pointer to a circular list of ele-

other bugs within the operating system.

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

ht try to
abstract
y delay-
ng [33],

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Z
Department of Computer Science, Un
{babic,ajh,zrakamar}

Abstract

We describe a simple and efficient algorithm fo
the termination of a class of loops with nonlinea
ments to variables. The method is based on diverge
ing for each variable in the cone-of-influence of t
termination condition. The analysis allows us to a
cally prove the termination of loops that cannot be
using previous techniques. The paper closes wit
mental results using short examples drawn from i
code.

1 Introduction

From the very beginnings of the formal analysi
ware [12, 14], the task of formally verifying the co
of a program has been decomposed into the tasks
ing correctness *if* the program terminates, and s
proving termination. Deciding termination, in ge
obviously undecidable, but thanks to considerable
progress over the years (e.g., [9, 20, 5, 23, 3, 6, 1
18, 21, 8, 7]), a variety of techniques and heuristics
automatically prove termination of many loops that
practice.

Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
EPFL and MPI
rybal@mpi-sb.mpg.de

Abstract

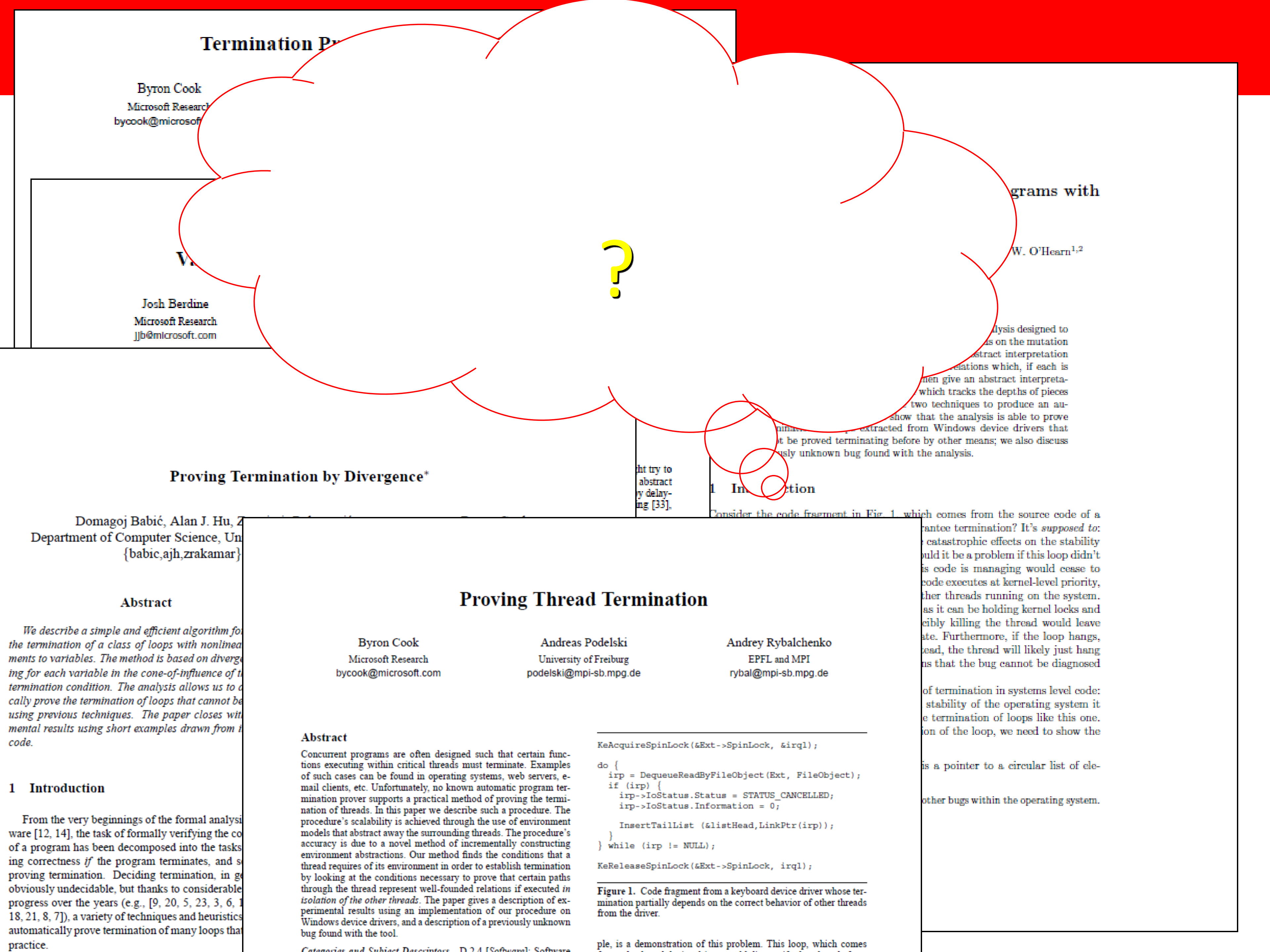
Concurrent programs are often designed such that certain functions executing within critical threads must terminate. Examples of such cases can be found in operating systems, web servers, e-mail clients, etc. Unfortunately, no known automatic program termination prover supports a practical method of proving the termination of threads. In this paper we describe such a procedure. The procedure's scalability is achieved through the use of environment models that abstract away the surrounding threads. The procedure's accuracy is due to a novel method of incrementally constructing environment abstractions. Our method finds the conditions that a thread requires of its environment in order to establish termination by looking at the conditions necessary to prove that certain paths through the thread represent well-founded relations if executed *in isolation of the other threads*. The paper gives a description of experimental results using an implementation of our procedure on Windows device drivers, and a description of a previously unknown bug found with the tool.

Categories and Subject Descriptors D.2.4 [Software]: Software

```
KeAcquireSpinLock(&Ext->SpinLock, &irql);

do {
  irp = DequeueReadByFileObject(Ext, FileObject);
  if (irp) {
    irp->IoStatus.Status = STATUS_CANCELLED;
    irp->IoStatus.Information = 0;

    InsertTailList (&listHead,LinkPtr(irp));
  }
} while (irp != NULL);

KeReleaseSpinLock(&Ext->SpinLock, irql);
```

Figure 1. Code fragment from a keyboard device driver whose termination partially depends on the correct behavior of other threads from the driver.

ple, is a demonstration of this problem. This loop, which comes



Termination Pr

Byron Cook
Microsoft Research
bycook@microsof

V

Josh Berdine
Microsoft Research
jjb@microsoft.com

grams with

W. O'Hearn[1,2]

lysis designed to
is on the mutation
stract interpretation
relations which, if each is
hen give an abstract interpreta-
which tracks the depths of pieces
two techniques to produce an au-
show that the analysis is able to prove
ninat extracted from Windows device drivers that
ot be proved terminating before by other means; we also discuss
usly unknown bug found with the analysis.

1 In tion

Consider the code fragment in Fig. 1, which comes from the source code of a
rantee termination? It's *supposed to*:
catastrophic effects on the stability
uld it be a problem if this loop didn't
is code is managing would cease to
code executes at kernel-level priority,
ther threads running on the system.
as it can be holding kernel locks and
cibly killing the thread would leave
ate. Furthermore, if the loop hangs,
tead, the thread will likely just hang
ns that the bug cannot be diagnosed

of termination in systems level code:
stability of the operating system it
e termination of loops like this one.
ion of the loop, we need to show the

is a pointer to a circular list of ele-

other bugs within the operating system.

ht try to
abstract
y delay-
ing [33].

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Z
Department of Computer Science, Un
{babic,ajh,zrakamar}

Abstract

We describe a simple and efficient algorithm fo
the termination of a class of loops with nonlinea
ments to variables. The method is based on diverge
ing for each variable in the cone-of-influence of t
termination condition. The analysis allows us to a
cally prove the termination of loops that cannot be
using previous techniques. The paper closes wit
mental results using short examples drawn from i
code.

1 Introduction

From the very beginnings of the formal analysi
ware [12, 14], the task of formally verifying the co
of a program has been decomposed into the tasks
ing correctness *if* the program terminates, and s
proving termination. Deciding termination, in ge
obviously undecidable, but thanks to considerable
progress over the years (e.g., [9, 20, 5, 23, 3, 6, 1
18, 21, 8, 7]), a variety of techniques and heuristics
automatically prove termination of many loops that
practice.

Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
EPFL and MPI
rybal@mpi-sb.mpg.de

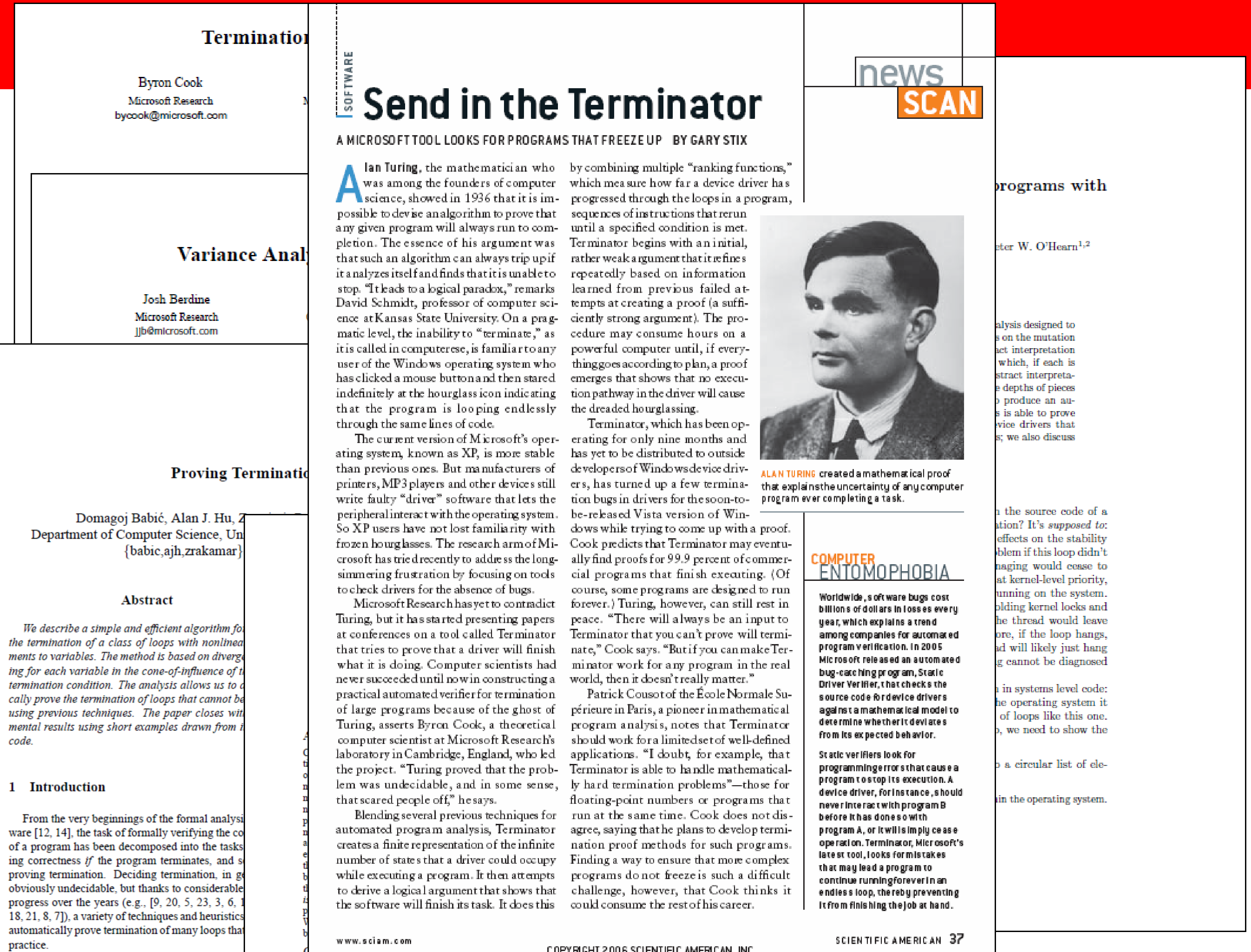
Abstract

Concurrent programs are often designed such that certain functions executing within critical threads must terminate. Examples of such cases can be found in operating systems, web servers, e-mail clients, etc. Unfortunately, no known automatic program termination prover supports a practical method of proving the termination of threads. In this paper we describe such a procedure. The procedure's scalability is achieved through the use of environment models that abstract away the surrounding threads. The procedure's accuracy is due to a novel method of incrementally constructing environment abstractions. Our method finds the conditions that a thread requires of its environment in order to establish termination by looking at the conditions necessary to prove that certain paths through the thread represent well-founded relations if executed *in isolation of the other threads*. The paper gives a description of experimental results using an implementation of our procedure on Windows device drivers, and a description of a previously unknown bug found with the tool.

```
KeAcquireSpinLock(&Ext->SpinLock, &irql);

do {
  irp = DequeueReadByFileObject(Ext, FileObject);
  if (irp) {
    irp->IoStatus.Status = STATUS_CANCELLED;
    irp->IoStatus.Information = 0;

    InsertTailList (&listHead,LinkPtr(irp));
  }
} while (irp != NULL);

KeReleaseSpinLock(&Ext->SpinLock, irql);
```

Figure 1. Code fragment from a keyboard device driver whose termination partially depends on the correct behavior of other threads from the driver.

ple, is a demonstration of this problem. This loop, which comes



Terminatio

Byron Cook
Microsoft Research
bycook@microsoft.com

Variance Anal

Josh Berdine
Microsoft Research
jjb@microsoft.com

Proving Terminatio

Domagoj Babić, Alan J. Hu, Z
Department of Computer Science, Un
{babic,ajh,zrakamar}

Abstract

We describe a simple and efficient algorithm fo the termination of a class of loops with nonlinea ments to variables. The method is based on diverge ing for each variable in the cone-of-influence of t termination condition. The analysis allows us to a cally prove the termination of loops that cannot be using previous techniques. The paper closes wit mental results using short examples drawn from i code.

1 Introduction

From the very beginnings of the formal analysi ware [12, 14], the task of formally verifying the co of a program has been decomposed into the tasks ing correctness *if* the program terminates, and s proving termination. Deciding termination, in ge obviously undecidable, but thanks to considerable progress over the years (e.g., [9, 20, 5, 23, 3, 6, 1 18, 21, 8, 7]), a variety of techniques and heuristics automatically prove termination of many loops that practice.

SOFTWARE

Send in the Terminator

A MICROSOFT TOOL LOOKS FOR PROGRAMS THAT FREEZE UP BY GARY STIX

Alan Turing, the mathematician who was among the founders of computer science, showed in 1936 that it is impossible to devise an algorithm to prove that any given program will always run to completion. The essence of his argument was that such an algorithm can always trip up if it analyzes itself and finds that it is unable to stop. "It leads to a logical paradox," remarks David Schmidt, professor of computer science at Kansas State University. On a pragmatic level, the inability to "terminate," as it is called in computerese, is familiar to any user of the Windows operating system who has clicked a mouse button and then stared indefinitely at the hourglass icon indicating that the program is looping endlessly through the same lines of code.

The current version of Microsoft's operating system, known as XP, is more stable than previous ones. But manufacturers of printers, MP3 players and other devices still write faulty "driver" software that lets the peripheral interact with the operating system. So XP users have not lost familiarity with frozen hourglasses. The research arm of Microsoft has tried recently to address the long-simmering frustration by focusing on tools to check drivers for the absence of bugs.

Microsoft Research has yet to contradict Turing, but it has started presenting papers at conferences on a tool called Terminator that tries to prove that a driver will finish what it is doing. Computer scientists had never succeeded until now in constructing a practical automated verifier for termination of large programs because of the ghost of Turing, asserts Byron Cook, a theoretical computer scientist at Microsoft Research's laboratory in Cambridge, England, who led the project. "Turing proved that the problem was undecidable, and in some sense, that scared people off," he says.

Blending several previous techniques for automated program analysis, Terminator creates a finite representation of the infinite number of states that a driver could occupy while executing a program. It then attempts to derive a logical argument that shows that the software will finish its task. It does this by combining multiple "ranking functions," which measure how far a device driver has progressed through the loops in a program, sequences of instructions that rerun until a specified condition is met. Terminator begins with an initial, rather weak argument that it refines repeatedly based on information learned from previous failed attempts at creating a proof (a sufficiently strong argument). The procedure may consume hours on a powerful computer until, if everything goes according to plan, a proof emerges that shows that no execution pathway in the driver will cause the dreaded hourglassing.

Terminator, which has been operating for only nine months and has yet to be distributed to outside developers of Windows device drivers, has turned up a few termination bugs in drivers for the soon-to-be-released Vista version of Windows while trying to come up with a proof. Cook predicts that Terminator may eventually find proofs for 99.9 percent of commercial programs that finish executing. (Of course, some programs are designed to run forever.) Turing, however, can still rest in peace. "There will always be an input to Terminator that you can't prove will terminate," Cook says. "But if you can make Terminator work for any program in the real world, then it doesn't really matter."

Patrick Cousot of the École Normale Supérieure in Paris, a pioneer in mathematical program analysis, notes that Terminator should work for a limited set of well-defined applications. "I doubt, for example, that Terminator is able to handle mathematically hard termination problems"—those for floating-point numbers or programs that run at the same time. Cook does not disagree, saying that he plans to develop termination proof methods for such programs. Finding a way to ensure that more complex programs do not freeze is such a difficult challenge, however, that Cook thinks it could consume the rest of his career.

ALAN TURING created a mathematical proof that explains the uncertainty of any computer program ever completing a task.

news SCAN

COMPUTER ENTOMOPHOBIA

Worldwide, software bugs cost billions of dollars in losses every year, which explains a trend among companies for automated program verification. In 2005 Microsoft released an automated bug-catching program, Static Driver Verifier, that checks the source code for device drivers against a mathematical model to determine whether it deviates from its expected behavior.

Static verifiers look for programming errors that cause a program to stop its execution. A device driver, for instance, should never interact with program B before it has done so with program A, or it will simply cease operation. Terminator, Microsoft's latest tool, looks for mistakes that may lead a program to continue running forever in an endless loop, thereby preventing it from finishing the job at hand.

www.sciam.com

SCIENTIFIC AMERICAN 37

COPYRIGHT 2006 SCIENTIFIC AMERICAN, INC.

programs with

eter W. O'Hearn[1,2]

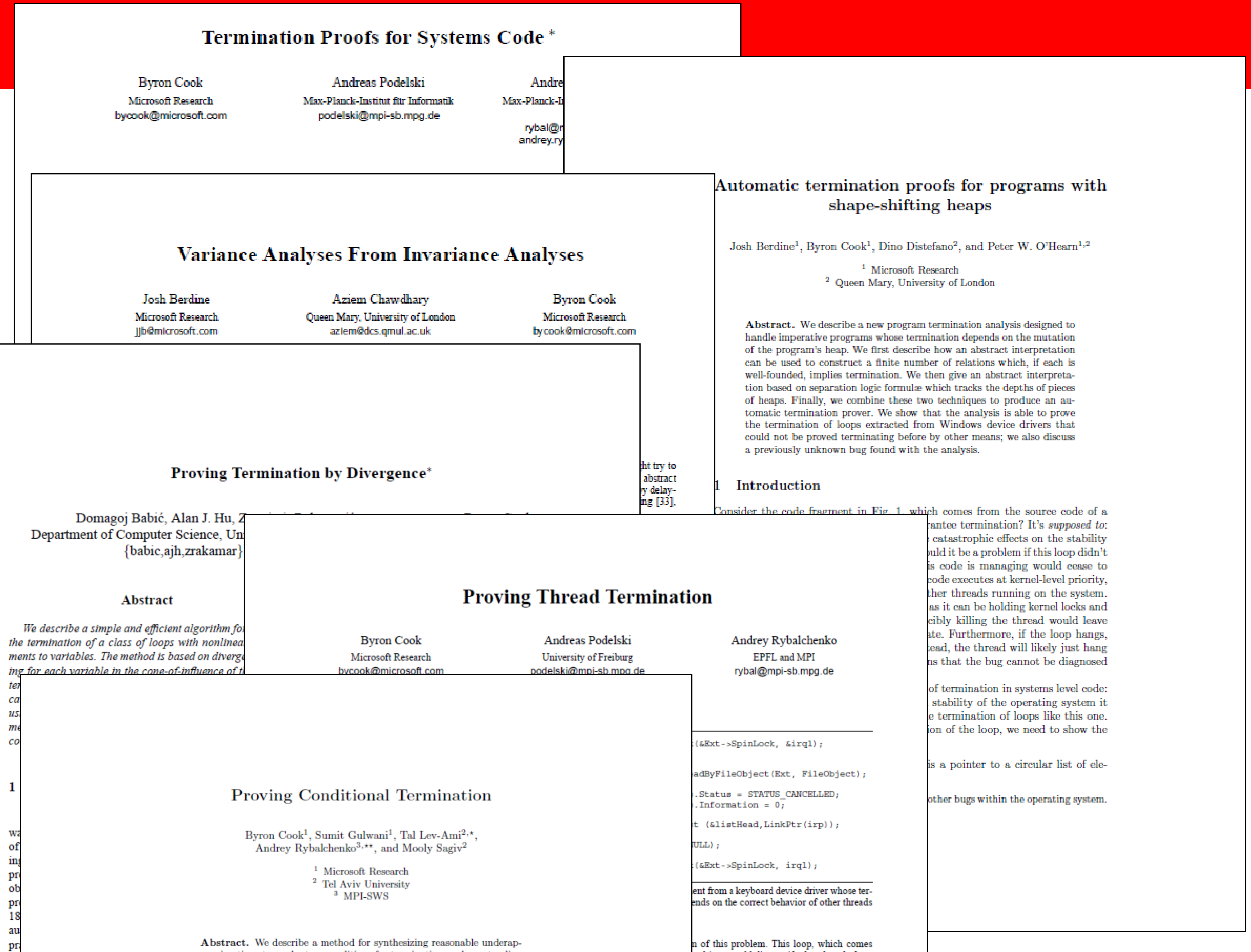
alysis designed to
s on the mutation
ct interpretation
which, if each is
stract interpreta-
e depths of pieces
produce an au-
s is able to prove
vice drivers that
; we also discuss

n the source code of a
ation? It's *supposed to*:
effects on the stability
blem if this loop didn't
naging would cease to
at kernel-level priority,
unning on the system.
olding kernel locks and
he thread would leave
ore, if the loop hangs,
d will likely just hang
g cannot be diagnosed

n in systems level code:
he operating system it
of loops like this one.
, we need to show the

a circular list of ele-

in the operating system.



Termination Proofs for Systems Code*

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I
rybal@r
andrey.ry

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

ht try to
abstract
y delay-
ng [33].

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a
rantee termination? It's *supposed to*:
catastrophic effects on the stability
uld it be a problem if this loop didn't
is code is managing would cease to
code executes at kernel-level priority,
ther threads running on the system.
as it can be holding kernel locks and
cibly killing the thread would leave
ate. Furthermore, if the loop hangs,
ead, the thread will likely just hang
ns that the bug cannot be diagnosed

of termination in systems level code:
stability of the operating system it
e termination of loops like this one.
ion of the loop, we need to show the

is a pointer to a circular list of ele-

other bugs within the operating system.

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Z
Department of Computer Science, Un
{babic,ajh,zrakamar}

Abstract

We describe a simple and efficient algorithm fo
the termination of a class of loops with nonlinea
ments to variables. The method is based on diverge
ing for each variable in the cone-of-influence of t
te
ca
us
me
co

1

wa
of
ing
pr
ob
pr
18
au
pr

Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
EPFL and MPI
rybal@mpi-sb.mpg.de

```
(&Ext->SpinLock, &irql);

adByFileObject(Ext, FileObject);

.Status = STATUS_CANCELLED;
.Information = 0;

t (&listHead,LinkPtr(irp));

ULL);

(&Ext->SpinLock, irql);
```

ent from a keyboard device driver whose ter-
nds on the correct behavior of other threads

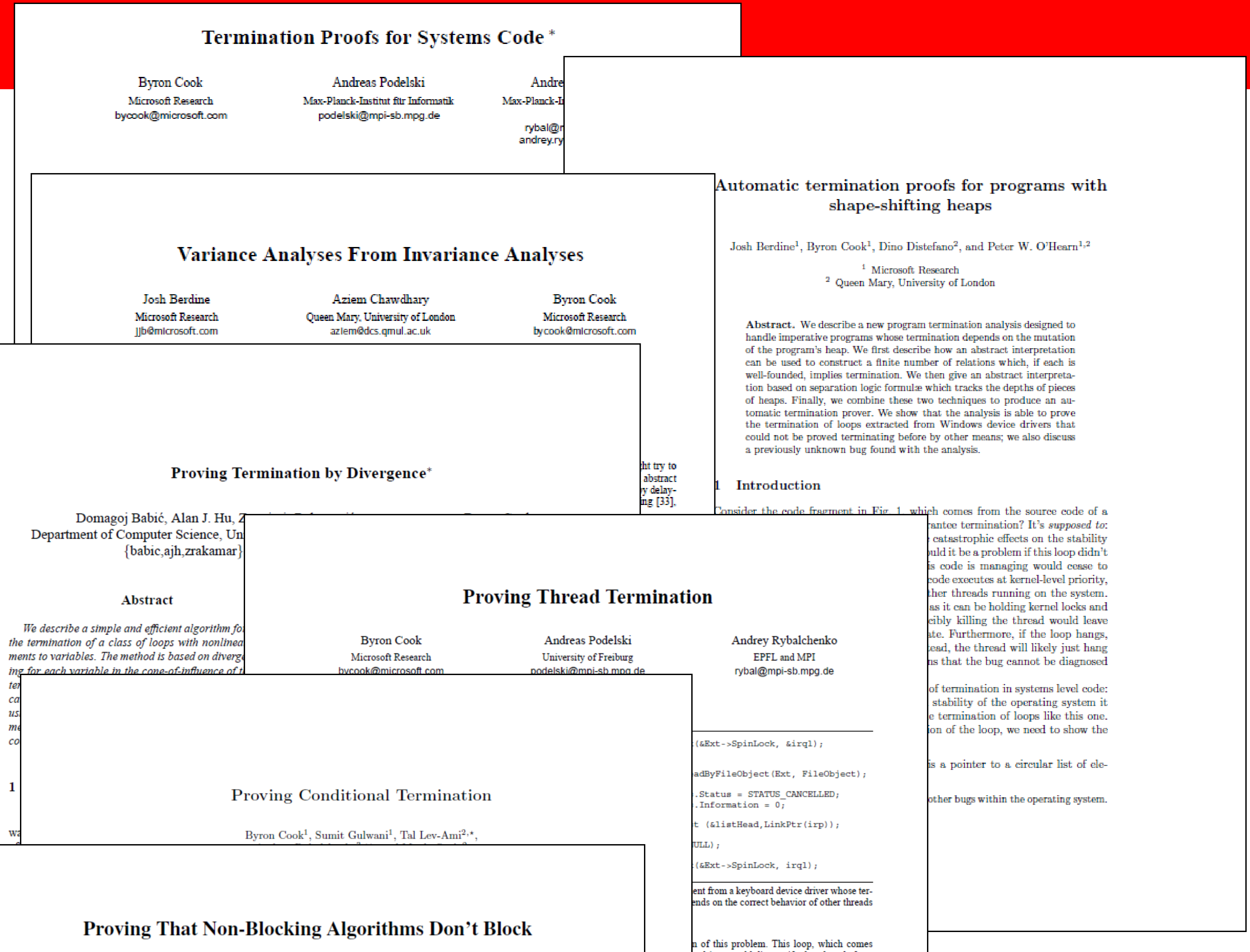
n of this problem. This loop, which comes

Proving Conditional Termination

Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,*],
Andrey Rybalchenko[3,**], and Mooly Sagiv[2]

[1] Microsoft Research
[2] Tel Aviv University
[3] MPI-SWS

Abstract. We describe a method for synthesizing reasonable underap-



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I
ybal@r
andrey.ry

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a
rantee termination? It's *supposed to*:
catastrophic effects on the stability
uld it be a problem if this loop didn't
s code is managing would cease to
ode executes at kernel-level priority,
her threads running on the system.
as it can be holding kernel locks and
ibly killing the thread would leave
te. Furthermore, if the loop hangs,
ead, the thread will likely just hang
s that the bug cannot be diagnosed

of termination in systems level code:
stability of the operating system it
e termination of loops like this one.
on of the loop, we need to show the

is a pointer to a circular list of ele-

other bugs within the operating system.

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

ht try to
abstract
y delay-
ng [33].

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Z
Department of Computer Science, Un
{babic,ajh,zrakamar}

Abstract

We describe a simple and efficient algorithm fo
the termination of a class of loops with nonlinea
ments to variables. The method is based on diverge
ing for each variable in the cone-of-influence of t
ter
ca
us
me
co

1

wa

Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
EPFL and MPI
rybal@mpi-sb.mpg.de

```
(&Ext->SpinLock, &irql);

adByFileObject(Ext, FileObject);

.Status = STATUS_CANCELLED;
.Information = 0;

t (&listHead,LinkPtr(irp));

ULL);

(&Ext->SpinLock, irql);
```

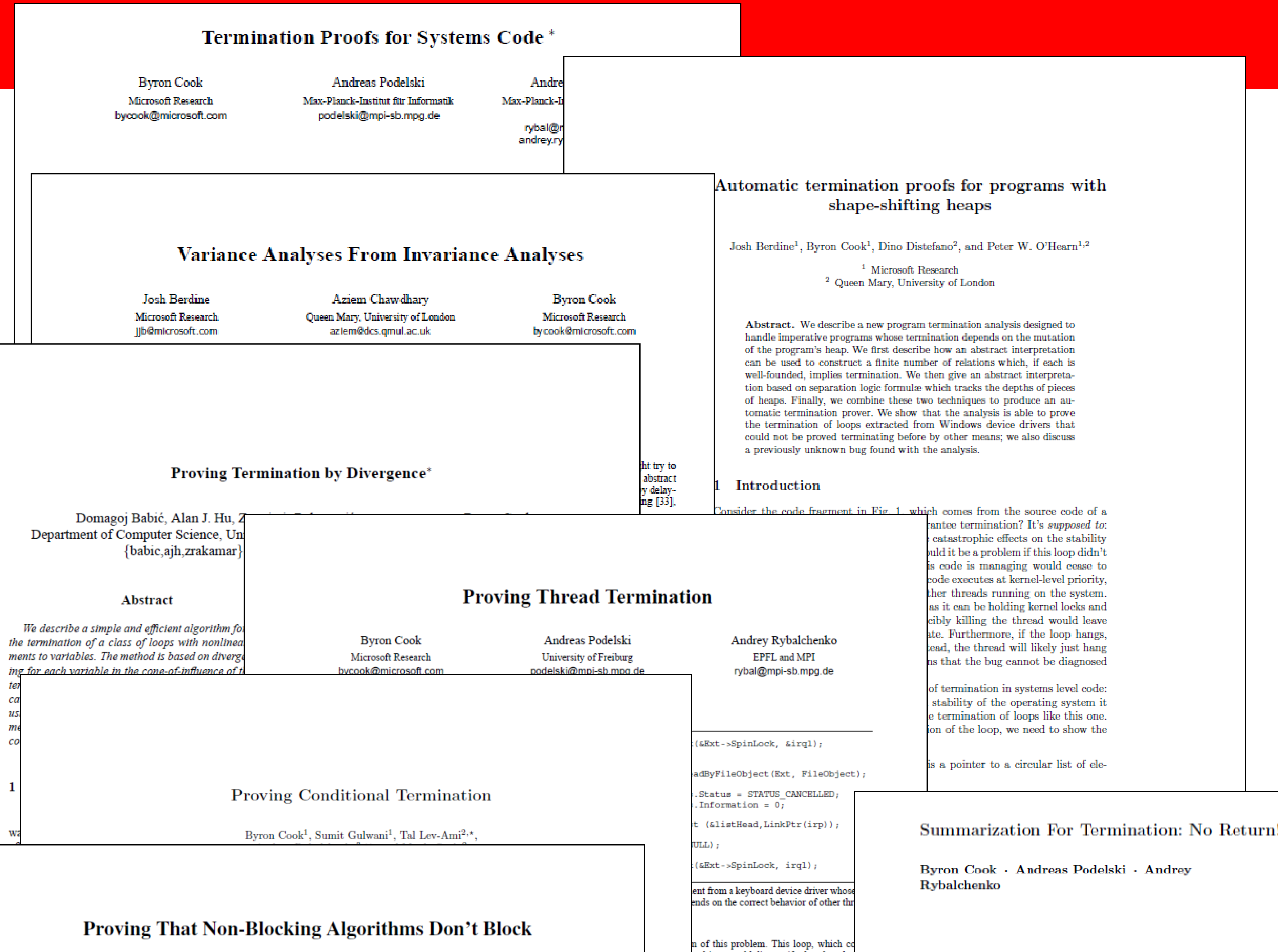
ent from a keyboard device driver whose ter-
ends on the correct behavior of other threads

n of this problem. This loop, which comes

Proving Conditional Termination

Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,*],

Proving That Non-Blocking Algorithms Don't Block



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I
rybal@r
andrey.ry

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research
[2] Queen Mary, University of London

Abstract. We describe a new program termination analysis designed to handle imperative programs whose termination depends on the mutation of the program's heap. We first describe how an abstract interpretation can be used to construct a finite number of relations which, if each is well-founded, implies termination. We then give an abstract interpretation based on separation logic formulæ which tracks the depths of pieces of heaps. Finally, we combine these two techniques to produce an automatic termination prover. We show that the analysis is able to prove the termination of loops extracted from Windows device drivers that could not be proved terminating before by other means; we also discuss a previously unknown bug found with the analysis.

1 Introduction

Consider the code fragment in Fig. 1, which comes from the source code of a
rantee termination? It's *supposed to*:
catastrophic effects on the stability
uld it be a problem if this loop didn't
is code is managing would cease to
code executes at kernel-level priority,
ther threads running on the system.
as it can be holding kernel locks and
cibly killing the thread would leave
ate. Furthermore, if the loop hangs,
ead, the thread will likely just hang
ns that the bug cannot be diagnosed

of termination in systems level code:
stability of the operating system it
e termination of loops like this one.
ion of the loop, we need to show the

is a pointer to a circular list of ele-

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

ht try to
abstract
y delay-
ng [33].

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Z
Department of Computer Science, Un
{babic,ajh,zrakamar}

Abstract

We describe a simple and efficient algorithm fo the termination of a class of loops with nonlinea ments to variables. The method is based on diverge ing for each variable in the cone-of-influence of t
ter
ca
us
me
co

1

wa

Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
EPFL and MPI
rybal@mpi-sb.mpg.de

```
(&Ext->SpinLock, &irql);

adByFileObject(Ext, FileObject);

.Status = STATUS_CANCELLED;
.Information = 0;

t (&listHead,LinkPtr(irp));

ULL);

(&Ext->SpinLock, irql);
```

ent from a keyboard device driver whose
ends on the correct behavior of other thr

n of this problem. This loop, which co

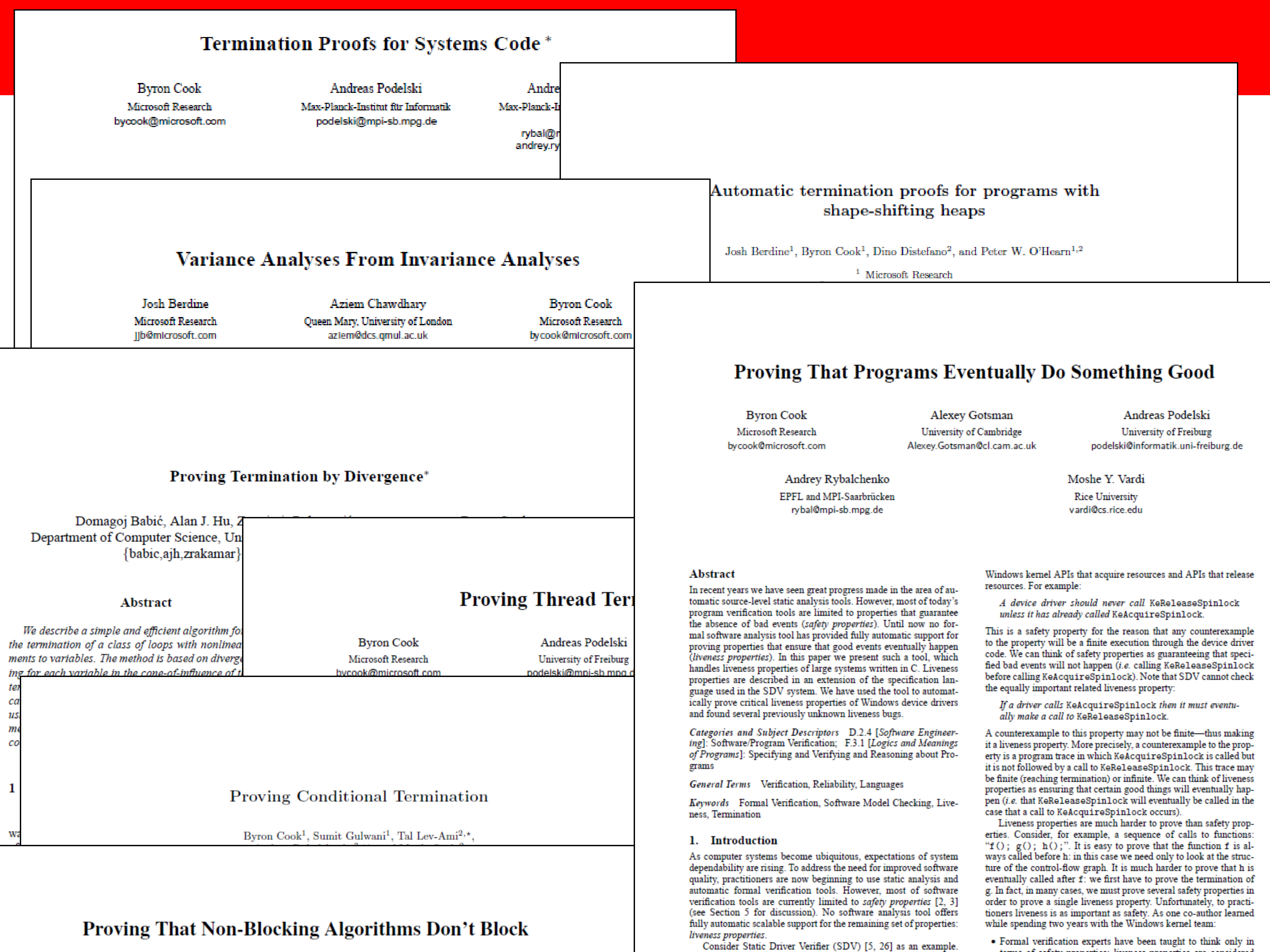
Proving Conditional Termination

Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,*],

Summarization For Termination: No Return!

Byron Cook · Andreas Podelski · Andrey Rybalchenko

Proving That Non-Blocking Algorithms Don't Block



Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I
rybal@
andrey.ry

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research

Variance Analyses From Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Proving Termination by Divergence*

Domagoj Babić, Alan J. Hu, Z
Department of Computer Science, Un
{babic,ajh,zrakamar}

Abstract

We describe a simple and efficient algorithm fo the termination of a class of loops with nonlinea ments to variables. The method is based on diverge ing for each variable in the cone-of-influence of t
te
ca
us
me
co

1

wa

Proving Thread Ter

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.d

Proving Conditional Termination

Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,*],

Proving That Non-Blocking Algorithms Don't Block

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk

Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de

Andrey Rybalchenko
EPFL and MPI-Saarbrücken
rybal@mpi-sb.mpg.de

Moshe Y. Vardi
Rice University
vardi@cs.rice.edu

Abstract

In recent years we have seen great progress made in the area of automatic source-level static analysis tools. However, most of today's program verification tools are limited to properties that guarantee the absence of bad events (*safety properties*). Until now no formal software analysis tool has provided fully automatic support for proving properties that ensure that good events eventually happen (*liveness properties*). In this paper we present such a tool, which handles liveness properties of large systems written in C. Liveness properties are described in an extension of the specification language used in the SDV system. We have used the tool to automatically prove critical liveness properties of Windows device drivers and found several previously unknown liveness bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Liveness, Termination

1. Introduction

As computer systems become ubiquitous, expectations of system dependability are rising. To address the need for improved software quality, practitioners are now beginning to use static analysis and automatic formal verification tools. However, most of software verification tools are currently limited to *safety properties* [2, 3] (see Section 5 for discussion). No software analysis tool offers fully automatic scalable support for the remaining set of properties: *liveness properties*.

Consider Static Driver Verifier (SDV) [5, 26] as an example. Windows kernel APIs that acquire resources and APIs that release resources. For example:

> *A device driver should never call* KeReleaseSpinlock *unless it has already called* KeAcquireSpinlock.

This is a safety property for the reason that any counterexample to the property will be a finite execution through the device driver code. We can think of safety properties as guaranteeing that specified bad events will not happen (*i.e.* calling KeReleaseSpinlock before calling KeAcquireSpinlock). Note that SDV cannot check the equally important related liveness property:

> *If a driver calls* KeAcquireSpinlock *then it must eventually make a call to* KeReleaseSpinlock.

A counterexample to this property may not be finite—thus making it a liveness property. More precisely, a counterexample to the property is a program trace in which KeAcquireSpinlock is called but it is not followed by a call to KeReleaseSpinlock. This trace may be finite (reaching termination) or infinite. We can think of liveness properties as ensuring that certain good things will eventually happen (*i.e.* that KeReleaseSpinlock will eventually be called in the case that a call to KeAcquireSpinlock occurs).

Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions: "f(); g(); h();". It is easy to prove that the function f is always called before h: in this case we need only to look at the structure of the control-flow graph. It is much harder to prove that h is eventually called after f: we first have to prove the termination of g. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team:

- Formal verification experts have been taught to think only in

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I

Automatic termination proofs for programs with shape-shifting heaps

Josh Berdine[1], Byron Cook[1], Dino Distefano[2], and Peter W. O'Hearn[1,2]

[1] Microsoft Research

Temporal property verification as a program analysis task

Byron Cook[1], Eric Koskinen[2], and Moshe Vardi[3]

[1] Microsoft Research and Queen Mary University of London
[2] University of Cambridge
[3] Rice University

Abstract. We describe a reduction from temporal property verification to a program analysis problem. We produce an encoding which, with the use of recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (*e.g.* backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.). Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (*e.g.* abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (*e.g.* [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (*e.g.* [3, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique leads to speedups by orders of magnitude for the universal fragment of CTL (∀CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential non-recursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* ∀CTL rather than CTL).

Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk

Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de

Andrey Rybalchenko
EPFL and MPI-Saarbrücken
rybal@mpi-sb.mpg.de

Moshe Y. Vardi
Rice University
vardi@cs.rice.edu

Abstract

In recent years we have seen great progress made in the area of automatic source-level static analysis tools. However, most of today's program verification tools are limited to properties that guarantee the absence of bad events (*safety properties*). Until now no formal software analysis tool has provided fully automatic support for proving properties that ensure that good events eventually happen (*liveness properties*). In this paper we present such a tool, which handles liveness properties of large systems written in C. Liveness properties are described in an extension of the specification language used in the SDV system. We have used the tool to automatically prove critical liveness properties of Windows device drivers and found several previously unknown liveness bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Liveness, Termination

1. Introduction

As computer systems become ubiquitous, expectations of system dependability are rising. To address the need for improved software quality, practitioners are now beginning to use static analysis and automatic formal verification tools. However, most of software verification tools are currently limited to *safety properties* [2, 3] (see Section 5 for discussion). No software analysis tool offers fully automatic scalable support for the remaining set of properties: *liveness properties*.

Consider Static Driver Verifier (SDV) [5, 26] as an example. Windows kernel APIs that acquire resources and APIs that release resources. For example:

> *A device driver should never call* KeReleaseSpinlock *unless it has already called* KeAcquireSpinlock.

This is a safety property for the reason that any counterexample to the property will be a finite execution through the device driver code. We can think of safety properties as guaranteeing that specified bad events will not happen (*i.e.* calling KeReleaseSpinlock before calling KeAcquireSpinlock). Note that SDV cannot check the equally important related liveness property:

> *If a driver calls* KeAcquireSpinlock *then it must eventually make a call to* KeReleaseSpinlock.

A counterexample to this property may not be finite—thus making it a liveness property. More precisely, a counterexample to the property is a program trace in which KeAcquireSpinlock is called but it is not followed by a call to KeReleaseSpinlock. This trace may be finite (reaching termination) or infinite. We can think of liveness properties as ensuring that certain good things will eventually happen (*i.e.* that KeReleaseSpinlock will eventually be called in the case that a call to KeAcquireSpinlock occurs).

Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions: "f(); g(); h();". It is easy to prove that the function f is always called before h: in this case we need only to look at the structure of the control-flow graph. It is much harder to prove that h is eventually called after f: we first have to prove the termination of g. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team:

- Formal verification experts have been taught to think only in

Dep

We des
the termi
ments to v
ing for ea
te
ca
us
me
co

1

wa

Proving That Non-Blocking Algorithms Don't Block

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I

Temporal property verification as a program analysis task

Byron Cook[1], Eric Koskinen[2], and Moshe Vardi[3]

[1] Microsoft Research and Queen Mary University of London
[2] University of Cambridge
[3] Rice University

Abstract. We describe a reduction from temporal property verification to a program analysis problem. We produce an encoding which, with the use of recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (*e.g.* backtracking, eventuality checking, tree counterexamples for branching-time properties, abstraction refinement, etc.). Using examples drawn from the PostgreSQL database server, Apache web server, and Windows OS kernel, we demonstrate the practical viability of our work.

1 Introduction

We describe a method of proving temporal properties of (possibly infinite-state) transition systems. We observe that, with subtle use of recursion and nondeterminism, temporal reasoning can be encoded as a program analysis problem. All of the tasks necessary for reasoning about temporal properties (*e.g.* abstraction search, backtracking, eventuality checking, tree counterexamples for branching-time, etc.) are then naturally performed by off-the-shelf program analysis tools. Using known safety analysis tools (*e.g.* [2, 5, 8, 24, 32]) together with techniques for discovering termination arguments (*e.g.* [3, 6, 17]), we can implement temporal logic provers whose power is effectively limited only by the power of the underlying tools.

Based on our method, we have developed a prototype tool for proving temporal properties of C programs and applied it to problems from the PostgreSQL database server, the Apache web server, and the Windows OS kernel. Our technique leads to speedups by orders of magnitude for the universal fragment of CTL (∀CTL). Similar performance improvements result when proving LTL with our technique in combination with a recently described iterative symbolic determinization procedure [15].

Limitations. While in principle our technique works for all classes of transition systems, our approach is currently geared to support only sequential nonrecursive infinite-state programs as its input. Furthermore, we currently only support the universal fragments of temporal logics (*i.e.* ∀CTL rather than CTL).

Making Prophecies with Decision Predicates

Byron Cook
Microsoft Research &
Queen Mary, University of London
bycook@microsoft.com

Eric Koskinen
University of Cambridge
ejk39@cam.ac.uk

Abstract

We describe a new algorithm for proving temporal properties expressed in LTL of infinite-state programs. Our approach takes advantage of the fact that LTL properties can often be proved more efficiently using techniques usually associated with the branching-time logic CTL than they can with native LTL algorithms. The caveat is that, in certain instances, nondeterminism in the system's transition relation can cause CTL methods to report counterexamples that are spurious with respect to the original LTL formula. To address this problem we describe an algorithm that, as it attempts to apply CTL proof methods, finds and then removes problematic nondeterminism via an analysis on the potentially spurious counterexamples. Problematic nondeterminism is characterized using *decision predicates*, and removed using a partial, symbolic determinization procedure which introduces new prophecy variables to predict the future outcome of these choices. We demonstrate—using examples taken from the PostgreSQL database server, Apache web server, and Windows OS kernel—that our method can yield enormous performance improvements in comparison to known tools, allowing us to automatically prove properties of programs where we could not prove them before.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; Correctness proofs; Reliability; D.4.5 [*Operating Systems*]: Reliability—Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Verification, Theory, Reliability

Keywords Linear temporal logic, formal verification, termination, program analysis, model checking

1. Introduction

The common wisdom amongst users and developers of tools that prove temporal properties of systems is that the linear specification logic LTL [33] is more intuitive than CTL [10], but that properties expressed in the universal fragment of CTL (∀CTL) without fairness constraints are often easier to prove than their LTL cousins [3, 32, 44][1]. Properties expressed in CTL without fairness can be proved in a purely syntax-directed manner using state-based reasoning techniques, whereas LTL requires deeper reasoning about whole sets of traces and the subtle relationships between families of them.

In this paper we aim to make an LTL prover for infinite-state programs with performance closer to what one would expect from a CTL prover. We use the observation that ∀CTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don't always fit together: there are cases when, due to some instances of nondeterminism in the transition system, ∀CTL alone is not powerful enough to prove an LTL property.

In these cases our LTL prover works around the problem using something we call *decision predicates*, which are used to characterize and treat such instances of nondeterminism. A decision predicate is represented as a pair of first-order logic formulae (a, b), where the formula a defines the decision predicate's presupposition (*i.e. when* the decision is made), and b characterizes the binary choice made when this presupposition holds. Any transition from state s to state s' in the system that meets the constraint $a(s) \wedge b(s')$ is distinguished by the decision predicate (a, b) from $a(s) \wedge \neg b(s')$.

We use decision predicates as the basis of a partial symbolic determinization procedure: for each predicate we introduce a new prophecy variable [3] to predict the future outcome of the decision. After partially determinizing with respect to these prophecy variables, we find that CTL proof methods succeed, thus allowing us to prove LTL properties with CTL proof techniques in cases where this strategy would have previously failed. To synthesize the decision predicates we employ a form of symbolic execution on spurious ∀CTL counterexamples together with an application of Farkas' lemma [23].

With our new approach we can automatically prove properties of infinite-state programs in minutes or seconds which were intractable using existing tools. Examples include code fragments drawn from the PostgreSQL database server, the Apache web server, and the Windows OS kernel.

Limitations. In practice, the applicability and performance of our technique is dependent on the heuristic used to choose new decision predicates when given an abstract representation of a specific point in a spurious counterexample. The predicate synthesis mechanism implemented in our tool is applicable primarily to infinite-state programs over arithmetic variables with commands that only contain linear arithmetic. However, no matter which predicate selection mechanism is used, our predicate-based determinization strategy is sound. Thus, unsound approximations to predicate synthesis could potentially be used in instances where the systems considered do not meet the constraints given above. Our technique is also based

[1] Abadi and Lamport [3] make this point using the terminology of "refinement mappings" and "trace equivalence" instead of phrasing it in the context of temporal logics.

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01…$10.00

d

ki
rg
eiburg.de

s that release
pinlock
nterexample
device driver
g that speci-
seSpinlock
cannot check
t eventu-
-thus making
e to the prop-
is called but
his trace may
k of liveness
entually hap-
called in the
safety prop-
to functions:
ction f is al-
at the struc-

Dep

We des
the termi
ments to
ing for e
te
ca
us
me
co

1

wa

quality, practitioners are now beginning to use static analysis and automatic formal verification tools. However, most of software verification tools are currently limited to *safety properties* [2, 3] (see Section 5 for discussion). No software analysis tool offers fully automatic scalable support for the remaining set of properties: *liveness properties*.

Consider Static Driver Verifier (SDV) [5, 26] as an example.

eventually called after f: we first have to prove the termination of g. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team:

- Formal verification experts have been taught to think only in

Proving That Non-Blocking Algorithms Don't Block

Termination Proofs for Systems Code *

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andre
Max-Planck-I

Temporal
as a pro

Byron Cook[1], E

[1] Microsoft Research
[2] U

Abstract. We describe a
to a program analysis pro
the use of recursion and n
analysis tools to naturally
temporal properties (*e.g.* b
terexamples for branching-
Using examples drawn fro
web server, and Windows
bility of our work.

1 Introduction

We describe a method of provin
transition systems. We observe
minism, temporal reasoning ca
of the tasks necessary for reaso
search, backtracking, eventualit
time, etc.) are then naturally p
Using known safety analysis to
for discovering termination arg
poral logic provers whose powe
underlying tools.
Based on our method, we h
poral properties of C programs
database server, the Apache we
nique leads to speedups by ord
CTL (∀CTL). Similar performa
our technique in combination w
minimization procedure [15].

Limitations. While in principl
tion systems, our approach is c
recursive infinite-state program
support the universal fragments

Dep

We des
the termi
ments to
ing for e
te
ca
us
me
co

1

wa

Proving That Non-Blocking

Proving stabilization for biological systems

Byron Cook[1,2], Jasmin Fisher[1], Elzbieta Krepska[1,3], and Nir Piterman[4]

[1] Microsoft Research
[2] Queen Mary, University of London
[3] VU University Amsterdam
[4] Imperial College London

Abstract. We describe an efficient procedure for proving stabilization of biological systems modeled as qualitative networks. For scalability, our procedure uses modular proof techniques, where state-space exploration is applied only locally to small pieces of the system rather than the entire system as a whole. Our procedure exploits the observation that, in practice, the form of modular proofs required can be restricted to a very limited set. Using our new procedure, we have solved a number of challenging published examples, including a 3-D model of the mammalian epidermis, a model of metabolic networks operating in type-2 diabetes, and a model of fate determination of vulval precursor cells in the *C. elegans* worm. Our results show many orders of magnitude speedup in cases where previous stabilization proving techniques were known to succeed, and new results in cases where tools had previously failed.

1 Introduction

Biologists are increasingly turning to techniques from computer science in their quest to understand and predict the behavior of complex biological systems [2–4]. In particular, the application of formal verification tools to models of biological processes is gaining impetus among biologists. In some cases known formal verification techniques work well (*e.g.* [5–7]). Unfortunately in other cases—such as proving stabilization [8]—we find that existing abstractions and heuristics are not effective.

In this paper we address the open challenge to find scalable algorithms for proving stabilization of biological systems. In computer science terms, we are trying to prove a liveness property similar to termination of large parallel systems. The sizes of these systems forces us to use some form of modular reasoning. Unfortunately, because stabilization is a liveness property, we must be careful when using the more powerful cyclic modular proof rules (*e.g.* [9,10]), as they are formally only sound in the context of safety [11]. Furthermore, we find that the complex temporal interactions between the modules are crucial to the stabilization of the system as a whole; meaning that we cannot use scalable techniques that simply abstract away the interactions altogether.

In this paper we show that in practice non-circular modular proofs can be found using local liveness lemmas of a limited form:

$$[FG(p_1) \wedge \ldots \wedge FG(p_n)] \Rightarrow FG(q)$$

cision Predicates

Eric Koskinen
University of Cambridge
ejk39@cam.ac.uk

[3, 32, 44][1]. Properties expressed in CTL without fair-
n be proved in a purely syntax-directed manner using state-
easoning techniques, whereas LTL requires deeper reason-
ut whole sets of traces and the subtle relationships between
of them.
is paper we aim to make an LTL prover for infinite-state
s with performance closer to what one would expect from
prover. We use the observation that ∀CTL without fairness
useful abstraction of LTL. The problem with this strategy
he pieces don't always fit together: there are cases when,
ome instances of nondeterminism in the transition system,
lone is not powerful enough to prove an LTL property.
ese cases our LTL prover works around the problem using
ng we call *decision predicates*, which are used to character-
treat such instances of nondeterminism. A decision predi-
represented as a pair of first-order logic formulae (a, b),
he formula a defines the decision predicate's presupposi-
when the decision is made), and b characterizes the binary
made when this presupposition holds. Any transition from
o state s' in the system that meets the constraint $a(s) \wedge b(s')$
guished by the decision predicate (a, b) from $a(s) \wedge \neg b(s')$.
use decision predicates as the basis of a partial symbolic
nization procedure: for each predicate we introduce a new
y variable [3] to predict the future outcome of the decision.
artially determinizing with respect to these prophecy vari-
e find that CTL proof methods succeed, thus allowing us
LTL properties with CTL proof techniques in cases where
tegy would have previously failed. To synthesize the deci-
dicates we employ a form of symbolic execution on spuri-
TL counterexamples together with an application of Farkas'
[23].
our new approach we can automatically prove properties
ite-state programs in minutes or seconds which were in-
e using existing tools. Examples include code fragments
from the PostgreSQL database server, the Apache web
and the Windows OS kernel.

ions. In practice, the applicability and performance of our
ue is dependent on the heuristic used to choose new decision
es when given an abstract representation of a specific point
rious counterexample. The predicate synthesis mechanism
ented in our tool is applicable primarily to infinite-state pro-
ver arithmetic variables with commands that only contain
rithmetic. However, no matter which predicate selection
ism is used, our predicate-based determinization strategy is
Thus, unsound approximations to predicate synthesis could
lly be used in instances where the systems considered do
t the constraints given above. Our technique is also based

nd Lamport [3] make this point using the terminology of "refine-
ppings" and "trace equivalence" instead of phrasing it in the con-
mporal logics.

d

ki
rg
eiburg.de

s that release

pinlock

nterexample
device driver
g that speci-
seSpinlock
cannot check

st eventu-

-thus making
e to the prop-
k is called but
his trace may
k of liveness
entually hap-
called in the

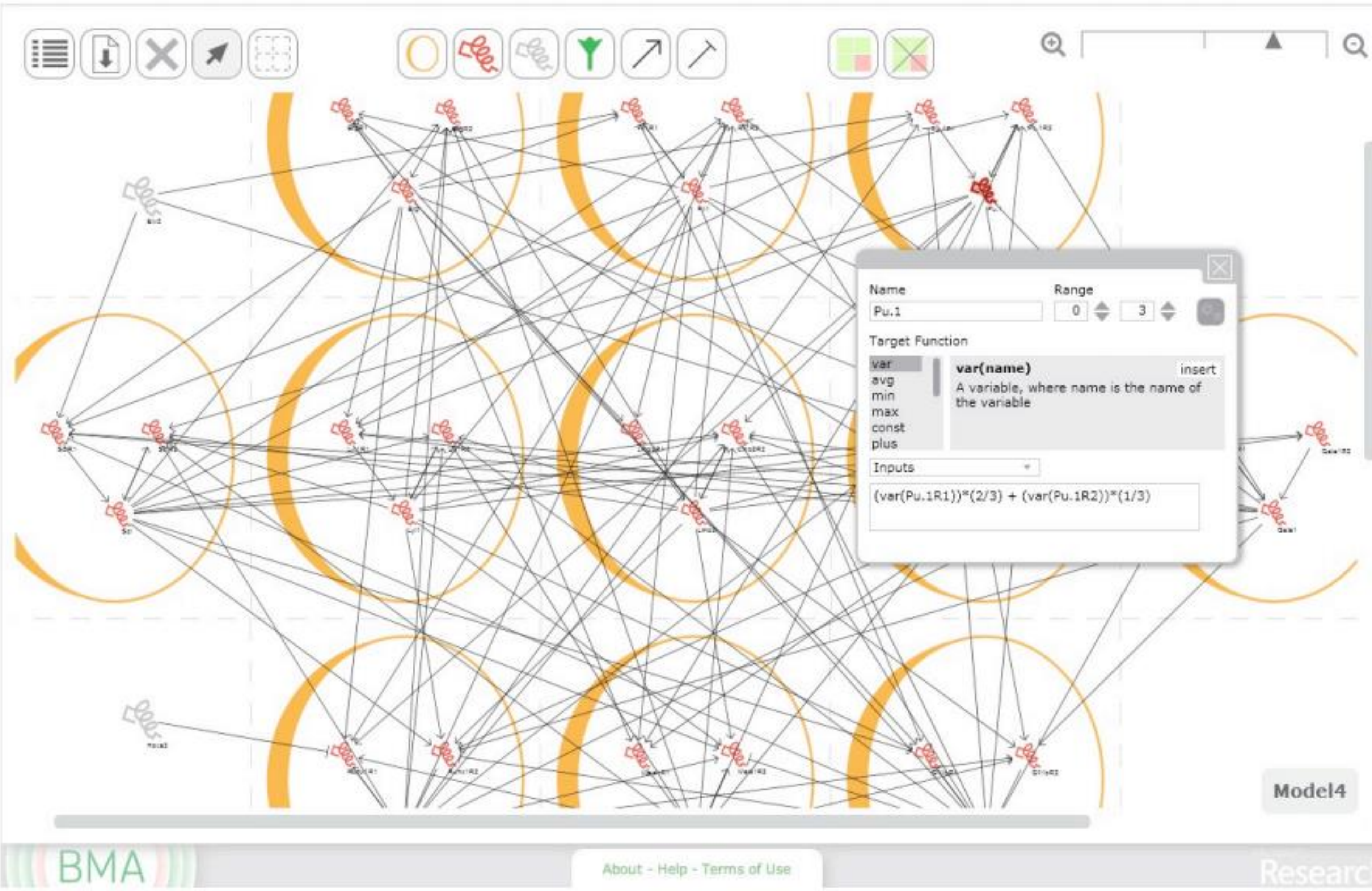
safety prop-
to functions:
ction f is al-
k at the struc-
prove that h is
eventually called after f: we first have to prove the termination of g. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two years with the Windows kernel team:

- Formal verification experts have been taught to think only in

ftware
is and
ftware
[2, 3]
offers
erties:

ample.

Termination Proofs for Systems Code *



Name

Pu.1

Range

0

3

Target Function

var(name)

A variable, where name is the name of the variable

Inputs

(var(Pu.1R1))*(2/3) + (var(Pu.1R2))*(1/3)

Byron Cook · Andreas Podelski · Andrey Rybalchenko

Proving That Non-Blocking Algorithms Don't Block

uses ranking relations to conser

program states. One of the attra

Outline

→ Introduction

→ Termination basics & history

→ New advances for program termination proving

- Proving termination argument validity
- Finding termination arguments

→ Conclusion

Outline

→ Introduction

→ Termination basics & history

→ New advances for program termination proving

- Proving termination argument validity
- Finding termination arguments

→ Conclusion

Future work

- Previous wisdom: proving termination for industrial systems code is impossible

- Now people are beginning to think that it’s effectively “solved”.

- Much left to do, including
  - Complex data structures (safety)
  - Infinite-state systems w/ bit vectors (safety)
  - Binaries (safety)
  - Non-linear systems (liveness and safety)
  - Better support for concurrent programs
  - Modern programming features (*e.g.* closures)
  - Scalability, performance, precision

Future work

- Termination proving is at the heart of many undecidable problems (*e.g.* Wang's tiling problem)
- Modern termination proving techniques could potentially be used to building working tools
- Challenge: "black-box" solutions to undecidable problems die in the most unpredictable ways

Conclusion

→ Conventional wisdom about termination overturned

- Undecidable does not mean we cannot soundly approximate a solution

→ TERMINATOR shows that automatic termination proving is not hopeless for industrial systems code

→ Current state-of-the-art solutions based on

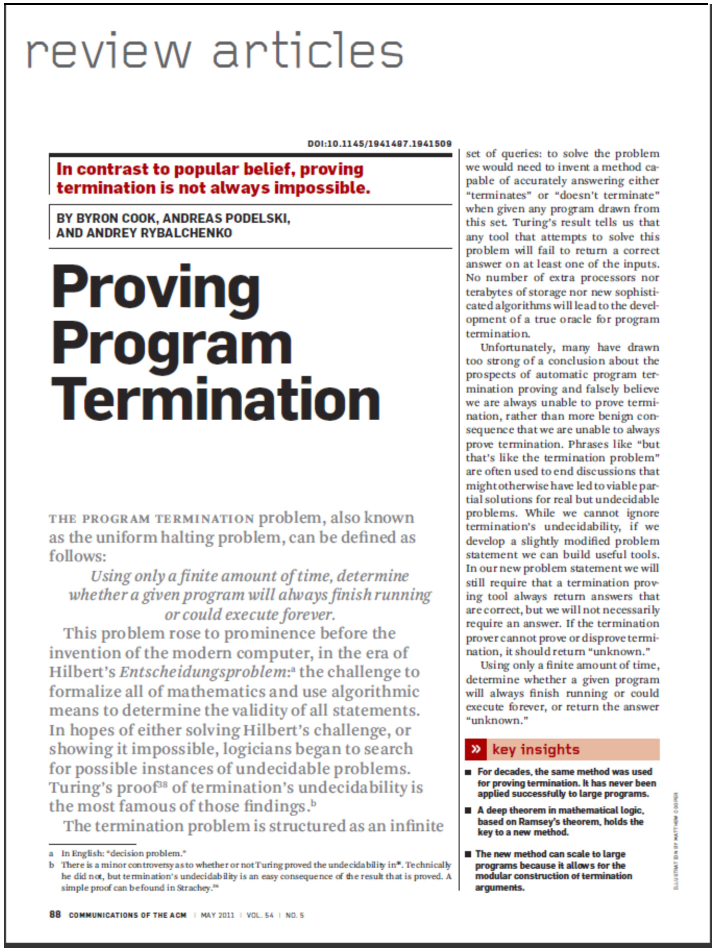
- Abstraction search for safety property verification (*e.g.* SLAM)
- Farkas-based linear rank function synthesis
- Ramsey-based Refinement-based termination proving
- Separation Logic based data structure analysis

For more information

→ http://research.microsoft.com/TERMINATOR

- Research papers
- Recorded technical lectures
- Contact details
- T2 source-code available

→ CACM review article



review articles

DOI:10.1145/1941487.1941509

In contrast to popular belief, proving termination is not always impossible.

BY BYRON COOK, ANDREAS PODELSKI, AND ANDREY RYBALCHENKO

Proving Program Termination

THE PROGRAM TERMINATION problem, also known as the uniform halting problem, can be defined as follows:

> *Using only a finite amount of time, determine whether a given program will always finish running or could execute forever.*

This problem rose to prominence before the invention of the modern computer, in the era of Hilbert's *Entscheidungsproblem*:[a] the challenge to formalize all of mathematics and use algorithmic means to determine the validity of all statements. In hopes of either solving Hilbert's challenge, or showing it impossible, logicians began to search for possible instances of undecidable problems. Turing's proof[38] of termination's undecidability is the most famous of those findings.[b]

The termination problem is structured as an infinite set of queries: to solve the problem we would need to invent a method capable of accurately answering either "terminates" or "doesn't terminate" when given any program drawn from this set. Turing's result tells us that any tool that attempts to solve this problem will fail to return a correct answer on at least one of the inputs. No number of extra processors nor terabytes of storage nor new sophisticated algorithms will lead to the development of a true oracle for program termination.

Unfortunately, many have drawn too strong of a conclusion about the prospects of automatic program termination proving and falsely believe we are always unable to prove termination, rather than more benign consequence that we are unable to always prove termination. Phrases like "but that's like the termination problem" are often used to end discussions that might otherwise have led to viable partial solutions for real but undecidable problems. While we cannot ignore termination's undecidability, if we develop a slightly modified problem statement we can build useful tools. In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer. If the termination prover cannot prove or disprove termination, it should return "unknown."

Using only a finite amount of time, determine whether a given program will always finish running or could execute forever, or return the answer "unknown."

» key insights

- **For decades, the same method was used for proving termination. It has never been applied successfully to large programs.**
- **A deep theorem in mathematical logic, based on Ramsey's theorem, holds the key to a new method.**
- **The new method can scale to large programs because it allows for the modular construction of termination arguments.**

ILLUSTRATION BY MATTHEW COOPER

[a] In English: "decision problem."

[b] There is a minor controversy as to whether or not Turing proved the undecidability in[38]. Technically he did not, but termination's undecidability is an easy consequence of the result that is proved. A simple proof can be found in Strachey.[36]

88 COMMUNICATIONS OF THE ACM | MAY 2011 | VOL. 54 | NO. 5

Outline

- → 10 years work on termination
- → More broadly about me, my career, jobs, other things I work on
- → Some notes on Evergreen & me
- → Questions
  - though please interrupt when things are unclear!

Outline

→ 10 years work on termination

→ More broadly about me, my career, jobs, other things I work on

→ Some notes on Evergreen & me

→ Questions
  - though please interrupt when things are unclear!

Career

- Evergreen: 1990 – 95
  - First year: literature in fall term, then dropped out
  - Second year: Spanish literature/history, then moved to Spain
  - Third year year: physics/chemistry/math
  - Fourth year: computer science
  - Fifth year: logic
  - NOTE: no previous background in subjects. Went to anti-dropout program and did NO technical things in high school
  - Honestly: only a so-so student

- PhD at Oregon Graduate Institute: 1995-2005
  - Weird choice since people were making $$$ in companies
  - Failed to get into PhD program first try (should have applied broadly)
  - I struggled as a PhD student

Career

- Internship, Intel: 1997

- Sales engineer, Prover technology : 2000-2002
  - Hadn’t finished PhD.
  - Had to get job for money reasons
  - Abysmal failure

- Developer, Microsoft Windows OS product group: 2002-2004
  - Still not done with PhD!

- Researcher, Microsoft Research Cambridge (UK), 2004-2014
  - Lucky break based on networking!
  - Big chance.
  - Oh, and I *had* to finish PhD

- University professorship: 2008-Current
  - Based on fame and networks

Job description

→ Goal: innovative new ideas, impact, fame/leadership

→ Work on what I want:

- Termination and temporal logic
- Constraint solving, automated reasoning
- Cancer research
- Art and its use to help facilitate proof
- Gender diversity in computer science
- Connections to programming languages, machine learning, ecology, etc.
- ……………….

Job description practicalities

- → Teaching and supervising PhD students (through university)
- → Postdocs (both at Microsoft Research and University)
- → Interns
- → Thinking/writing/coding
- → Research visits
- → Lectures
- → Conference/journal reviewing
- → Networking
- → Leadership/supervision/management
- → The search for NEW problems

Outline

- 10 years work on termination
- More broadly about me, my career, jobs, other things I work on
- Some notes on Evergreen & me
- Questions
  - though please interrupt when things are unclear!

Outline

- 10 years work on termination
- More broadly about me, my career, jobs, other things I work on
- Some notes on Evergreen & me
- Questions
  - though please interrupt when things are unclear!

Why Evergreen was so great for me

- True & healthy diversity at an unprecedented level
  - All aspects, including economic/privilege diversity
  - Not a privileged person’s view of diversity
  - I understand people in a much deeper way than my colleagues
- No prerequisites (or negotiable ones at least)
- Subsidized and high quality childcare
- Anti-competitive, pro teamwork work environment
- Emphasis on interdisciplinary studies, and on new things
- Generally high quality teaching
- Freedom to fail
- Filled with freaks & dreamers

Outline

→ 10 years work on termination

→ More broadly about me, my career, jobs, other things I work on

→ Some notes on Evergreen & me

→ Questions

- though please interrupt when things are unclear!

Outline

- 10 years work on termination

- More broadly about me, my career, jobs, other things I work on

- Some notes on Evergreen & me

- Questions
  - though please interrupt when things are unclear!