



# **Specification Editing and Discovery Assistant for C/C++ Software Development (SPEEDY) *and related verification projects***

**Frama-C Day – June 20, 2016, Paris**

**Presented by: David Cok, PI**

**Contributors: Gunjan Aggarwal, Andreas Stahlbauer, Scott Johnson**

**Pilot use: Ian Blissard, Joshua Robbins, external users**

GrammaTech, Inc.  
531 Esty Street  
Ithaca, NY 14850  
Tel: 607-273-7340  
E-mail: [info@grammatech.com](mailto:info@grammatech.com)

# Support

- This work was supported by NASA contract NNX14CL05C
- A portion of this material is based upon work supported by the National Science Foundation under Grant No. ACI-1314674.
- Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or NASA.
- Permission to distribute this presentation verbatim is granted; reproduction or distribution of portions or removing copyright or support markings is expressly prohibited.

# Original NASA Solicitation

- Solicitation topic: A1.06: Aviation Safety
- Key elements:
  - › rigorous, systematic, scalable, and repeatable **V&V**
  - › “Techniques, tools and policies to enable **efficient and accurate analysis of safety aspects** of software-intensive systems, ultimately reducing the cost of software V&V” ...
  - › “Tools and techniques that can **facilitate the use of formal methods in V&V** throughout the lifecycle such as graphical-based development environments (e.g., Eclipse plug-ins for static analyzers, model checkers, or theorem provers)” ...
- Ended 4/2016 – Eclipse plugin incorporating formal methods technology (Frama-C)

# The (long-term) vision: formal methods behind the curtain

A Programmer's workbench (for multiple languages):

- Thoroughly integrated with commonly used **SW development IDE(s)**
- Makes **thinking/acting about correctness** a natural part of the process, via specification checking, bug-finding, code style checking tools
- Provides **UI assists** in generating, editing, refactoring, checking, ... specifications in concert with code
- Automatic tools to **generate** specifications **and check** specs+code, seamless combination of sound(-ish) verification and bug-finding
- **Plug-in interfaces** to add new back-end tools as technology improves (or customization is desired)
- Means to connect specifications with other kinds of **design information** and **development processes** (e.g., from model checking, proof by construction)

GammaTech provides commercial static analysis tools for bug-finding.

This vision extends our current capabilities in additional directions.

# Background - GrammaTech

- GrammaTech produces commercial tools
  - › Heuristic flaw-finding tools (CodeSonar/C, x86, [Ada])
  - › Reverse engineering tools (CodeSurfer/C, x86)
  - › Integrates research prototypes (e.g., visualization tools, new analyses)
  - › Incorporating more sound/verification techniques where automated analysis is possible, at industrial scale
  
- GrammaTech does contract research in program analysis
  - › safety (assurance) and security
  - › commercial and government
  - › static and dynamic analysis
  
- David Cok, PI
  - › 15+ years history in Java specification (e.g., JML, ESC/Java2, OpenJML) and use of SMT solvers (e.g., contributed to SMT-LIBv2; jSMTLIB, SMT-COMP 2012, SMT-EVAL 2013)
  - › Focused on application to industrial-scale problems, everyday use
  - › [Prior research: automated reasoning for intelligent (imaging) systems, image processing, research leadership, leading commercial SW development teams]

# Random Related points

- **Connection of verified algorithms to actual implementations in code is sometimes (often?) missing**
- **Specifications serve as an aid to code understanding**
- **Automatically derived specifications serve as an aid to reverse engineering, debugging, code understanding**
- **Non-verification oriented engineers find specifications a nuisance, and more time-consuming than thinking and debugging – the goal has to be to minimize this overhead and objection**
  - › **automate**
  - › **integrate**
  - › **low performance overhead**
- **All of this is needed regardless of the underlying logic**

Hoare triples, separation logic, matching logic, temporal logic, abstract interpretation, symbolic execution, ...

  - › Every technique needs induction and/or user input to go beyond straight line or bounded path exploration – loops, recursive calls, gotos

# Context

Top-down:  
Correct by construction  
Get the algorithms right

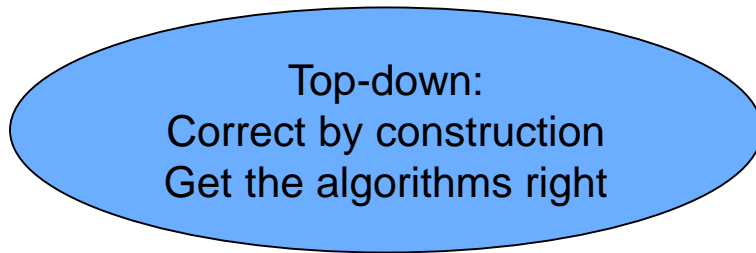
Complex algorithms need  
to be 'got right' before  
implementation begins

Need both approaches  
working together

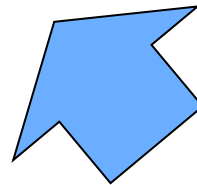
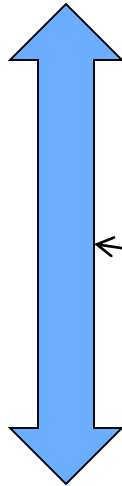
Bottom-up:  
Apply formal  
methods to code

A lot of implementation is more  
conveniently designed & partially  
implemented before the work  
of annotating and proving begins

# Context

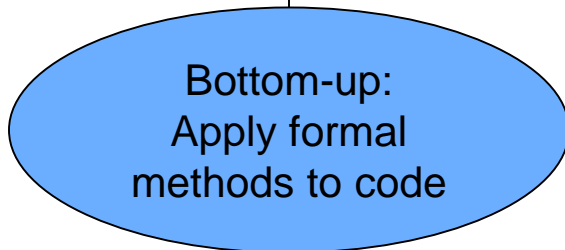


Complex algorithms need to be 'got right' before implementation begins



Need both approaches working together

And need a hierarchy of abstractions (expressed in logical annotations) that connects high- and low level specs.



A lot of implementation is more conveniently designed & partially implemented before the work of annotating and proving begins

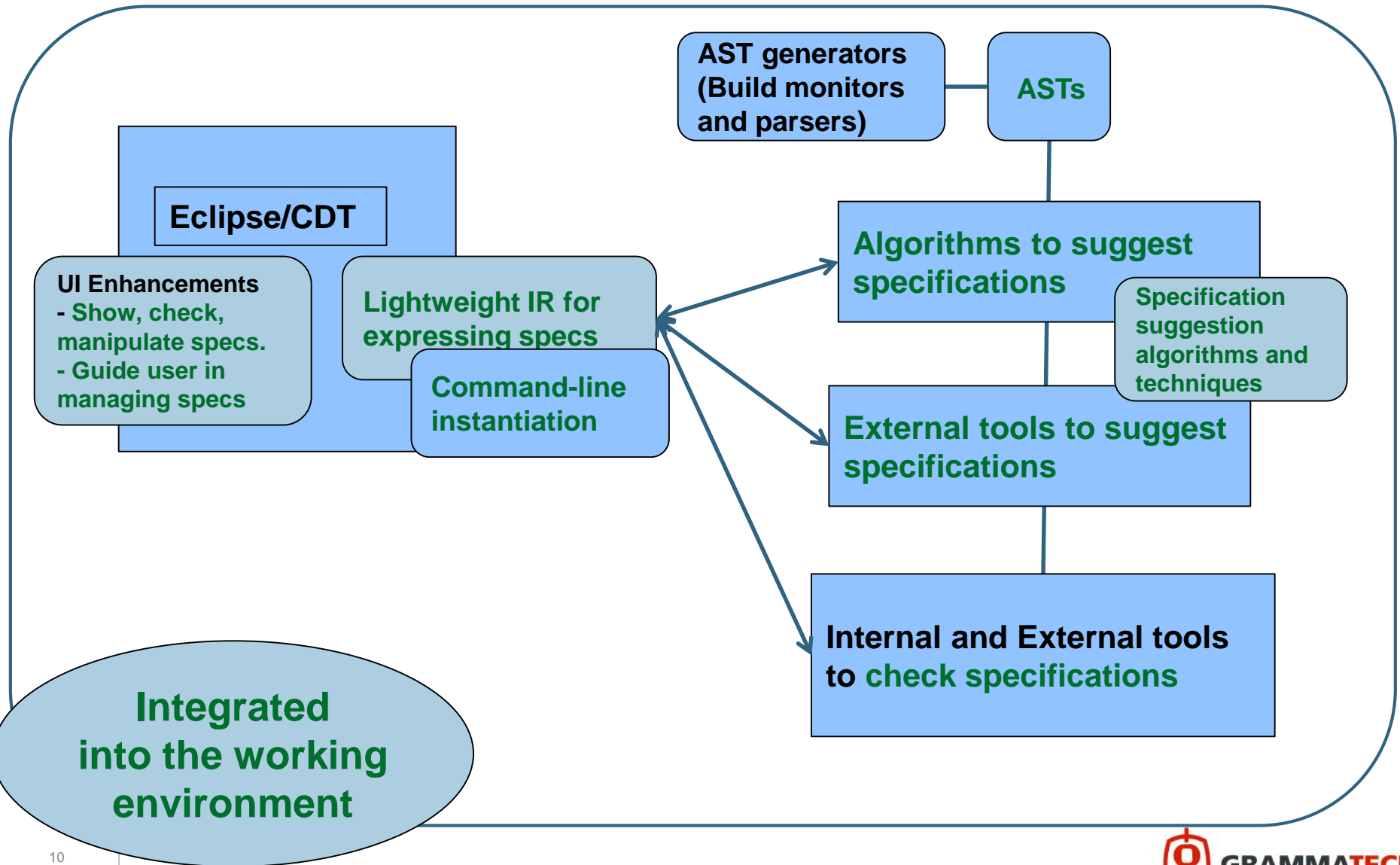


# Programming productivity research

Michael Ernst's (and Todd Schiller) work on programmer productivity (VeriWeb):


- › Drag and drop editing interface <<< SPEEDY: templates and content assist, keyboard short cuts
- › Concrete counterexamples (from execution traces) <<< SPEEDY: from static analysis, displayed directly in source code editor
- › Specification inlining (in Web interface) <<< SPEEDY: in source code editor
- › Context clues <<< SPEEDY: suggested spec locations
- › Specification suggestions from Daikon <<< SPEEDY: from various tools
- › Active guidance <<< SPEEDY: using Eclipse cheatsheets, quick fixes, ...

# SPEEDY Architecture



# 1a. Technical underpinnings

## – Specification language

- Choice: **ACSL (ANSI-C Specification Language)**
    - › A BISO in the style of JML (Java), SPARK (Ada), Spec# (C#), CodeContracts (.NET), ...
    - › Effort led by CEA-LIST (France)
    - › More readily understandable than logic languages (such as Coq)
  - Characteristics:
    - › Classic invariant/precondition/postcondition/frame condition form
    - › Expressions include first-order-logic (quantification)
    - › Additional primitives to express memory allocation, frame conditions
    - › Substantial user base, applications to safety-critical code
    - › Connection to Coq for interactive/assisted proof
  - We contributed bugs, fixes, documentation, discussion
- 

# 1a. Technical underpinnings

## – Specification language: C++

- C++ : No C++ specification language yet  
[Project goal was to follow/study external work and experiment with any artifacts produced.]
  - › CEA participated in the EC STANCE project to develop SL for C++
    - STANCE is an international, multi-partner project, with commercialization goals – so not feasible to just join in
    - STANCE project concluded, with some tools, but without a clear definition of a C++ specification language
    - SPEEDY project held discussions but there was no path to collaboration
  - › Still looking for ways to advance this goal
    - [Beyond the scope and time-frame of SPEEDY]

# 1b. Technical underpinnings

## – Parsing infrastructure

- Built a standalone AST, parsing, typechecking, error reporting infrastructure for ACSL
  - › Java-based (Antlr)
    - for easy integration into Eclipse
    - for ease of deployment across platforms
  - › Independent of Eclipse
    - supports standalone tools as well
    - supports other tools (e.g., a specification translator being built in another project; support for Fortran for an about-to-start AF project)
  
- Status: met SPEEDY goals with a few corners of the ACSL grammar left for future work:
  - › ghost program elements, preprocessing annotations, modules, sum types and patterns



## 2. GUI – Support for ACSL in Eclipse



Key goal: Integrate specifications as first-class elements of an IDE

Lots of ‘little/moderate’ features, built on Eclipse capabilities:

- Syntax and semantic coloring of ACSL
- As-you-type syntax and type errors
- Autoindenting
- Problems and markers
- Semantic searching (based on indexing)
- Commands with keyboard bindings
- Menu items
- Quick Fixes
- Icons
- Custom Console
- Preference and property pages
- Custom views, perspectives, folding
- Help (a view on the external documentation); help documentation index; cheat sheets
  - reasonable start on material, but always more to write and more examples to produce
- Showing counterexample paths and values
  - Prototype for built-in SMT solving
  - Not feasible in Phase II for Frama-C
- Refactoring
  - Renaming within specs
  - Specification and expression manipulation
    - More refactorings possible
- Internationalization
  - structure in place; more strings to localize; process to document
- Code completion (based on indexing), keyboard shortcuts
- Various informational hovers
- Low priorities or not needed: Context-sensitive help, project nature, project decoration, project builder

# 3. Specification Checking

- Various specification checking technologies
  - › Frama-C WP (weakest precondition) checks of consistency of modular ACSL specs and C implementation
  - › Frama-C + Why (above plan)
  - › Frama-C Value set plug-in
  - › CodeSonar
  - › CpaChecker (above plan)
  - › Direct translation to SMT
    - allows experimentation with different representations
    - allows for easier specification debugging
    - Goal met: proof-of-concept and demonstration, as a fall-back alternative to Frama-C
    - Future work: complete all of ACSL and a memory model
- Evaluation of scalability and performance on realistic code



# 3. Specification Checking – Frama-C WP

```
swap.c  NavigateToD...  NavigateToD...
1 /*@ requires \valid(a) ;
2   requires \valid(b);
3   ensures *a == \old(*b);
4   ensures *a == \old(*a); // WRONG
5   ensures *b == \old(*a);
6   ensures *b == \old(*b); // WRONG
7   assigns *a, *b;
8 */
9 void swap(int *a, int *b){
10  int tmp = *a ;
11  *a = *b;
12  *b = tmp;
13  return;
14 }
15
16 void main(){
17  int a =1;
18  int b=2;
19  swap(&a, &b);
20 }
21
```



# 3. Specification Checking – Frama-C Value Analysis

```
README  .c arp.c  .c behaviorBug
1
2 /*@ requires \true;
3 */
4 int div(int y) {
5     return 10/y;
6 }
7
8
9
10 /*@ requires y != 0;
11 */
12 int div2(int y) {
13     return 10/y;
14 }
15
16 int main() {
17     div(0);
18     div2(0);
19 }
20
21
22
```

```
README  .c arp.c  .c behaviorBug.
1
2 /*@ requires \true;
3 */
4 int div(int y) {
5     return 10/y;
6 }
7
8
9
10 /*@ requires y != 0;
11 */
12 int div2(int y) {
13     return 10/y;
14 }
15
16 int main() {
17     div(1);
18     div2(0);
19 }
20
21
22
```

# 3. Specification Checking - SMT

```
negate.c combinePredi... SyntaxColori...
1 float qq;
2 struct { int q; } a;
3
4 /*@ ensures y >= 0 ==> \result <= 0;
5    @ ensures y <= 0 ==> \result >= 0;
6 */
7 int negate(int y) {
8     int j;
9     j = -y;
10    return j;
11 }
12
13
14
15 /*@ requires y != -2147483648;
16    ensures y >= 0 ==> \result <= 0;
17    @ ensures y <= 0 ==> \result >= 0;
18 */
19 int negate2(int y) {
20     int j = -y;
21     return j;
22 }
23
24
25
```

Postcondition is false when the input is the most negative integer (which does not have a corresponding positive value in machine integers)

# 3'. Specification Checking – scaling up

- Specify and check publicly reported bugs
- Task:
  - › Find reported bugs of significance/interest
  - › Obtain code before and after fix
  - › Is it possible to specify the code so that it fails to check before and does validate afterwards
  - › i.e. – if specification-style development had been used, would the bug have been noted in development
  - › Some small modification to code needed for features not supported (e.g. variadic argument lists)
- Several examples – two shown here
  - › arp (BusyBox) : *busybox arp -Ainet* in version 1.6.2 results in *Segmentation fault*
  - › pr (CoreUtils) : crashed when too many backspaces were typed before a tab character was entered

# 3'. Specification Checking – arp (BusyBox)

Bug present since 2007  
Fixed 2008

```
483 /*@
484     requires argc >= 0;
485     requires \valid(argv+(0..argc-1));
486 */
487 int arp_main(int argc, char **argv)
488 {
489     // hwtypes set up correctly:|
490     char *hw_type;
491     char *protocol;
492
493     /* Initialize variables... */
494     ap = get_aftype(DFLT_AF);
495     if (!ap)
496         bb_error_msg_and_die("%s: %s not supported", DFLT_AF, "address family");
497
498     getopt32_arp(argc, argv, "A:p:H:t:i:adnDsv", &protocol, &protocol,
499                 &hw_type, &hw_type, &device);
500     argv += optind;
501     if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
502         ap = get_aftype(protocol);
503         if (ap == NULL)
504             bb_error_msg_and_die("%s: unknown %s", protocol, "address family");
505     }
506     if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
507         // XXX: FIX:
508         // if (option_mask32 & ARP_OPT_H || option_mask32 & ARP_OPT_t) {
509             hw = get_hwtype(hw_type);
510             if (hw == NULL)
511                 bb_error_msg_and_die("%s: unknown %s", hw_type, "hardware type");
512             hw_set = 1;
513         }
514     }
```

```
879 /*@
880     requires valid_name: \valid(name);
881 */
882 const struct hwtype *get_hwtype(const char *name)
883 {
884     // XXX: Workaround for global invariant
885     //@ assert \valid(hwtypes+(0..hwtypes_size));
886     //@ assert \forall integer i; (0 <= i < hwtypes_size) =
887     //@ assert \forall integer i; (0 <= i < hwtypes_size) =
888     //@ assert *(hwtypes+hwtypes_size) == 0;
889
890     const struct hwtype *const *hwp;
891
892     hwp = hwtypes;
893     /** Invariant to show that the argument name remains a valid string,
894         * and all iterated values of hwp have valid names.
```

```
534 /*@
535     // results flag & argument validity
536     ensures \result == option_mask32;
537     ensures (option_mask32 & (0x1)) <==> \valid(*protocol);
538     ensures (option_mask32 & (0x2)) <==> \valid(*protocol2);
539     ensures (option_mask32 & (0x4)) <==> \valid(*hw_type);
540     ensures (option_mask32 & (0x8)) <==> \valid(*hw_type2);
541 */
542 uint32_t
543 getopt32_arp(int argc, char **argv, char *applet_opts, char **protocol, char **protocol2,
544              char **hw_type, char **hw_type2, char **device) {
```

# 3'. Specification Checking – arp

```
483 /*@
484   requires argc >= 0;
485   requires \valid(argv+(0..argc-1));
486 */
487 int arp_main(int argc, char **argv)
488 {
489   // hwtypes set up correctly:
490   char *hw_type;
491   char *protocol;
492
493   /* Initialize variables... */
494   ap = get_aftype(DFLT_AF);
495   if (!ap)
496     bb_error_msg_and_die("%s: %s not supported", DFLT_AF, "address family");
497
498   getopt32_arp(argc, argv, "A:p:H:t:i:adnDsv", &protocol, &protocol,
499               &hw_type, &hw_type, &device);
500   argv += optind;
501   if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
502     ap = get_aftype(protocol);
503     if (ap == NULL)
504       bb_error_msg_and_die("%s: unknown %s", protocol, "address family");
505   }
506   // if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
507   // XXX: FIX:
508   if (option_mask32 & ARP_OPT_H || option_mask32 & ARP_OPT_t) {
509     hw = get_hwtype(hw_type);
510     if (hw == NULL)
511       bb_error_msg_and_die("%s: unknown %s", hw_type, "hardware type");
512     hw_set = 1;
513   }
}
```

```
879 /*@
880   requires valid_name: \valid(name);
881 */
882 const struct hwtype *get_hwtype(const char *name)
883 {
884   // XXX: Workaround for global invariant
885   //@ assert \valid(hwtypes+(0..hwtypes_size));
886   //@ assert \forall integer i; (0 <= i < hwtypes_size) =
887   //@ assert \forall integer i; (0 <= i < hwtypes_size) =
888   //@ assert *(hwtypes+hwtypes_size) == 0;
889
890   const struct hwtype *const *hwp;
891
892   hwp = hwtypes;
893   /** Invariant to show that the argument name remains a valid string,
894     * and all iterated values of hwp have valid pointers
```

```
534 /*@
535   // results flag & argument validity
536   ensures \result == option_mask32;
537   ensures (option_mask32 & (0x1)) <==> \valid(*protocol);
538   ensures (option_mask32 & (0x2)) <==> \valid(*protocol2);
539   ensures (option_mask32 & (0x4)) <==> \valid(*hw_type);
540   ensures (option_mask32 & (0x8)) <==> \valid(*hw_type2);
541 */
542 uint32_t
543 getopt32_arp(int argc, char **argv, char *applet_opts, char **protocol, char
544             char **hw_type, char **hw_type2, char **device) {
```

# 3'. Specification Checking – pr (CoreUtils)

Bug present since 1992  
Fixed 2008

```
2656 /*@
2657  requires valid input position;
2658  ensures valid input position;
2659 */
2660 static int
2661 char_to_clump (char c)
2662 {
2663     unsigned char uc = c;
2664     char *s = clump_buff;
2665     int i;
2666     char esc_buff[4];
2667     int width;
2668     int chars;
2669     int chars_per_c = 8;
2670
2671     if (c == input_tab_char)
2672         chars_per_c = chars_per_input_tab;
```

```
592 /* Horizontal position relative to the current file.
593     (output_position depends on where we are on the page;
594     input_position depends on where we are in the file.)
595     Important for converting tabs to spaces on input. */
596 static int input_position;
597 /*@ predicate valid_input_position = (input_position >= 0); */
```

```
2741 // 6.10 BUG
2742 input_position += width;
2743 // Initial fix:
2744 /* Too many backspaces must put us in position 0 -- never negative. */
2745 // if (width < 0 && input_position == 0)
2746 // {
2747 //     chars = 0;
2748 //     input_position = 0;
2749 // }
2750 // else if (width < 0 && input_position <= -width)
2751 //     input_position = 0;
2752 // else
2753 //     input_position += width;
2754
2755 return chars;
2756 }
```

```
2441 /*@
2442  requires valid input position;
2443  ensures valid input position;
2444 */
2445 static bool
2446 read_line (COLUMN *p)
2447 {
2448     int c;
2449     int chars IF_LINT (= 0);
2450     int last_input_position;
2451     int j, k;
2452     COLUMN *q;
```

# 3'. Specification Checking – pr

```
2656 /*@
2657  requires valid_input_position;
2658  ensures valid_input_position;
2659 */
2660 static int
2661 char_to_clump (char c)
2662 {
2663     unsigned char uc = c;
2664     char *s = clump_buff;
2665     int i;
2666     char esc_buff[4];
2667     int width;
2668     int chars;
2669     int chars_per_c = 8;
2670
2671     if (c == input_tab_char)
2672         chars_per_c = chars_per_input_tab;
```

```
2741 // 6.10 BUG
2742 // input_position += width;
2743 // Initial fix:
2744 /* Too many backspaces must put us in position 0 -- never negative. */
2745 if (width < 0 && input_position == 0)
2746 {
2747     chars = 0;
2748     input_position = 0;
2749 }
2750 else if (width < 0 && input_position <= -width)
2751     input_position = 0;
2752 else
2753     input_position += width;
2754
2755 return chars;
2756 }
```

# 4. Specification Assistance

## Programming productivity research

Michael Ernst's (and Todd Schiller) work on programmer productivity (VeriWeb):

- › Drag and drop editing interface <<< SPEEDY: templates and content assist, keyboard short cuts
- › Concrete counterexamples (from execution traces) <<< SPEEDY: from static analysis, displayed directly in source code editor
- › Specification inlining (in Web interface) <<< SPEEDY: in source code editor
- › Context clues <<< SPEEDY: suggested spec locations
- › Specification suggestions from Daikon <<< SPEEDY: from various tools
- › Active guidance <<< SPEEDY: using Eclipse cheatsheets, quick fixes,



# 4. Specification Assistance

## Documentation resources



- Online User guide/Reference manual
  - › Standalone web pages using GrammaTech’s commercial-grade manual production
  - › Integrated as Eclipse Help
  - › Future Work: Tutorial (with help of NSF grant)
  
- Guide to writing specifications
  - › Step-by-step tasks as Eclipse “cheatsheets”
  
- Guide to ACSL in the GUI
  - › Keyword/feature descriptions using Eclipse dynamic help

## 4. Specification Assistance IDE assistance



Key goal: Present needed information about specifications *in the engineer's development environment*

- Code completion (that works within specifications)
- Semantics-aware searching (finding declarations and uses)
- Context-sensitive information/help in the IDE
- Refactoring. Future Work: Even more refactoring

# 4. Specification Assistance

## Debugging resources



Key goal: Present information about specification/code inconsistency *in the engineer's development environment*

- Parsing/typechecking errors in ACSL are just like compiler errors
- Specification inconsistencies are presented as Eclipse 'markers'
- **Going beyond 'proof failed'** : Counterexample values presented in the context of the source code as Eclipse hovers and dialogs
  - › Future Work: Frama-C counterexamples
  - › Direct SMT checking. Future Work: Fill out features

# 5. Automated Specification Discovery

## What is needed?

Prior to implementing a bunch of algorithms...

- Took time to assess what kinds of code structures might be encountered
- Academic work tends to focus on a particular kind of code structure (say, nested array indexing) and develop an algorithm for that (perhaps difficult) situation, without reference to how common that situation is.
  
- Survey
  - › Examples in literature: most loops are not that complicated
  - › Our results as well: Most loops
    - simple indexing by 1 from fixed lower to upper bound
    - pointer equivalent
    - pointer indexing along a string to null terminator
    - [pointer indexing along a linked structure]
  - › Loop content is often
    - array processing (do same thing to every element)
    - accumulator (compute some summary value while not changing array)

# 5. Automated Specification Discovery Tool & algorithm integrations

[ specification discovery == invariant inference == function summarization ]

Key goal: Infer specs from code for the user to review/edit

- *gives a head start in writing specifications*
- *serves as an assistant in code review and understanding*

## Tools

- CodeSonar's function summarization as preconditions
- CodeSurfer's analysis as frame conditions
- Frama-C val: frame conditions
- Daikon: spec suggestion from analysis of runtime traces
- CpaChecker for predicate abstraction



# 5. Automated Specification Discovery Tool & algorithm integrations

## Algorithms

- Symbolic execution for simple pre/frame/post conditions
- Loop invariants for the index and variant
- Other algorithms ...



# Automated Specification Discovery

## - Algorithms: Symbolic execution

- Symbolic execution for simple pre/frame/post conditions
- Fast way to derive specs for simple functions (no loops, not too many branches)
- Key question: how much simplification is needed to make the derived expressions human-readable (and how intelligent must the simplification be)

# More Algorithms for suggesting specs

[Note: All techniques will need ability to simplify and present automatically generated formulae]

[Note 2: It is still helpful if 'easy' cases are handled automatically leaving a small number to be completed by hand.]

- Daikon with custom predicates; use later work to help select predicates for precondition-postcondition implications
  - › Variation: Daikon with inputs from concolic execution (Grace)
- Symbolic execution (many variations)
  - › Simple, for simple procedures
  - › Axiom Meister algorithm (Tillman et al.) – simplification in terms of observers
- Dynamic Symbolic Execution
  - › For polynomial invariants and array relationships (Nguyen, Kapur, Weimer, Forrest)
  - › “Universal Symbolic Execution” (Kannan and Sen)
  - › DySy (Csallner et al.)
- Template-based Predicate abstraction (Srivastava & Gulwani, 2009)
  - › Uses an SMT solver to solve for satisfying values that are *predicates*
  - › Also: VS<sup>3</sup> (Srivastava, Fulwani, Foster, 2009)
- Loop invariants for loops with parallelizable state updates (Gedell, Hähnle)
- Quantified abstract domains for loop invariants (Gulwani et al.)
- Inferred type annotations with the Checker framework
- Houdini-style static analysis with Daikon’s approach to selecting predicates

There is LOTS  
of active  
literature



# 7. Assessment – Pilot use

Key goal: Use teams outside of the principal developers as pilot users

- Internal use on other projects within GrammaTech 

- External users (academic or industrial) 

- › concentrated on IDE and Frama-C integration
- › regular users of Frama-C
- › academic users less familiar with Frama-C

- Future Work

- › External user evaluation is ongoing and we are seeking additional participants

# Related projects

- SPEEDY is being incorporated into a follow-on project wanting tools to review critical code (and we're going to sneak in some verification)
- An important impediment to wider use of modular bottom-up verification is the need for specifications of libraries (as well as more robust and scalable tools).

Have an NSF grant with which we are trying to address both the tool and specification aspects.

The NSF project is OpenSource << contributors welcome.

# Key needs regarding ACSL & Frama-C

## ACSL

- Common base of library specifications.
  - › An earlier call for requesting information had little response.
  - › I would like to propose a common (open source) repository of library specifications in ACSL (or is the community going to go with Coq instead...?)
- Clearer definition of ACSL than is in the current description document
  - › I propose joint collaborative work on resolving ambiguities in the description and omissions in the language.

## Frama-C

- Clear statements regarding what ACSL is supported by plugins (such as WP) and plans toward closing gaps.
- Counterexample information returned from subsidiary SMT solvers

# Summary

- Put verification ('formal methods') tools in the hands of software developers (especially if they do not know they are using it)
- Make tools robust, scalable and highly automated
  - › rely on decision procedures and automated heuristics as much as possibleSpecification inference can help greatly
  - › takes care of simple cases without needing engineer input
  - › provides a starting point for complex specifications
  - › is an aid to understanding
- Need extensive libraries with verified specifications
- Make correct-by-construction and verification of legacy code approaches work together

# Questions?