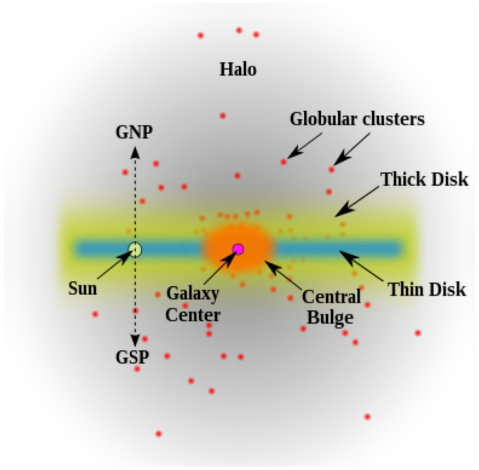
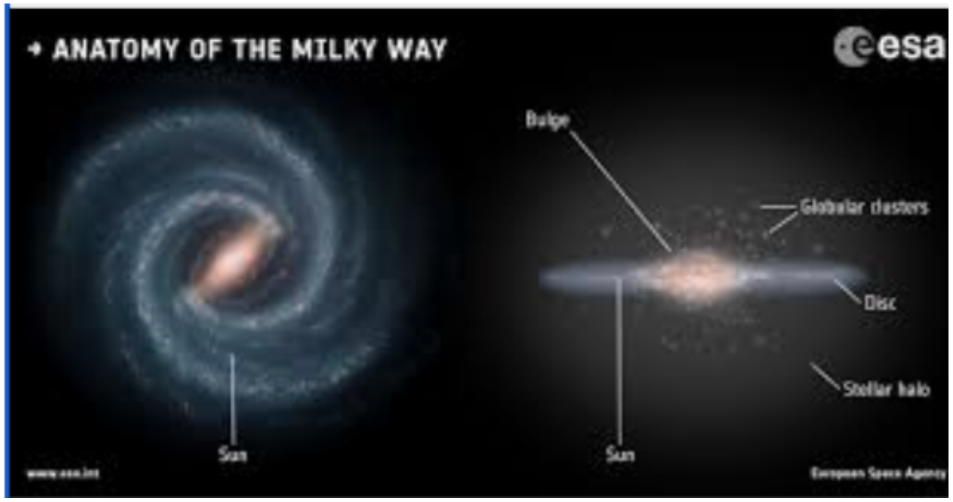
Refinement Checking
Implementation Correctness
Simulation Relations
Homomorphisms
(whatever)

Ravi Hosabettu (Univ. of Utah) [thanks!]
Ganesh Gopalakrishnan (Univ. of Utah)
Mandayam Srivas (SRI International) [thanks!]

Where are we today?

Where are we today? (fun to follow the JWST news…to feel suddenly good!). https://jwst.nasa.gov/ . Reading about their "salt lenses" was fun!





(more down-to-earth things now) Talk Organization

- Motivation for studying implementation correctness in all its guises..
- Completion Functions Approach
- Key contribution of the talk
- Detailed illustration
- Conclusions

Motivations

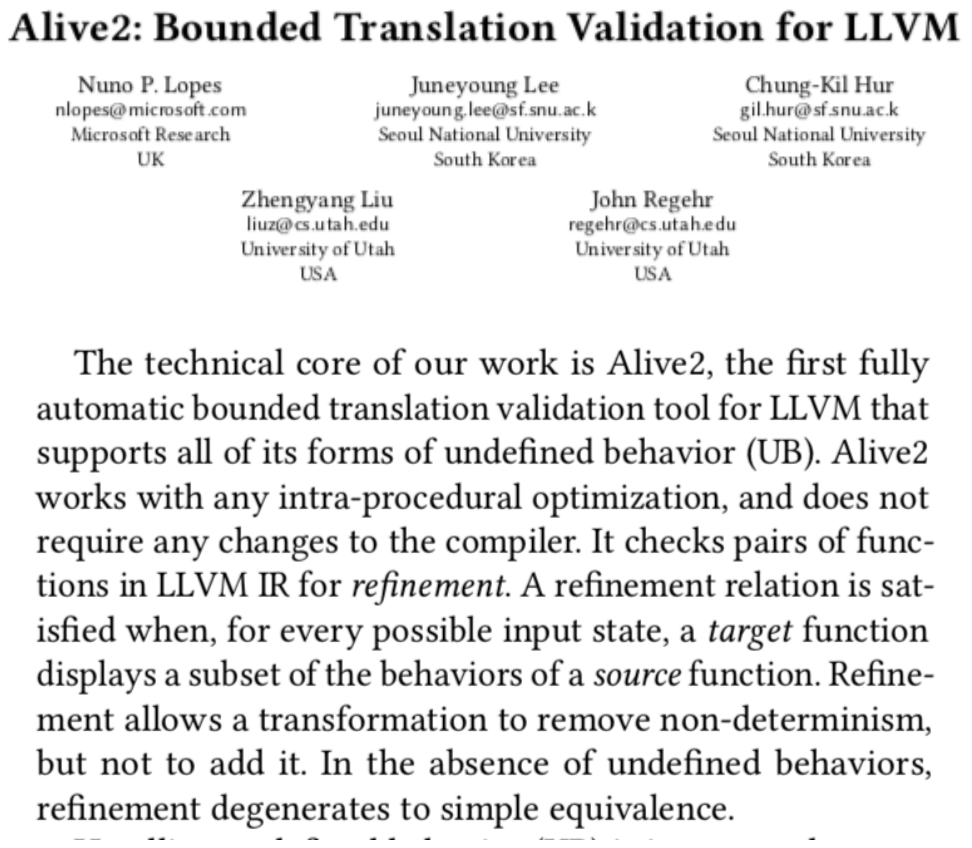
- Often, correctness is adherence to a more abstract specification
- This notion recurs so much in formal verification that we have to be armed with a collection of techniques
- In this talk I'll present three methods:
  - Completion functions
  - Aggregation abstractions
  - Predicate abstraction

Motivations

- The generality of these methods are (simply put) stunning!
  - Two examples from my readings in just the past week now follow

Alive2 (pldi'21) correctness

- See how the paper introduces the problem



Alive2: Bounded Translation Validation for LLVM

Nuno P. Lopes
nlopes@microsoft.com
Microsoft Research
UK

Juneyoung Lee
juneyoung.lee@sf.snu.ac.k
Seoul National University
South Korea

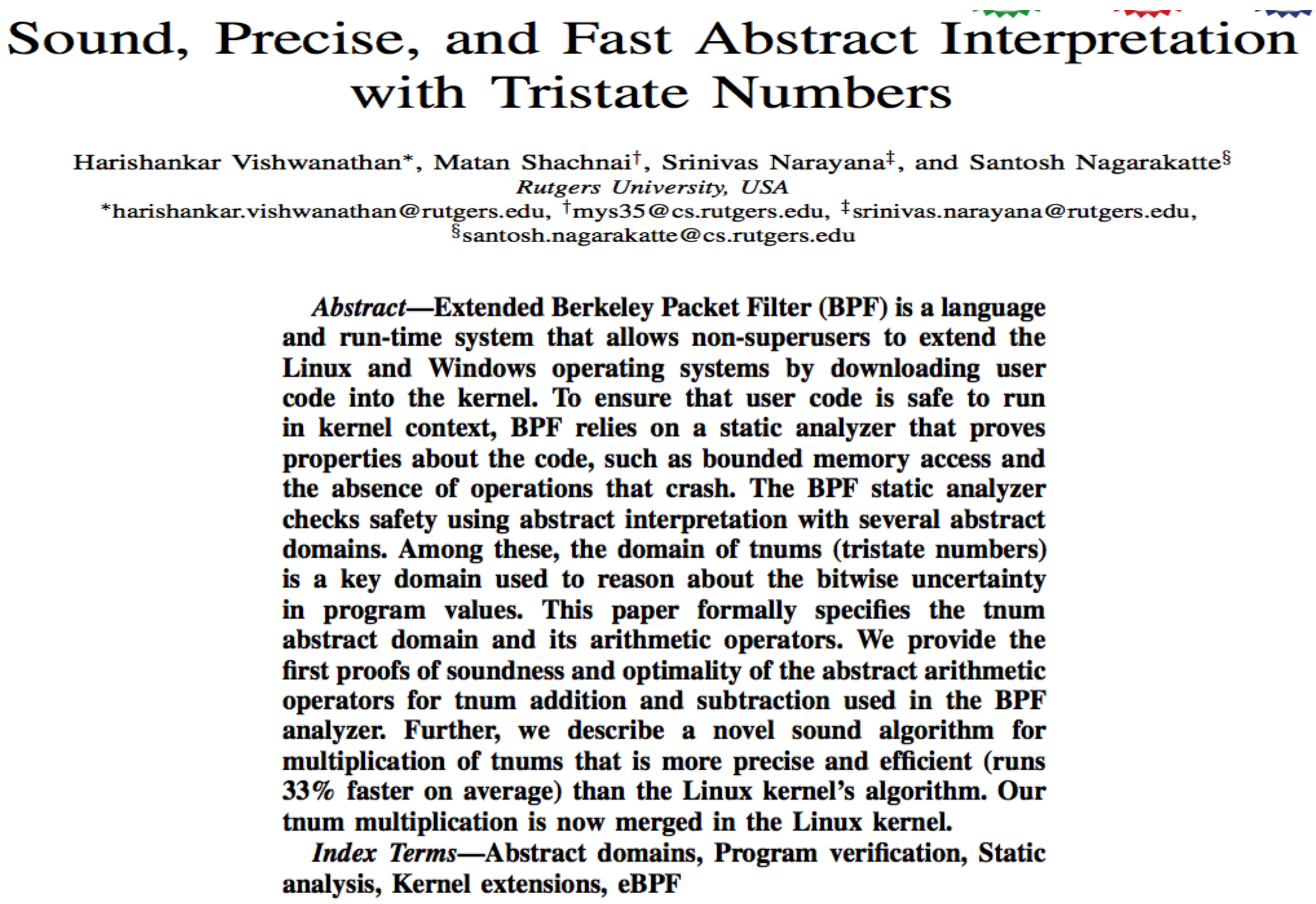
Chung-Kil Hur
gil.hur@sf.snu.ac.k
Seoul National University
South Korea

Zhengyang Liu
liuz@cs.utah.edu
University of Utah
USA

John Regehr
regehr@cs.utah.edu
University of Utah
USA

The technical core of our work is Alive2, the first fully automatic bounded translation validation tool for LLVM that supports all of its forms of undefined behavior (UB). Alive2 works with any intra-procedural optimization, and does not require any changes to the compiler. It checks pairs of functions in LLVM IR for *refinement*. A refinement relation is satisfied when, for every possible input state, a *target* function displays a subset of the behaviors of a *source* function. Refinement allows a transformation to remove non-determinism, but not to add it. In the absence of undefined behaviors, refinement degenerates to simple equivalence.

Tristate Values (cgo’22 best paper)



Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers

Harishankar Vishwanathan*, Matan Shachnai†, Srinivas Narayana‡, and Santosh Nagarakatte§
Rutgers University, USA
*harishankar.vishwanathan@rutgers.edu, †mys35@cs.rutgers.edu, ‡srinivas.narayana@rutgers.edu,
§santosh.nagarakatte@cs.rutgers.edu

***Abstract*—Extended Berkeley Packet Filter (BPF) is a language and run-time system that allows non-superusers to extend the Linux and Windows operating systems by downloading user code into the kernel. To ensure that user code is safe to run in kernel context, BPF relies on a static analyzer that proves properties about the code, such as bounded memory access and the absence of operations that crash. The BPF static analyzer checks safety using abstract interpretation with several abstract domains. Among these, the domain of tnums (tristate numbers) is a key domain used to reason about the bitwise uncertainty in program values. This paper formally specifies the tnum abstract domain and its arithmetic operators. We provide the first proofs of soundness and optimality of the abstract arithmetic operators for tnum addition and subtraction used in the BPF analyzer. Further, we describe a novel sound algorithm for multiplication of tnums that is more precise and efficient (runs 33% faster on average) than the Linux kernel’s algorithm. Our tnum multiplication is now merged in the Linux kernel.**

***Index Terms*—Abstract domains, Program verification, Static analysis, Kernel extensions, eBPF**

A. *Primer on Abstract Interpretation*

Abstract interpretation [38] is a form of static analysis that captures the values of program variables in all executions of the program. Abstract interpretation employs *abstract values* and *abstract operators*. Abstract values are drawn from an *abstract domain*, each element of which is a concise representation of a set of concrete values that a variable may take across executions. For example, an abstract value from the interval abstract domain [47] $\{[a, b] \mid a, b \in \mathbb{Z}, a \leq b\}$ models the set of all concrete integer values (*i.e.*, $x \in \mathbb{Z}$) such that $a \leq x \leq b$.

Abstraction and Concretization functions. An *abstraction function* α takes a concrete set and produces an abstract value, while a *concretization function* γ produces a concrete set from an abstract value. For example, the abstraction of the set $\{2, 4, 5\}$ in the interval domain is $[2, 5]$, which produces the set $\{2, 3, 4, 5\}$ when concretized.

A value $a \in \mathbb{A}$ is a *sound* abstraction of a value $c \in \mathbb{C}$ if and only if $c \sqsubseteq_{\mathbb{C}} \gamma(a)$. Moreover, a is an *exact* abstraction of c if $c = \gamma(a)$. Abstractions are often not exact, over-approximating the concrete set to permit concise representation and efficient analysis in the abstract domain. For example, the interval $[2, 5]$ is a sound but inexact abstraction of the set $\{2, 4, 5\}$.

Galois connection. Pairs of abstraction and concretization functions (α, γ) are said to form a Galois connection if [45]:
1) α is monotonic, *i.e.*, $x \sqsubseteq_{\mathbb{C}} y \implies \alpha(x) \sqsubseteq_{\mathbb{A}} \alpha(y)$
2) γ is monotonic, $a \sqsubseteq_{\mathbb{C}} b \implies \gamma(a) \sqsubseteq_{\mathbb{A}} \gamma(b)$
3) $\gamma \circ \alpha$ is extensive, *i.e.*, $\forall c \in \mathbb{C} : c \sqsubseteq_{\mathbb{C}} \gamma(\alpha(c))$
4) $\alpha \circ \gamma$ is reductive, *i.e.*, $\forall a \in \mathbb{A} : \alpha(\gamma(a)) \sqsubseteq_{\mathbb{A}} a$

The Galois connection is denoted as $(\mathbb{C}, \sqsubseteq_{\mathbb{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \sqsubseteq_{\mathbb{A}})$. The existence of a Galois connection enables reasoning about the soundness and the precision of any abstract operator.

An abstract operator $g : \mathbb{A} \rightarrow \mathbb{A}$ is a *sound* abstraction of a concrete operator $f : \mathbb{C} \rightarrow \mathbb{C}$ if $\forall a \in \mathbb{A} : f(\gamma(a)) \sqsubseteq_{\mathbb{C}} \gamma(g(a))$. Further, g is *exact* if $\forall a \in A : f(\gamma(a)) = \gamma(g(a))$.

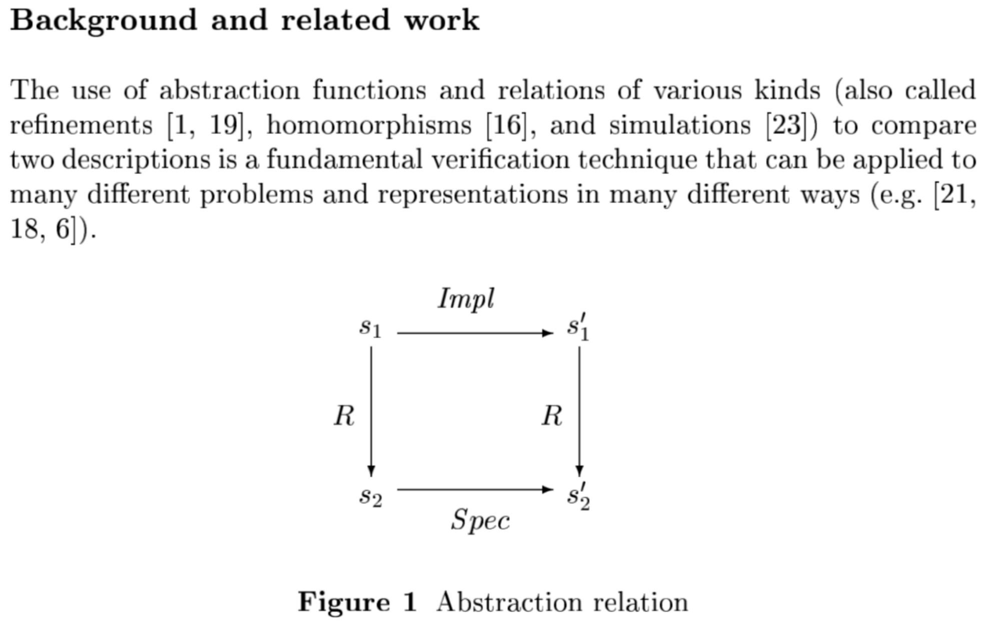
Optimality. Suppose $(\mathbb{C}, \sqsubseteq_{\mathbb{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \sqsubseteq_{\mathbb{A}})$ is a Galois connection. Given a concrete operator $f : \mathbb{C} \rightarrow \mathbb{C}$, the abstract operator $\alpha \circ f \circ \gamma$ is the smallest sound abstraction of f: that is, for any sound abstraction $g : \mathbb{A} \rightarrow \mathbb{A}$ of f, we have $\forall a \in \mathbb{A} : \alpha(f(\gamma(a))) \sqsubseteq_{\mathbb{A}} g(a)$. We call $\alpha \circ f \circ \gamma$ the *optimal*, or maximally precise abstraction, of f.

Motivations

- In this talk I’ll present three methods:
  - Completion functions
  - Aggregation abstractions
  - Predicate abstraction

How can we model processor pipelining for FV?

- The old idea for non-pipelined processors was the "standard commute diagram";



Background and related work

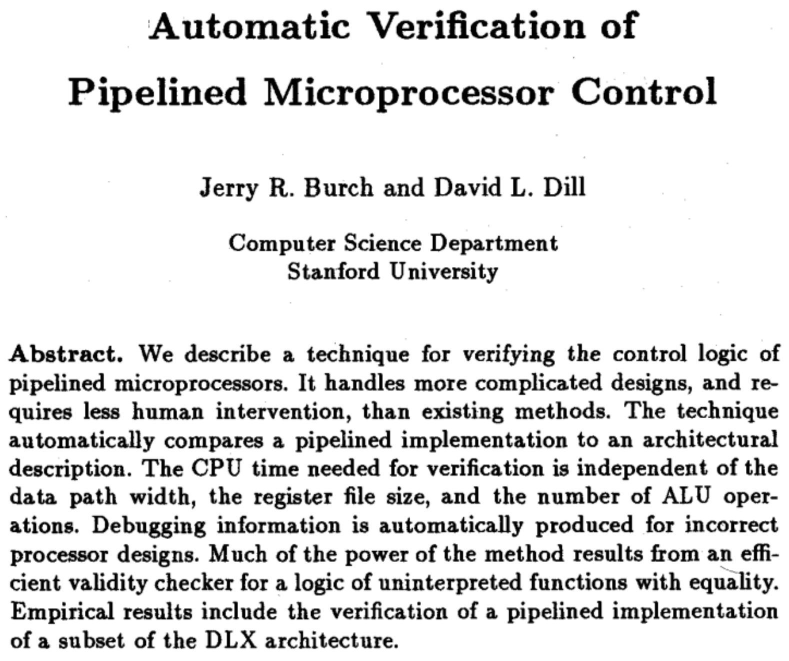
The use of abstraction functions and relations of various kinds (also called refinements [1, 19], homomorphisms [16], and simulations [23]) to compare two descriptions is a fundamental verification technique that can be applied to many different problems and representations in many different ways (e.g. [21, 18, 6]).

$$\begin{tikzcd}
s_1 \arrow[r, "Impl"] \arrow[d, "R"'] & s'_1 \arrow[d, "R"'] \\
s_2 \arrow[r, "Spec"'] & s'_2
\end{tikzcd}$$

Figure 1 Abstraction relation

- but with pipelining, the operation boundaries are not "crisp" and rigid … things keep oozing along…

Burch and Dill 1994 introduced the idea of "flushing" – a breakthrough!


Automatic Verification of Pipelined Microprocessor Control

Jerry R. Burch and David L. Dill

Computer Science Department
Stanford University

Abstract. We describe a technique for verifying the control logic of pipelined microprocessors. It handles more complicated designs, and requires less human intervention, than existing methods. The technique automatically compares a pipelined implementation to an architectural description. The CPU time needed for verification is independent of the data path width, the register file size, and the number of ALU operations. Debugging information is automatically produced for incorrect processor designs. Much of the power of the method results from an efficient validity checker for a logic of uninterpreted functions with equality. Empirical results include the verification of a pipelined implementation of a subset of the DLX architecture.

The new idea: let the implementation be in a 'half-baked' state. Flush it and project it; then claim as induction hypothesis that it matches the spec state. Then let the implementation process one more instruction. Now flush it and project it. As induction step, claim that the "diagram now commutes!"

$$\begin{tikzcd}
\boxed{\text{Old Impl State}} \arrow[r, "F_{\mathrm{Impl}}(\cdot, I_{\mathrm{Stall}})"] \arrow[d, "F_{\mathrm{Impl}}(\cdot, I)"] & \cdots \arrow[r, "F_{\mathrm{Impl}}(\cdot, I_{\mathrm{Stall}})"] & \boxed{\text{Flushed Old Impl State}} \arrow[r, "proj"] & \boxed{\text{Old Spec State}} \arrow[d, "F_{\mathrm{Spec}}(\cdot, I)"'] \\
\boxed{\text{New Impl State}} \arrow[r, "F_{\mathrm{Impl}}(\cdot, I_{\mathrm{Stall}})"'] & \cdots \arrow[r, "F_{\mathrm{Impl}}(\cdot, I_{\mathrm{Stall}})"'] & \boxed{\text{Flushed New Impl State}} \arrow[r, "proj"'] & \boxed{\text{New Spec State}}
\end{tikzcd}$$

Fig. 1. Commutative diagram for showing our correctness criteria.

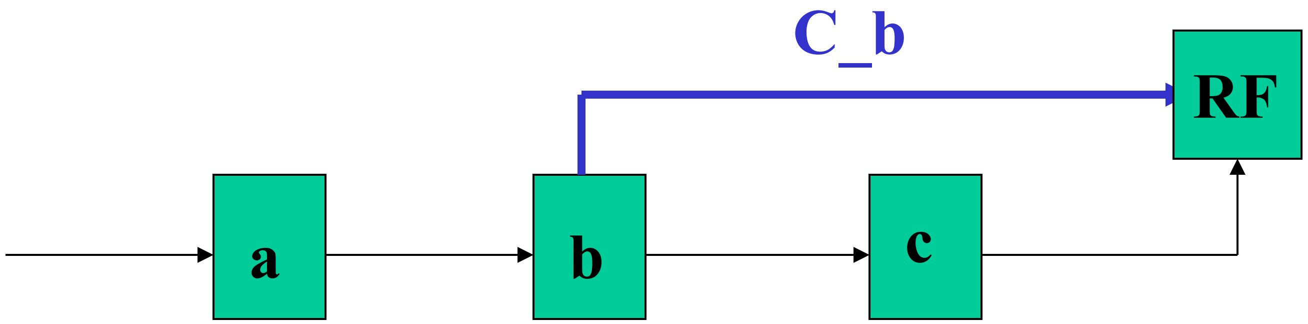
The new idea: let the implementation be in a ‘half-baked’ state. Flush it and project it; then claim as induction hypothesis that it matches the spec state. Then let the implementation process one more instruction. Now flush it and project it. As induction step, claim that the ”diagram now commutes!”

But ”flushing” was not incremental

- Choked theorem provers
- Did not provide a stage-wise proof
  - Bug-localization
- Enter Ravi Hosabettu’s completion functions!
  - A “ladder commutes-diagram”!!

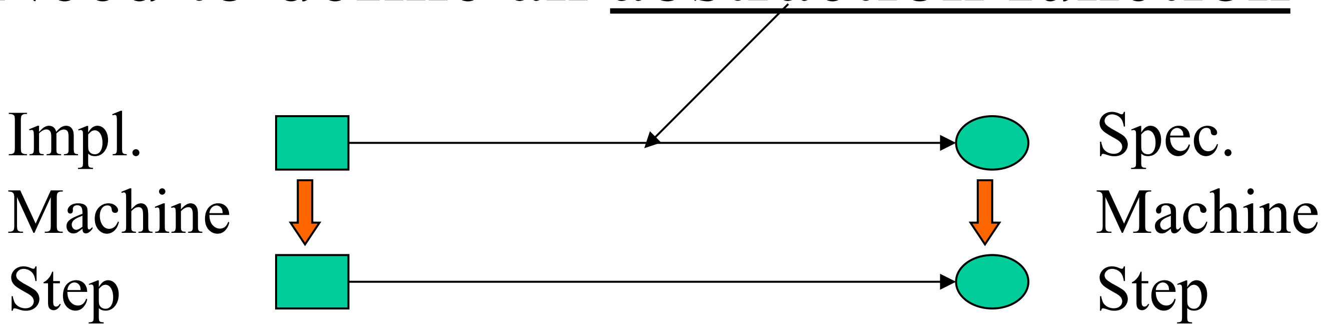
What are Completion Functions?

- Desired effect of retiring an unfinished instruction in an atomic fashion



Abstraction Function

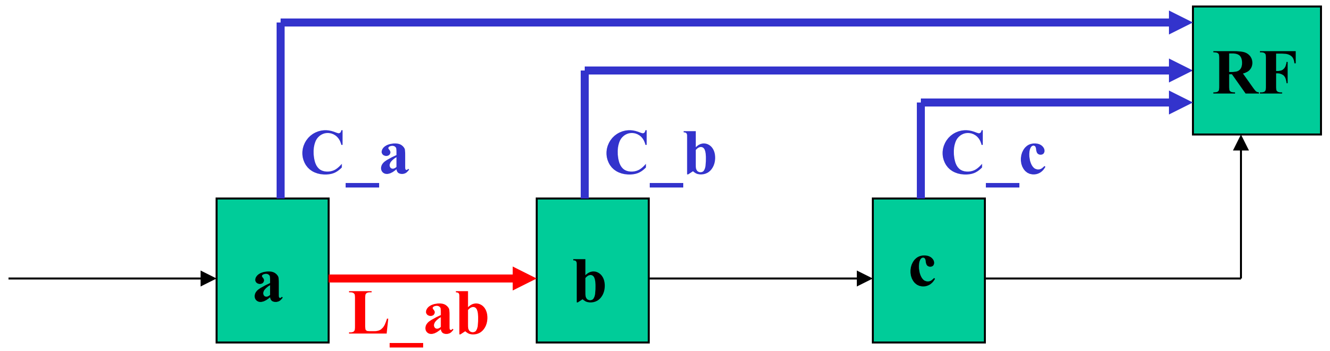
- Need to define an <u>abstraction function</u>



- Flushing the pipeline
- Our idea: Define abstraction function as a Composition of Completion Functions

Main Features

- Decomposition into verification conditions



Abs. fn = C_a o C_b o C_c

One VC is: C_a == L_ab o C_b

Motivations

- In this talk I'll present three methods:
  - Completion functions
  - Aggregation abstractions
  - Predicate abstraction

Automatic Checking of Aggregation Abstractions Through State Enumeration

Seungjoon Park   Satyaki Das   David L. Dill
Computer Systems Laboratory, Stanford University
Gates {358, 312, 314}, Stanford University, Ca 94305, U. S. A.
{park@turnip, satyaki@turnip, dill@cs}.stanford.edu

Formal verification of a system design compares two different descriptions of the system: the *specification* describes the desired behavior, and the *implementation* describes the actual behavior of the system. The implementation is usually given in some (potentially) executable form. There are many specification methods, such as assertions in the implementation code, temporal logic or the other logical properties, or automata. However, the most appropriate specification for a protocol is often an abstract version of the protocol with coarser-grained atomicity. For example, most cache coherence protocols are

Background and related work

The use of abstraction functions and relations of various kinds (also called refinements [1, 19], homomorphisms [16], and simulations [23]) to compare two descriptions is a fundamental verification technique that can be applied to many different problems and representations in many different ways (e.g. [21, 18, 6]).

$$\begin{tikzcd}
s_1 \arrow[r, "Impl"] \arrow[d, "R"'] & s'_1 \arrow[d, "R"'] \\
s_2 \arrow[r, "Spec"'] & s'_2
\end{tikzcd}$$

Figure 1 Abstraction relation

$$\forall s_1, s'_1, s_2 : \exists s'_2 : R(s_1, s_2) \wedge Impl(s_1, s'_1) \Rightarrow R(s'_1, s'_2) \wedge Spec(s_2, s'_2), \qquad (1)$$

and the specification. The implementation description contains a set of state variables; the set Q of states of the implementation is the set of assignments of values to the state variables. The specification description may contain a subset of the state variables of the implementation. Each description also

able commit step. Based on the reasoning about the commit steps, the user defines an aggregation function *aggr* which maps an implementation state to a specification state by first completing any committed but incomplete transactions, then hiding variables that do not appear in the specification. The

The aggregation abstraction works when the computation can be thought of as implementing a set of transactions and each transaction has an identifiable commit step. Based on the reasoning about the commit steps, the user defines an aggregation function *aggr* which maps an implementation state to a specification state by first completing any committed but incomplete transactions, then hiding variables that do not appear in the specification. The

$$\forall q \in Q : aggr(Impl_i(q)) = Spec_i(aggr(q)). \tag{2}$$

The requirements (2) will generally not hold for some absurd states that cannot actually occur during a computation. Hence, it is usually necessary to provide an *invariant* predicate, which characterizes a superset of all the

$$\forall q \in Q : Inv(q) \Rightarrow aggr(Impl_i(q)) = Spec_i(aggr(q)). \tag{3}$$

To check the aggregation abstraction automatically, we use a finite-state enumerator which explores all and only the reachable states of the implementation on-the-fly. Because state enumeration generates the exact invariant of the system while searching the state space, the user can check property (2) above without proving property (3).

```
Rule "Transition relation for Impl_i"
  CONDITION
==>
Var i0, i1: ImplState;
Begin
  i0 := current_state;  ACTION-STATEMENT;  i1 := current_state;
  Assert Spec_i(aggr(i0)) = aggr(i1);
Endrule;
```

The two local variables i_0 and i_1 contain the implementation state before and after the execution of the rule respectively. The assert statement expresses the corresponding commutativity requirement using the functions defined for specification steps and the aggregation function.

```
Function Faggr(ist:ImplState): SpecState;
Var  sst: SpecState;     -- specification state to be returned
Begin
 For i:Proc Do                                 --
   sst.State[i] := ist.Procs[i].Cache.State;   -- Copy the specification
   sst.Data[i]  := ist.Procs[i].Cache.Value;   -- variables from the current
 EndFor;                                       -- state of implementation
 sst.Memory   := ist.Memory;                   --

 For i:Queue do          -- Check each message in the network
   If i < ist.PNet.Count then   -- for the reply queue
     If ist.PNet.Message[i].Mtype=Putt then
       -- Simulate processing a 'put' reply
       If ist.PNet.Message[i].dst=Home then
         sst.Memory := ist.PNet.Message[i].Data;
       Endif;
       If ! ist.Procs[ist.PNet.Message[i].dst].Cache.InvMarked then
         sst.Data[ist.PNet.Message[i].dst]  := ist.PNet.Message[i].Data;
         sst.State[ist.PNet.Message[i].dst]  := Shared;
       Else
         sst.State[ist.PNet.Message[i].dst]  := Invalid;
       EndIf;
     Elsif ist.PNet.Message[i].Mtype=PutX then
       -- Simulate processing a 'putx' reply
       sst.Data[ist.PNet.Message[i].dst]  := ist.PNet.Message[i].Data;
       sst.State[ist.PNet.Message[i].dst]  := Exclusive;
     endif;
   Endif;
   If i < ist.QNet.Count then   -- for the request queue
     -- Simulate processing a 'WB' or a 'ShWB' sent to the home
     if ist.QNet.Message[i].Mtype=WB | ist.QNet.Message[i].Mtype=ShWB
     then sst.Memory := ist.QNet.Message[i].Data;
     endif;
   Endif;
 EndFor;
 Return sst;
End;
```

```
Function Atomic_GetX(st:SpecState;  oldproc,newproc:Proc): SpecState;
Begin
  If st.State[oldproc] = Exclusive & oldproc != newproc then
    st.State[oldproc] := Invalid;
    st.State[newproc] := Exclusive;
    st.Data[newproc] := st.Data[oldproc];
    Return st;
  Elsif Forall i:Proc Do st.State[i] != Exclusive EndForall then
    st.State[Home] := Invalid;
    st.State[newproc] := Exclusive;
    st.Data[newproc] := st.Memory;
    Return st;
  Else Return st;
  Endif;
End;
```

Figure 3 The specification step processing a write miss in the FLASH protocol

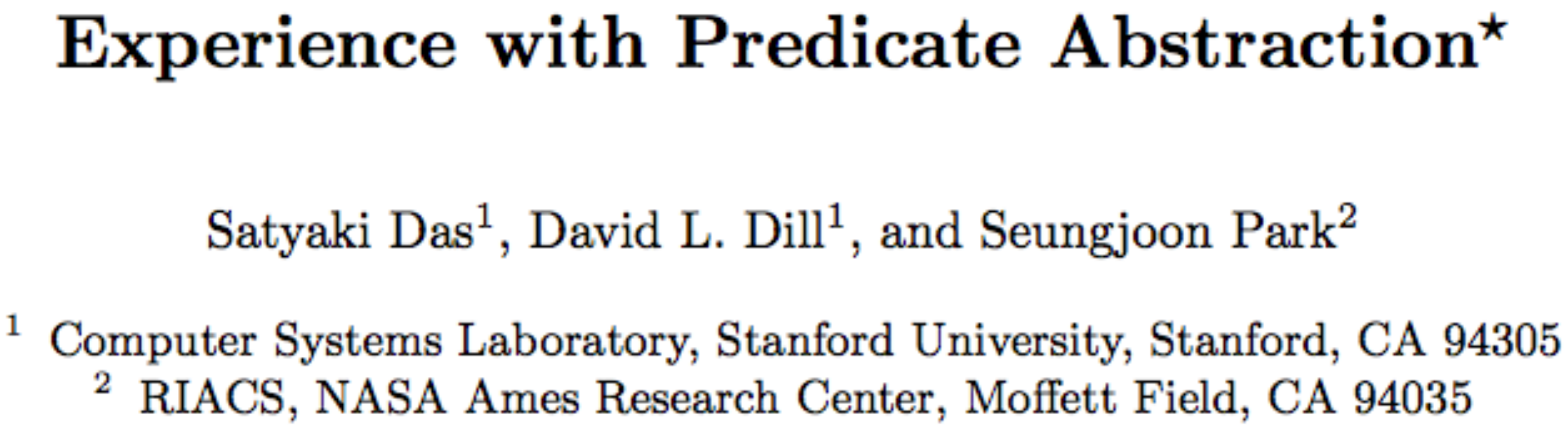
Motivations

- In this talk I’ll present three methods:
  - Completion functions
  - Aggregation abstractions
  - Predicate abstraction

Predicate Abstraction

- Method to compute Galois-Connections nicely using a bunch of predicates
  - The "shatter the mirror" analogy of Ashok Chandra (IBM)
    - Each mirror fragment is like an equivalence class under predicate abstraction
- Graf and Saidi introduced Pred Abs
- Das, Dill, Park explained it well!

Predicate Abstraction using BDDs and SMT (cav'99 paper)



Experience with Predicate Abstraction*

Satyaki Das[1], David L. Dill[1], and Seungjoon Park[2]

[1] Computer Systems Laboratory, Stanford University, Stanford, CA 94305
[2] RIACS, NASA Ames Research Center, Moffett Field, CA 94035

Predicate Abstraction Search (DDP)

- Say a system has variable X,Y,Z over INT32, and say we want to observe the system via X<Y, X==Y+1, and X=Z-2 (p1,p2,p3 for now)
- Form the 8 obvious truth-combinations of p1,p2,p3 (all Boolean cubes)
- Each is like the ”Ashok Chandra Mirror Facet”
- Say you are in facet 5 (p123 = 101)
  - A BDD path
- Say you want to know what happens when we do
  - Atomic { X = Y ; Y = Z-1 ; Z = X+2 }
- We do this
  - Concretize 101 into the pred combinations
  - Compute the strongest post-condition transformer … in a sense
  - Then discover the abstract state (mirror facet) we must be in!
- If done “religiously” we will get ”nice” (strongest, I think - check) invariants wrt p1,2,3

- Automated nicely in this paper by DDP’99

Conclusions

- There are many ways to compare systems
  - Language containment
  - Simulation
  - Bisimulation
  - …
- This is really the essence of safety verification
  - As Manna told me once, “my progress book is still in progress” ☺