https://runtimeverification.com

runtime
verification

Technology and Products

Grigore Rosu
Founder, President and CEO
Professor of Computer Science, University of Illinois

Overview

- Runtime Verification
  - Company
    - Licensed by University of Illinois at Urbana-Champaign
  - Scientific Field
    - Co-pioneered with NASA colleagues and collaborators
- Products and Demos
  - RV-Match
  - RV-Predict
  - RV-Monitor
- Conclusion

The Company

Runtime Verification, Inc.

Description of Company

Runtime Verification, Inc. (RV): startup company aimed at bringing the best ideas and technology developed by the runtime verification community to the real world as mature and competitive products; licensed by the University of Illinois at Urbana-Champaign (UIUC), USA.

Mission: To offer the best possible solutions for reliable software development and analysis.

Computer Science @ UIUC

Ranked **top 5 in USA** (US News)

#1 in USA in Soft. Eng. (csrankings.org)

RV technology is licensed by UIUC

RV employees are former UIUC students

Technology

- *Runtime verification* is a new field aimed at verifying computing systems as they execute
  - Good scalability, rigorous, *no false alarms*
- We helped shape the field
  - Co-Founded the Runtime Verification conference
  - >100 RV publications
  - Raised $7M+ funding to develop technology

The Field

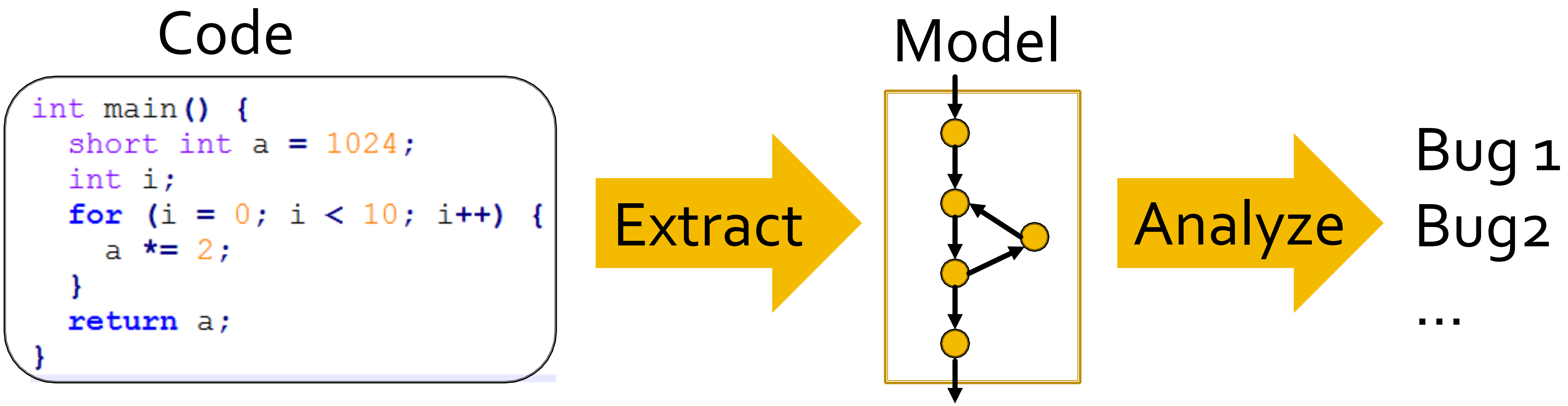
Runtime Verification

What is Runtime Verification (RV)?

- Subfield of program analysis and verification
  - So is static analysis (SA)
  - SA and RV *complement* each other
- Main idea of RV is different from that of SA:
  - *Execute* program to analyze
    - Using instrumentation or in a special runtime environment
  - *Observe* execution trace
  - *Build model* from execution trace
  - *Analyze* model

Steps above may be combined (e.g., online analysis)

Recall Static Analysis (including model-checking)



Code

```
int main() {
  short int a = 1024;
  int i;
  for (i = 0; i < 10; i++) {
    a *= 2;
  }
  return a;
}
```

Extract

Model

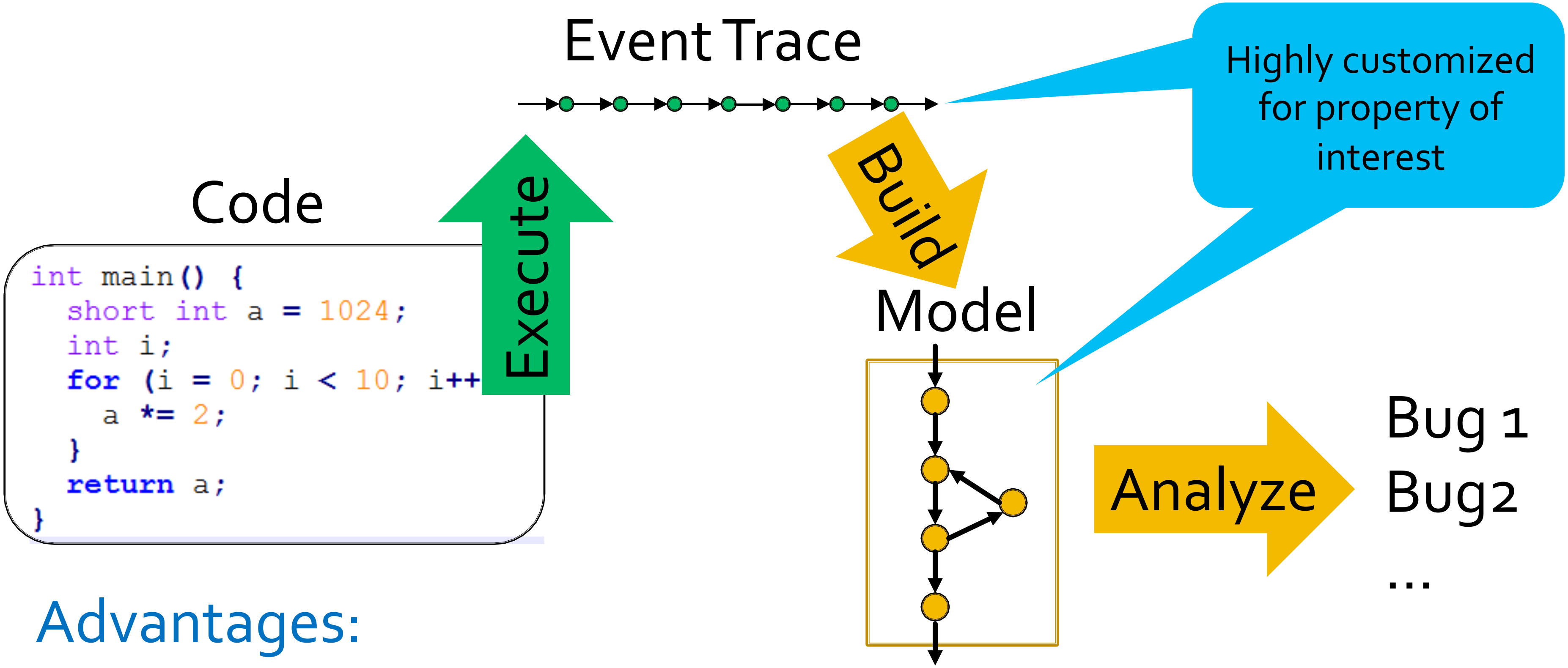
Analyze

Bug 1
Bug2
...

Advantages:
+ good code coverage
+ early in development
+ mature field

Limitations:
- undecidable problem, so
- false positives/negatives or
- does not scale

Runtime Verification



```
int main() {
  short int a = 1024;
  int i;
  for (i = 0; i < 10; i++
    a *= 2;
  }
  return a;
}
```

Highly customized for property of interest

Advantages:
+ precise (no false alarms)
+ good scalability and rigor
+ recovery possible

Limitations:
- code must be executable
- less code coverage

Addressing the Limitations

- Code must be executable
  - Use *complementary*, static analysis, earlier in process
  - Use symbolic execution (RV-Match)
- Less code coverage
  - *Integrate RV tools with your existing testing infrastructure*: your unit tests should already provide good code coverage; invoke RV tools on each test
  - Systematic re-execution: cover new code each time
  - Symbolic execution covers many inputs at once

The Products

Runtime Verification

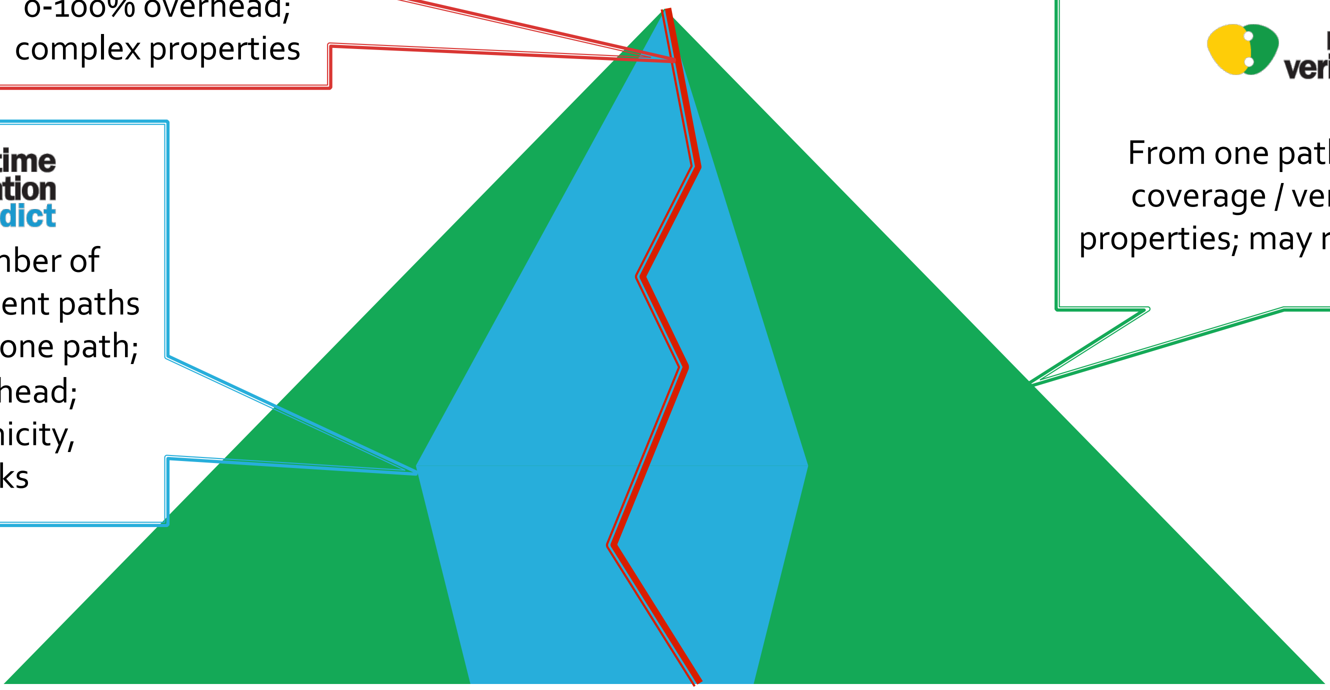
Runtime Verification Products https://runtimeverification.com

RV-Match is a semantics-based automatic debugger for common and subtle C errors, and an automatic dynamic checker for all types of ISO C11 undefinedness.
- C (mature); Java and JavaScript (prototypes)

RV-Predict is an automatic dynamic data-race detector for Java, which is sound (no false positives) and maximal (no other sound dynamic tool can find more races).
- Java (mature), C/C++ with interrupts (prototype)

RV-Monitor is a runtime monitoring tool that allows for checking and enforcement of safety properties over the execution of your software.
- Java (prototype), C/C++ (prototype)

Runtime Verification Products
Coverage vs. Performance vs. Expressiveness



Big triangle: all runtime behaviors of your program

runtime verification monitor

One path only;
0-100% overhead;
complex properties

runtime verification predict

Maximal number of causally equivalent paths predicted from one path;
$1\text{-}10^2$x overhead;
races, atomicity,
deadlocks

runtime verification match

From one path to complete coverage / verification; any properties; may require user input

Semantics-based runtime verification

RV-Match

RV-Match Overview

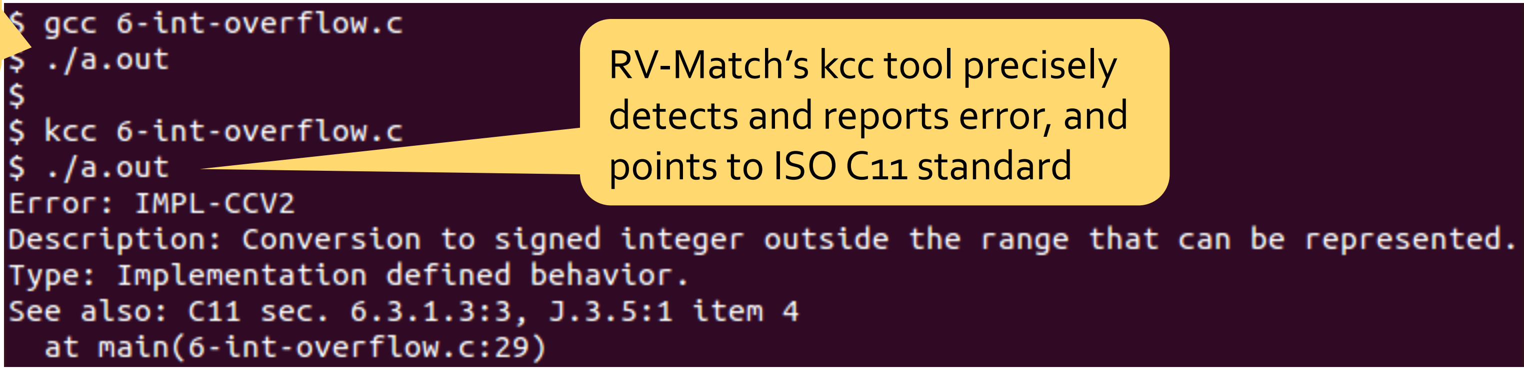
Code (6-int-overflow.c)

```
…
int main() {
  short int a = 1;
  int i;
  for (i = 0; i < 15; i++) {
    a *= 2;
  }
  return a;
}
```

RV-Match gives you:

- an automatic debugger for subtle bugs other tools can't find, with no false positives
- seamless integration with unit tests, build infrastructure, and continuous integration
- a platform for analyzing programs, boosting standards compliance and assurance

Conventional compilers do not detect problem

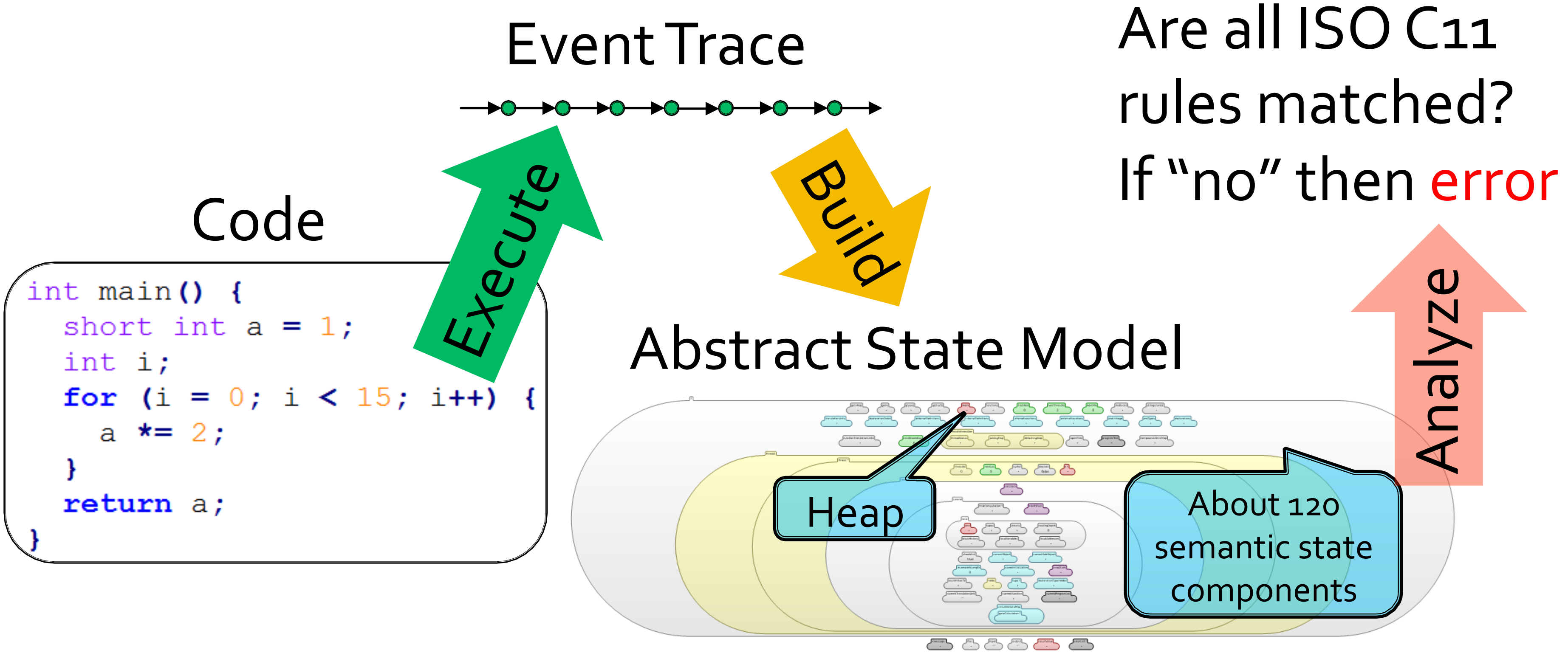


```
$ gcc 6-int-overflow.c
$ ./a.out
$
$ kcc 6-int-overflow.c
$ ./a.out
Error: IMPL-CCV2
Description: Conversion to signed integer outside the range that can be represented.
Type: Implementation defined behavior.
See also: C11 sec. 6.3.1.3:3, J.3.5:1 item 4
  at main(6-int-overflow.c:29)
```

RV-Match's kcc tool precisely detects and reports error, and points to ISO C11 standard

RV-Match Approach

1. Execute program within precise mathematical model of ISO C11
2. Build abstract program state model during execution
3. Analyze each event, performing consistency checks on state



Event Trace

Are all ISO C11 rules matched?

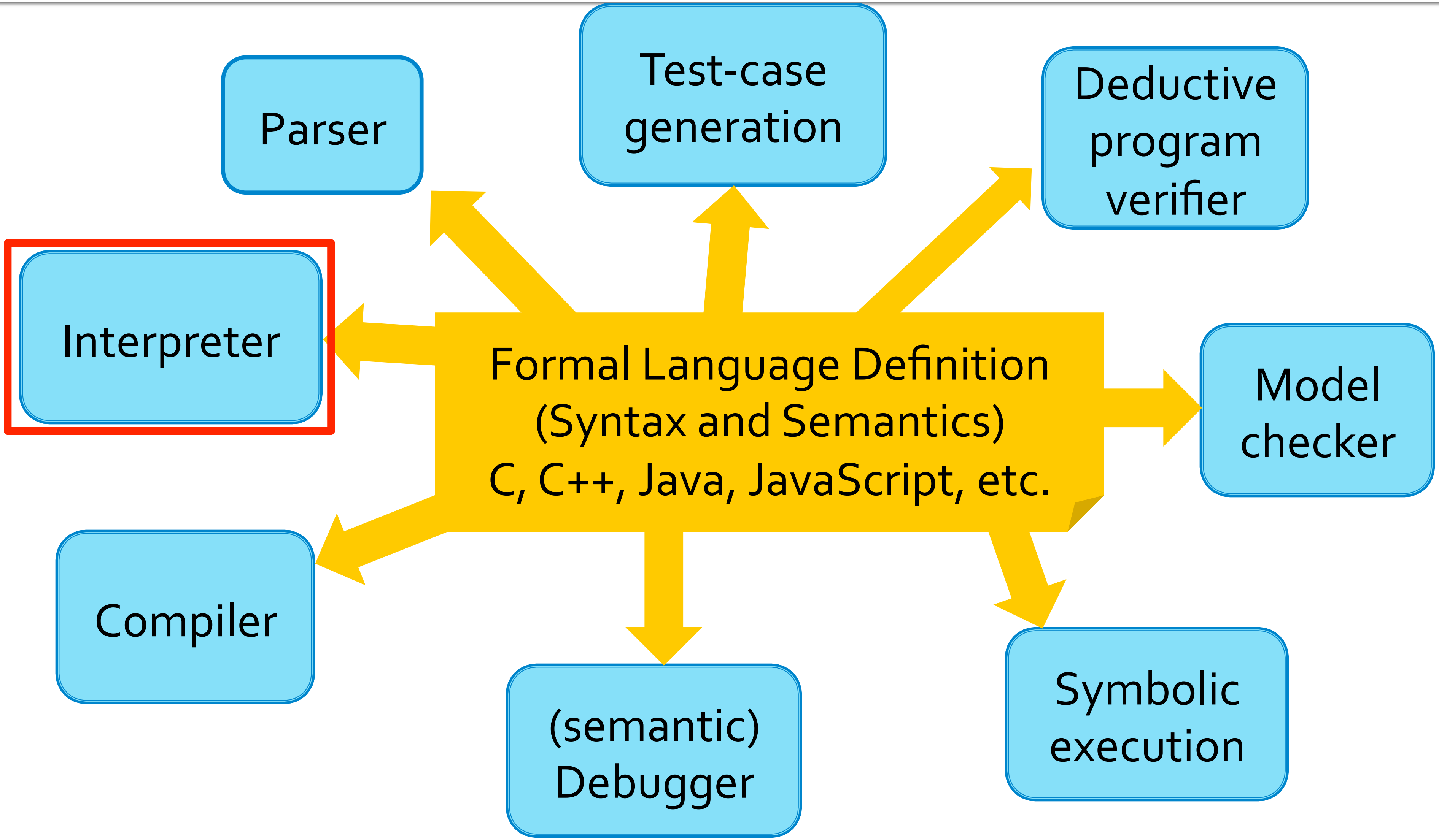
If "no" then error

Code

```
int main() {
  short int a = 1;
  int i;
  for (i = 0; i < 15; i++) {
    a *= 2;
  }
  return a;
}
```

Abstract State Model

RV-Match: Bigger Picture



Formal Language Definition
(Syntax and Semantics)
C, C++, Java, JavaScript, etc.

Parser

Test-case generation

Deductive program verifier

Interpreter

Model checker

Compiler

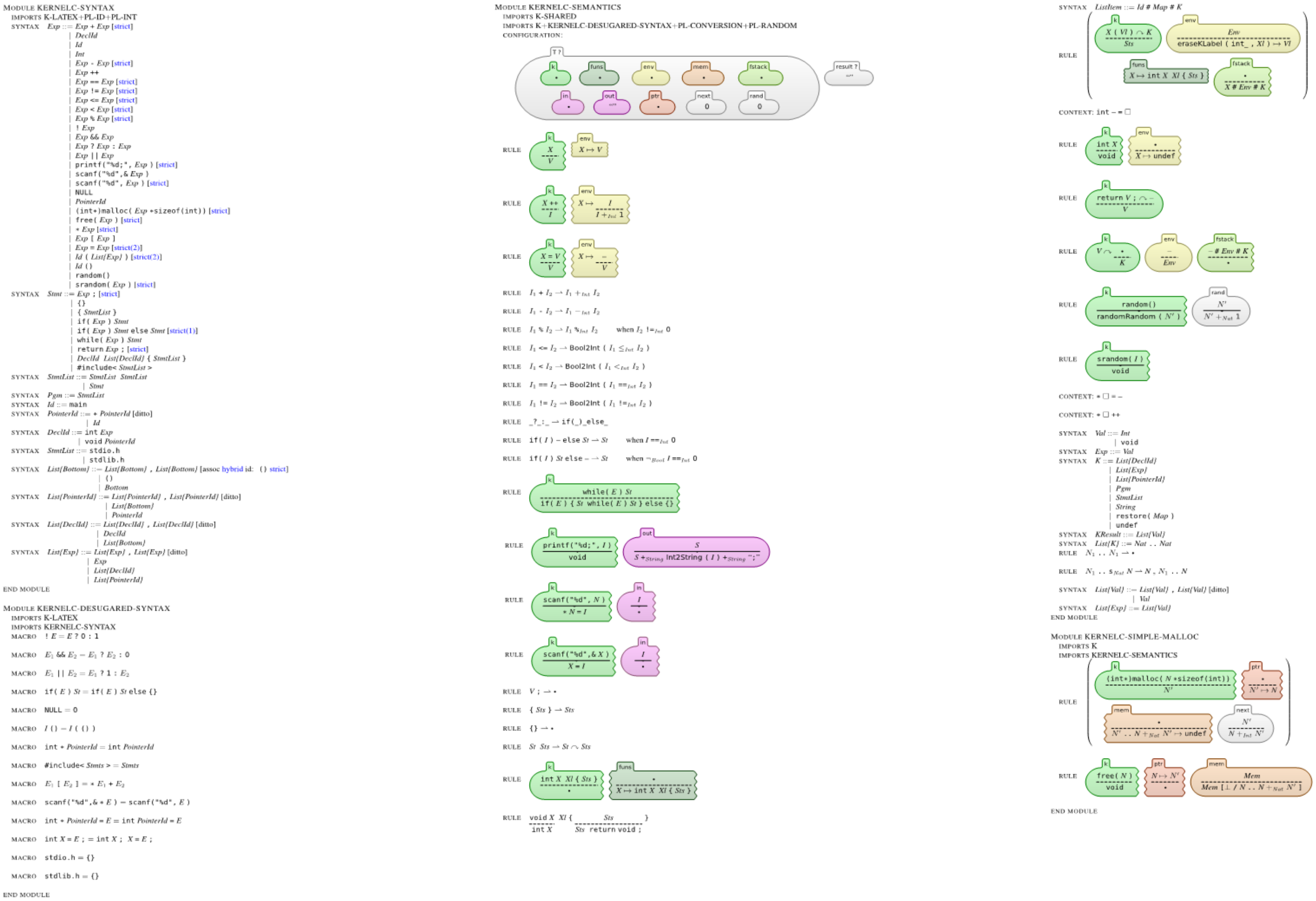
(semantic) Debugger

Symbolic execution

Formal Language Definitions

- To define programming languages formally, we use the academic K tool and notation
  - http://kframework.org
  - Developed in the Formal Systems Laboratory (my research group) at the University of Illinois
  - Open source

Complete K Definition of KernelC



MODULE KERNELC-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX *Exp* ::= *Exp* + *Exp* [strict]
| *DeclId*
| *Id*
| *Int*
| *Exp* - *Exp* [strict]
| *Exp* ++
| *Exp* == *Exp* [strict]
| *Exp* != *Exp* [strict]
| *Exp* <= *Exp* [strict]
| *Exp* < *Exp* [strict]
| *Exp* % *Exp* [strict]
| ! *Exp*
| *Exp* && *Exp*
| *Exp* ? *Exp* : *Exp*
| *Exp* || *Exp*
| `printf("%d;",` *Exp* `)` [strict]
| `scanf("%d",&` *Exp* `)`
| `scanf("%d",` *Exp* `)` [strict]
| `NULL`
| *PointerId*
| `(int*)malloc(` *Exp* `*sizeof(int))` [strict]
| `free(` *Exp* `)` [strict]
| * *Exp* [strict]
| *Exp* [*Exp*]
| *Exp* = *Exp* [strict(2)]
| *Id* (*List{Exp}*) [strict(2)]
| *Id* ()
| `random()`
| `srandom(` *Exp* `)` [strict]
SYNTAX *Stmt* ::= *Exp* ; [strict]
| {}
| { *StmtList* }
| `if(` *Exp* `)` *Stmt*
| `if(` *Exp* `)` *Stmt* `else` *Stmt* [strict(1)]
| `while(` *Exp* `)` *Stmt*
| `return` *Exp* ; [strict]
| *DeclId* *List{DeclId}* { *StmtList* }
| `#include<` *StmtList* >
SYNTAX *StmtList* ::= *StmtList* *StmtList*
| *Stmt*
SYNTAX *Pgm* ::= *StmtList*
SYNTAX *Id* ::= `main`
SYNTAX *PointerId* ::= * *PointerId* [ditto]
| *Id*
SYNTAX *DeclId* ::= `int` *Exp*
| `void` *PointerId*
SYNTAX *StmtList* ::= `stdio.h`
| `stdlib.h`
SYNTAX *List{Bottom}* ::= *List{Bottom}* , *List{Bottom}* [assoc hybrid id: () strict]
| ()
| *Bottom*
SYNTAX *List{PointerId}* ::= *List{PointerId}* , *List{PointerId}* [ditto]
| *List{Bottom}*
| *PointerId*
SYNTAX *List{DeclId}* ::= *List{DeclId}* , *List{DeclId}* [ditto]
| *DeclId*
| *List{Bottom}*
SYNTAX *List{Exp}* ::= *List{Exp}* , *List{Exp}* [ditto]
| *Exp*
| *List{DeclId}*
| *List{PointerId}*
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
IMPORTS K-LATEX
IMPORTS KERNELC-SYNTAX
MACRO ! E = E ? 0 : 1

MACRO E_1 && E_2 = E_1 ? E_2 : 0

MACRO E_1 || E_2 = E_1 ? 1 : E_2

MACRO `if(` E `)` *St* = `if(` E `)` *St* `else {}`

MACRO `NULL` = 0

MACRO I () = I (())

MACRO `int` * *PointerId* = `int` *PointerId*

MACRO `#include<` *Stmts* > = *Stmts*

MACRO E_1 [E_2] = * E_1 + E_2

MACRO `scanf("%d",&` * E `)` = `scanf("%d",` E `)`

MACRO `int` * *PointerId* = E = `int` *PointerId* = E

MACRO `int` X = E ; = `int` X ; X = E ;

MACRO `stdio.h` = {}

MACRO `stdlib.h` = {}

END MODULE

MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:

$$\overset{\text{T ?}}{\boxed{\overset{\text{k}}{\boxed{\cdot}}\ \overset{\text{funs}}{\boxed{\cdot}}\ \overset{\text{env}}{\boxed{\cdot}}\ \overset{\text{mem}}{\boxed{\cdot}}\ \overset{\text{fstack}}{\boxed{\cdot}}\ \overset{\text{in}}{\boxed{\cdot}}\ \overset{\text{out}}{\boxed{\text{""}}}\ \overset{\text{ptr}}{\boxed{\cdot}}\ \overset{\text{next}}{\boxed{0}}\ \overset{\text{rand}}{\boxed{0}}}}\quad \overset{\text{result ?}}{\boxed{\text{""}}}$$

RULE $\overset{\text{k}}{\boxed{\frac{X}{V}}}\ \overset{\text{env}}{\boxed{X \mapsto V}}$

RULE $\overset{\text{k}}{\boxed{\frac{X\,\texttt{++}}{I}}}\ \overset{\text{env}}{\boxed{X \mapsto \frac{I}{I +_{Int} 1}}}$

RULE $\overset{\text{k}}{\boxed{\frac{X = V}{V}}}\ \overset{\text{env}}{\boxed{X \mapsto \frac{\_}{V}}}$

RULE $I_1 + I_2 \rightharpoonup I_1 +_{Int} I_2$

RULE I_1 - $I_2 \rightharpoonup I_1 -_{Int} I_2$

RULE I_1 % $I_2 \rightharpoonup I_1$ $\%_{Int}$ I_2 when I_2 $!=_{Int}$ 0

RULE I_1 <= $I_2 \rightharpoonup$ Bool2Int ($I_1 \leq_{Int} I_2$)

RULE I_1 < $I_2 \rightharpoonup$ Bool2Int ($I_1 <_{Int} I_2$)

RULE I_1 == $I_2 \rightharpoonup$ Bool2Int (I_1 $==_{Int}$ I_2)

RULE I_1 != $I_2 \rightharpoonup$ Bool2Int (I_1 $!=_{Int}$ I_2)

RULE _?_:_ $\rightharpoonup$ `if(_)_else_`

RULE `if(` I `)` _ `else` *St* $\rightharpoonup$ *St* when I $==_{Int}$ 0

RULE `if(` I `)` *St* `else` _ $\rightharpoonup$ *St* when $\neg_{Bool}$ I $==_{Int}$ 0

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{while(}\ E\ \texttt{)}\ St}{\texttt{if(}\ E\ \texttt{) \{}\ St\ \texttt{while(}\ E\ \texttt{)}\ St\ \texttt{\} else \{\}}}}}$

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{printf("\%d;",}\ I\ \texttt{)}}{\texttt{void}}}}\ \overset{\text{out}}{\boxed{\frac{S}{S +_{String} \text{Int2String (} I \text{) } +_{String} \text{";"}}}}$

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{scanf("\%d",}\ N\ \texttt{)}}{*\ N = I}}}\ \overset{\text{in}}{\boxed{\frac{I}{\cdot}}}$

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{scanf("\%d",\&}\ X\ \texttt{)}}{X = I}}}\ \overset{\text{in}}{\boxed{\frac{I}{\cdot}}}$

RULE V ; $\rightharpoonup \cdot$

RULE { *Sts* } $\rightharpoonup$ *Sts*

RULE {} $\rightharpoonup \cdot$

RULE *St* *Sts* $\rightharpoonup$ *St* $\curvearrowright$ *Sts*

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{int}\ X\ \ Xl\ \texttt{\{}\ Sts\ \texttt{\}}}{\cdot}}}\ \overset{\text{funs}}{\boxed{\frac{\cdot}{X \mapsto \texttt{int}\ X\ \ Xl\ \texttt{\{}\ Sts\ \texttt{\}}}}}$

RULE $\underset{\texttt{int}\ X}{\underline{\texttt{void}\ X}}\ \ Xl\ \texttt{\{}\ \underset{Sts\ \texttt{return void ;}}{\underline{Sts}}\ \texttt{\}}$

SYNTAX *ListItem* ::= *Id* # *Map* # *K*

RULE $\left(\overset{\text{k}}{\boxed{\frac{X\ (\ Vl\ ) \curvearrowright K}{Sts}}}\ \overset{\text{env}}{\boxed{\frac{Env}{\text{eraseKLabel (int\_ , } Xl \text{)} \mapsto Vl}}}\ \overset{\text{funs}}{\boxed{X \mapsto \texttt{int}\ X\ \ Xl\ \texttt{\{}\ Sts\ \texttt{\}}}}\ \overset{\text{fstack}}{\boxed{\frac{\cdot}{X\ \#\ Env\ \#\ K}}}\right)$

CONTEXT: `int` – = □

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{int}\ X}{\texttt{void}}}}\ \overset{\text{env}}{\boxed{\frac{\cdot}{X \mapsto \texttt{undef}}}}$

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{return}\ V\ \texttt{;} \curvearrowright \_}{V}}}$

RULE $\overset{\text{k}}{\boxed{V \curvearrowright \frac{\cdot}{K}}}\ \overset{\text{env}}{\boxed{\frac{\_}{Env}}}\ \overset{\text{fstack}}{\boxed{\frac{\_\ \#\ Env\ \#\ K}{\cdot}}}$

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{random()}}{\texttt{randomRandom (}\ N'\ \texttt{)}}}}\ \overset{\text{rand}}{\boxed{\frac{N'}{N' +_{Nat} 1}}}$

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{srandom(}\ I\ \texttt{)}}{\texttt{void}}}}$

CONTEXT: * □ = –

CONTEXT: * □ ++

SYNTAX *Val* ::= *Int*
| `void`
SYNTAX *Exp* ::= *Val*
SYNTAX *K* ::= *List{DeclId}*
| *List{Exp}*
| *List{PointerId}*
| *Pgm*
| *StmtList*
| *String*
| `restore(` *Map* `)`
| `undef`
SYNTAX *KResult* ::= *List{Val}*
SYNTAX *List{K}* ::= *Nat* .. *Nat*
RULE N_1 .. $N_1 \rightharpoonup \cdot$

RULE N_1 .. s_{Nat} $N \rightharpoonup N$, N_1 .. N

SYNTAX *List{Val}* ::= *List{Val}* , *List{Val}* [ditto]
| *Val*
SYNTAX *List{Exp}* ::= *List{Val}*
END MODULE

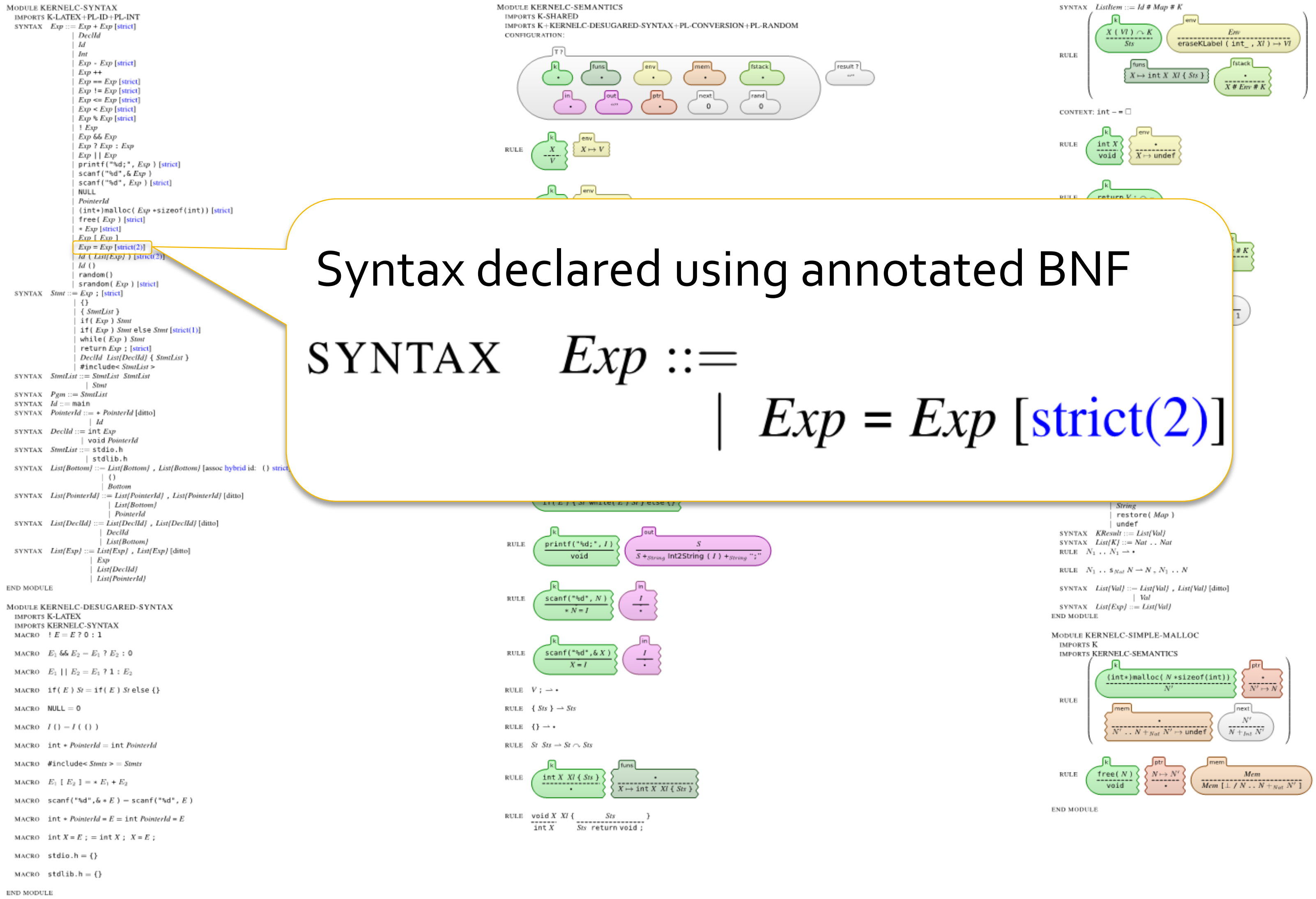
MODULE KERNELC-SIMPLE-MALLOC
IMPORTS K
IMPORTS KERNELC-SEMANTICS

RULE $\left(\overset{\text{k}}{\boxed{\frac{\texttt{(int*)malloc(}\ N\ \texttt{*sizeof(int))}}{N'}}}\ \overset{\text{ptr}}{\boxed{\frac{\cdot}{N' \mapsto N}}}\ \overset{\text{mem}}{\boxed{\frac{\cdot}{N'\ ..\ N +_{Nat} N' \mapsto \texttt{undef}}}}\ \overset{\text{next}}{\boxed{\frac{N'}{N +_{Int} N'}}}\right)$

RULE $\overset{\text{k}}{\boxed{\frac{\texttt{free(}\ N\ \texttt{)}}{\texttt{void}}}}\ \overset{\text{ptr}}{\boxed{\frac{N \mapsto N'}{\cdot}}}\ \overset{\text{mem}}{\boxed{\frac{Mem}{Mem\ [\bot\ /\ N\ ..\ N +_{Nat} N'\ ]}}}$

END MODULE

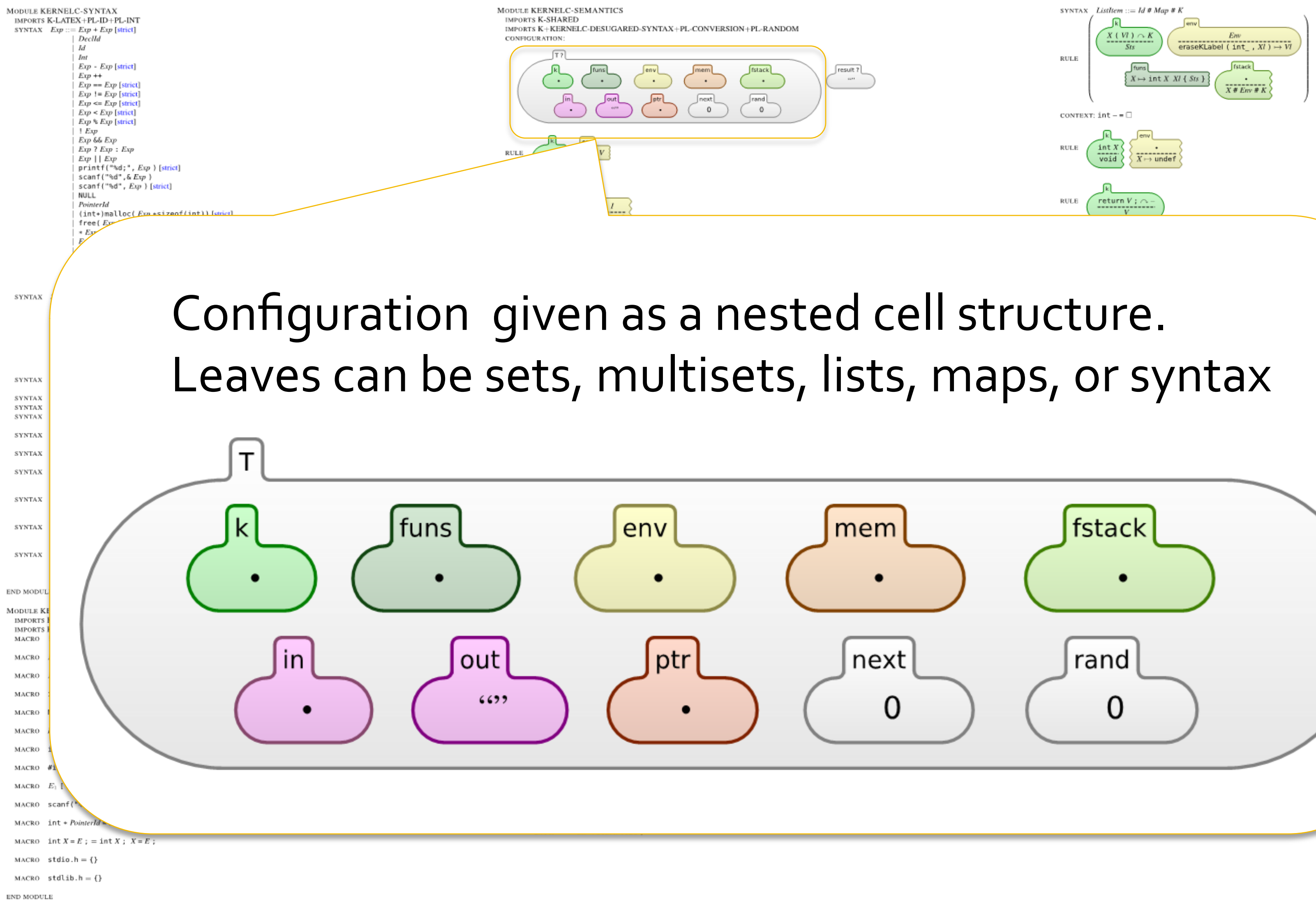
Complete K Definition of KernelC



Syntax declared using annotated BNF

$$\text{SYNTAX} \quad Exp ::= \\ \quad | \quad Exp = Exp \; [\text{strict(2)}]$$

Complete K Definition of KernelC



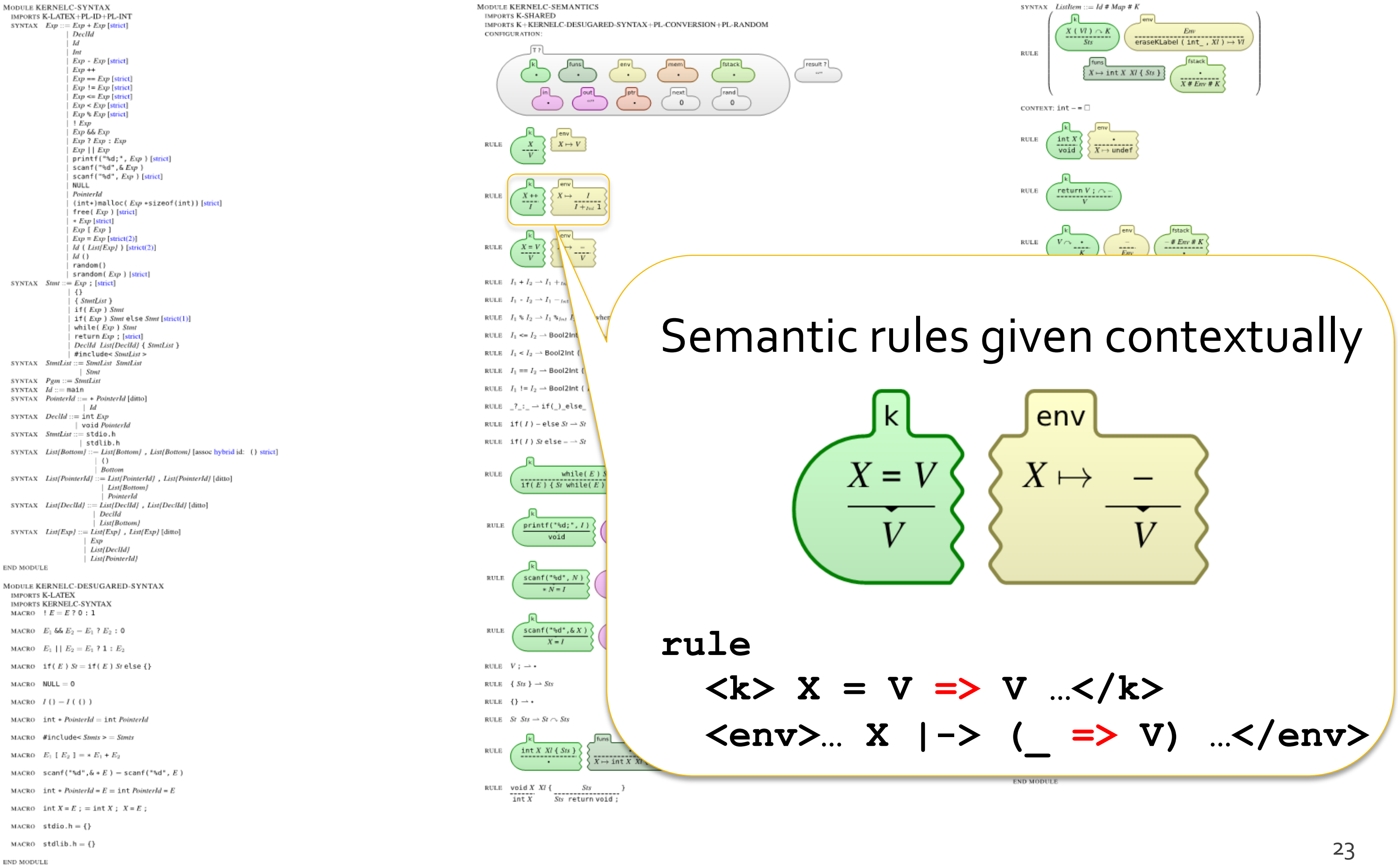
```
MODULE KERNELC-SYNTAX
  IMPORTS K-LATEX+PL-ID+PL-INT
  SYNTAX  Exp ::= Exp + Exp [strict]
                | DeclId
                | Id
                | Int
                | Exp - Exp [strict]
                | Exp ++
                | Exp == Exp [strict]
                | Exp != Exp [strict]
                | Exp <= Exp [strict]
                | Exp < Exp [strict]
                | Exp % Exp [strict]
                | ! Exp
                | Exp && Exp
                | Exp ? Exp : Exp
                | Exp || Exp
                | printf("%d;", Exp ) [strict]
                | scanf("%d",& Exp )
                | scanf("%d", Exp ) [strict]
                | NULL
                | PointerId
                | (int*)malloc( Exp *sizeof(int)) [strict]
                | free( Exp
                | * Exp
                | E
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
  SYNTAX
END MODUL

MODULE KE
  IMPORTS
  IMPORTS
  MACRO
  MACRO
  MACRO
  MACRO
  MACRO
  MACRO
  MACRO
  MACRO  #i
  MACRO  E1 [
  MACRO  scanf("
  MACRO  int * PointerId
  MACRO  int X = E ; = int X ;  X = E ;
  MACRO  stdio.h = {}
  MACRO  stdlib.h = {}
END MODULE
```

```
MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
  RULE
```

```
SYNTAX  ListItem ::= Id # Map # K
RULE
CONTEXT: int − = □
RULE
RULE
```

Configuration given as a nested cell structure.
Leaves can be sets, multisets, lists, maps, or syntax

Complete K Definition of KernelC



```
MODULE KERNELC-SYNTAX
  IMPORTS K-LATEX+PL-ID+PL-INT
  SYNTAX   Exp ::= Exp + Exp [strict]
                 | DeclId
                 | Id
                 | Int
                 | Exp - Exp [strict]
                 | Exp ++
                 | Exp == Exp [strict]
                 | Exp != Exp [strict]
                 | Exp <= Exp [strict]
                 | Exp < Exp [strict]
                 | Exp % Exp [strict]
                 | ! Exp
                 | Exp && Exp
                 | Exp ? Exp : Exp
                 | Exp || Exp
                 | printf("%d;", Exp ) [strict]
                 | scanf("%d",& Exp )
                 | scanf("%d", Exp ) [strict]
                 | NULL
                 | PointerId
                 | (int*)malloc( Exp *sizeof(int)) [strict]
                 | free( Exp ) [strict]
                 | * Exp [strict]
                 | Exp [ Exp ]
                 | Exp = Exp [strict(2)]
                 | Id ( List{Exp} ) [strict(2)]
                 | Id ()
                 | random()
                 | srandom( Exp ) [strict]
  SYNTAX   Stmt ::= Exp ; [strict]
                  | {}
                  | { StmtList }
                  | if( Exp ) Stmt
                  | if( Exp ) Stmt else Stmt [strict(1)]
                  | while( Exp ) Stmt
                  | return Exp ; [strict]
                  | DeclId  List{DeclId} { StmtList }
                  | #include< StmtList >
  SYNTAX   StmtList ::= StmtList  StmtList
                      | Stmt
  SYNTAX   Pgm ::= StmtList
  SYNTAX   Id ::= main
  SYNTAX   PointerId ::= * PointerId [ditto]
                       | Id
  SYNTAX   DeclId ::= int Exp
                    | void PointerId
  SYNTAX   StmtList ::= stdio.h
                      | stdlib.h
  SYNTAX   List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id:  () strict]
                          | ()
                          | Bottom
  SYNTAX   List{PointerId} ::= List{PointerId} , List{PointerId} [ditto]
                             | List{Bottom}
                             | PointerId
  SYNTAX   List{DeclId} ::= List{DeclId} , List{DeclId} [ditto]
                          | DeclId
                          | List{Bottom}
  SYNTAX   List{Exp} ::= List{Exp} , List{Exp} [ditto]
                       | Exp
                       | List{DeclId}
                       | List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS K-LATEX
  IMPORTS KERNELC-SYNTAX
  MACRO   ! E = E ? 0 : 1

  MACRO   E1 && E2 = E1 ? E2 : 0

  MACRO   E1 || E2 = E1 ? 1 : E2

  MACRO   if( E ) St = if( E ) St else {}

  MACRO   NULL = 0

  MACRO   I () = I ( () )

  MACRO   int * PointerId = int PointerId

  MACRO   #include< Stmts > = Stmts

  MACRO   E1 [ E2 ] = * E1 + E2

  MACRO   scanf("%d",& * E ) = scanf("%d", E )

  MACRO   int * PointerId = E = int PointerId = E

  MACRO   int X = E ; = int X ;  X = E ;

  MACRO   stdio.h = {}

  MACRO   stdlib.h = {}

END MODULE
```

```
MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
```

Semantic rules given contextually

```
rule
   <k> X = V => V …</k>
   <env>… X |-> (_ => V) …</env>
```

K Scales

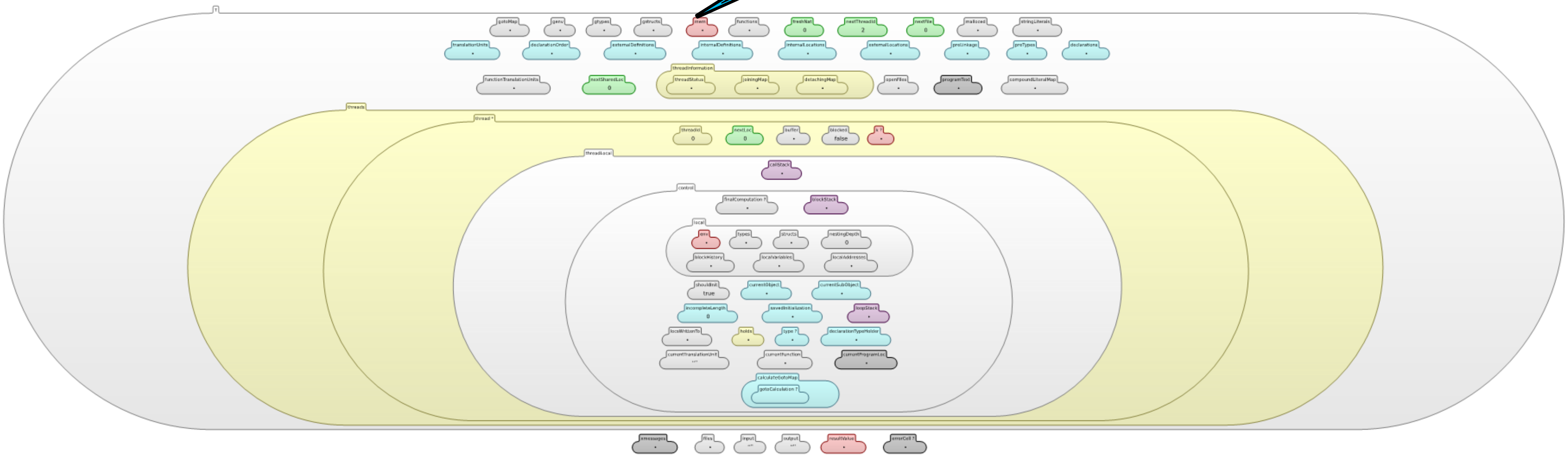
Several large languages were recently defined in K:

- Java 1.4: by Bogdanas etal [POPL'15]
  - 800+ program test suite that covers the semantics
- JavaScript ES5: by Park etal [PLDI'15]
  - Passes existing conformance test suite (2872 pgms)
  - Found (confirmed) bugs in Chrome, IE, Firefox, Safari
- C11: Ellison etal [POPL'12, PLDI'15]
  - It defines the ISO C11 standard
  - Including all *undefined behaviors*

…

K Configuration and Definition of C

Heap



120 Cells!

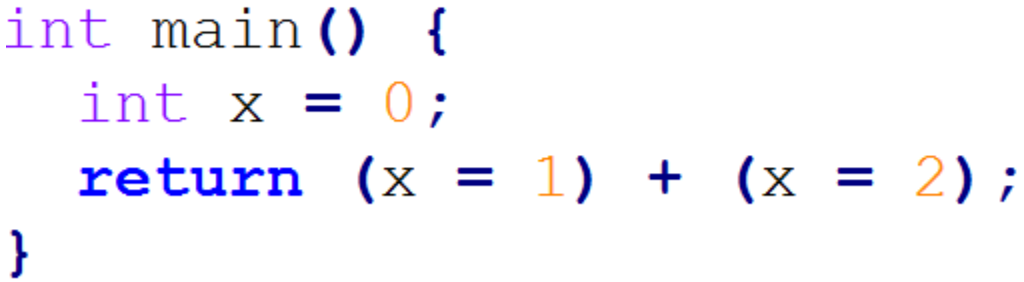
... plus ~2500 rules ...

Advantages of RV-Match Approach

- No need to re-implement tools as language changes
  - Easy to customize tools
    - E.g., embedded C for a specific micro-controller
  - Programming languages continuously evolve (C90 → C99 → C11 → …; or Java 1.4 → Java 5 → … → Java 8 → …)
- Tools are correct by construction
  - Tools are language-independent and can produce correctness certificates based on language semantics only
  - Language definitions are open-source and public
    - Experts worldwide can “validate” them
    - No developer “interpretation” of language meaning (e.g., C)

Does it Work?

- Let’s use RV-Match with (extended) C11 semantics
- Goal: catch undefined behavior!
  - You should always avoid undefined behavior in your code!!!
  - Undefined behavior → lack of portability, security risks, non-determinism
- Wrapped RV-Match[C11] as an ICO C11 compliant drop-in replacement of C compilers (e.g., `gcc`), called `kcc`
- Example: what does the following return?

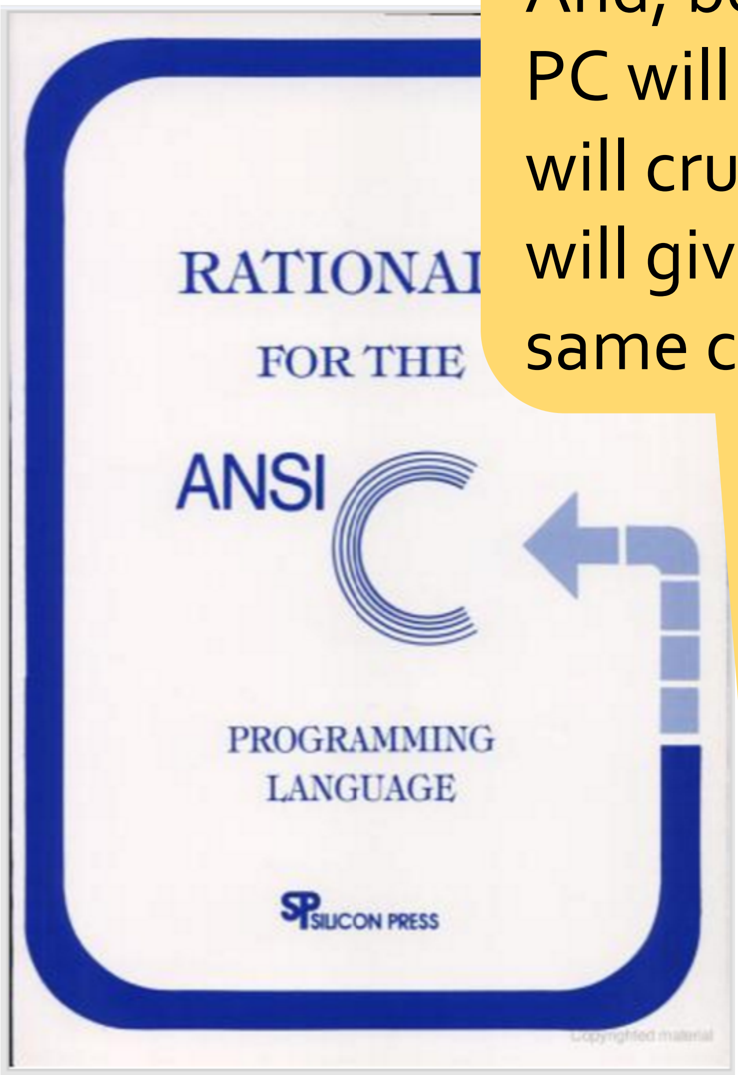


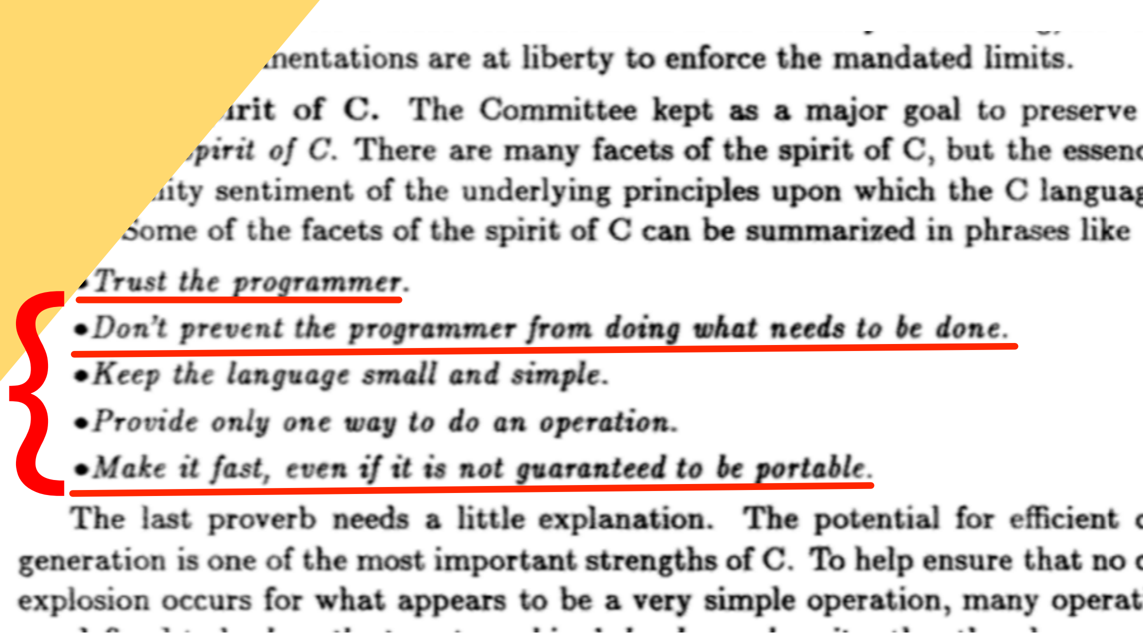
```
int main() {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

4 with `gcc`
3 with `clang` (LLVM)
ISO C11: **undefined!**
`kcc` reports **error**

Why Undefined Behavior Matters?

And, because of that, your code tested on PC will not port on embedded platform, will crush when you change compiler, and will give you different results with even the same compiler but different options …





nentations are at liberty to enforce the mandated limits.

irit of C. The Committee kept as a major goal to preserve *pirit of C*. There are many facets of the spirit of C, but the essenc nity sentiment of the underlying principles upon which the C languag Some of the facets of the spirit of C can be summarized in phrases like

- *Trust the programmer.*
- *Don't prevent the programmer from doing what needs to be done.*
- *Keep the language small and simple.*
- *Provide only one way to do an operation.*
- *Make it fast, even if it is not guaranteed to be portable.*

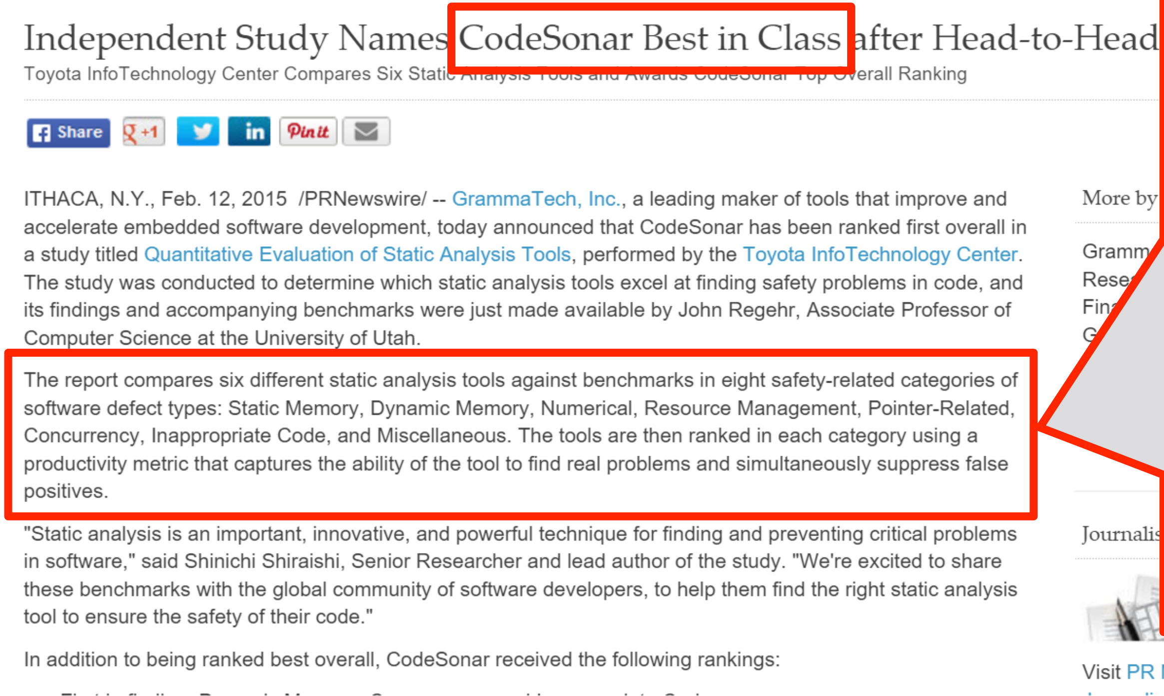
The last proverb needs a little explanation. The potential for efficient c generation is one of the most important strengths of C. To help ensure that no c explosion occurs for what appears to be a very simple operation, many operat

RV-Match DEMO

- Go to https://runtimeverification.com/match to download RV-Match (currently only C11 version available); kcc and then execute the C programs under `examples/demo` in the given order
  - Most of the examples above are also discussed, with detailed comments, at
  https://runtimeverification.com/match/docs/runningexamples
- You can also run the Toyota ITC benchmark: https://runtimeverification.com/match/docs/benchmark

Does it *Really* Work? Let's Evaluate it!

- Evaluated RV-Match on the *Toyota ITC benchmark*, aimed at quantitatively evaluating static analysis t
  - By Shin'ichi Shiraishi and collaborators
  - ISSRE'14 original paper, compared six tools; pa
  - Press release by Grammatech, available at PRN



Independent Study Names CodeSonar Best in Class after Head-to-Head

Toyota InfoTechnology Center Compares Six Static Analysis Tools and Awards CodeSonar Top Overall Ranking

ITHACA, N.Y., Feb. 12, 2015 /PRNewswire/ -- GrammaTech, Inc., a leading maker of tools that improve and accelerate embedded software development, today announced that CodeSonar has been ranked first overall in a study titled Quantitative Evaluation of Static Analysis Tools, performed by the Toyota InfoTechnology Center. The study was conducted to determine which static analysis tools excel at finding safety problems in code, and its findings and accompanying benchmarks were just made available by John Regehr, Associate Professor of Computer Science at the University of Utah.

The report compares six different static analysis tools against benchmarks in eight safety-related categories of software defect types: Static Memory, Dynamic Memory, Numerical, Resource Management, Pointer-Related, Concurrency, Inappropriate Code, and Miscellaneous. The tools are then ranked in each category using a productivity metric that captures the ability of the tool to find real problems and simultaneously suppress false positives.

"Static analysis is an important, innovative, and powerful technique for finding and preventing critical problems in software," said Shinichi Shiraishi, Senior Researcher and lead author of the study. "We're excited to share these benchmarks with the global community of software developers, to help them find the right static analysis tool to ensure the safety of their code."

In addition to being ranked best overall, CodeSonar received the following rankings:

More by

Gramm
Rese
Fin
G

Journalis

Visit PR Newswire for

... report compares six different static analysis tools against benchmarks in eight safety-related categories of software defect types: Static Memory, Dynamic Memory, Numerical, Resource Management, Pointer-Related, Concurrency, ...

Toyota ITC Benchmark Paper
- Static Analysis Tools -

Shiraishi etal published revised version in *ISSRE 2015*

1276 programs; 3 static analysis tools compared

- Grammatech CodeSonar wins again (numbers below from ISSRE'15 paper)

Shiraishi et al., ISSRE '15	**RV-Match**			**GrammaTech CodeSonar**			**MathWorks Code Prover**			**MathWorks Bug Finder**			**GCC**			**Clang**		
				DR	**$\underline{\text{FPR}}$**	**PM**	**DR**	**$\underline{\text{FPR}}$**	**PM**	**DR**	**$\underline{\text{FPR}}$**	**PM**	**DR**	**$\underline{\text{FPR}}$**	**PM**	**DR**	**$\underline{\text{FPR}}$**	**PM**
Static memory				100	100	100	97	100	98	97	100	98	0	100	0	15	100	39
Dynamic memory				89	100	94	92	95	93	90	100	95	0	100	0	0	100	0
Stack-related				0	100	0	60	70	65	15	85	36	0	100	0	0	100	0
Numerical		?		48	100	69	55	99	74	41	100	64	12	10				33
Resource management				61	100	78	20	90	42	55	100	74	6	10				18
Pointer-related				52	96	71	69	93	80	69	100	83	9	10				36
Concurrency				70	77	73	0	100	0	0	100	0	0	100		0	0	0
Inappropriate code				46	99	67	1	97	10	28	94	51	2	100	1	0	0	0
Miscellaneous				69	100	83	83	100	91	69	100	83	11	100	34	11	10	34
AVERAGE (Unweighted)				59	97	76	53	94	71	52	98	71	4	100	20	6	100	24
AVERAGE (Weighted)				68	98	82	53	95	71	62	99	78	5	100	22	7	100	26

What you get for free

DR: Percent of programs with defects where defects are reported

FPR: Percent of programs without defects, with defects incorrectly reported; $\underline{\text{FPR}} = 100 - \text{FPR}$

PM: Productivity metric: $\sqrt{\text{DR}\times(100-\text{FPR})}$

RV-Match on Toyota ITC Benchmark
- Comparison with Static Analysis Tools -

- We do not have semantics for “inappropriate code” yet
- We miss defects because inherent limited code coverage of RV
  - No false positives for RV-Match!

Shiraishi et al., ISSRE ’15	RV-Match			GrammaTech CodeSonar			MathWorks Code Prover			MathWorks Bug Finder			GCC			Clang		
	DR	FPR	PM	DR	FPR	PM	DR	FPR	PM	DR	FPR	PM	DR	FPR	PM	DR	FPR	PM
Static memory	100	100	100	100	100	100	97	100	98	97	100	98	0	100	0	15	100	39
Dynamic memory	94	100	97	89	100	94	92	95	93	90	100	95	0	100	0	0	100	0
Stack-related	100	100	100	0	100	0	60	70	65	15	85	36	0	100	0	0	100	0
Numerical	96	100	98	48	100	69	55	99	74	41	100	64	12	100	35	11	100	33
Resource management	93	100	96	61	100	78	20	90	42	55	100	74	6	100	25	3	100	18
Pointer-related	98	100	99	52	96	71	69	93	80	69	100	83	9	100	30	13	100	36
Concurrency	67	100	82	70	77	73	0	100	0	0	100	0	0	100	0	0	100	0
Inappropriate code	0	100	0	46	99	67	1	97	10	28	94	51	2	100	13	0	100	0
Miscellaneous	63	100	79	69	100	83	83	100	91	69	100	83	11	100	34	11	100	34
AVERAGE (Unweighted)	79	100	89	59	97	76	53	94	71	52	98	71	4	100	20	6	100	24
AVERAGE (Weighted)	82	100	91	68	98	82	53	95	71	62	99	78	5	100	22	7	100	26

DR: Percent of programs with defects where defects are reported

FPR: Percent of programs without defects, with defects incorrectly reported; $\underline{\mathrm{FPR}} = 100 - \mathrm{FPR}$

PM: Productivity metric: $\sqrt{\mathrm{DR} \times (100 - \mathrm{FPR})}$

RV-Match on Toyota ITC Benchmark

- Comparison with Other Analysis Tools -

- We have also evaluated other free analysis tools on the Toyota ITC benchmark
- Numbers for other tools may be slightly off; they were not manually checked yet
- Clang cannot be run with UBSan, ASan and TSan together; we ran them separately

Shiraishi et al., ISSRE '15	RV-Match			Valgrind + Helgrind (GCC)			UBSan + TSan + MSan + ASan (Clang)			Frama-C (Value Analysis Plugin)			Compcert Interpreter		
	DR	$\underline{\text{FPR}}$	PM	DR	$\underline{\text{FPR}}$	PM	DR	$\underline{\text{FPR}}$	PM	DR	$\underline{\text{FPR}}$	PM	DR	$\underline{\text{FPR}}$	PM
Static memory	100	100	100	9	100	30	79	100	89	82	96	89	97	82	89
Dynamic memory	94	100	97	80	95	87	16	95	39	79	27	46	29	80	48
Stack-related	100	100	100	70	80	75	95	75	84	45	65	54	35	70	49
Numerical	96	100	98	22	100	47	59	100	77	79	47	61	48	79	62
Resource management	93	100	96	57	100	76	47	96	67	63	46	54	32	83	52
Pointer-related	98	100	99	60	100	77	58	97	75	81	40	57	87	73	80
Concurrency	67	100	82	72	79	76	67	72	70	7	100	26	58	42	49
Inappropriate code	0	100	0	2	100	13	0	100	0	33	63	45	17	83	38
Miscellaneous	63	100	79	29	100	53	37	100	61	83	49	63	63	71	67
AVERAGE (Unweighted)	79	100	89	44	95	65	51	93	69	61	59	60	52	74	62
AVERAGE (Weighted)	82	100	91	42	97	65	47	95	67	66	55	60	51	76	63

DR: Percent of programs with defects where defects are reported

FPR: Percent of programs without defects, with defects incorrectly reported; $\underline{\text{FPR}} = 100 - \text{FPR}$

PM: Productivity metric: $\sqrt{\text{DR} \times (100 - \text{FPR})}$

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

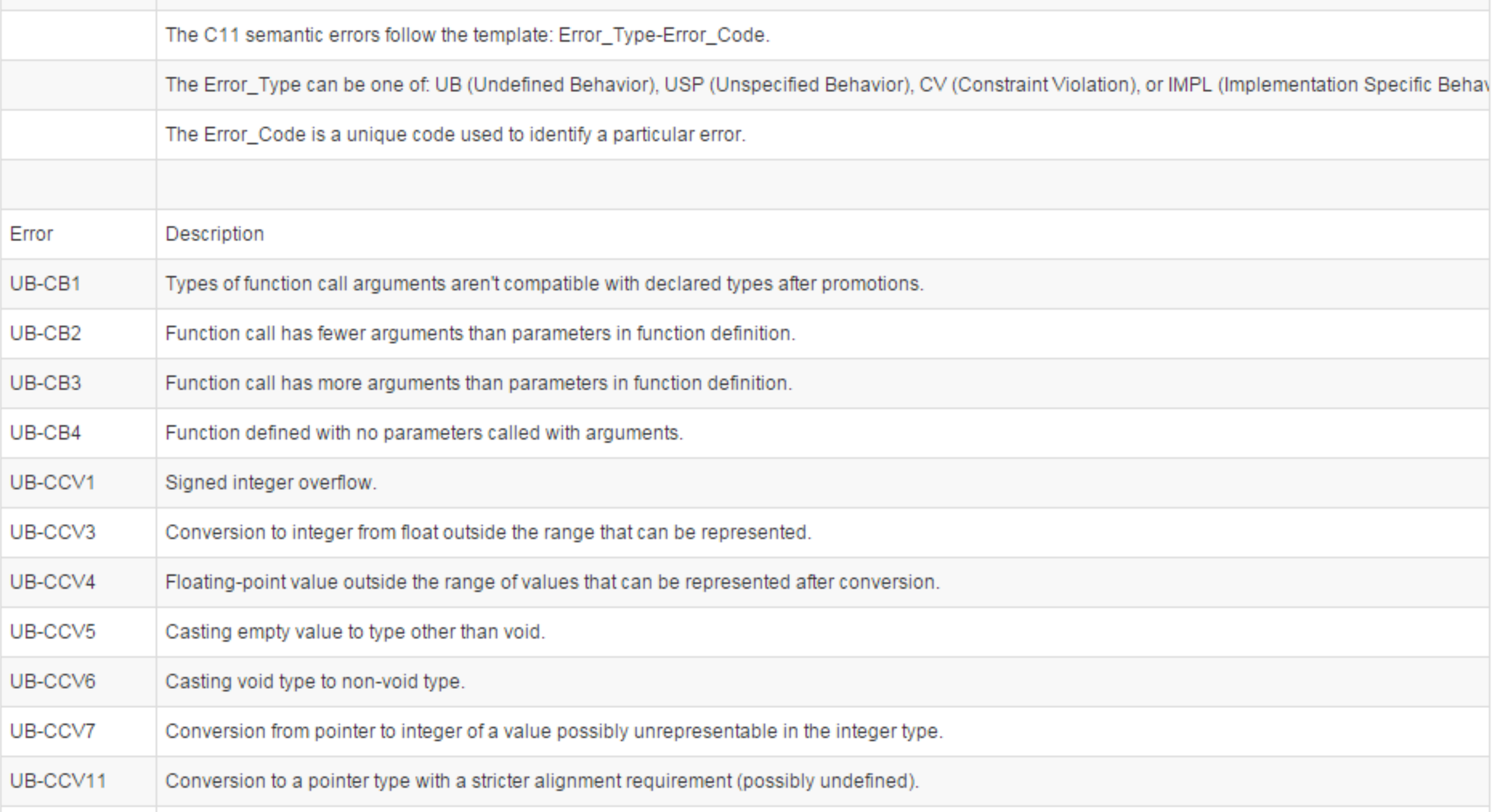
TACAS 2016

Competition on Software Verification (SV-COMP)

RV-Match on SV-Comp

- We had a tutorial at ETAPS'16 Congress. We heard colleagues at ETAPS'16 complaining that some of the *correct* SV-Comp benchmark programs are undefined
  - SV-Comp = benchmark for evaluating C program verifiers
  - Annual competition of program verification
- So we run the correct SV-Comp programs with kcc
- Unexpected results
  - Out of 1346 "correct programs", 188 (14%) were undefined, that is, wrong! So most program verifiers these days prove wrong programs correct. Think about it …

RV-Match Error Reports



	The C11 semantic errors follow the template: Error_Type-Error_Code.
	The Error_Type can be one of: UB (Undefined Behavior), USP (Unspecified Behavior), CV (Constraint Violation), or IMPL (Implementation Specific Behav
	The Error_Code is a unique code used to identify a particular error.
Error	Description
UB-CB1	Types of function call arguments aren't compatible with declared types after promotions.
UB-CB2	Function call has fewer arguments than parameters in function definition.
UB-CB3	Function call has more arguments than parameters in function definition.
UB-CB4	Function defined with no parameters called with arguments.
UB-CCV1	Signed integer overflow.
UB-CCV3	Conversion to integer from float outside the range that can be represented.
UB-CCV4	Floating-point value outside the range of values that can be represented after conversion.
UB-CCV5	Casting empty value to type other than void.
UB-CCV6	Casting void type to non-void type.
UB-CCV7	Conversion from pointer to integer of a value possibly unrepresentable in the integer type.
UB-CCV11	Conversion to a pointer type with a stricter alignment requirement (possibly undefined).

…

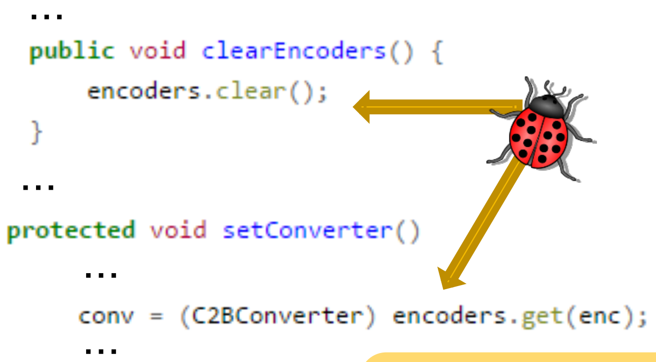
~200 different error reports

Predicting Concurrency Errors from Correct Executions without false alarms

RV-Predict

RV-Predict Overview

Tomcat (OutputBuffer.java)



```
 ...
 public void clearEncoders() {
     encoders.clear();
 }
...
protected void setConverter()
    ...
    conv = (C2BConverter) encoders.get(enc);
    ...
```

Automatically detect the rarest and most difficult data races in your Java/C code, saving on development effort with the most precise race finder available. **RV-Predict** gives you**:**

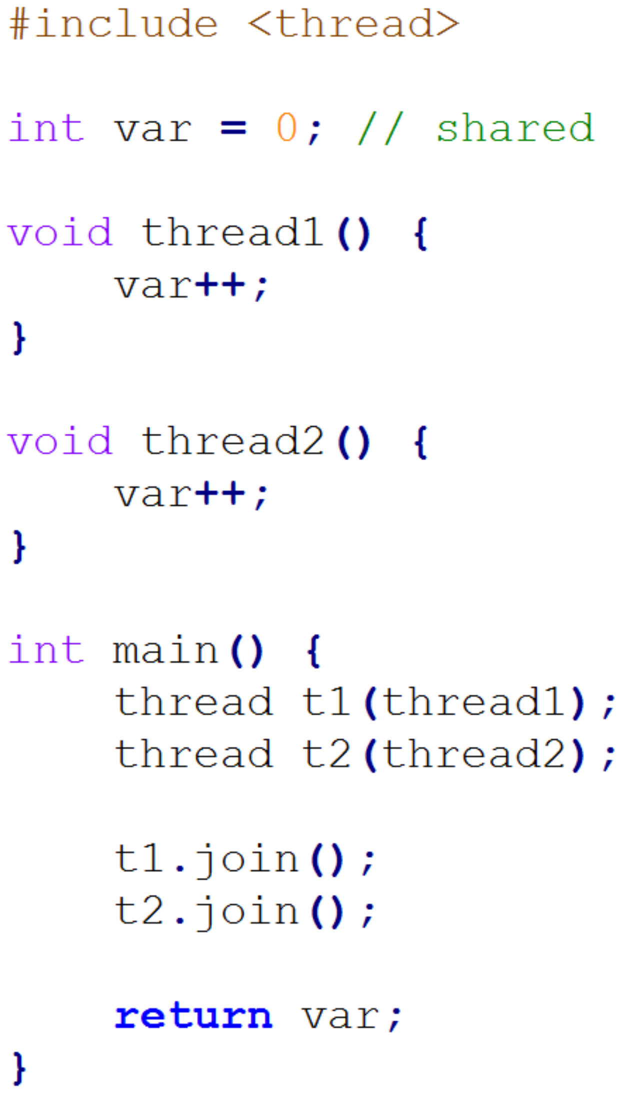
- an automatic debugger for subtle Java/C data races with no false positives
- seamless integration with unit tests, build infrastructure, and continuous integration
- a maximal detection algorithm that finds more races than any sound dynamic tool

Conventional testing approaches do not detect the data-race

RV-Predict precisely detects the data-race, and reports the relevant stack-traces

```
------------------                    --------
 T E S T S
-----------------------   ----------------------
Results :
Tests run: 0, Failures: 0, Errors: 0, Skipped: 0
...
...

Data race on field java.util.HashMap.$state:
{{{ Concurrent write in thread T83 (locks held: {Monitor@67298f15})
 ----> at org.apache.catalina.connector.OutputBuffer.clearEncoders(OutputBuffer.java:255)
...
Concurrent read in thread T61 (locks held: {})
 ----> at org.apache.catalina.connector.OutputBuffer.setConverter(OutputBuffer.java:604)...
```

Simple C Data Race Example



```
#include <thread>

int var = 0; // shared

void thread1() {
    var++;
}

void thread2() {
    var++;
}

int main() {
    thread t1(thread1);
    thread t2(thread2);

    t1.join();
    t2.join();

    return var;
}
```

- What value does it return?
- Data race on shared `var`
- This one is easy to spot, but data races can be evil
  - Non-deterministic
  - Rare
  - Hard to reproduce
- Led to catastrophic failures
  - Human life (Therac 25, Northeastern blackout, …)

Expected Execution

Code

```
#include <thread>

int var = 0; // shared

void thread1() {
  var++;
}

void thread2() {
  var++;
}

int main() {
  thread t1(thread1);
  thread t2(thread2);

  t1.join();
  t2.join();

  return var;
}
```

Event Trace

main	thread1	thread2
1. write(var,0)		
2. fork(thread1)		
3. fork(thread2)		
	4. read(var,0)	
	5. write(var,1)	
		6. read(var,1)
		7. write(var,2)
8. join(thread1)		
9. join(thread2)		
return(2)		

UnExpected Execution (Rare)

Code

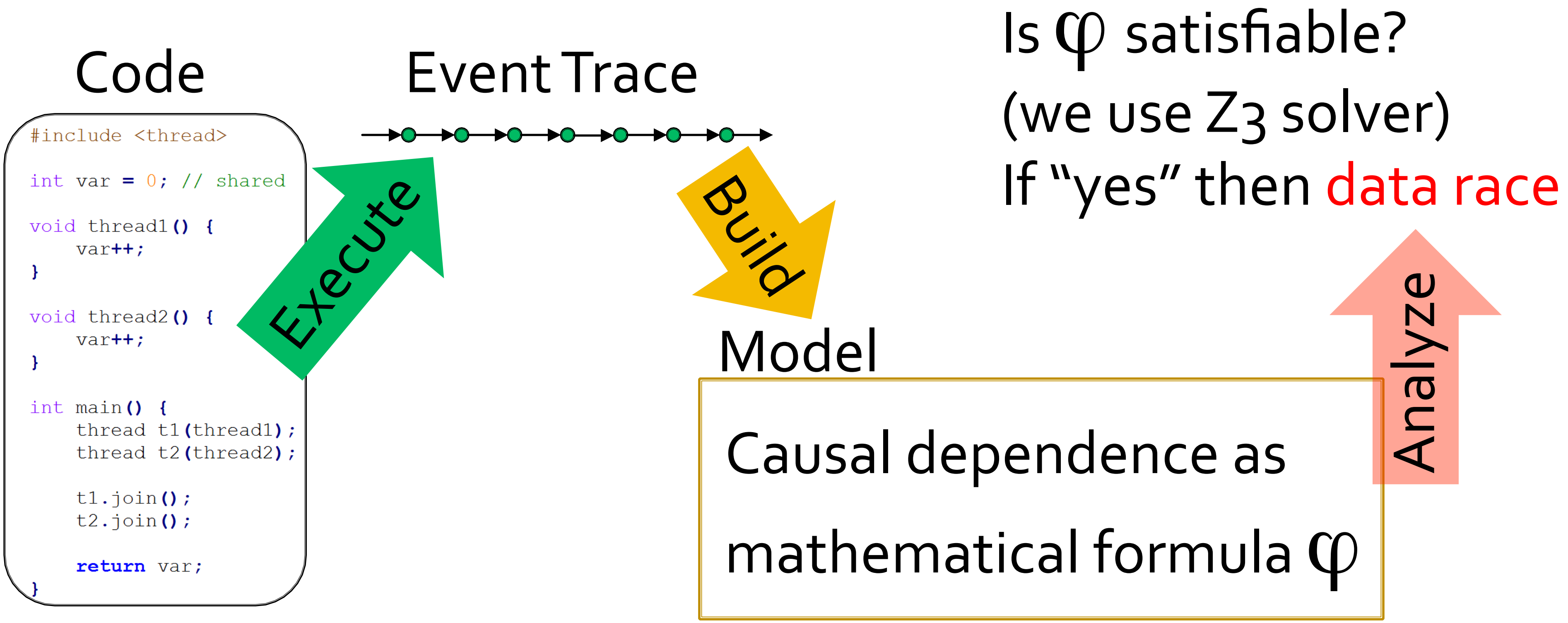
```
#include <thread>

int var = 0; // shared

void thread1() {
   var++;
}

void thread2() {
   var++;
}

int main() {
   thread t1(thread1);
   thread t2(thread2);

   t1.join();
   t2.join();

   return var;
}
```

Event Trace

main	thread1	thread2
1. write(var,0)		
2. fork(thread1)		
3. fork(thread2)		
	4. read(var,0)	
		5. read(var,0)
	6. write(var,1)	
		7. write(var,1)
8. join(thread1)		
9. join(thread2)		
return(1)		

RV-Predict Approach

1. Instrument program to emit event trace when executed
2. Give every observed event an order variable
3. Encode event causal ordering and data race as constraints
4. Solve constraints with SMT solver



Code

```
#include <thread>

int var = 0; // shared

void thread1() {
    var++;
}

void thread2() {
    var++;
}

int main() {
    thread t1(thread1);
    thread t2(thread2);

    t1.join();
    t2.join();

    return var;
}
```

Event Trace

Model

Causal dependence as mathematical formula φ

Is φ satisfiable?

(we use Z3 solver)

If “yes” then data race

Predicting Data Races

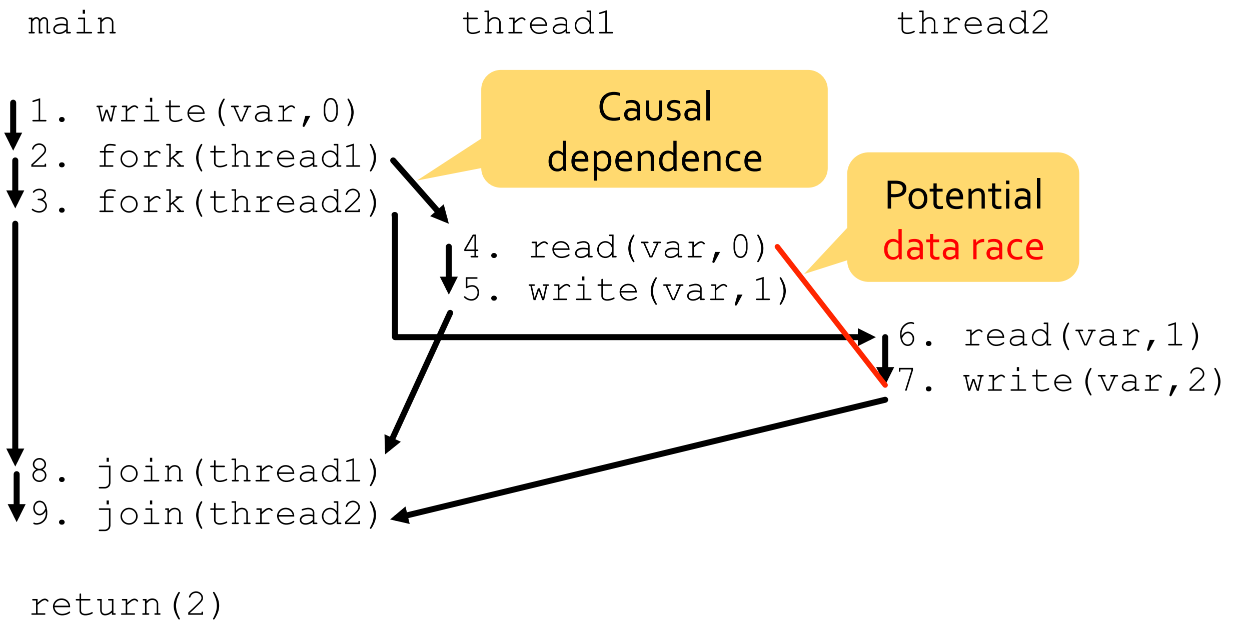
Code

```
#include <thread>

int var = 0; // shared

void thread1() {
    var++;
```

If φ satisfiable then **data race** is possible (no false alarm)

```
    t1.join();
    t2.join();

    return var;
}
```

Assume Expected Execution Trace



main thread1 thread2

1. write(var,0)
2. fork(thread1)
3. fork(thread2)
4. read(var,0)
5. write(var,1)
6. read(var,1)
7. write(var,2)
8. join(thread1)
9. join(thread2)

return(2)

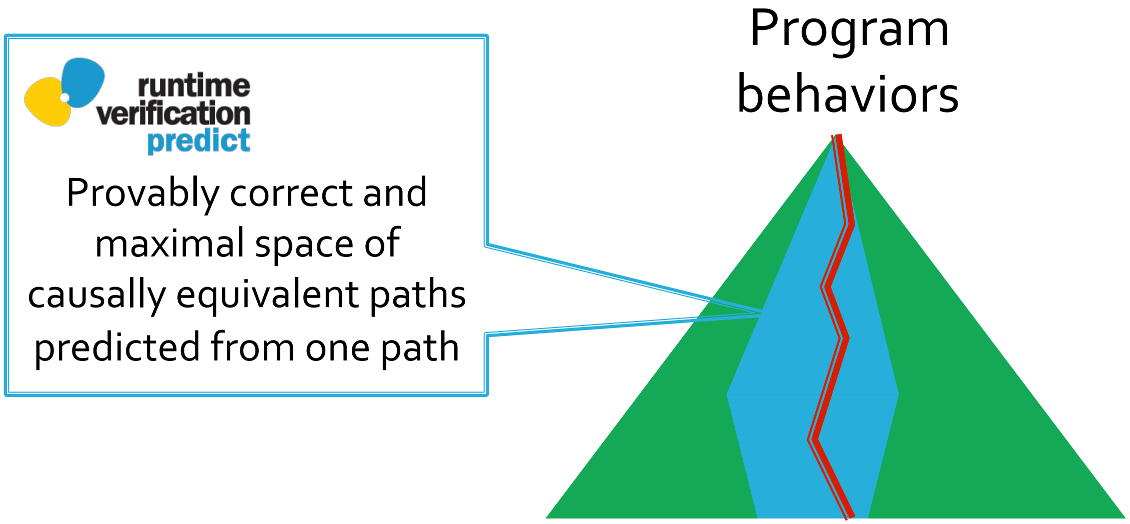
Causal dependence

Potential data race

Encode causal dependence and data race as constraints:

φ = O1<O2<O3<O8<O9 /\ O4<O5 /\ O6<O7
/\ O2<O4 /\ O3<O6 /\ O5<O8 /\ O7<O9
/\ O4=O7 // only one out of 3 races

RV-Predict Features



- Also synchronization, interrupts; see demo
- No false alarms: all predicted races are real
- Maximal: Impossible to precisely (without false alarms) predict more races than RV-Predict does from the same execution trace

[PLDI'14]
[RV'12]

RV-Predict DEMO

- Go to https://runtimeverification.com/predict to download RV-Predict (currently only Java 8 version available); javac and then execute the Java programs under folder `examples`
    - Most of the examples above are also discussed, with detailed comments, at

https://runtimeverification.com/predict/docs/runningexamples
https://runtimeverification.com/blog/?p=58

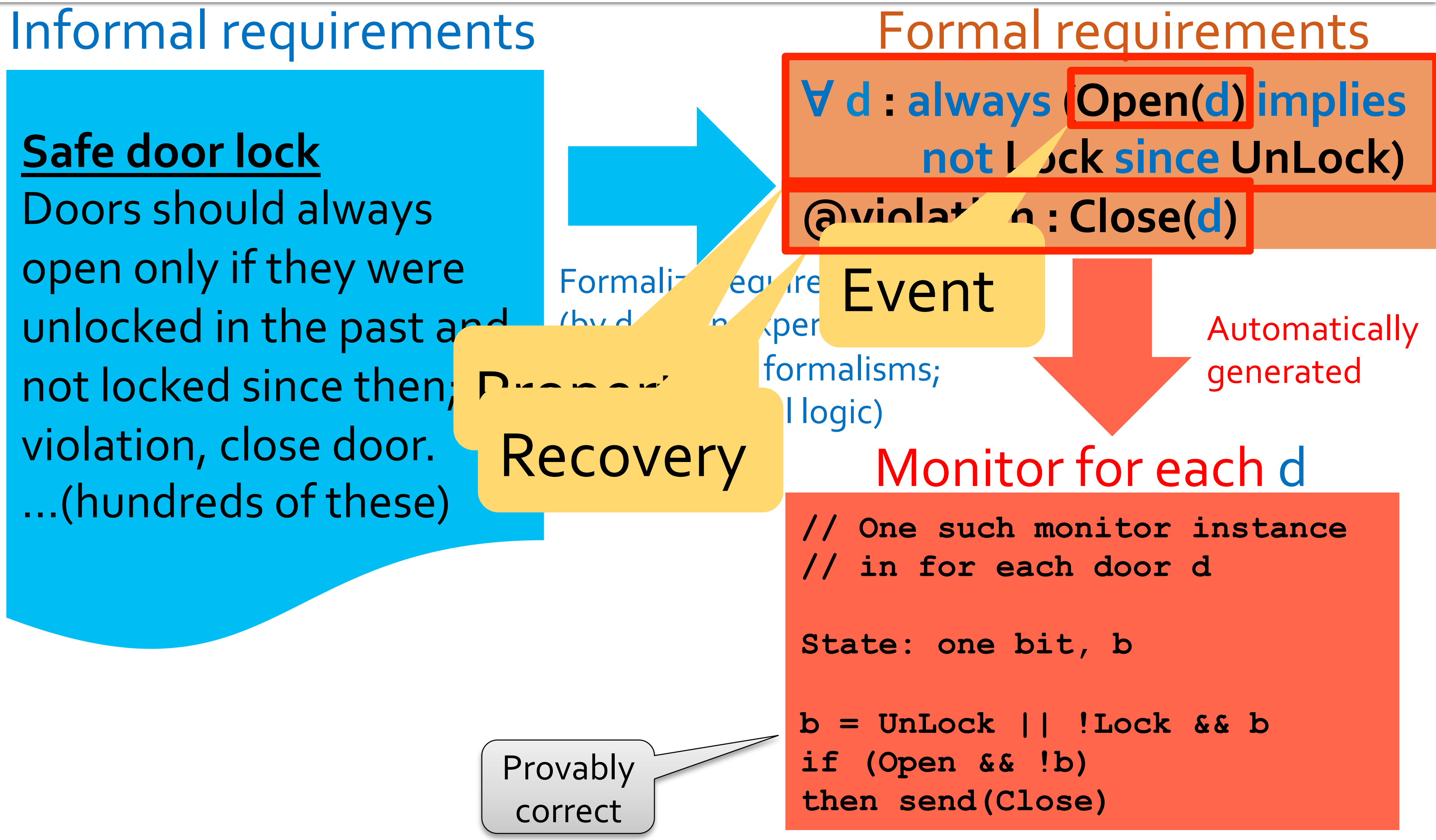
Monitor Safety Requirements and Recover when Violations Detected

RV-Monitor

RV-Monitor for C

- RV-Monitor is a code generator
  - Takes safety property specifications as input
  - Generates efficient monitoring code library as output
    - Provably correct: proof certificate can also be generated
- Specifications can be implicit (generic API protocols) or explicit (application-specific)
- RV-Monitor specifications consist of
  - *Events*: snapshots of system execution
  - *Properties*: desired sequences of events
  - *Recovery*: what to do when property violated

RV-Monitor Example



Informal requirements

Safe door lock

Doors should always open only if they were unlocked in the past and not locked since then; Propert

violation, close door.

…(hundreds of these)

Formali equ re (by d n xper formalisms; l logic)

Formal requirements

∀ d : always (Open(d) implies not Lock since UnLock)

@violat n : Close(d)

Event

Recovery

Automatically generated

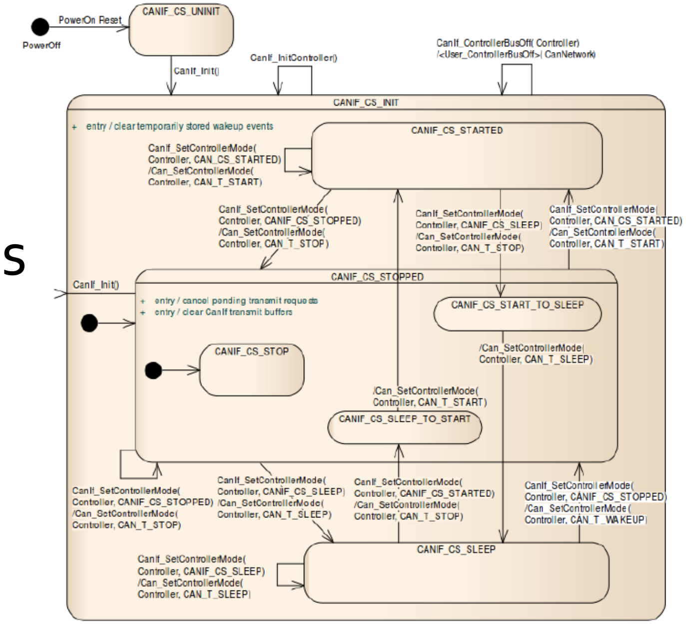
Monitor for each d

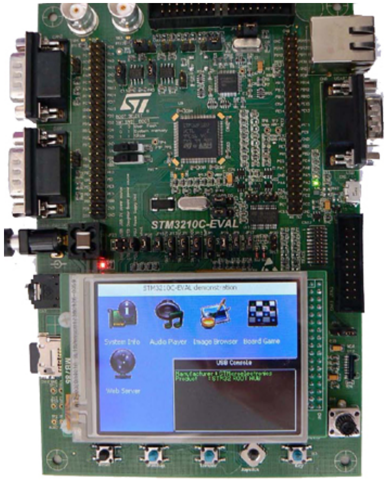
```
// One such monitor instance
// in for each door d

State: one bit, b

b = UnLock || !Lock && b
if (Open && !b)
then send(Close)
```

Provably correct

RV-Monitor Applications

- RV-AUTOSAR
  - Monitor AUTOSAR compliance
  - Formalized 20+ CAN interface properties
- RV-ECU
  - ECU in charge of safety on CAN bus
  - Runs LLVM
  - All code generated automatically from safety specifications; provably correct
  - Built prototype using STM ECU board STM3210C-EVAL
    - Currently runs in an actual car (model omitted)





RV-ECU DEMO

- Go to https://runtimeverification.com/ecu and watch video

Conclusion

- Runtime Verification, Inc., is a new startup company licensed by the University of Illinois
- Offers solutions for reliable and safe software
- Technology based on runtime verification
  - Scalable, rigorous, automatic, no false alarms
  - Can also be done exhaustively: full verification
- Business model
  - *General-purpose* libraries and tools
  - *Custom tools and services* to select customers