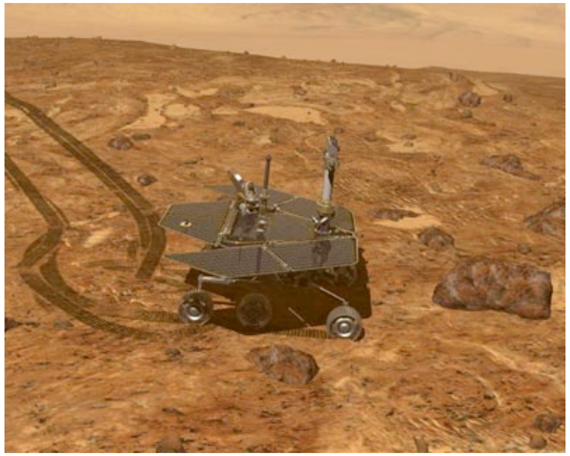
Software Verification for Space Applications

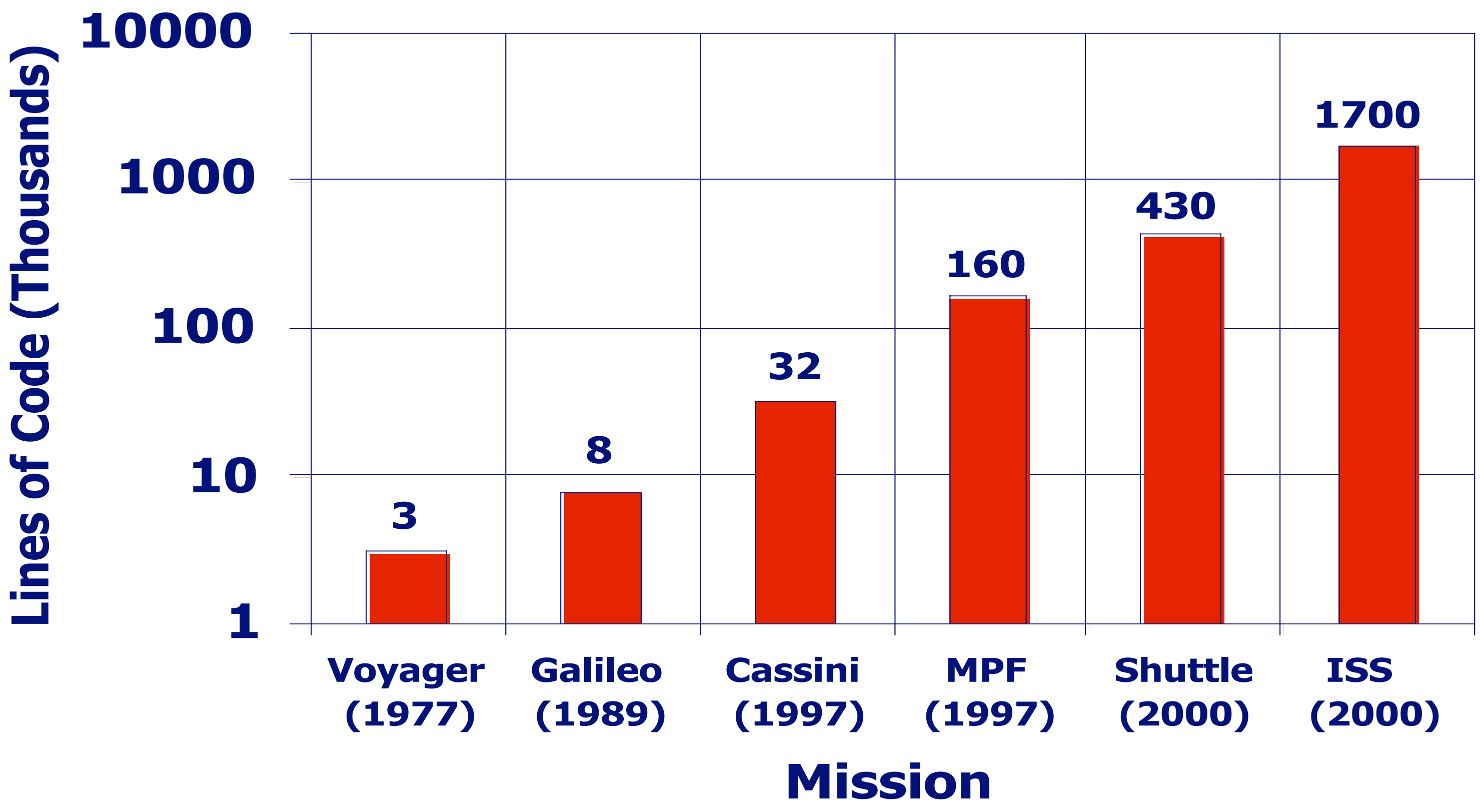
Part 1. Static Analysis



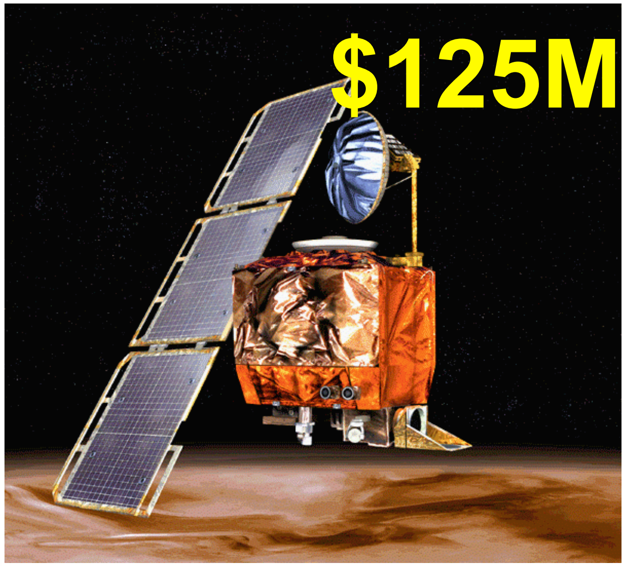
Guillaume Brat

USRA/RIACS

Software blowup



Famous aerospace failures



NASA Software Challenges

- Need to develop three systems for each mission:
  - Flight software
  - Ground software
  - Simulation software
- Flight software
  - Has to fit on radiation-hardened processors
  - Limited memory resources
  - Has to provide enough information for diagnosis
  - Can be patched (or uploaded) during the mission
- Each mission has its own goals, and therefore, each software system is unique!
- Cannot benefit from opening its source code to the public because of security reasons.
  - No open-source V&V
- Mission software is getting more complex.
  - Large source code (~1 MLOC)
  - The structure of the code is more complex

International Space Station

- International Space Station:
  - Attitude control system, 1553 bus, science payloads
  - International development (interface issues)
  - Codes ranging from 10-50 KLOC
  - A failure in a non critical system can cause a hazardous situation endangering the whole station
  - Enormous maintenance costs
  - Over 500 defects reported
  - Over 3 MLOC by now

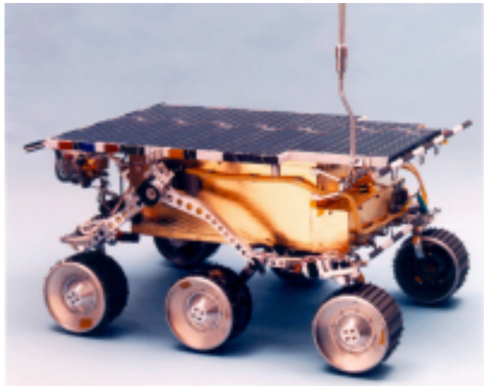
ISS problem example

- SCR 25345 describes an issue where GNC Redundancy Management (RM) does not appropriately reset "Indicate Attitude Control Handover to RS" Flag .
  - Flag set (4 occurrences since Feb'03 CCS R3 uplink)
    - On GNC MDM failure
    - SMTC loss of communication (triggers GNC failure response)
    - Planned GNC MDM swaps
  - If flag set, Autohandover to RS Enabled, RS is in Mode of CMG TA or Indicator, and US is Master; FDIR will send an Off Nominal US to RS H/O command.
  - If this flag is not reset an attitude control force fight will occur.

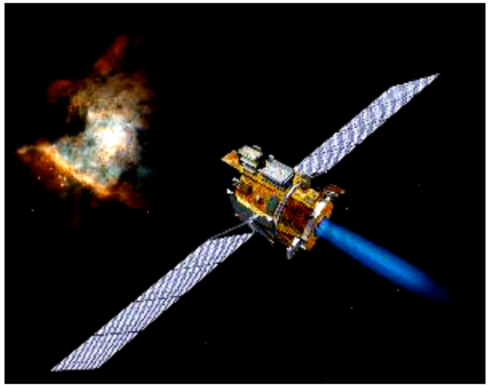
Dan Duncavage, NASA JSC, June 2003

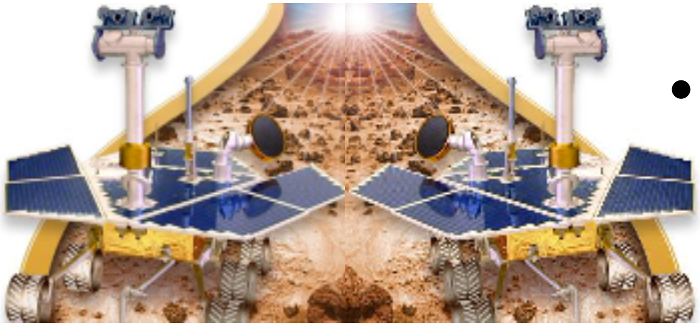
"Are these problems that ANY sort of computational assistance will help? I always knew that we couldn't build a complete system that would automatically tell us what problems would occur with this or that software change. But I am hoping that we can build tools that make things a whole lot faster than they are now. "

Mars mission software



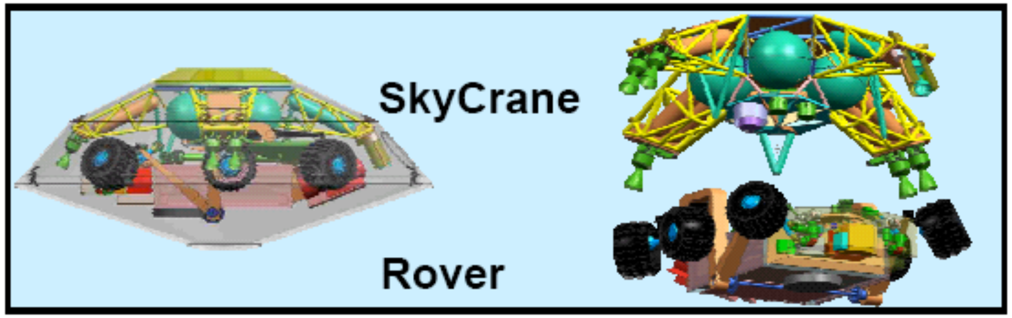
- Mars Path Finder:
  - Code size: 140 KLOC
  - Famous bug: priority inversion problem
- Deep Space One:
  - Code size: 280 KLOC
  - Famous bug: race condition problem in the RAX software



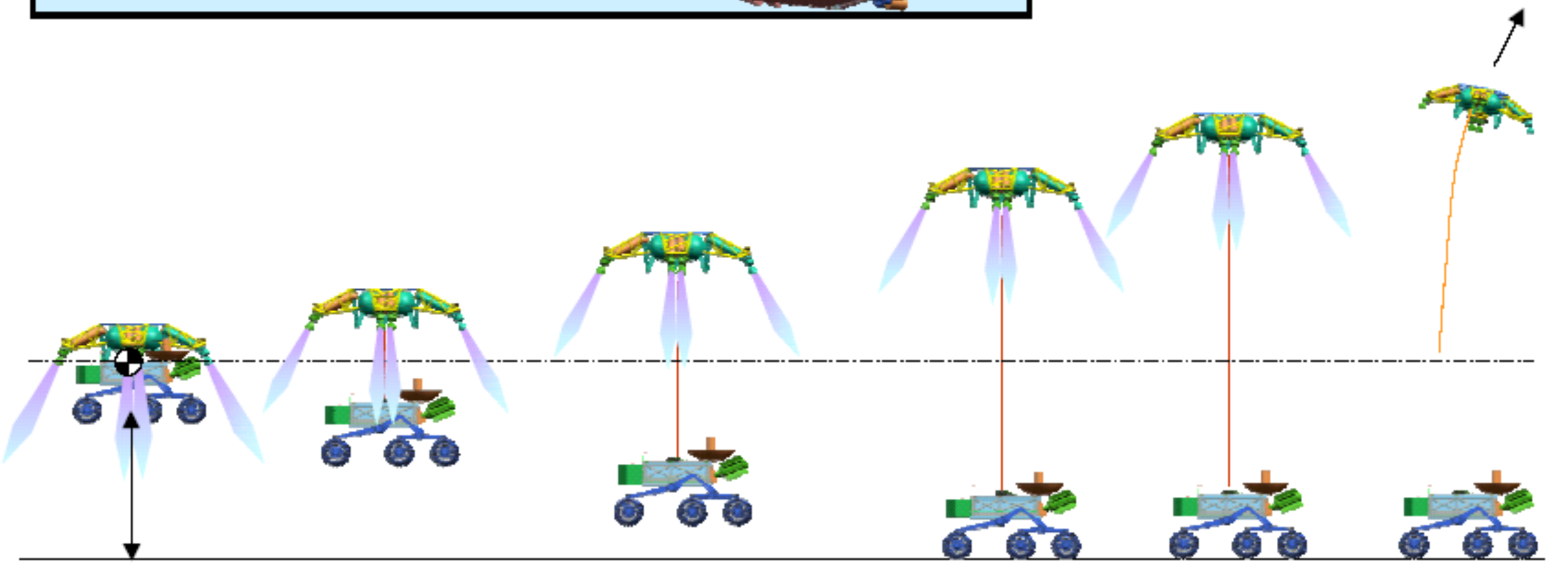


- Mars Exploration Rovers:
  - Code size: > 650 KLOC
  - Famous bug: Flash memory problem

Mars Science Laboratory



PRE-DECISIONAL DRAFT; For planning and discussion purposes only



Mars Science Laboratory

- Complicated Landing:
  - no ground real time control
  - The rover lands, the crane flies away
- Long autonomous traverses
  - Automatic obstacle avoidance
  - Recognize possible interesting science along the way
- Critical systems:
  - Uses RTG (no solar panels) for power
- It’s a long mission, almost 2 years of rover operation
  - Needs to be durable
  - Plenty of time to recover in case of problems

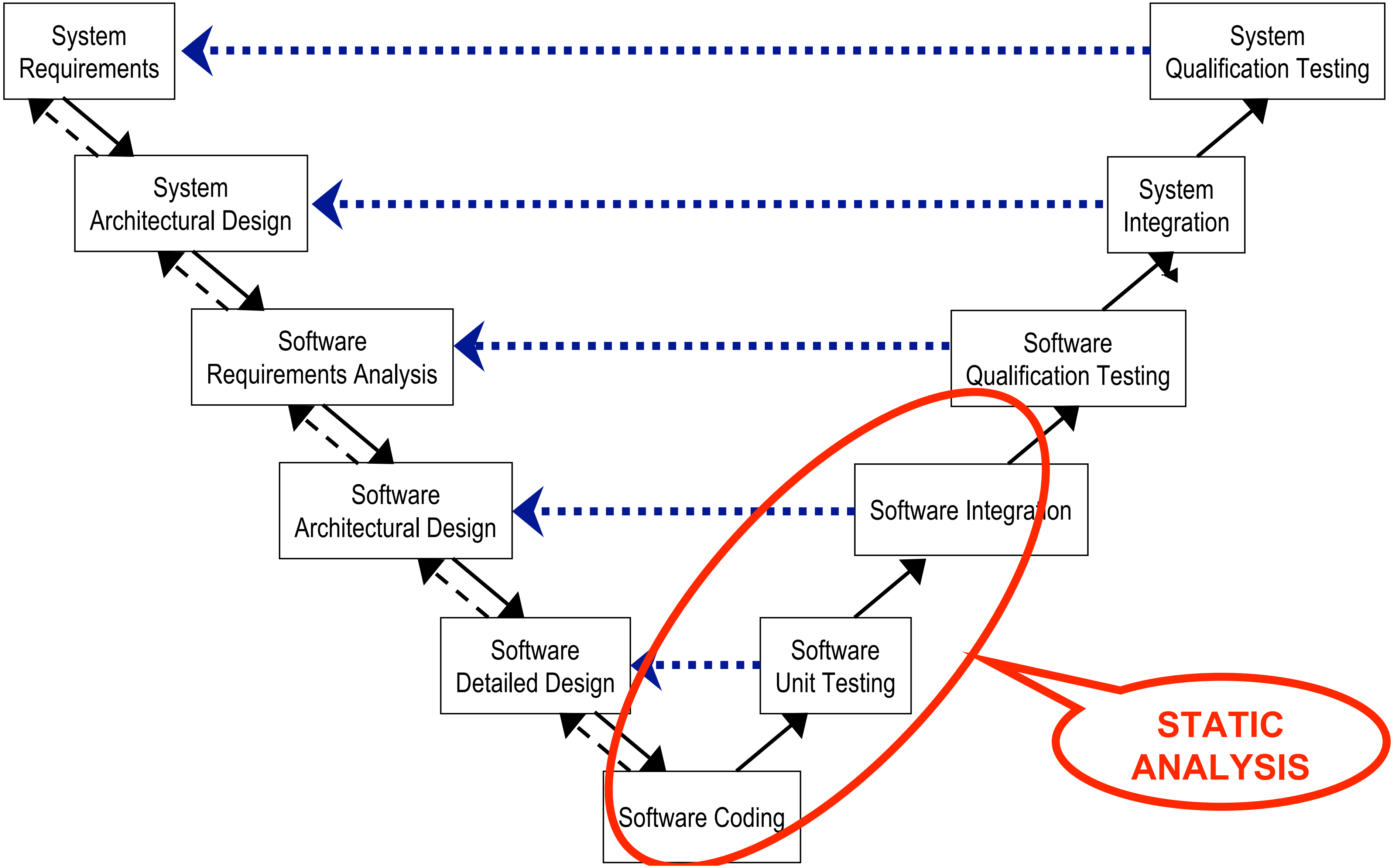
How is the Software Verified?

- Testing
- Mars missions: high-fidelity test bench
  - Runs 24 hours a day
  - 8 hour test sessions: lost if a runtime error occurs
- Space Station:
  - Critical software: on-ground simulator maintained at Marshall Space Center
  - Payloads:
    - Independently verified by contractors
    - NASA test requirement document

How effective is this?

- Badly re-initialized state variable for MPL: caused the crash of the lander ($150M)
- Unit mismatch for MCO: caused the orbiter to miss its orbit insertion and burn during re-entry ($85M)
- Thread priority inversion problem for MPF: 24 hours of science data lost
- Flash memory problem for MER: rover paralyzed during several days
- Science mission for the ISS currently under validation:
  - Passes NASA test requirements
  - But… 500+ defects reported

Software Development Process



Static Analysis

all possible values
(and more) are computed

the analysis is done
without executing the program

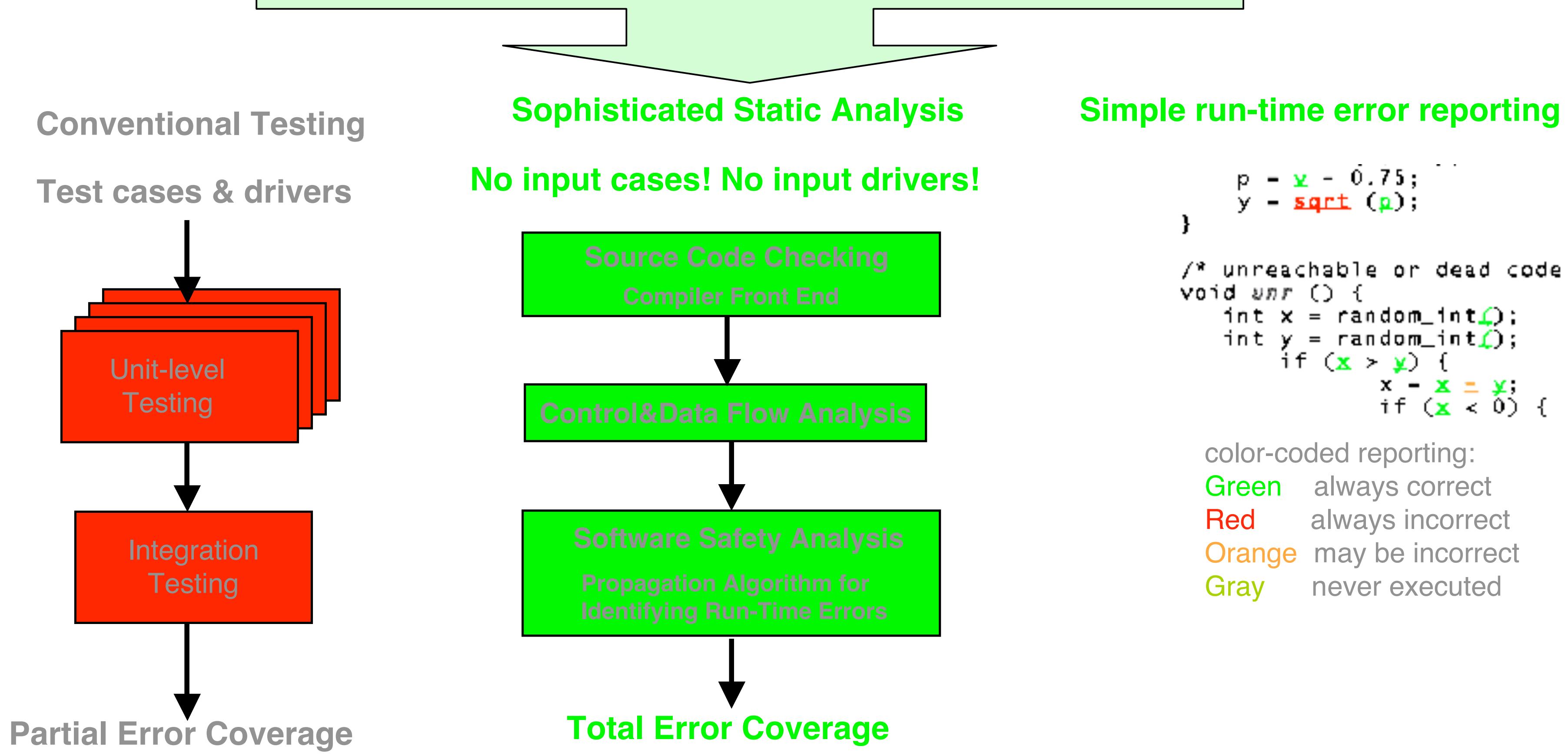
Static analysis offers compile-time techniques for predicting safe and computable approximations to the set of values arising dynamically at run-time when executing the program

We use abstract interpretation techniques
to extract a safe system of semantic equations
which can be resolved using lattice theory techniques
to obtain numerical invariants for each program point

Static analysis

Static analyzers find runtime errors in programs.

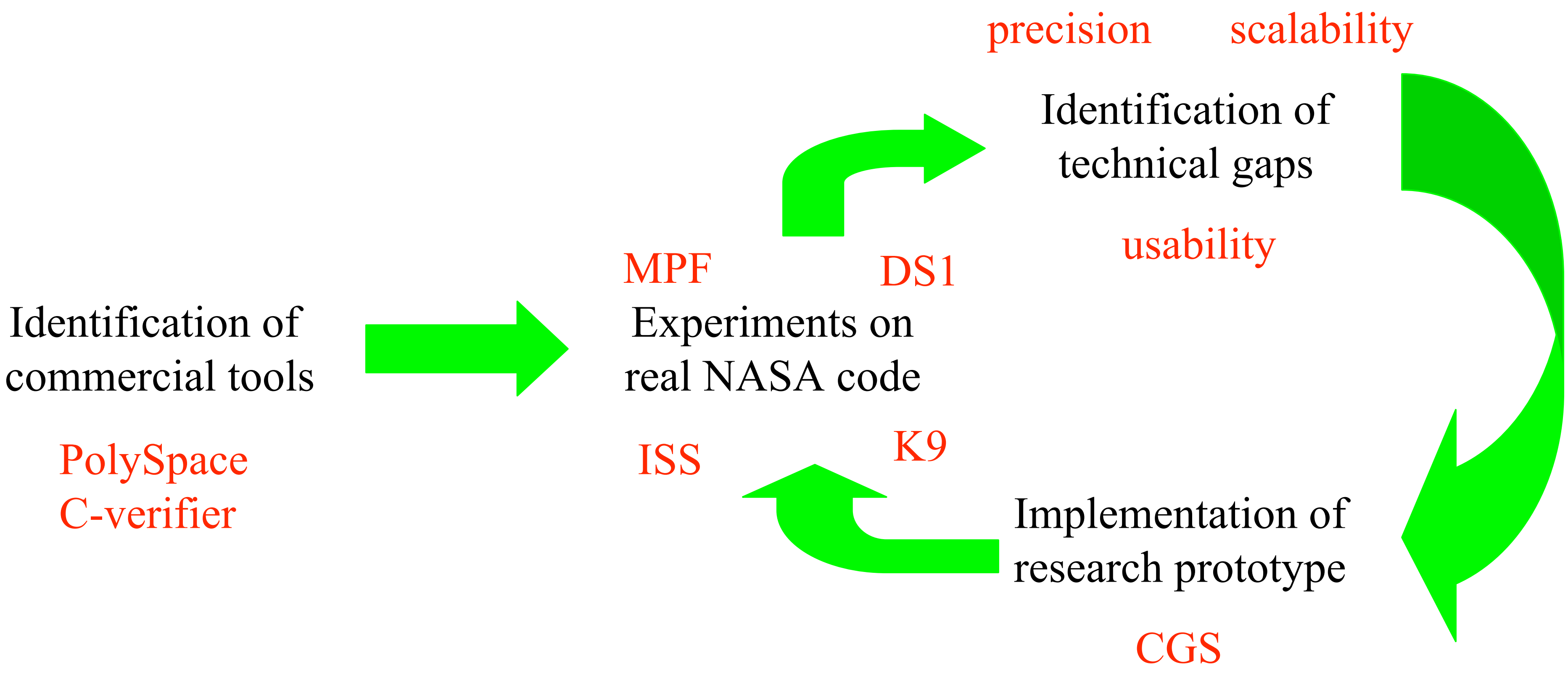
They work like sophisticated compilers.



Defect Classes

- Static analysis is well-suited for catching runtime errors
  - Array-out-bound accesses
  - Un-initialized variables/pointers
  - Overflow/Underflow
  - Invalid arithmetic operations
- Also for program understanding
  - Data dependences
  - Control dependences
  - Slicing
  - Call graphs
- Potential applications to
  - Convergence/divergence in floating point computations
  - Unit mismatching
  - Execution time predictions
  - Memory usage predictions

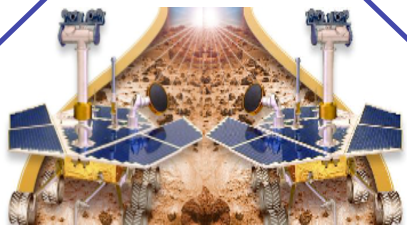
Static Analysis Research Process



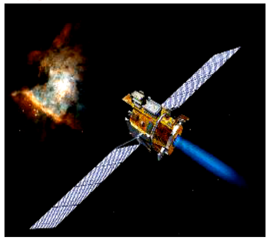
Analysis of MPF family

POLYSPACE C-VERIFIER

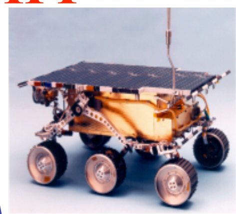
MER
650KLoc



DS1 285KLoc



MPF 134KLoc



Found errors!

Un-initialized variables
Out-of-bound array accesses
Overflow/underflow problems

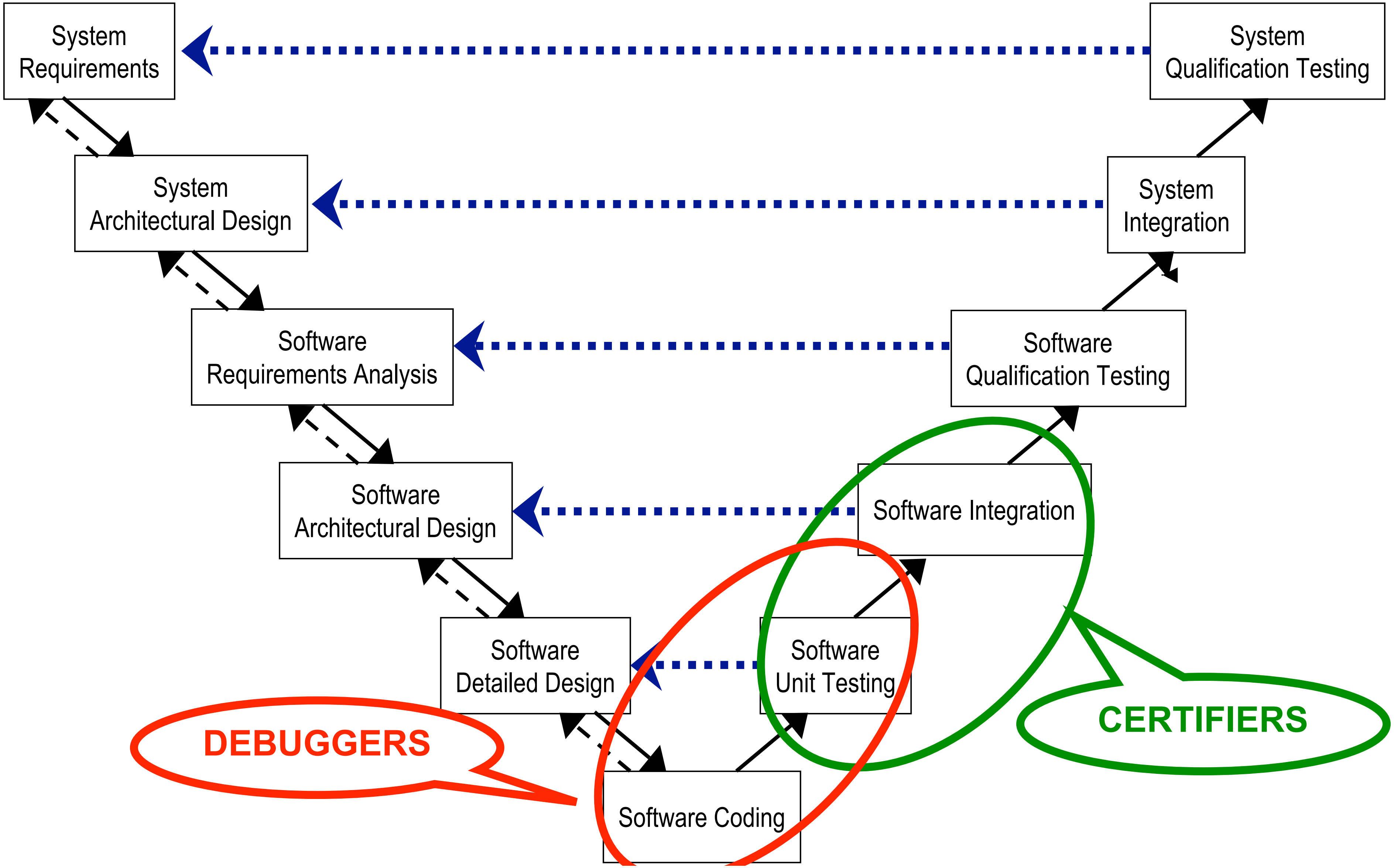
Limitations

Needed to modify the code slightly
Limited code size to ~40 KLoc
Got too many false positive

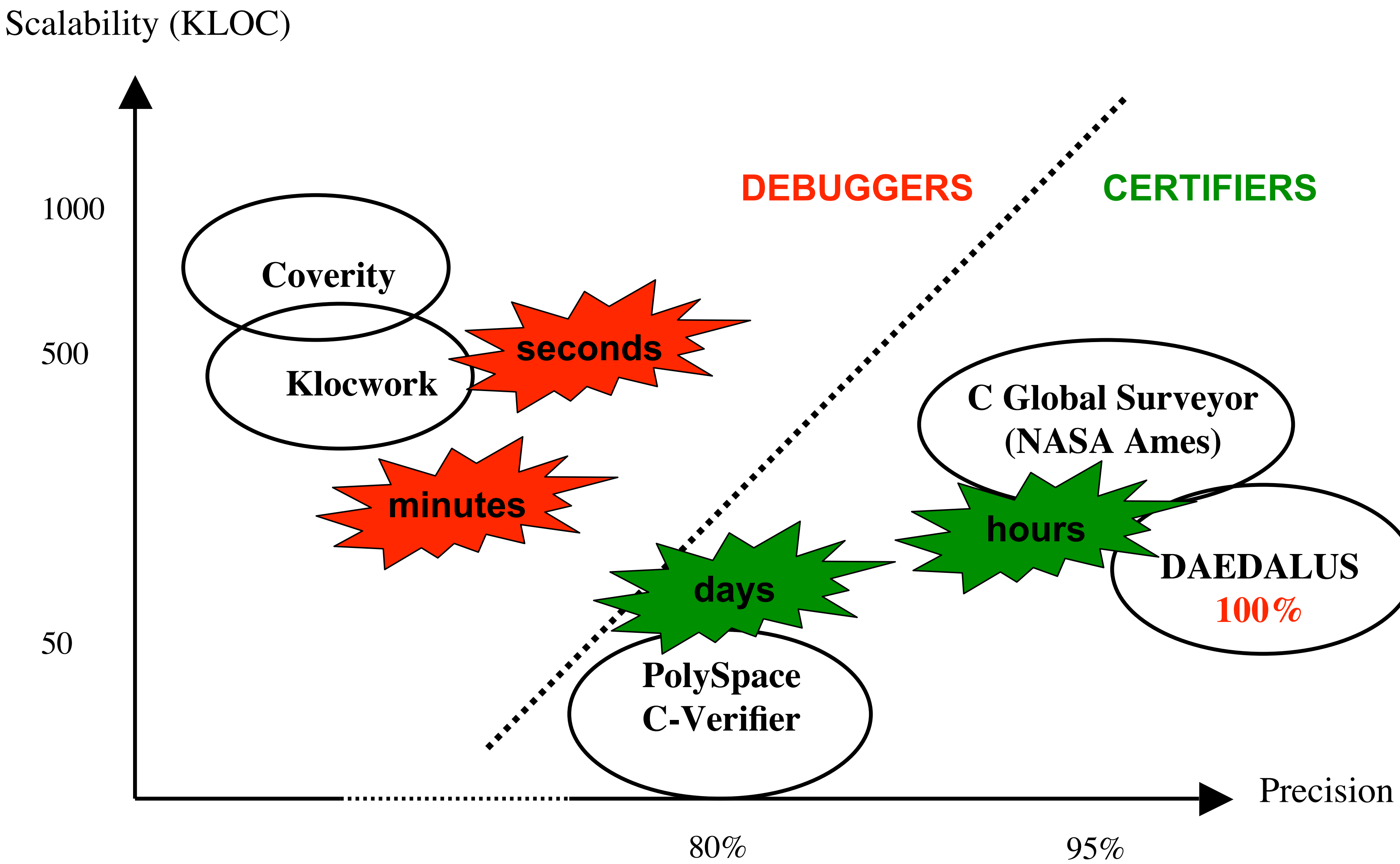
The MER Experiment

- We conducted extensive experiments with PolySpace Verifier:
  - Minors bugs found in MER
  - Serious out-of-bounds array accesses found in an ISS Science Payload
- Absence of runtime errors (80% precision)
- Useful: yes
- Effective: no
  - It takes 24 hours to analyze 40 KLOC
  - Difficulty to break down large systems into small modules

What type of static analysis?



Practical Static Analysis



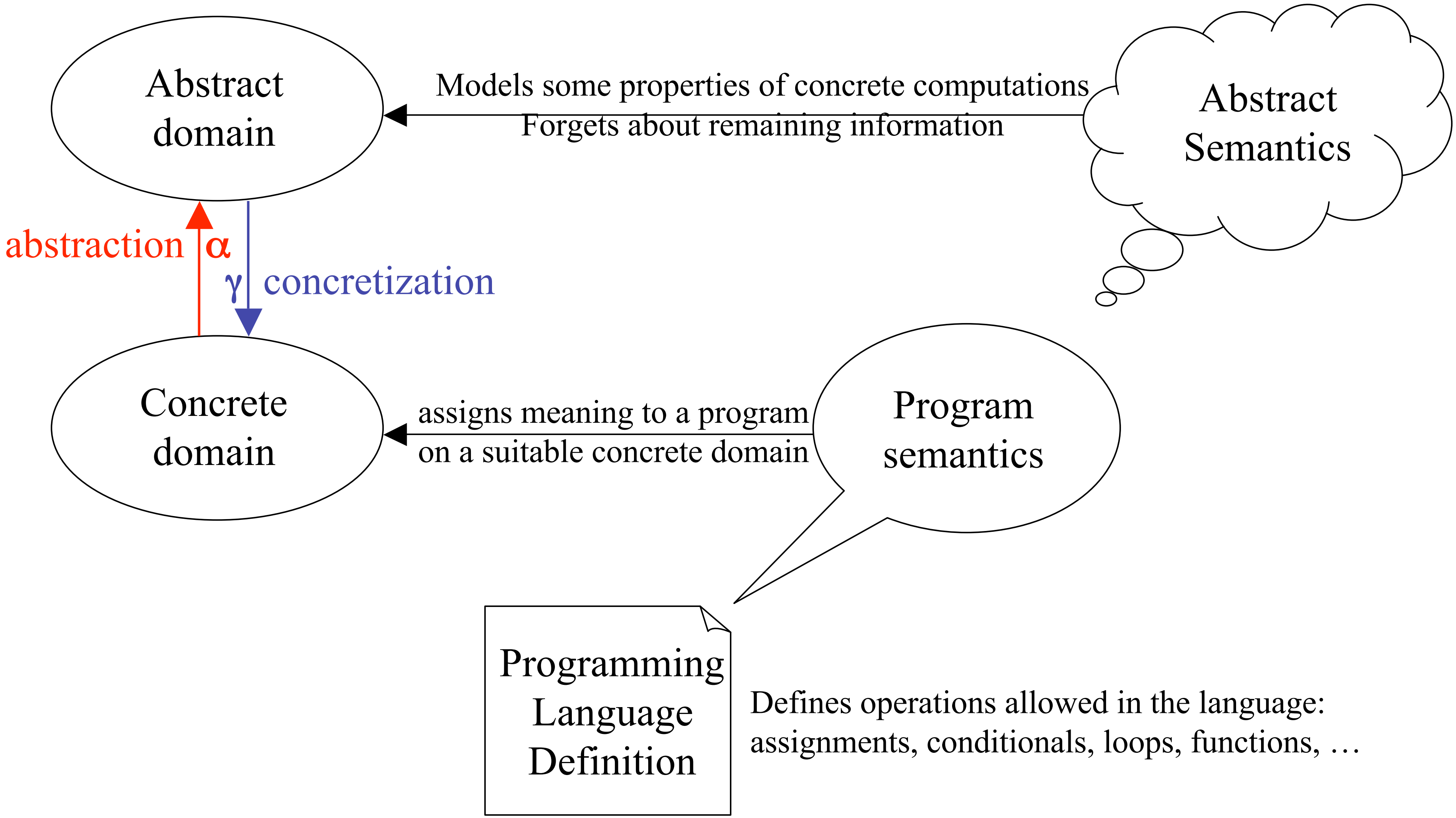
NASA Requirements

- Scalability:
  - Analyze large systems in less than 24 hours
  - Analysis time similar to compilation time for mid-size programs
- Precision:
  - At least 80%
  - Informative: the analysis provides enough information to diagnose a warning

C Global Surveyor

- Prototype analyzer
  - Based on abstract interpretation
  - specialized for NASA flight software
- Covers major pointer manipulation errors:
  - Out-of-bounds array indexing
  - Uninitialized pointer access
  - Null pointer access
- Keeps all intermediate results of the analysis in a human readable form: huge amount of artifacts

Abstract Interpretation



Simple Example

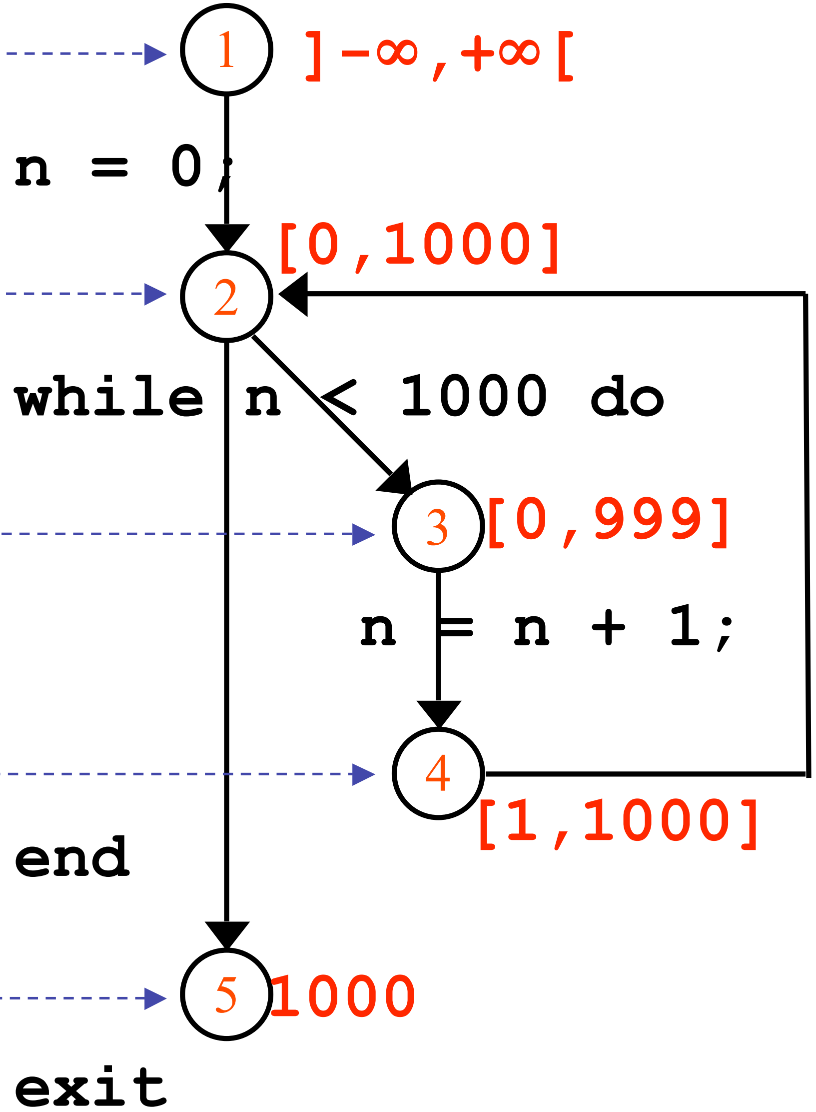
$$E_1 = \{n \Rightarrow \Omega\}$$

$$E_2 = [\![\mathbf{n = 0}]\!]\ E_1 \cup E_4$$

$$E_3 = E_2 \cap\ ]-\infty, 999]$$

$$E_4 = [\![\mathbf{n = n + 1}]\!]\ E_3$$

$$E_5 = E_2 \cap [1000, +\infty[$$



Simple Example

In effect, the analysis has automatically computed numerical invariants!

```
]-∞,+∞[

n = 0;

[0,1000]

while n < 1000 do

         [0,999]

         n = n + 1;

         [1,1000]

end

1000

exit
```

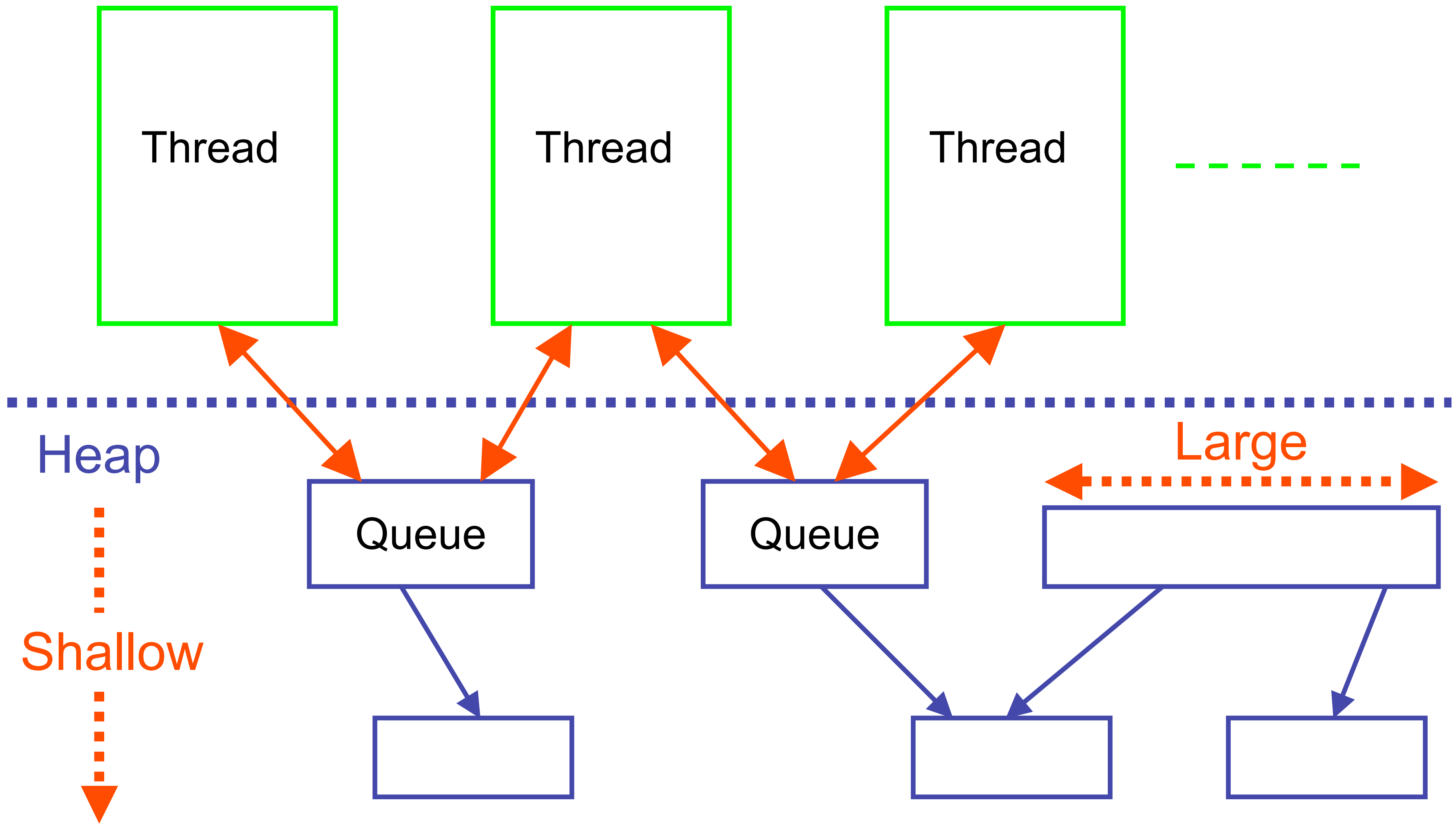
Array Bound Checking

- Arrays are the basic data structures in embedded programs
- Out-of-bounds array access:
  - One of the most common runtime errors
  - One the most difficult to trace back

Bug found in an ISS Science Payload

```
double a[10];
for (i = 0; i < 10; i++)
   a[i] = ...;        <------   0 <= i < 10
if (...)
   a[i] = ...;        <------   i = 10
```

Runtime Structure



MPF Flight Software Family

10...1000 call sites

```
assign (A, B, 10)        assign (&pS->f, &A[2], m)
```

```
assign (double *p, double *q, int n) {
  int i;
  for (i = 0; i < n; i++)
    p[i] = q[i];
}
```

Thousands of such functions
Almost all of them contain loops

Fast Context Sensitivity

- Context-sensitivity is required
- We can't afford performing 1000 fixpoint iterations with widening and narrowing for each function
- Compute a summary of the function using a relational numerical lattice

```
access(p[i], 0 <= i < n)

access(q[i], 0 <= i < n)
```

Byte-Based Pointer Model

- Pointer analyses commonly use symbolic access paths into structures
- Mixing symbolic and numerical information is difficult and costly
- We use a uniform byte-based representation (sufficient for array bound checking)

`&S.f[2][3]`

↓

&S + *offset*(**`f`**) + 2 * *size*(row) + 3 * *size*(elem)

Relational Domain

- Convex polyhedra are too costly (exponential complexity)

- Weakly relational domain of Difference-Bound Matrices (Mine 01):
  - $\{x - y \leq c, z - t \leq c', ...\}$
  - Floyd-Warshall algorithm (shortest path):
    - $x - y \leq c \;\&\; y - z \leq c' \Rightarrow x - z \leq c + c'$
    - $x - y \leq c, x - y \leq c' \Rightarrow x - y \leq \min(c, c')$
  - Cubic time, quadratic space complexity

Expressiveness Problem

- Cannot express the invariant:

  $0 \leq$ **offset** $\leq$ **`n * sizeof (double)`**

- Solution: use auxiliary variables
  - Split up the offset: **offset = b + δ * u**

base offset

relative offset

unit

  - New invariant:
    - **b** = 0
    - **u** = **`sizeof (double)`**
    - **`0 ≤ δ ≤ n`**

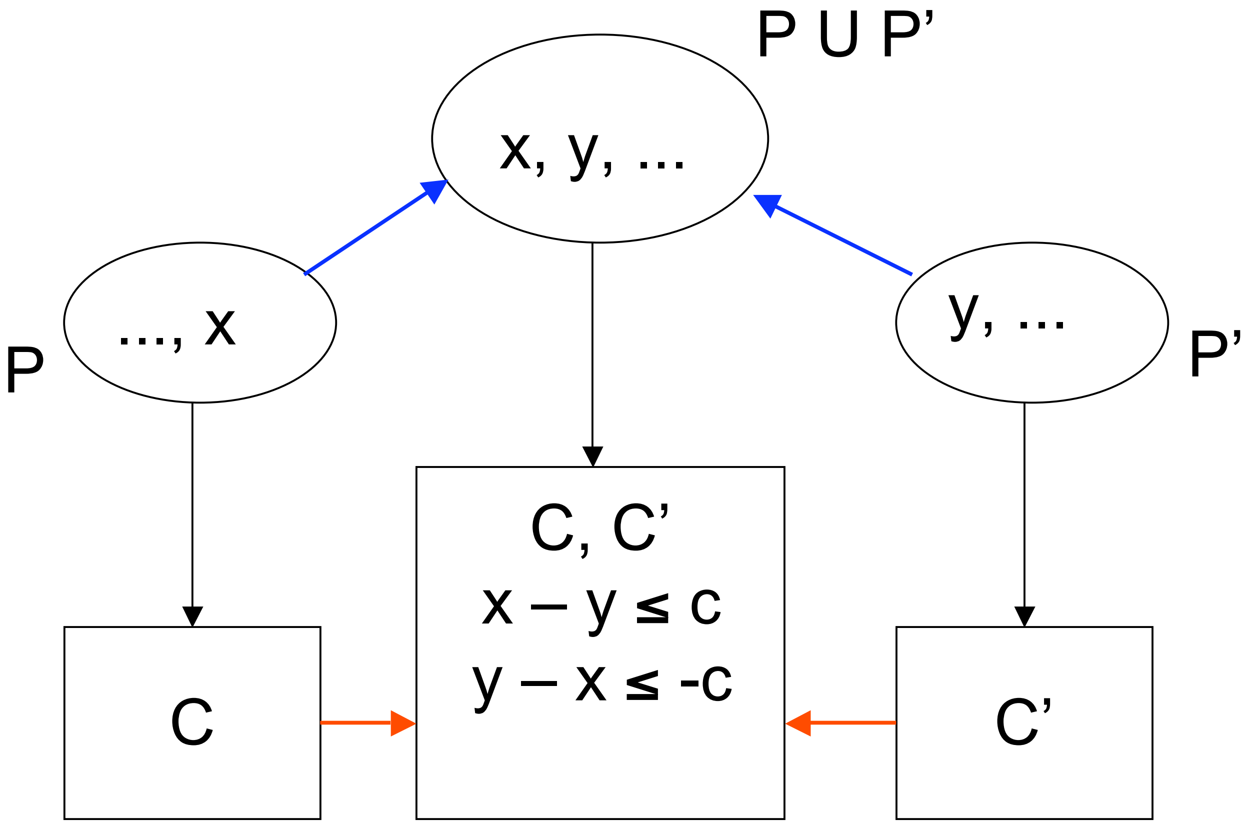
Expressible as a DBM

Scalability Issues

- The domain of Difference Bound Matrices do not scale
- Problem: strongly polynomial (worst-case bounds always attained)
- Solution: split up the relations into small packets using computational dependencies

Adaptive Variable Clustering

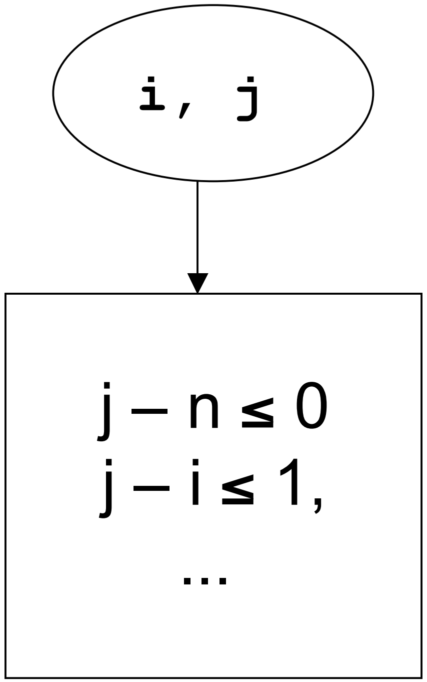
- x = y + c



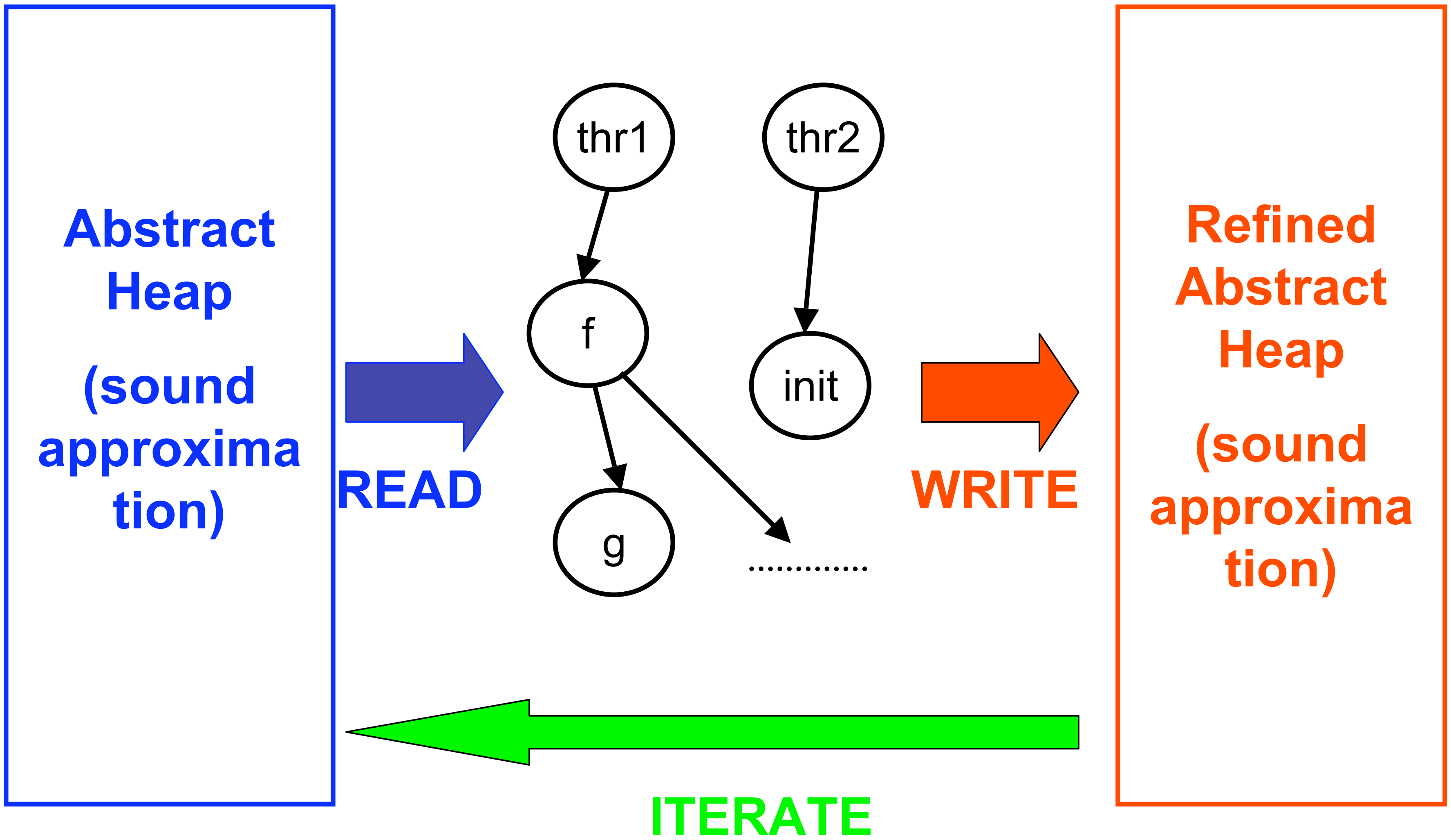
Loops

- All variable modified within a loop are clustered (implicit dependencies)

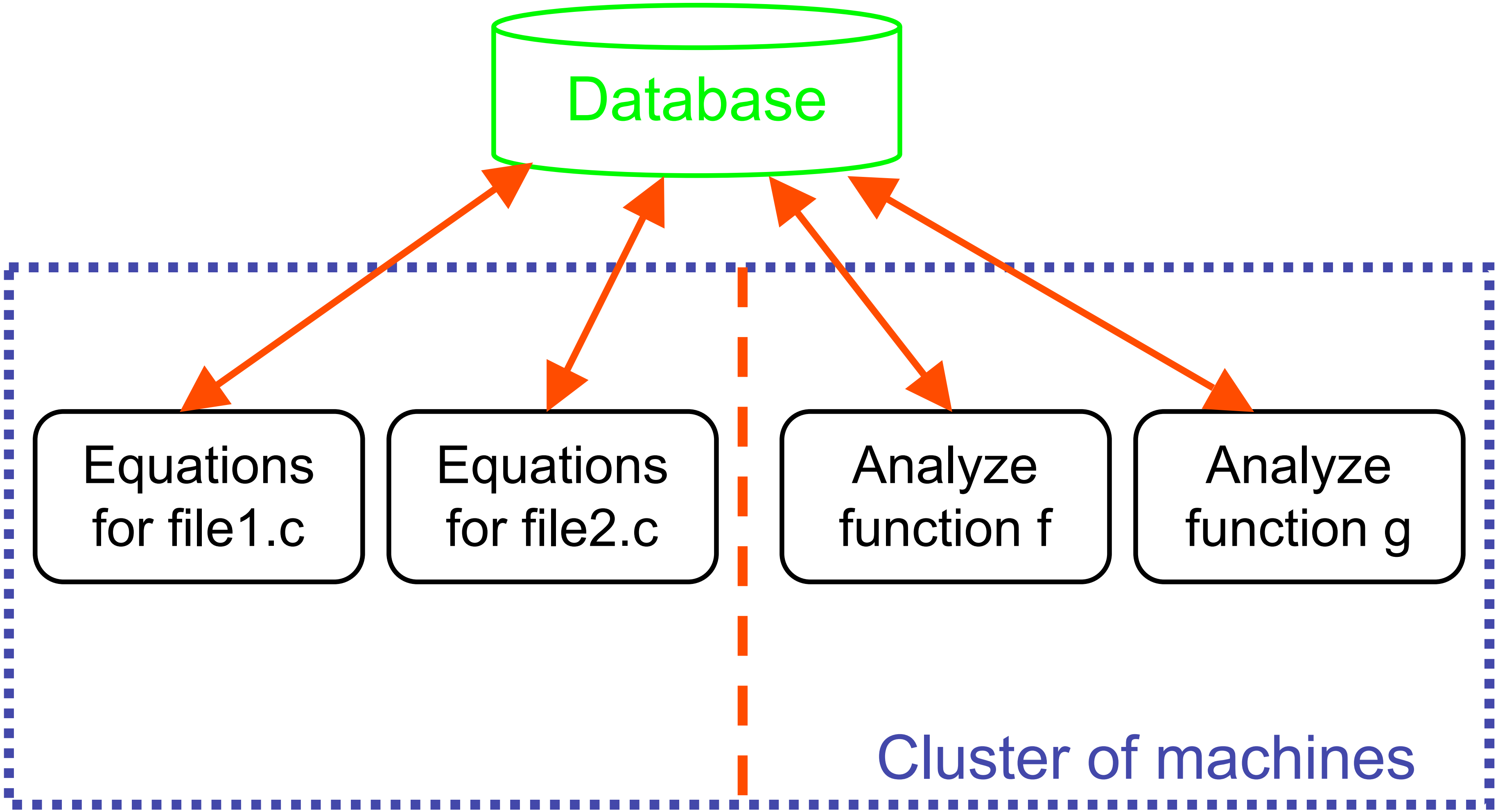
```
j = 1;
for (i = 0; i < n; i++) {
  j++;
  a[j] = ...;
}
```



Memory Graph Construction



Implementation of CGS



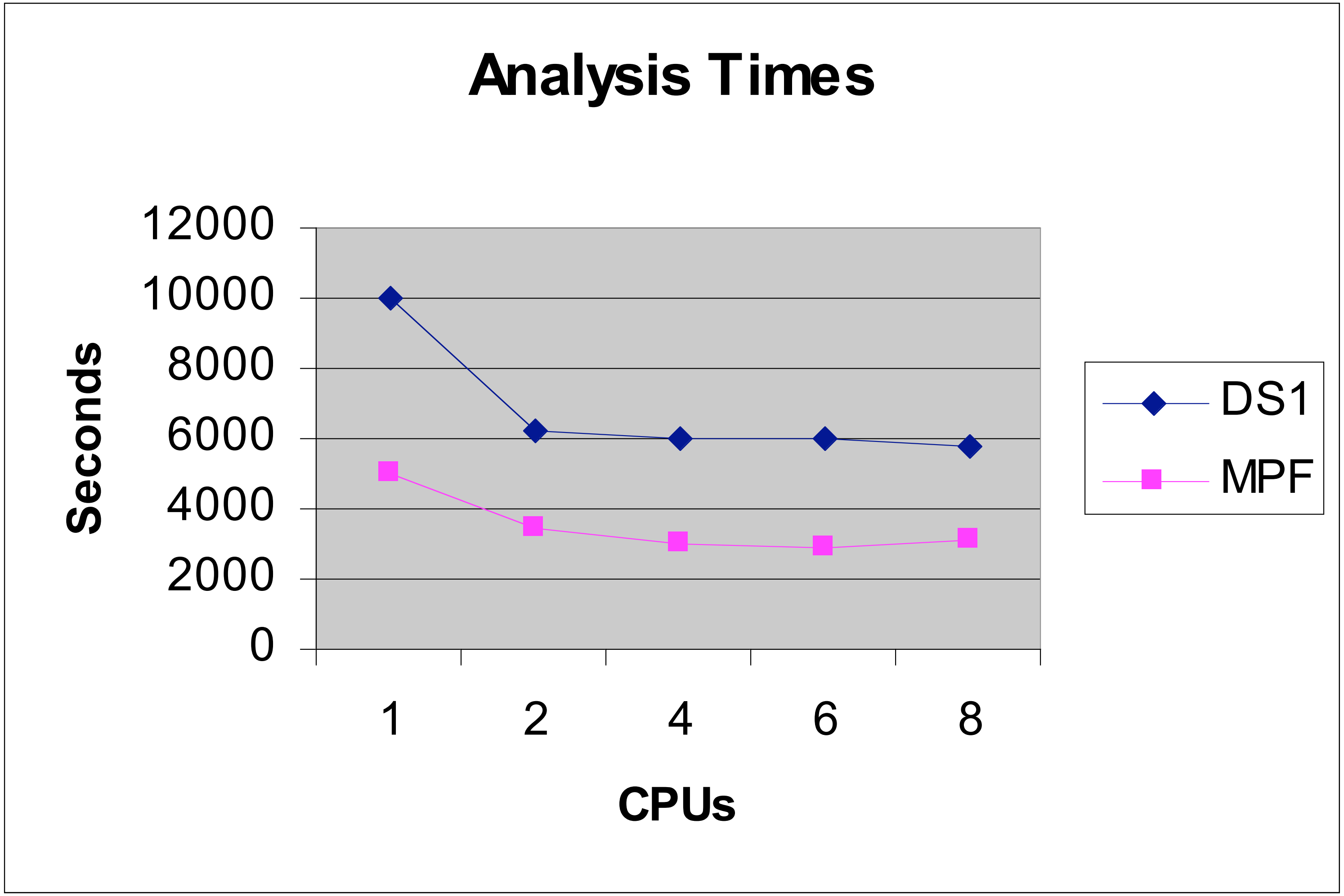
Working with a Database

- We use PostgreSQL
- Mutual exclusion problems are cared for by the database
- Simple interface using SQL queries
- Efficient communications require index structures (B-Trees):
  - Populating tables is slower
  - Difficult to manage
- Granularity problems: splitting up large tables into smaller ones

Parallel implementation

- We use the Parallel Virtual Machine (PVM)
- High-level interface for process creation and communication
- Allows heterogeneous implementation: currently a mix of C and OCaml
- Remote debugging is extremely difficult
- Design is difficult:
  - Scheduling policies
  - Granularity of computations

Effectiveness of Parallelization



The I/O Bottleneck

- The performance curve flattens: overhead of going through the network
- MER takes a bit less than 24 hours to analyze:
  - 70% of the time is spent in the interprocedural propagation
  - I/O times dominate (loading/unloading large tables)
- Under investigation: caching tables on machines of the cluster and using PVM communication mechanism (faster than concurrent database access)

Experimental Results

	Size (KLOC)	Max Size Analyzed	Precision	Analysis Time (hours)
MPF	140	20	80%	8-12
MPF	140	140	80%	1.5
DS1	280	280	80%	2.5
MER	550	550	80%	20

Commercial tool

C Global Surveyor

CGS Users

- Mars & Solar System Exploration (JPL)
  - MER
  - MSL
- Manned space missions: International Space Station & Shuttle
  - Urine Processing Assembly (20KLOC)
  - Material Science Research Rack (82KLOC)
  - Advanced Video Guidance System (12KLOC)
  - Space Shuttle Main Engine Controller(?)
  - Biological Research Project Rack Interface Controller (40KLOC)
  - Centrifuge Rack Interface Controller (40KLOC)
- Independent Verification & Validation Center

Will include some C++ (MSL)

Done without daily expert help (Urine Processing Assembly, Material Science Research Rack, Advanced Video Guidance System, Space Shuttle Main Engine Controller)

Heavy expert help (Biological Research Project Rack Interface Controller, Centrifuge Rack Interface Controller)

CGS fact sheet

- Static analyzer for finding runtime errors in C programs
  - Out-of-bound array accesses
  - De-referencing null pointers
  - Tested on MPF, DS1, and ISS flight software systems
- Developed (20 KLoc of C) at NASA Ames in ASE group
  - A. Venet: no longer working at NASA
  - G. Brat: brat@email.arc.nasa.gov
  - S. Thompson: thompson@email.arc.nasa.gov
- Runs on Linux and Solaris platforms
  - RedHat Linux 2.4
- Analysis can be distributed over several CPUs
  - Using PVM distribution system
- Results available using SQL queries
  - To the PostgreSQL database
  - Graphical user interface

Future directions

- Need to move to analyzing C++
  - C is the legacy language
  - New development (CEV, CLV) is in C++
- C++ is a complicated language
  - Dynamic allocation
  - Virtual functions
  - Object-oriented
  - No thread standard package
- Our strategy
  - Develop more than just static analysis tools
  - Based them on the same language compilation framework

A C++ tool suite

Common Interface (Eclipse, Web-based)

MCP
Model Checker

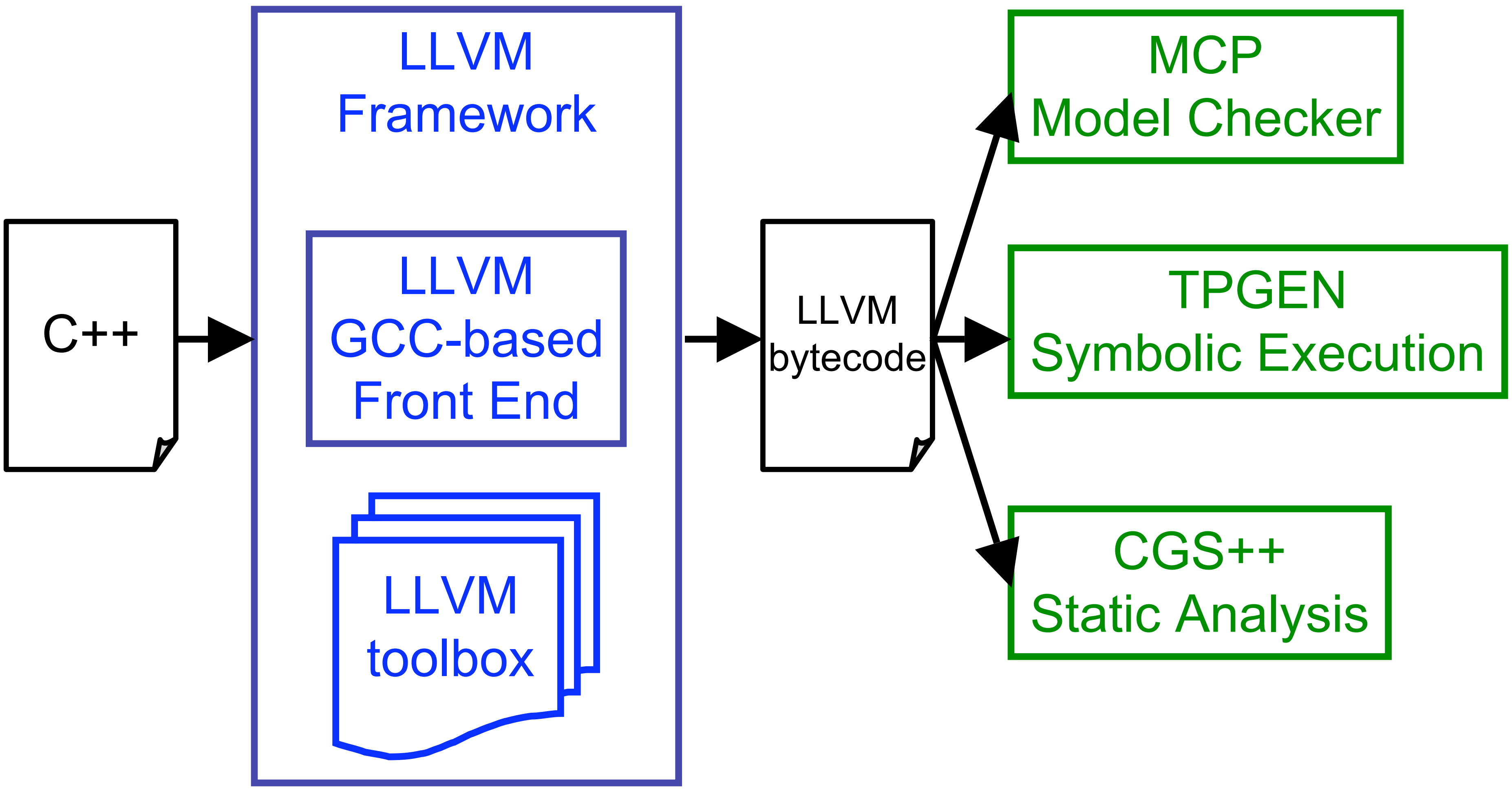
TPGEN
Symbolic Execution

CGS++
Static Analysis

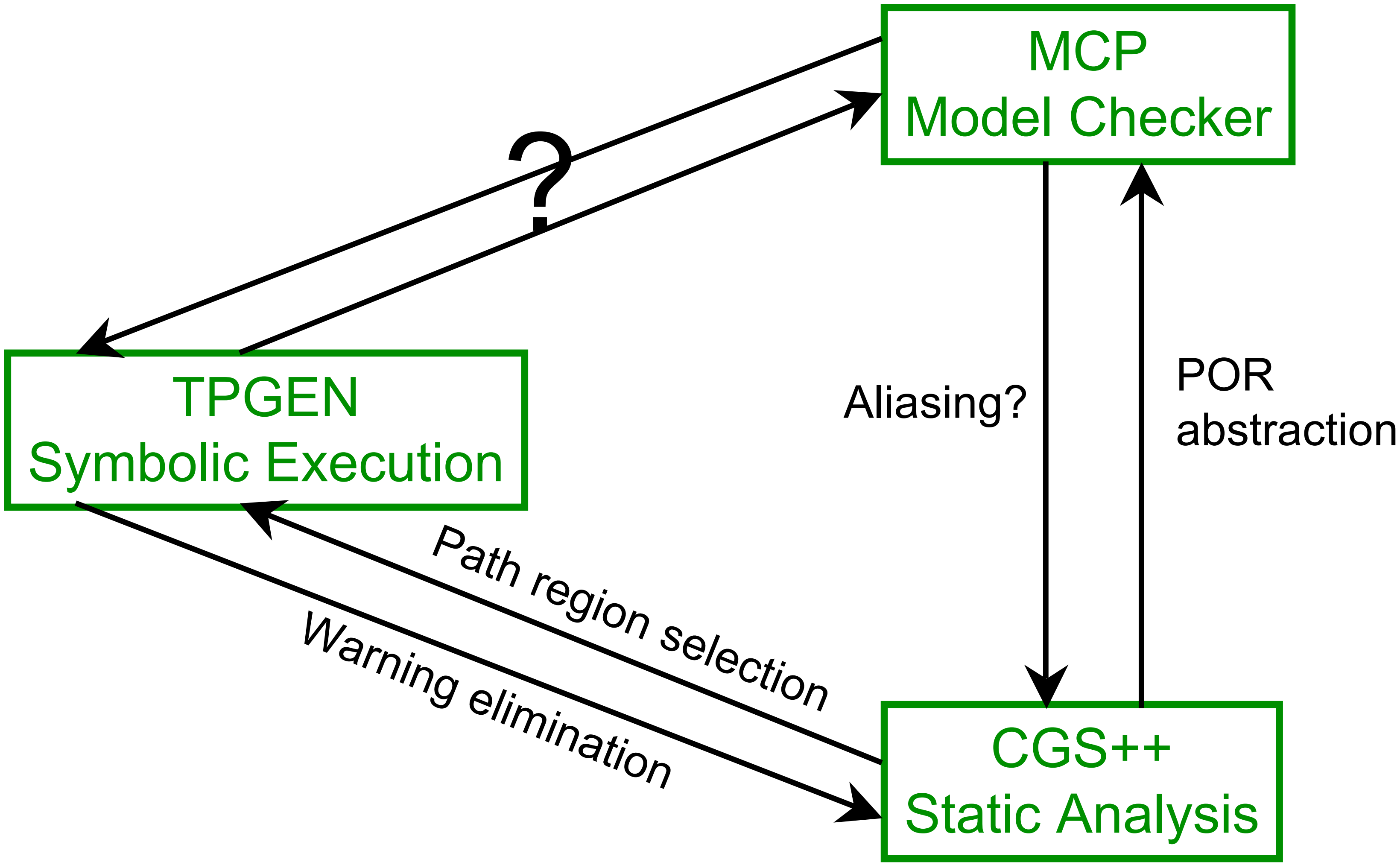
LLVM Compilation and Analysis Framework

- Open source
- Used by Apple for commercial products

C++ tool suite flow



Tool interactions



Conclusions

- Static analysis is useful for NASA software
  - Certifies the absence of errors
  - Does not require testing/simulation environment
- Static analysis is becoming practical
  - Scales to large software (e.g., MER)
  - Number of false positives is greatly reduced
  - Analysis times are less than a day even for large software
- CGS (developed at NASA Ames Research Center)
  - Catches pointer manipulation errors in embedded C programs
  - Is applicable to large flight software
- We are working on a suite of C++ analysis tools
  - Model checker
  - Symbolic execution
  - Static analysis