CS-Note* 64

Bill J. Ellis and Andrew Ireland

October 22, 2003

Trip Report: Automated Software Engineering 2003 (Montreal)

Contents

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

Automated Software Engineering 2003 took place in Montreal between the 8^{th} (Wednesday) and 10^{th} (Friday) of October 2003. Before the conference started, a series of tutorials were arranged.

1.1 Surviving Montreal

When Montreal is cold, it is cold, and when Montreal is hot, it is hot. It is thus fortunate that we found ourselves somewhere in between in a familiar Scottish Autumn climate. A great way to get about Montreal is via the Metro system. It is possibly cheaper to buy individual tickets if the Metro system is used efficiently and you have a sense of direction. For everyone else, it is probably best to buy a tourist pass, giving unlimited travel on the metro for 2 or 3 days. Note that access to the Montreal underground city is via the Metro access points. You do not need a Metro ticket to get into the shops. Avoid giving credit cards to taxi drivers, avoid the bus and shuttle service from the airport, the service charge is 15%, stay well clear of street performers wielding metal cubes, avoid downtown souvenir shops (go to the port instead), avoid downtown restaurants (go to Chinatown, the port or maybe the pedestrian district), do visit the botanical gardens, also the sphere at Parc Jean-Drapeau, and maybe the giant Montreal casino[1].

2 Tuesday

2.1 Tutorials

2.1.1 Program analysis using abstract interpretation

Guillaume Brat	Kestrel Technology
Arnaud Venet	Kestrel Technology

Guillaume and Arnaud work for Kestrel Technology, a division of NASA Ames Research Centre. They have been using abstract interpretation techniques to reason about NASA software systems. A booklet collecting their slides was provided for the tutorial[2]. The aim of the tutorial was to provide a clear description of abstract interpretation theory and its use in practise.

Abstract interpretation is a static program analysis technique. It aims to automatically discover program properties that hold for all possible execution paths of a program. These properties can then be used to perform various tasks including:

- **Optimising compilers -** Properties found can be used to effectively control optimising compilers.
- **Program understanding -** Properties found can be presented to the programmer to assist program understanding.
- **Semantic pre processing -** Properties found are used to constrain the program prior to performing further analysis, such as model checking.
- **Program verification -** Properties found are used to demonstrate that some property holds within a program, such as exception freedom.

It is noted that program analysis is necessarily incomplete as discovering a sufficient set of properties to check every operation of a program is an undecidable problem. This leads to the recognition of a trade off between:

[1]Imagine Blackpool, crammed into a five story building.
[2]Photocopy available on request! – Bill.

- **Precision -** Taken as the number of program operations that can be decided safe automatically. High precision gives high verification coverage.
- **Scalability -** The practical execution of the program analysis.

Unsurprisingly, greater precision tends to result in greater computational complexity. This complexity is a real problem, limiting the size of programs that can be realistically analysed. A solution to this problem of achieving both precision and scalability is to specialise the analysis for a particular software system. Such specialisation occurs frequently in practise and is key to the viability of abstract interpretation.

The abstract interpretation methodology is divided into a collection of steps.

- **Concrete semantics -** The program is translated into a semantically equivalent flowchart.
- **Collecting semantics -** The collecting semantics is the set of all observable behaviours in the operational semantics. Depending on the collecting semantics selected the set of all observable behaviours will change and hence the scope of program analysis will change.
- **Partitioning -** The nature of partitioning depends on the collecting semantics adopted. Its role is to relate the collecting semantics to the control flow of the program.
- **Abstract semantics -** The abstract semantics provides a sound approximation of the collecting semantics.
- **Abstract domains -** The abstract domain describes the kind of approximation to be collected.

Abstract interpretation involves using the abstract semantics to establish approximate values for the abstract domains. This is achieved by a finite chain of widening operations followed by a finite chain of narrowing operations.

The key to successful abstract interpretation is not the strict implementation of the technique. Rather, various heuristics are selected, typically targeting the properties of a particular program rather than programming language. For example, work by P. Cousot [1] considered variables referred to in the same syntactic statement as being related and exploited this to constrain the abstract interpretation. Further, the application of widening and narrowing requires tuning. Typically a few iterations without widening are employed, followed by a widening and then a narrowing.

3 Wednesday

3.1 Keynote Session

3.1.1 Towards a Science of Software Design

John Mylopoulos	University of Toronto

This talk began by comparing the typical views of applying science in the natural and artificial domains. The observation was that the natural sciences were analytic (analysing the natural thing) and generally well respected. While the artificial sciences were synthetic (analysing the design of an artificial thing) and generally regarded as woolly or cookbook like.

The artificial sciences were labelled a science of design. References were made throughout the talk to Herbert Simon's 1968 lecture *"The Sciences of the Artificial"* (published in 1969). It was noted that current design techniques do not describe alternatives and as such warrant the woolly/cookbook view. To address this the idea of goal oriented requirements engineering was presented, in order to explicitly consider alternatives during the design process. Initial ideas involved the use of temporal logic to express real tangible goals for a system (such as some area of functionality) while vague positive and negative influences where used to describe softer goals (such as implementing a change colours module would have a positive influence on a programs configurability).

A neat observation was made that traditional approaches to software design have tended to come upstream from changes at the lower level. For example, the introduction of structured programming constructs led to the structured programming design and the development of object oriented programming led to object oriented design. The interesting question was asked, what would happen if we let the science of design propagate downstream? What would the resulting programming constructs look like?

Unfortunately, this question was used to turn attention to the TROPOS project. The TROPOS project advocates agent oriented software engineering, apparently once again building design concepts from upstream low level ideas[3]. The TROPOS system supports the goal driven design approach outlined above.

[3]Perhaps agent oriented programming is sufficiently immature allowing for its science of design to have an upstream impact? It seems odd that there would be such a close fit between the natural science of design and agents.

- **Feature oriented techniques?** Can use these to describe sets of alternatives.
- **Multiple stake holders?** Requires different goal hierarchies.
- **Software engineering design?** Hard to get principals independent to object being considered.
- **Runtime design?** Agent could adapt at runtime to cope with changes in environment.
- **So, how describe runtime design alternatives?** Undecided.

3.2 Session 1: Requirements, Interfaces and Groupware

3.2.1 An Empirical Study on Groupware Support for Software Inspection Meetings

Paul Gruenbacher	Johannes Kepler University
Michael Halling	University of Vienna / Johannes Kepler University
Stefan Biffl	Vienna University of Technology

It is generally better to detect faults as early as possible in the development cycle. For this purpose it is common to perform a series of inspection meetings to minimise the number of errors in software requirements documents. This talk described a new tool called GRIP which aims to assist in the inspection planning process. It is built around existing Commercial Off The Shelf (COTS) tools for supporting collaborative processes. The aim is to improve the quality of the inspection meetings, correctly identifying the significant errors.

GRIP allows designers to grade there opinion of various detected defects and have these opinions automatically collected and compared. Meeting schedules are automatically produced to focus attention on the areas where designers have differing opinions. Experiments have been conducted which give strong evidence to support the applicability of the technique.

The argument presented for GRIP was convincing. It makes the process repeatable, recorded and controlled. However, given that the current system employed is essentially pen and paper based, it is not all that surprising that gains were made.

- **Results for different artifacts (code)?** Not sure. The techniques in GRIP target requirements documents. These are different.
- **What is the result?** The result of the inspection meetings is an agreed list of defects.

3.3 Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach

Mehrdad Sabetzadeh	University of Toronto
Steve Easterbrook	University of Toronto

Assuming that specifications are graph based, each interested parties view of a specification can be presented as a graph. The problem is then relating these disjoint viewpoints together to form a single unified specification. Here this problem is tackled using fuzzy logic to identify related views and highlight conflicts.

- **Why!?** Resulting model can be inputted to a multi-valued model checker.

3.3.1 Deriving User Interface Requirements from Densely Interleaved Scientific Computing Applications

Andrew Strelzoff	California University
Linda Petzold	California University

This talk was cancelled.

3.4 Session 2: Testing 1

3.4.1 Fault Localization With Nearest Neighbor Queries

Manos Renieris	Brown University
Steven Reiss	Brown University

A technique for finding software faults was presented. Given a number of correct program runs and a faulty program run the technique generates a localized fault report for the programmer. Using a "nearest criteria", the technqiue selects a successful program run that most closely matches with the faulty program run. Differences between the successful and failing runs are then identified. These differences are filtered in order to narrow the search for faulty code. A variety of filters could be used, the filter advocated involves reporting features that are only detected within the failing run. The technique was evaluated using a set of buggy programs provided by Siemens. The results were quite impressive.

3.4.2 Tool-Assisted Unit Test Selection Based on Operational Violations

Tao Xie	University of Washington
David Notkin	University of Washington

The generation of unit tests is expensive. Tool support for generating unit tests exists. For instance the commercial Java unit testing tool Jtest (ParaSoft) can automatically generate unit tests given a Java class. To be effective, however, specifications must also be provided. In the case of Jtest, specifications take the form of pre- and postconditions and class invariants. In practise, however, such formal specifications are hard to obtain. In this paper an invariant detection program called Daikon [2] was exploited. Daikon works by looking for patterns in program traces. The idea here is to use Daikon to detect program invariants based upon a small test suit. These invariants are then added to the program under test and passed to Jtest. Any tests generated by Jtest that violate the Daikon detected invariants are passed to the human tester for inspection. The tests that do not violate the invariants are fed back into Daikon and a fresh set of invariants are detected. The process is repeated until no detected invariants are violated by the tests generated by Jtest. In some sense the approach can be viewed as a mechanism for boot-strapping the Daikon invariant detection technique[4]. This is an interesting integration of Jtest and Daikon. I am not completely convinced by the results, *i.e.* it still feels like the quality of the initial test suit is crucial to success.

3.4.3 A New Structural Coverage Criteria for Dynamic Detection of Program Invariants

Neelam Gupta	University of Arizona
Zachary V. Heideprie	University of Arizona

Structural coverage criteria can be used to generate test cases, *e.g.* branch coverage, definition-use pair coverage *etc.* A new structural coverage criteria is proposed that provides test cases that support the accurate detection of program invariants. Yes another, Daikon related paper! An initial test suit is generated using a conventional coverage criteria. This test suit is used by Daikon to detect invariants. These invariants are then used with the invariant coverage criteria to generate a second test suit. Daikon is then applied to the second test suit. The intersection between both sets of invariants is then computed and reported. Basically the invariant coverage criteria provides a mechanism for filtering-out false invariants, *i.e.* properties that are true on the initial test runs but which are not invariants of the program under analysis.

3.5 Session 3: Software Architectures and Distributed Systems

3.5.1 DeCo: A Declarative Coordination Framework for Scientific Model Federations

Dean Herington	University of North Carolina
David Stotts	University of North Carolina

A federation is an collection of programs interacting to provide a single program. Such federations are common where a Unix like policy of several small interacting programs are used to solve a particular problem. However, most federations are currently created by writing an informal shell script. DeCo aims to provide a more formal approach for creating federations.

DeCo exploits the Haskell programming language to create types for key architectural elements, such as streams. By using these types a formal type checked federation can be created. An example was presented of a series of FORTRAN programs running to retrieve data about a river. This involved 850 lines of code, 70% of which could be classified as establishing complex data formats.

Plans for the future include trying more examples, to purse distribution (current approach is many processes on same system), provide a clean GUI and XML interface to hide the complexities of the underlying Haskell.

[4] Usually Daikon requires many hundreds of test runs in order to detect useful invariants.

3.5.2 A Programmable Client-Server Model: Robust Extensibility via DSLs

Laurent Reveillere[5]	INRIA/LbRI (NASA Ames)
Charles Consel	INRIA/LbRI (NASA Ames)

The client server model provides a one size fits all service. It is insensitive to client terminal capabilities, network features, client preferences and the needs of a rapidly evolving market. The aim is to address these concerns by parameterising the service on the server side.

To ensure a consistent and useful interface a domain specific language (DSL) is created for each service. Currently a language called Pems has been developed for email and Hadis for the web. The DSL can be implemented in whichever language is appropriate and the interface it offers can be verified to ensure the safety of the server under all client requests.

- **How work in practise? Who downloads the server program?** The server administrator. We do not envisage clients uploading their desired interface onto the server.
- **Proliferation of languages?** Yes. But is this a bad thing? Having an identified interface language for each service is good.
- **Meta-language? Language for these DSLs?** Yes. There does seem to be common themes between different DSLs for different services. This will be explored in future work.

3.5.3 Specification and Synthesis of Hybrid Automata for Physics-Based Animation

Thomas Ellman	Vassar College

Hybrid automata link discrete and continuous applications. These are used to precisely describe events for presentation in a physics based model. The physics are described as a series of equations solved through numeric integration. the transitions between states in the hybrid automata are protected by guards, triggered by particular solutions to the equations.

4 Thursday

4.1 Keynote Session

4.1.1 Many a slip: from research prototype to industry-strength software

Anthony Finkelstein	University College London

This entertaining talk focused on crossing the void ('here be dragons') between research funding stopping and sales of the related finished system starting. The speaker emphasised the importance of targeting the mundane, undramatic experienced problems of the day. Apparently some took this as a suggestion that research should be inspired by the wants of industry. An alternative interpretation is that research should maintain a practical slant.

It was also noted how many systems tend to require a number of iterations for industrial applicability. However, such sustained development is academically discouraged by the reluctance of conferences to publish papers that essentially reduce to 'it does the same thing as version 1, but now its more efficient/easier to use/etc.'. Z was highlighted as a success as a result of its sustained development over many years. Another issues raised is competition squeezing out novel research. Large tool suites offer substantial features and high integration. Further the open source movement provides free systems performing popular tasks. Another obstacle to industrial adoption is that tools must fit within any middle-ware employed by the industry. This leads to different technical constraints from separate organisations and poses a technical problem in practise.

4.2 Session 4: Software Evolution & Maintenance

4.2.1 On the automatic evolution of an OS kernel using temporal logic and AOP

Rickard A. Aberg	Ecole des Mines de Nantes/INRIA
Julia L. Lawall	University of Copenhagen
Mario Sudholt	Ecole des Mines de Nantes/INRIA
Gilles Muller	Ecole des Mines de Nantes/INRIA
Anne-Francoise Le Meur	INRIA/LaBRI

[5] Some of the letter 'e's here should have accents. No attempt is made at encoding the few hundred names in this note into LaTeX.

The aim is to allow application programs to implement kernel process schedulers. To achieve this an operating system kernel is translated to adopt a more flexible configurable process scheduler. This operation is made time consuming and complex as use of process schedulers are not centralised in a kernel.

To automate the kernel translation a set of rewrite rules are created to target the features of a particular kernel, in the example presented Linux. Temporal logic is used to describe paths in a control flow graph. The rewrite rules target patterns of code and contain temporal logic guards to perform transformations at the appropriate points.

The approach is not applicable to general code. Rather it aims to automatically translate kernels within the same version strain. By sticking to the same version strain the changes seen are less significant and hence less likely to break the rewrite rules.

- **Rewrite rule development?** Learn by doing.
- **Document the rules and their development?** Not yet. But, yes, this might be done.
- **Correctness?** Not always correct. Works within kernel versions.

4.2.2 Unspeculation

Noah Snavely	University of Arizona
Saumya Debray	University of Arizona
Gregory Andrews	University of Arizona

Modern computer architectures apply novel features to improve performance. In particular, they employ speculative load and speculative checks. Such binary systems can not be analysed using existing tools that are not configured to support such novel features. Thus the aim is to try and normalise speculation without affecting program semantics.

The solution involves hoisting speculative loads up the data stream, taking into account any branches accordingly. The technique has successfully removed about 75% of all speculation, resulting in an efficiency boost.

This seems an unusual activity. The compiler might be modified to describe its actions rather than get involved in reverse engineering. It was noted that this is not applicable where the source code is not available or the binary is subjected to binary paths. Essentially if you want to reason at the binary level about systems that involve speculation, then this will be helpful[6].

4.2.3 SPQR: Flexible Automated Design Pattern Extraction From Source Code

Jason McC. Smith	University of North Carolina
David Stotts	University of North Carolina

The aim is to automatically discover known patterns in source code. Such patterns can assist comprehension during maintenance.

The technique works by describing the building blocks of patterns and using a theorem prover to identify combinations of these building blocks as recognisable patterns. Interestingly, the speaker noted that out of the Gang Of Four[7] patterns the only pattern not supported by this technique is the iterator pattern.

- **Only first order theorem prover?** Sufficient for now.

4.2.4 Automatically Inferring Concern Code from Program Investigation Activities

Martin P. Robillard	University of British Columbia
Gail C. Murphy	University of British Columbia

Concern is apparently another software engineering word to add to your vocabulary. It is a fairly abstract notion of a concept, goal or purpose seen within a software system. Here the aim is to automatically find and document concerns by monitoring a programmer in tackling some software update.

The idea is that the programmer will predominantly visit the areas of code relevant to there problem or concern. By recording the programmers actions and applying some statistical analysis it is possible to identify areas of the code that are most likely related to the same concern.

- **Impact of different programmers?** Accept that different programmers may search in different ways. However, the eventual concern of focus should be the same. It is possible to delay recording programmers actions until a lead has been found, minimising the initial noise in getting started.

[6]But it seems there is bigger problems if we really want to be doing this...

[7]A slang reference to an apparently famous book on Design Patterns [3].

- **Different concerns in same session?** Can finding different concerns as search will branch into separate concern areas.

4.3 Posters 1

A Model-Driven Approach to the Non-Functional Analysis of Software Architectures,
James Skene, Wolfgang Emmerich

Extending Diagnosis to Debug Programs With Exceptions,
Wolfgang Mayer, Markus Stumptner

Generating Design Pattern Detectors from Pattern Specifications,
Dirk Heuzeroth, Welf Lwe, Stefan Mandel

Predicting Fault Prone Modules by the Dempster-Shafer Belief
Networks, Lan Guo, Bojan Cukic, Harshinder Singh

Automatic Fault Localization and Behavior Verification for Physical System Simulation Models,
Peter Bunus, Peter Fritzson

Automatic Generation of Content Management Systems from EER-Based Specifications,
Sebastiano Vigna

Automated Requirements-based Generation of Test Cases for Product Families,
Clmentine Nebut, Simon Pickin, Yves Le Traon, Jean-Marc Jezequel

XRay Views: Understanding the Internals of Classes,
Gabriela Arevalo, Stephane Ducasse, Oscar Nierstrasz

Visual Constraint Diagrams: Runtime Conformance Checking of UML Object Models versus Implementations,
Christopher Turner, T.C. Nicholas Graham, Christopher Wolfe, Julian Ball, David Holman, Arthur G. Ryman

A Pragmatic Study of Binary Class Relationships,
Yann-Gael Gueheneuc, Herve Albin-Amiot

The Features Signatures of Evolving Programs,
Daniel Licata, Christopher D. Harris, Shriram Krishnamurthi

Theoretical Foundations of Updating Systems,
Hannah Harris, Mark Ryan

An Incremental Approach to Task-Specific Information Delivery
in SE Processes, Harald Holz

4.4 Demo 1

AGENDA tool set,
eng Yuetang

GOTECH:Aspectizing Server-Side Distribution,
Eli Tilevich

Model-Based Test Generator BZ-Testing-Tools,
Bruno Legeard

GReAT: A Metamodel Based Model Transformation Language,
Aditya Agrawal

UCEd: a tool for Use Cases based requirements acquisition,
Stephane S. Some

4.5 Session 5: Model Checking

4.5.1 Debugging Overconstrained Declarative Models Using Unsatisfiable Cores

Ilya Shlyakhter	Massachusetts Institute of Technology
Robert Seater	Massachusetts Institute of Technology
Manu Sridharan	University of California
Mana Taghdiri	Massachusetts Institute of Technology
Daniel Jackson	Massachusetts Institute of Technology

Declarative models expressed using conjunction and negation may become overconstrained. That is, aspects of the model may be subsumed by others. A tool, built around the new ability of SAT solvers to find an unsatisfiable core of a satisfiable set of clauses, can automatically identify the points where a model is overconstrained. This work builds directly upon Daniel Jackson's Alloy system. Note: This paper won the best paper award.

4.5.2 Model Checking Parallel Breadth-First Search LTL Model-Checking

Jakub Chaloupka	Masaryk University Brno
Jiri Barnat	Masaryk University Brno
Lubos Brim	Masaryk University Brno

The title is quite descriptive. This involves using a a parallel breadth first search to finding counter examples in linear temporal logic. The need for parallelism was to make the process viable rather than improve performance. Note that breadth first search is more naturally parallelisable and leads to shorter counter examples being found.

4.5.3 Automated Environment Generation for Software Model Checking

Oksana Tkachuk	Kansas State University
Matthew Dwyer	Kansas State University
Corina Pasareanu	NASA Ames

Software model checking is not applicable for large programs. Thus there is a desire to model subsets of a large program instead. Here a technique is presented to extract units from Java programs, automatically creating drivers and stubs to allow these units to be analysed as a single entity. The tool is called Bandera.

4.6 Session 6: Software Components

4.6.1 Aspectizing Server-Side Distribution

Eli Tilevich	Georgia Institute Of Technology
Stephan Urbanski	Georgia Institute Of Technology
Yannis Smaragdakis	Georgia Institute Of Technology
Marc Fleury	The JBoss Group

This talk described a means for easier distributed software development in Java. This involved extending Java with annotations that are used to generate various low level code that would otherwise be written by hand. The full system involves various Java components such as Enterprise Java Beans, AspectJ, XDoclet and NRMI.

The strategy presented is viable. However, it does all seem a little complicated. More familiarity with Java might help in accepting the strategy.

4.6.2 Automating Component Adaptation for Reuse

Brandon Morel	University of Kansas
Perry Alexander	University of Kansas

This talk described the Spartacus system for software reuse. In theory software reuse should reduce development time. However, effective software reuse is difficult and consequently adoption in software engineering has been slow.

Spartacus accepts a specification for a software component and then searches for a suitable component in its library. The clever part involves the automatic configuration of existing software components to create the component desired. This is done in two ways: parallel or serial. For example if wanting a double adder, two single adders could be serialised. Or if wanting an adder or subtractor depending on a value, an adder and subtractor component can be put in parallel, synthesising a suitable interface. Central to the technique is establishing matching specifications by dispatching and solving problems within a theorem prover.

This is a neat idea, building larger software components from mixed software components. However, the search for components can be expensive and there is a need for specifications. Currently the system has worked successfully with small examples, demonstrating scalability would make these more exciting[8].

- **Not modify code? But modify architecture?** Yes.
- **If no match found?** Return partial match. Or fail.
- **Why fail?** Often because of limited theorem prover.

4.6.3 Model-based Verification of Web Service Compositions

Howard Foster	Imperial Collage London
Sebastian Uchitel	Imperial Collage London
Jeff Magee	Imperial Collage London
Jeff Krame	Imperial Collage London

An approach to verify web service implementations against service expectations. This analysis is built around the Business Process Execution Language for Web Services Specification (the BPEL4WS Specification). Model checking is used to identify potential troublesome sequences of interactions.

4.7 Posters 2

```
Test Suite Design for Code Generation Tools,
Ingo Stuermer, Mirko Conrad

Architecture Style-Based Calculi for Non-functional Properties,
David Wile

Certifying Measurement Unit Safety Policy,
Grigore Rosu, Feng Chen

Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search,
Eugenia Diaz, Javier Tuya, Raquel Blanco

Model Checking Software Requirement Specifications Using Domain Reduction Abstraction,
Yunja Choi, Mats Heimdahl

An Approach for Tracing and Understanding Asynchronous Systems,
Scott Hendrickson, Eric Dashofy, Adrita Bhor, Richard Taylor, Santiago Li, Nghi Nguyen

Refactoring C with Conditional Compilation,
Alejandra Garrido, Ralph Johnson

A Java Component Model for Evolving Software Systems,
Moacir Silva, Paulo Guerra, Cecilia Rubira

Depiction and Playout of Multi-threaded Program Executions,
Abhik Roychoudhury

Automating the Analysis of Planetary Nebulae Images,
Bernd Fischer, Johann Schumann

Automation for Exception Freedom Proofs,
Bill Ellis, Andrew Ireland

Overview of Open-model Based Validation with Partial Information,
Robert Hall, Andrea Zisman
```

The poster presented by Bernd Fischer described a system which relates to our work. The system is written in Prolog and automatically generates C code from a number of source code templates. The resulting C program is used to analyse and categorise images from the Hubble telescope.

The template code includes annotated properties. Using the Floyd/Hoare approach verification conditions are generated to demonstrate various specific properties. These include proving that variables are assigned to before use, that array bounds are not violated and proving the symmetry of matrices at particular points in the program.

We noticed the similarity between aspects of this verification and our work with SPARK. By modifying the back end of the Prolog program to generate SPARK rather than C it would be possible to apply the SPARK approach,

[8] I still think this is a neat idea. Could it be combined with program construction? Or focus only on reusing constructive proofs?

and our tools, to the same verification problems. Bernd was enthusiastic about a collaboration, probably involving some intern work at NASA to perform the required changes, to explore the

I think our poster presentation was well received. Some points made:

- **Is that possible?** This was asked by someone thinking of Java. Proving exception freedom in an large object oriented language that changes semantics at runtime (dynamic dispatch) is very difficult. SPARK is well constrained and makes this task tractable.

- **How do you prove arrays are initialised?** This was part of a longer discussion about a weakness of the SPARK approach to proving that variables are initialised before there use. SPARK employs decision procedure flow analysis to prove this property. However, it is computationally restrictive to consider each element of an array individually. The solution used is to consider arrays as a single unit, creating warnings for well formed initialisation routines, such as initialising an array within a loop. The solution in SPARK is to avoid such initialisations and rely on Ada statements that assign the whole array directly.

  However, it would be possible to create VCs to prove this property by modifying the Floyd/Hoare rules accordingly. This is the approach taken by Bernd and Steve German in the Runcheck system [4]. It has been thought that the SPARK approach is neater as it solves the majority of the problem automatically via a decision procedure. However, this affects the ability for diversity of proof. Further, as noted by Bernd, the theorem proving required to prove this property are fairly light.

4.8 Demos 2

```
PRIME: A Tool for Task-Specific Knowledge Discovery,
Harald Holz

A Pragmatic Study of Binary Class Relationships,
Herve Albin-Amiot, Yann-Gael Gueheneuc

The Alloy Analyzer,
Ilya A Shlyakhter

J-Orchestra: Automatic Java Application Partitioning,
Eli Tilevich
```

4.9 Session 7: Testing II

4.9.1 What Test Oracle Should I Use for Effective Testing?

Atif Memon	University of Maryland
Ishan Banerjee	University of Maryland
Adithya Nagarajan	University of Maryland

This describes an approach to the testing of graphical user interfaces. Test cases are described as input output vectors. The vectors record the states of various graphical components, such as the location of clicks or the status of radio buttons. These test cases are then executed on the system.

Different levels of testing are available by recording the states of active windows, visible windows, all windows or just final state. Evidence suggested that these catch different errors. A large test case took 1.5 months to execute (fully automatically).

- **Switch to user level actions?** Record 'Open file' rather than a series of clicks? Consider.

- **Compare faults to performance analysis?** Something to do.

4.9.2 A Type System for Statically Detecting Spreadsheets Errors

Yanif Ahmad	Brown University
Tudor Antoniu	Brown University
Sharon Goldwater	Brown University
Shriram Krishnamurthi	Brown University

This energetic talk described the use of a type system to statically find errors in spreadsheets. The type system exploited the conventional layout of spreadsheets, using the header and column axis to establish a type of a particular element. These types are then used to check that element calculations are working with the same types, reporting mismatches as errors. An implementation exists for Microsoft Excel.

4.9.3 Testing Database Transaction Concurrency

Yuetang Deng	Polytechnic University Brooklyn
Phyllis Frankl	Polytechnic University Brooklyn
Zhongqiang Chen	Polytechnic University Brooklyn

Want to test that items are read and written correct inside a database. Errors may occur as a result of a buggy database transaction mechanism.

4.10 Conference Close

The conference was closed, in French, with *nous sommes finis*, translated as 'we are finished' or 'we are dead'. Although a tiring few days, the consensus was that it had been a good conference.

A short advert for Automated Software Engineering 2004 was made. This will be held in Linz, Austria[9]. All of the conference will take place within the Johannes Kepler University. They encouraged short paper submissions this year to try for full papers next year.

References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, 2002.

[2] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariant to support program evolution. *IEEE Trans. on Software Engineering*, 27(2):1–25, February 2001.

[3] Erich Gamma, Richard Helm, and Ralph Johnson und John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986. ISBN: 0–201–63361–2.

[4] S.M. German. Automating proof of the absence of common runtime errors. In *POPL-5*. ACM, 1978.

[9] In request to some mumblings the speaker gave details on the location of Austria, confirming that it is not an island nor home to Kangaroos.