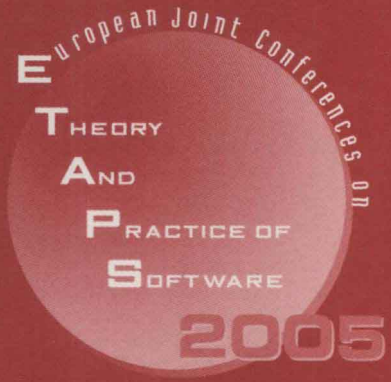
Mooly Sagiv (Ed.)

LNCS 3444

Programming Languages and Systems

14th European Symposium on Programming, ESOP 2005
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2005
Edinburgh, UK, April 2005, Proceedings



Springer

Mooly Sagiv (Ed.)

Programming Languages and Systems

14th European Symposium on Programming, ESOP 2005
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2005
Edinburgh, UK, April 4-8, 2005
Proceedings

Springer

Volume Editor

Mooly Sagiv
Tel Aviv University
School of Computer Science
Tel Aviv 69978, Israel
E-mail: msagiv@post.tau.ac.il

Library of Congress Control Number: 2005922810

CR Subject Classification (1998): D.3, D.1, D.2, F.3, F.4, E.1

ISSN 0302-9743
ISBN-10 3-540-25435-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25435-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11410553 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2005 was the eighth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 17 satellite workshops (AVIS, BYTECODE, CEES, CLASE, CMSB, COCV, FAC, FESCA, FINCO, GCW-DSE, GLPL, LDTA, QAPL, SC, SLAP, TGC, UITP), seven invited lectures (not including those that were specific to the satellite events), and several tutorials. We received over 550 submissions to the five conferences this year, giving acceptance rates below 30% for each one. Congratulations to all the authors who made it to the final program! I hope that most of the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2005 was organized by the School of Informatics of the University of Edinburgh, in cooperation with

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST).

The organizing team comprised:

- Chair: Don Sannella
- Publicity: David Aspinall
- Satellite Events: Massimo Felici

- Secretariat: Dyane Goodchild
- Local Arrangements: Monika-Jeannette Lekuse
- Tutorials: Alberto Momigliano
- Finances: Ian Stark
- Website: Jennifer Tenzer, Daniel Winterstein
- Fundraising: Phil Wadler

ETAPS 2005 received support from the University of Edinburgh.

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Perdita Stevens (Edinburgh, Chair), Luca Aceto (Aalborg and Reykjavík), Rastislav Bodik (Berkeley), Maura Cerioli (Genoa), Evelyn Duesterwald (IBM, USA), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Roberto Gorrieri (Bologna), Reiko Heckel (Paderborn), Holger Hermanns (Saarbrücken), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Kim Larsen (Aalborg), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Hanne Riis Nielson (Copenhagen), Fernando Orejas (Barcelona), Mooly Sagiv (Tel Aviv), Don Sannella (Edinburgh), Vladimiro Sassone (Sussex), Peter Sestoft (Copenhagen), Michel Wermelinger (Lisbon), Igor Walukiewicz (Bordeaux), Andreas Zeller (Saarbrücken), Lenore Zuck (Chicago).

I would like to express my sincere gratitude to all of these people and organizations, the program committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the organizer of ETAPS 2005, Don Sannella. He has been instrumental in the development of ETAPS since its beginning; it is quite beyond the limits of what might be expected that, in addition to all the work he has done as the original ETAPS Steering Committee Chairman and current ETAPS Treasurer, he has been prepared to take on the task of organizing this instance of ETAPS. It gives me particular pleasure to thank him for organizing ETAPS in this wonderful city of Edinburgh in this my first year as ETAPS Steering Committee Chair.

Edinburgh, January 2005

Perdita Stevens
ETAPS Steering Committee Chair

Preface

This volume contains the 29 papers presented at ESOP 2005, the 14th European Symposium on Programming, which took place in Edinburgh, UK, April 6–8, 2005. The ESOP series began in 1986 with the goal of bridging the gap between theory and practice, and the conferences continue to be devoted to explaining fundamental issues in the specification, analysis, and implementation of programming languages and systems.

The volume begins with a summary of an invited contribution by Andrew Myers titled "Programming with Explicit Security Policies," and continues with the 28 papers selected by the Program Committee from 114 submissions. Each submission was reviewed by at least three referees, and papers were selected during a 10-day electronic discussion phase.

I would like to sincerely thank the members of the Program Committee for their thorough reviews and dedicated involvement in the PC discussion. I would also like to thank the subreferees, for their diligent work. Martin Karusseit and Noam Rinetzky helped me with MetaFrame, used as the conference management software. Finally, I would like to thank Anat Lotan-Schwartz for helping me to collect the final papers and prepare these proceedings.

January 2005 Mooly Sagiv

Organization

Program Chair

Mooly Sagiv	Tel Aviv University, Israel

Program Committee

Martín Abadi	University of California at Santa Cruz, USA
Alex Aiken	Stanford University, USA
Bruno Blanchet	École Normale Supérieure, France
Luca Cardelli	Microsoft Research, UK
Patrick Cousot	École Normale Supérieure, France
Oege de Moor	Oxford University, UK
Manuel Fähndrich	Microsoft Research, USA
John Field	IBM, USA
Maurizio Gabbrielli	Università di Bologna, Italy
Chris Hankin	Imperial College London, UK
Manuel Hermenegildo	Universidad Politécnica de Madrid, Spain and University of New Mexico, USA
Xavier Leroy	INRIA Rocquencourt, France
Anders Møller	University of Aarhus, Denmark
Greg Morrisett	Harvard University, USA
David Naumann	Stevens Institute of Technology, USA
Hanne Riis Nielson	IMM, Technical University of Denmark
Peter O'Hearn	University of London, UK
Catuscia Palamidessi	INRIA Futurs Saclay and LIX, France
Thomas Reps	University of Wisconsin-Madison, USA
Martin Rinard	MIT, USA
Andrei Sabelfeld	Chalmers University and Göteborg University, Sweden
David Sangiorgi	Università di Bologna, Italy
David Schmidt	Kansas State University, USA
Scott Stoller	SUNY at Stony Brook, USA

Referees

A. Ahmed	Z. Ariola	N. Benton
E. Albert	A. Askarov	J. Berdine
A. Aldini	F. Barbanera	L. Bettini
J. Aldrich	M. Barnett	G. Bierman

D. Biernacki
C. Bodei
C. Brabrand
K. Bruce
M. Buscemi
N. Busi
B.C. Pierce
C. Calcagno
A. Cavalcanti
K. Chatzikokolakis
S.C. Mu
T. Chothia
M. Codish
A. Corradini
A. Cortesi
V. Cortiero
S. Crafa
F.D. Valenciao
O. Danvy
F. De Boer
P. Degano
G. Delzanno
D. Distefano
D. Dougherty
D. Duggan
R. Ettinger
G. File
C. Flanagan
M. Fluet
R. Focardi
C. Fourned
B. Francisco
J. Garrigue
D. Ghica
R. Giacobazzi
J.C. Godskesen
S. Goldsmith
G. Gonthier
J. Goubault-Larrecq
M.R. Hansen
J. Hickey
T. Hildebrandt
P. Hill
Y. Huenke
J. Hurd
M.J. Jaskelioff
L. Jagadeesan
A. Jeffrey
A. Kennedy
C. Kirkegaard
B. Klin
J. Kodumal
R. Komondoor
S. Krishnamurthi
B. Le Charlier
F. Levi
F. Logozzo
P. Lopez-Garcia
I. Lynagh
R. Majumdar
R. Manevich
M.C. Marinescu
A. Matos
L. Mauborgne
D. Miller
A. Miné
D. Monniaux
M. Naik
U. Neumerkel
F. Nielson
N. Nystrom
R. O'Callahan
L. Ong
L. Paolini
B. Pfitzmann
E. Poll
F. Pottier
M. Proietti
G. Puebla
S. Rajamani
A. Ravara
J. Rehof
J. Reppy
N. Rinetzky
C. Russo
D. Rémy
C. Sacerdoti Cohen
A. Sahai
A. Sasturkar
A. Schmitt
T. Schrijvers
A.S. Christensen
R. Solmi
M. Spivey
F. Spoto
T. Streicher
K. Støvring Sørensen
J.M. Talbot
T. Terauchi
L. Tesei
H. Thielecke
C. Urban
M. Vaziri
T. Veldhuizen
B. Victor
L. Vigano
J. Vouillono
Y. Wang
B. Warinschi
Y. Xie
E. Yahav
E. Zaffanella
S. Zdancewic
T. Zhao
E. Zucca

Table of Contents

Programming with Explicit Security Policies
Andrew C. Myers 1

Trace Partitioning in Abstract Interpretation Based Static Analyzers
Laurent Mauborgne, Xavier Rival 5

The ASTRÉE Analyzer
Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival 21

Interprocedural Herbrand Equalities
Markus Müller-Olm, Helmut Seidl, Bernhard Steffen 31

Analysis of Modular Arithmetic
Markus Müller-Olm, Helmut Seidl 46

Forward Slicing by Conjunctive Partial Deduction and Argument Filtering
Michael Leuschel, Germán Vidal 61

A New Foundation for Control-Dependence and Slicing for Modern Program Structures
Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, John Hatcliff 77

Summaries for While Programs with Recursion
Andreas Podelski, Ina Schaefer, Silke Wagner 94

Determinacy Inference for Logic Programs
Lunjin Lu, Andy King 108

Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis
Oukseh Lee, Hongseok Yang, Kwangkeun Yi 124

A Type Discipline for Authorization Policies
Cédric Fournet, Andrew D. Gordon, Sergio Maffeis 141

Computationally Sound, Automated Proofs for Security Protocols
Véronique Cortier, Bogdan Warinschi 157

Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries (Extended Abstract)
Romain Janvier, Yassine Lakhnech, Laurent Mazaré 172

Analysis of an Electronic Voting Protocol in the Applied Pi Calculus
Steve Kremer, Mark Ryan 186

Streams with a Bottom in Functional Languages
Hideki Tsuiki, Keiji Sugihara 201

Bottom-Up β-Reduction: Uplinks and λ-DAGs
Olin Shivers, Mitchell Wand 217

BI Hyperdoctrines and Higher-Order Separation Logic
Bodil Biering, Lars Birkedal, Noah Torp-Smith 233

Deciding Reachability in Mobile Ambients
Nadia Busi, Gianluigi Zavattaro 248

Denotational Semantics for Abadi and Leino's Logic of Objects
Bernhard Reus, Jan Schwinghammer 263

A Design for a Security-Typed Language with Certificate-Based Declassification
Stephen Tse, Steve Zdancewic 279

Adjoining Declassification and Attack Models by Abstract Interpretation
Roberto Giacobazzi, Isabella Mastroeni 295

Enforcing Resource Bounds via Static Verification of Dynamic Checks
Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, George Necula 311

Asserting Bytecode Safety
Martin Wildmoser, Tobias Nipkow 326

Subtyping First-Class Polymorphic Components
João Costa Seco, Luís Caires 342

Complexity of Subtype Satisfiability over Posets
Joachim Niehren, Tim Priesnitz, Zhendong Su 357

A Type System Equivalent to a Model Checker
Mayur Naik, Jens Palsberg 374

Instant Polymorphic Type Systems for Mobile Process Calculi: Just Add Reduction Rules and Close
Henning Makholm, J.B. Wells 389

Towards a Type System for Analyzing JavaScript Programs
Peter Thiemann 408

Java Jr.: Fully Abstract Trace Semantics for a Core Java Language
Alan Jeffrey, Julian Rathke 423

Author Index 439

Programming with Explicit Security Policies

Andrew C. Myers

Cornell University
andru@cs.cornell.edu

Abstract. Are computing systems trustworthy? To answer this, we need to know three things: what the systems are supposed to do, what they are not supposed to do, and what they actually do. All three are problematic. There is no expressive, practical way to specify what systems must do and must not do. And if we had a specification, it would likely be infeasible to show that existing computing systems satisfy it. The alternative is to design it in from the beginning: accompany programs with explicit, machine-checked security policies, written by programmers as part of program development. Trustworthy systems must safeguard the end-to-end confidentiality, integrity, and availability of information they manipulate. We currently lack both sufficiently expressive specifications for these information security properties, and sufficiently accurate methods for checking them. Fortunately there has been progress on both fronts. First, information security policies can be made more expressive than simple noninterference or access control policies, by adding notions of ownership, declassification, robustness, and erasure. Second, program analysis and transformation can be used to provide strong, automated security assurance, yielding a kind of security by construction. This is an overview of programming with explicit information security policies with an outline of some future challenges.

1 The Need for Explicit Policies

Complex computing systems now automate and integrate a constantly widening sphere of human activities. It is crucial for these systems to be trustworthy: both secure and reliable in the face of failures and malicious attacks. Yet current standard practices in software development offer weak assurance of both security and reliability. To be sure, there has been progress on automatic enforcement of simple safety properties, notably type safety. And this is valuable for protecting systems from code injection attacks such as buffer overruns. But many, perhaps most serious security risks do not rely on violating type safety. Often the exposed interface of a computing system can be used in ways unexpected by the designers. Insiders may be able to misuse the system using their privileges. Users can sometimes learn sensitive information when they should not be able to. These serious vulnerabilities are difficult to identify, analyze, and prevent.

Unfortunately, current practices for software development and verification do not seem to be on a trajectory that leads to trustworthy computing systems. Incremental progress will not lead to this goal; a different approach is needed. We have been exploring a language-based approach to building secure, trustworthy systems, in which programs are annotated with explicit, machine-checked information security policies relating to

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 1–4, 2005.

properties such as the confidentiality and integrity of information. These properties are both crucial to security and difficult to enforce. It is possible to write relatively simple *information flow* policies that usefully capture these aspects of security. These explicit policy annotations then support automatic enforcement through program analysis and transformation.

2 Limitations of Correctness

Of course, the idea of automatic verification has always been appealing—and somewhat elusive. The classic approach of verifying that programs satisfy specifications can be a powerful tool for producing reliable, correct software. However, as a way to show that programs are secure, it has some weaknesses. First, there is the well-known problem that the annotation burden is high. A second, less appreciated problem is that classic specifications with preconditions and postconditions are not enough to understand whether a program is secure. Correctness assertions abstract program behavior; if the abstraction leaves out security-relevant information, the actual program may contain security violations (especially, of confidentiality) invisible at the level of the abstraction. Thus, it's also important to understand not only what programs do but also what they *don't* do. Even if the program has no observable effect beyond what its specification describes, the specification itself may allow the confidential information to be released. A third problem is that correctness assertions don't address the possible presence of malicious users or code, which is particularly problematic for distributed systems.

3 End-to-End Information Security

If classic specification techniques are too heavyweight and yet not expressive enough, what are the alternatives? One possibility is information flow policies, which constrain how information moves through the system. For example, a policy that says some data is confidential means that the system may not let that data flow into locations where it might be viewed insecurely. This kind of policy implicitly controls the use of the data without having to name all the possible destinations, so it can be lightweight yet compatible with abstraction. Furthermore, it applies to the system as a whole, unlike access control policies, which mediate access to particular locations but do not control how information propagates. One can think of information flow policies as an application of the end-to-end principle to the problem of specifying computer security.

Information flow policies can express confidentiality and integrity properties of systems: confidentiality is about controlling where information flows to; integrity is about controlling where information flows from. Integrity is also about whether information is computed correctly, but even just an analysis of integrity as information flow is useful for ensuring that untrustworthy information is not used to update trusted information.

Fundamentally, information flow is about dependency [ABHR99], which makes sense because security cannot be understood without understanding how components depend on one another. The approach to enforcing information flow that has received the most attention is to analyze dependency at compile time using a *security type sys-*

tem [SM03]. The Jif programming language [Mye99], based on Java, is an example of a language with a type system for information security.

The other appealing aspect of information flow policies is that they can be connected to an underlying semantic security condition, noninterference. Noninterference says roughly that the low-security behavior of a system does not change when high-security inputs are changed. This condition (which has many variants) can be expressed in the context of a programming language operational semantics [VSI96], making possible a proof that a security type system constrains the behavior of the system.

4 Whole-System Security and Mutual Distrust

Many of the computing systems for which security is especially important are distributed systems serving many principals, typically distributed at least partly because of security concerns. For example, consider a web shopping service. At the least, it serves customers, who do not entirely trust the service, and the companies selling products, who do not trust the customers or each other. For this reason, the computational resources in use when a customer is shopping are located variously on the customer's computer, on the web service provider, and on the seller's computers. It is important to recognize that these principals have their own individual security requirements; the system as a whole must satisfy those requirements in order for them to participate.

To enforce information security for such a system, it is necessary to know the requirements of each of the principals. The decentralized label model [ML00] is an information flow policy language that introduces a notion of information flow policies owned by principals. For example, in the context of confidentiality, a policy $p_1 : p_2$ means that principal p_1 owns the policy and trusts principal p_2 to read the corresponding information. More generally, p_1 trusts p_2 to enforce the relevant security property on its behalf. This structure makes it possible to express a set of policies on behalf of multiple principals while keeping track of who owns (and can relax) each policy.

For example, suppose we are implementing the game of Battleship with two players, A and B. Player A wants to be able to read his own board but doesn't want B to read it, so the confidentiality label is $\{A : A\}$. For integrity, both principals want to make sure that the board is updated in accordance with the rules of the game, so the integrity label has two owned policies: $\{A : A \wedge B,\ B : A \wedge B\}$, where $A \wedge B$ is a conjunctive principal representing the fact that both A and B must trust the updates to A's board.

5 Security Through Transformation

Secure distributed systems achieve security through a variety of mechanisms, including partitioning code and data (as in the web shopping example), replication, encryption, digital signatures, access control, and capabilities. Analyzing the security of a complex system built in this fashion is currently infeasible.

Recently, we have proposed the use of automatic program transformation as a way to solve this problem [ZZNM02]. Using the security policies in a non-distributed program, the Jif/split compiler automatically partitions its code and data into a distributed system that runs securely on a collection of host machines. The hosts may be trusted to varying

degrees by the participating principals; a partitioning is secure if policies of each principal can be violated only by hosts it trusts. The transformation employs not only partitioning, but also all of the distributed security mechanisms above to generate distributed code for Jif programs. For example, given the labels above, Jif/split can split the code of a Battleship program into a secure distributed system.

6 Conclusions and Future Challenges

The ability to provably enforce end-to-end security policies with lightweight, intuitive annotations is appealing. Using policies to guide automatic transformation into a distributed system is even more powerful, giving a form of security by construction. However, research remains to be done before this approach can be put into widespread use.

Noninterference properties are too restrictive to describe the security of real-world applications. Richer notions of information security are needed: quantitative information flow, policies for limited information release, dynamic security policies [ZM04], and downgrading policies [CM04]. End-to-end analyses are also needed for other security properties, such as availability.

Checking information flow policies with a trusted compiler increases the size of the trusted computed base; techniques for certifying compilation would help.

The power of the secure program transformation technique could be extended by employing more of the tools that researchers on secure protocols have developed; secret sharing and secure function computation are obvious examples.

Strong information security requires analysis of how programs use information. Language techniques are powerful and necessary tools for solving this problem.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proc. 11th ACM Conference on Computer and Communications Security*, October 2004.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[SM03] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[ZM04] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust*, August 2004.

[ZZNM02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

Trace Partitioning in Abstract Interpretation Based Static Analyzers

Laurent Mauborgne and Xavier Rival

DI, École Normale Supérieure, 45 rue d'Ulm,
75 230 Paris cedex 05, France
{Laurent.Mauborgne, Xavier.Rival}@ens.fr

Abstract. When designing a tractable static analysis, one usually needs to approximate the trace semantics. This paper proposes a systematic way of regaining some knowledge about the traces by performing the abstraction over a partition of the set of traces instead of the set itself. This systematic refinement is not only theoretical but tractable: we give automatic procedures to build pertinent partitions of the traces and show the efficiency on an implementation integrated in the ASTRÉE static analyzer, a tool capable of dealing with industrial-size software.

1 Introduction

Usually, concrete program executions can be described with traces; yet, most static analyses abstract them and focus on proving properties of the set of reachable states. For instance, checking the absence of runtime errors in C programs can be done by computing an over-approximation of the reachable states of the program and then checking that none of these states is erroneous. When computing a set of reachable states, any information about the execution order and the concrete flow paths is lost.

However, this *reachable states abstraction* might lead to too harsh an approximation of the program behavior, resulting in a failure of the analyzer to prove the desired property. For instance, let us consider the following program:

$$\textbf{if}(x < 0)\{\, sgn = -1; \,\}$$
$$\textbf{else}\{\, sgn = 1; \,\}$$

Clearly sgn is either equal to 1 or -1 at the end of this piece of code; in particular sgn cannot be equal to 0. As a consequence, dividing by sgn is safe. However, a simple interval analysis [7] would not discover it, since the lub (least upper bound) of the intervals $[-1,-1]$ and $[1,1]$ is the interval $[-1,1]$ and $0 \in [-1,1]$. A simple fix would be to use a more expressive abstract domain. For instance, the disjunctive completion [8] of the interval domain would allow the property to be proved: an abstract value would be a finite union of intervals; hence, the analysis would report x to be in $[-1,-1] \cup [1,1]$ at the end of the above program. Yet, the cost of disjunctive completion is prohibitive. Other domains

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 5–20, 2005.

could be considered as an alternative to disjunctive completion; yet, they may also be costly in practice and their design may be involved. For instance, common relational domains like octagons [16] or polyhedra [11] would not help here, since they describe convex sets of values, so the abstract union operator is an imprecise over-approximation of the concrete union. A reduced product of the domain of intervals with a congruence domain [13] succeeds in proving the property, since -1 and 1 are both in $\{1 + 2 \times k \mid k \in \mathbb{N}\}$. However, a more intuitive way to solve the difficulty would be to relate the value of *sgn* to the way it is computed. Indeed, if the *true* branch of the conditional was executed, then $sgn = -1$; otherwise, $sgn = 1$. This amounts to keeping *some* disjunctions based on control criteria. Each element of the disjunction is related to some property about the history of concrete computations, such as "which branch of the conditional was taken". This approach was first suggested by [17]; yet, it was presented in a rather limited framework and no implementation result was provided. The same idea was already present in the context of data-flow analysis in [14] where the history of computation is traced using an automaton chosen before the analysis.

Choosing of the relevant partitioning (which explicit disjunctions to keep during the static analysis) is a rather difficult and crucial point. In practice, it can be necessary to make this choice at analysis time. Another possibility presented in [1] is to use profiling to determine the partitions, but this approach is relevant in optimization problems only.

The contribution of the paper is both theoretical and practical:

- We introduce a *theoretical framework* for trace partitioning, that can be instantiated in a broad series of cases. More partitioning configurations are supported than in [17] and the framework also supports *dynamic partitioning* (choice of the partitions during the abstract computation);
- We provide detailed practical information about the use of the trace partitioning domain. First, we describe the implementation of the domain; second, we review some strategies for partition creation during the analysis.

All the results presented in the paper are supported by the experience of the design, implementation and practical use of the ASTRÉE static analyzer [2, 15]. This analyzer aims at certifying the absence of run-time errors (and user-defined non-desirable behaviors) in very large synchronous embedded applications such as avionics software. Trace partitioning turned out to be a very important tool to reach that goal; yet, this technique is not specific to the families of software addressed here and can be applied to almost any kind of software.

In Sect. 2, we set up a general theoretical framework for trace partitioning. The main choices for the implementation of the partitioning domain are evoked in Sect. 3; we discuss strategies for partitioning together with some practical examples in Sect. 4. Finally, we conclude in Sect. 5.

2 Theoretical Framework

This section supposes basic knowledge of the abstract interpretation framework [5]. For an introduction, the reader is referred to [9].