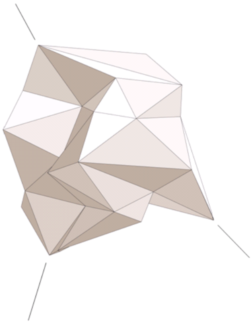
Safe and Secure Programming Using SPARK

Angela Wallenburg

Chalmers Tech Talk, 16 February 2015



Our World - Critical Software

No bugs please!

Motivating Example

Consider the following few lines of code from the original release of the Tokeneer code:

```
if Success and then
   (RawDuration * 10 <= Integer(DurationT'Last) and
    RawDuration * 10 >= Integer(DurationT'First)) then
   Value := DurationT(RawDuration * 10);
else
```

Can you see the problem? This error escaped lots of testing!

Tokeneer

- NSA-funded demonstrator of high-security software engineering
- biometric system for user verification and access control
- formal methods used: system specification and security properties in Z, implementation in SPARK
- small system (budget), about 10 kloc logical (2623 VCs)
- 2513 VCs were proven automatically (95.8 %), with 43 left to the an interactive prover and 67 discharged by review
- `http://www.adacore.com/sparkpro/tokeneer/`
- Open source (code, formal spec, project docs). Go and download!

The Verifying Compiler

The grand challenge of (building) the verifying compiler [Hoa03]:

> *A verifying compiler uses mathematical and logical reasoning to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program. The compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components.*

Practical Issues in Formal Verification

Failed proof attempt. Reasons:

1. There is an error in the contract and/or the program, or
2. The contract and the program are correct but the prover can not prove it

Need to:

- Analyse failed proof attempts
- Help the prover by providing additional facts, for example finding the correct loop invariant
- Time-consuming!

How to get enough confidence in the correctness to justify these tasks?

How to find sooner if there is an error?

Problem

Testing approach flawed... Proving approach flawed...

Two hurdles in the take-up of verifying compiler technology:

1. the lack of a convincing cost-benefit argument
2. the difficulty of reaching non-expert users

Solution?

Mixing Test and Proof

What is SPARK?

SPARK is...

- A programming language...
- A set of program verification tools...
- A design approach for high-integrity software...

What is SPARK?

SPARK is...

- A programming language...
- A set of program verification tools...
- A design approach for high-integrity software...
- All of the above!

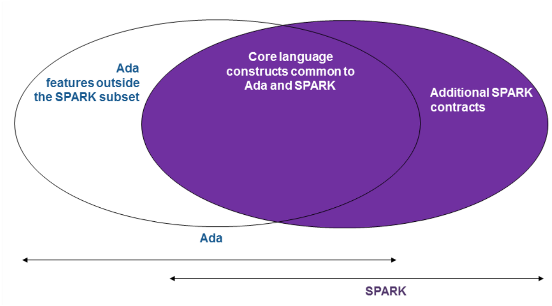
Static Verification Goals

Ideally we would like static verification to deliver analyses which are:

- Deep (tells you something useful...)
- Sound (with no false negatives...)
- Fast (tells you *now*...)
- Complete (with as few false alarms/positives as possible...)
- Modular and Constructive (and works on incomplete programs.)

SPARK is designed with these goals in mind. Since the 80ies!

SPARK Tradition - Analysable Subset of Ada


Ada features outside the SPARK subset
Core language constructs common to Ada and SPARK
Additional SPARK contracts
Ada
SPARK

- Exclude language features difficult to specify/verify
  - Pointers and aliasing
  - Exceptions
- Eliminate sources of ambiguity
  - Functions (not procedures) cannot have side-effects
  - Expressions cannot have side-effects

SPARK Application Domains

- Designed for embedded and real-time systems.
- Typical systems:
  - Hard real-time requirements
  - Little or no Operating System on target (no disk or VM...)
  - Fixed, known amount of storage
- Application domains:
  - The size of the problem is known in advance i.e. how many wings, engines, targets, tracks, etc.
- SPARK was not designed for building GUIs, database applications, web-servers and so on.
- Recently used for large, server-side, safety-critical system using tasking and richer data types (iFACTS).

Contracts

- *Contract*: agreement between the client and the supplier of a service
- *Program contract*: agreement between the caller and the callee subprograms

- Assigns responsibilities
- A way to organise your code
- Not a new idea (Floyd, Hoare, Dijkstra, Meyer)

Example Contract

Contracts are about *what* your code does rather than *how* it does it. Example:

```
procedure Sqrt (Input : in Integer; Res: out Natural)
with
   pre  => Input >= 0,
   post => (Res * Res) <= Input and
           (Res + 1) * (Res + 1) > Input;
```

Question: What difference do types make?

Types and Contracts

```
procedure Sqrt (Input : in Integer; Res: out Natural)
with
   pre  => Input >= 0,
   post => (Res * Res) <= Input and
           (Res + 1) * (Res + 1) > Input;
```

With the help of types:

```
procedure Sqrt (Input : in Natural; Res: out Natural)
with
   post => (Res * Res) <= Input and
           (Res + 1) * (Res + 1) > Input;
```

Less to write!

Some More Type Examples

Enumerations

```
type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

Subtypes

```
subtype Positive is Integer range 1 .. Integer'Last;
subtype Week_Day is Day range Mon .. Fri;
subtype Buffer_Range is Integer range 0 .. Max;
```

Array types

```
type My_Positive_Buffer is array (Buffer_Range) of Positive;

type String is array (Positive range <>) of Character;
```

discriminant and variant records, tagged types, array literals, and much more...

Observation: Good Fit!

Ada already offers a wide range of contracts:

- Type ranges
- Accessibility
- Parameter Modes
- Generic Parameters
- Interfaces
- Privacy
- Not Null Access
- ...

Contracts are Not Only Pre/Post

```
procedure Open (Customer : in      Identity.Name;
                Id       : in      Identity.Id;
                Cur      : in      Money.CUR;
                Account  :     out Account_Num)
with
   Pre  => not Max_Account_Reached,
   Post => Existing (Account) and then
           Belongs_To (Account, Customer, Id)
           and then
           Money.Is_Empty (Balance (Account));
```

- Strong typing
- Parameters not aliased
- Parameters initialised

Strong Typing (SPARK vs C)

Example in C:

```
int A = 10 * 0.9;
```

in Ada:

```
A : Integer := 10 * Integer (0.9);
A : Integer := Integer (Float (10) * 0.9);
```

- Types are at the base of the SPARK (Ada) model
- Semantic is different from representation
- SPARK (Ada) types are named
- Associated with properties (ranges, attributes) and operators

Run-Time Errors

A simple assignment statement

```
A (I + J) := P / Q;
```

Which are the possible run-time errors for this example?

Run-Time Errors

A simple assignment statement

```
A (I + J) := P / Q;
```

The following errors could occur:

1. `I + J` might overflow the base-type of the index range's subtype (arithmetic overflow)
2. `I + J` might be outside the index range's subtype
3. `P/Q` might overflow the base-type of the element type (arithmetic overflow)
4. `P/Q` might be outside the element subtype
5. `Q` might be zero

Verification Condition Generation

- Type safety (aka No run-time errors)
  - Arithmetic overflow
  - Division by zero
  - Array index range error (buffer overflow)
  - And many more...
  - ...for every statement in your program...
- Partial correctness with respect to pre- and post- conditions

Motivating Example Revisited

```
if Success and then
   (RawDuration * 10 <= Integer(DurationT'Last) and
    RawDuration * 10 >= Integer(DurationT'First)) then
   Value := DurationT(RawDuration * 10);
else
```

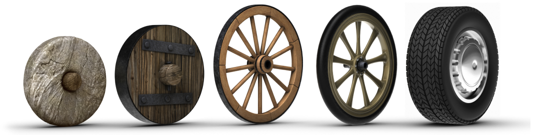
Failed VC:

```
procedure_readduration_4.
H1:     rawduration__1 >= - 2147483648 .
H2:     rawduration__1 <= 2147483647 .
 ...
        ->
C1:     success__1 -> rawduration__1 * 10 >= - 2147483648 and
           rawduration__1 * 10 <= 2147483647 .
```

Scaling Up — Interfacing

SPARK has solutions for scaling up: interfacing to other languages, systems, volatiles...

Scaling Up — Model and Implementation



... Containers, Abstraction, Design, Refinement...

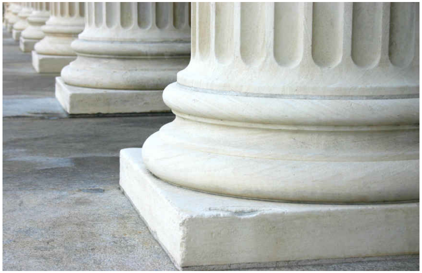
Programming Language and Program Verification Research

- Automated theorem proving progress: model checking, SAT, SMT
- Challenges in reasoning about mainstream programming language features [HLL+12]
  - Aliasing, exceptions, object invariants, subclasses, aggregations
- "Non-expert" users studied
- Many users are happy to write assertions for dynamic checking
- Can those users be convinced to write contracts?
- Could they do formal verification?

Executable Contracts

- Eiffel, JML, Spec#
- Executable contract vs formal contract?
- The same contract interpreted in two different worlds [Cha10]:
    1. Executable boolean expression
    2. First-order logic formula
- Ada 2012 has executable and formal contracts as part of the language

SPARK 2014 - Lessons Learned



Keep the good foundations...

but move a bit in this direction...

Let's combine the best of two worlds.

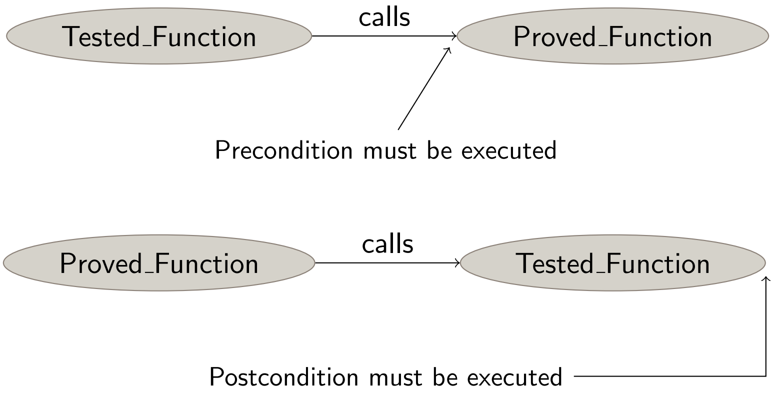
Motivation - Industry Safety Standards

- Drives the safety critical software business
- Important for cost-benefit argument
- DO-178B very successful (what does that mean?)
- Basis for certification of airborne software
  - FAA (Federal Aviation Administration)
  - European Aviation Safety Agency (EASA)
- DO-178B often used by other industries as well
- New standard supplement DO-178C [RTC11a, RTC11b]
- Using formal methods needs to be “as good as testing”

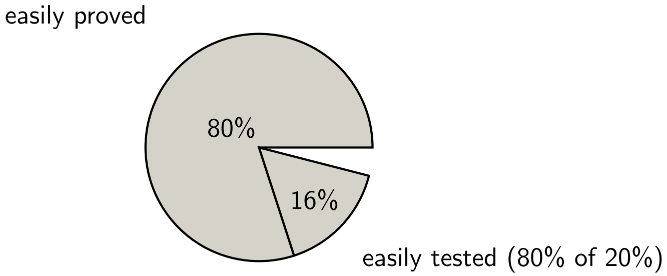
Mixing Test and Proof

- Modular verification
- Low-level requirements expressed as contracts
- Successful execution of postcondition $\rightarrow$ test successful
- Successful proof of postcondition $\rightarrow$ low-level requirement verified for all input
- Some low-level requirements are tested, some are proved
- Is the combination as “strong” as all low level requirements tested?

Hybrid Verification



Benefits of Hybrid Verification



- Helps with gradual introduction to formal proof
- The traditional 80/20% rule holds for both formal verification and testing
- More about this approach in [CKM12]

SPARK 2014 Architecture

- Joint development between Altran and AdaCore
- Built using the GNAT compiler front-end
- Why3 [BFPM11] is the intermediate proof language
- Modern implementation of data and flow analysis
- GNATprove, the end user tool, can be run from GPS IDE
- Under the hood: gnat2why translation
- Tools ship with Alt-Ergo and CVC4
- More on SPARK 2014 architecture: the convergence of compiler technology and program verification [KSD12]

SPARK Applications

For an overview of SPARK applications, see [O'N12].

SHOLIS Project (1995)

- assists Naval crew with the safe operation of helicopters at sea
- safety limits on wind vectors, ship's roll and pitch for landing, in-air refuelling
- 27 kloc (logical) of SPARK code, 54 kloc of information-flow contracts, and 29 kloc of proof contracts, 9000 VCs
- no operating system and no COTS libraries of any kind
- 75.5% proven automatically by the Simplifier
- lesson: state abstraction and refinement needed, later implemented, read more about SHOLIS: [KHCP00]

Lockheed-Martin C130J Project

- most recent generation of the very successful “Hercules” military transport aircraft
- the Mission Computer application software is written in SPARK
- subject to a large verification effort in the UK (UK RAF)
- SPARK analysis, including verification of partial correctness for most critical functions with respect to the system’s functional specification, which was expressed using the “Parnas Tables” notation.

iFACTS Project - Scaling up

- tool for NATS en-route air-traffic controllers in the UK [RoI]
- most ambitious SPARK project to date, starting in 2006
- a formal functional specification (Z)
- 250 kloc logical lines of code, in SPARK
- proof of type-safety, few functional correctness proofs
- 152927 VCs, of which 151026 (98.76 %) are proven automatically, 1701 VCs discharged with user rules
- proof is reconstructed daily (and overnight)

SPARKSkein Project - Fast, Formal, Nonlinear

- 2010, we implemented the Skein hash algorithm in SPARK [CBW11]
- goal: show that a “low-level” cryptographic algorithm like Skein could be implemented in SPARK
- complete proof of type safety, readable and “obviously correct” with respect to the Skein mathematical specification
- as fast or faster than the reference implementation in C
- unexpected discovery of a subtle corner-case bug in the designers’ C implementation: an arithmetic overflow, which leads to a loop iterating zero times, which leads to an undefined output

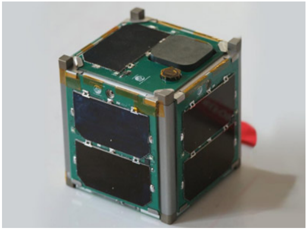
Secunet's Isabelle SPARK Plugin

- Secunet kept pushing the boundaries of expressiveness in SPARK contract language
- Stefan Berghofer (Secunet) implemented an Isabelle/HOL plug-in [Ber11] for SPARK proof functions
- released under free software license
- fully verified big-number library which they have used to implement RSA

Muen Project

- FLOSS separtion-kernel for the x86_64 architecture
- span off from the work of Secunet
- almost the entire kernel is written in SPARK
- automated proof of type-safety
- `http://muen.codelabs.ch/`

Vermont Tech CubeSat



- 14 mini satellites launched in November 2013
  - NASA ELaNa IV (Educational Launch of Nanosatellites)
- the only still fully operational mini satellite from this launch
- programmed mostly by undergraduate students
- several students with little or no overlap in time
- Prof. Carl Brandon attributes success of project SPARK
- slides about project: `http://www.cubesatlab.org/doc/Essex-High-School-2014-10-28.pdf`

Ironsides

- Ironsides is an authoritative DNS server implemented in SPARK [CF12]
- provably invulnerable to many of the problems that plague other servers
- by Dr. Martin C. Carlisle, director of the Academy Center for Cyberspace Research at the United States Air Force Academy
- `http://ironsides.martincarlisle.com`

Robot Navigation Software

- Piotr Trojanek and Kerstin Eder re-implemented well-known robot navigation algorithms in SPARK [TE14]
- aimed at automatic verification of absence of run-time errors
- errors found in the SPARK implementation were also errors in the widely used C versions
- errors found simply by testing, same simulation environment as with the C programs, SPARK (and Ada in general) had run-time checks detect out of bounds errors
- bonus effect: Piotr Trojanek started to contribute to the SPARK tools (open source), switched from a career in robotics, and is now hired as a permanent SPARK team member at Altran UK!

Important Steps on the Way, 20 years

Organisational:

- consultants, users, customer support, partnership, open-source

Technical:

- “user rules”, user-defined axioms
- quantifier support for completeness
- performance: use of processors, caching of VCs [BS12]

Counter Example Generation

- Riposte by Martin Brain (Bath/Oxford) and Florian Schanda (SPARK, Altran) [SB12]
- Used ASP (Answer Set Programming) on vintage SPARK to find counter examples
- sound and could be used as a prover too
- Fab side-effect: test suite with large portion of falsifiable VCs
- Currently experimenting with CVC4 counter examples

SMT and Victor

- by Paul Jackson, University of Edinburgh
- Victor [JP09] is an SMT translator and prover driver
- complementary to Simplifier (in-house Prolog prover), part of earlier SPARK tools, to discharge more VCs
- for SPARKSkein, 100 % of all VCs discharged automatically
- used also to audit user rules, in search of contradictions among axioms [JSW13]
- shipped with SPARK tools

Research Productisation Experiences

Large effort:

- porting to large number of platforms, can be very nasty
- nightly build & test, ability to fix bugs and ship wavefronts
- tuning 50+ command line flags and write user-friendly wrapper
- adapt POGS (proof obligation summary)
- large testing effort: adapt test suites, scripts to analyse test results
- bugs: parsing, type checking, axiom generation, soundness bugs etc.
- deterministic and predictable use of resources by provers

There is a lot of complexity in real systems. Question for any seemingly elegant theory: does it scale?

Challenges

- False alarm rate
- Usability: what to do with failed proof attempts?
- Managing Assumptions
- Benchmarks with falsifiable properties desirable
- Deterministic and predictable results
- Loops
- Floating-point verification
- Teaching...

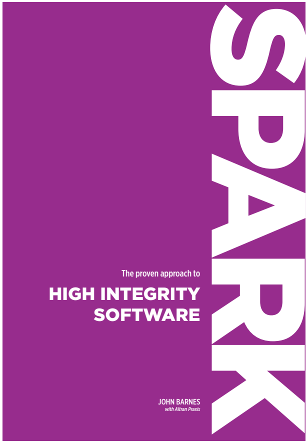
Recent/Ongoing Research Projects

- Formal semantics of SPARK 2014 in Coq [CAC^{+}13], CNAM, KSU
- Early work on dealing with assumptions [KCC^{+}14], Altran and AdaCore
- Floating-point verification, Oxford, Altran
  - SMT theory for IEEE-754 floating point reasoning: `http://smtlib.cs.uiowa.edu/theories/FloatingPoint.smt2`
  - Current version of semantics document: `http://smt-lib.org/papers/BTRW14.pdf`
- More advanced information-flow analysis, KSU, Rockwell Collins, [BHR^{+}11]
- Declassification policy for information-flow analysis, KSU, AdaCore, Altran
- Case studies with SPARK 2014: Mitsubishi Electric, Astrium
- ProofInUse `http://www.spark-2014.org/proofinuse/`, INRIA and AdaCore: bitvectors and counter examples

SPARK - Teaching

Consider teaching SPARK:

- formal and sound
- contracts, programming language based program verification
- industrially used
- open source
- mature tools
- support for academic faculty
- code examples, problems and sample answers
- excellent books; new book 2015 (Chapin, McCormick) in press, Barnes' book 3rd edition



Resources & Getting Started

- `http://www.spark-2014.org/`
- SPARK Community Page: `http://libre.adacore.com/tools/spark-gpl-edition/community/`
- GAP - GNAT Academic Program
  - Open-source, GPL release of SPARK tools
  - `http://libre.adacore.com/home/academia/`
  - Support from SPARK team for faculty
- Getting Started
  - Download the tools: `http://libre.adacore.com/download/`
  - User Guide: `http://docs.adacore.com/spark2014-docs/html/ug/`, chapter 5, SPARK tutorial, is a good start
  - SPARK 2014 Reference Manual: `http://docs.adacore.com/spark2014-docs/html/lrm/`
  - New to Ada? See `http://university.adacore.com/`

References I

[Ber11] Stefan Berghofer.
Verification of dependable software using SPARK and Isabelle.
In *SSV*, pages 15–31, 2011.

[BFPM11] François Bobot, Jean-Christophe Filliâtre, Andrei Paskevich, and Claude Marché.
Why3: Shepherd your herd of provers.
In *Proceedings of the First International Workshop on Intermediate Verification Languages, Boogie*, 2011.

[BHR$^+$11] Jason Belt, John Hatcliff, Robby, Patrice Chalin, David Hardin, and Xianghua Deng.
Enhancing SPARK's contract checking facilities using symbolic execution.
Ada Lett., 31(3):47–60, November 2011.

References II

[BS12] Martin Brain and Florian Schanda.
A lightweight technique for distributed and incremental program verification.
In *Verified Software: Theories, Tools, Experiments*, pages 114–129. Springer, 2012.

[CAC$^+$13] Pierre Courtieu, Maria Virginia Aponte, Tristan Crolard, Zhi Zhang, Fnu Robby, Jason Belt, John Hatcliff, Jerome Guitton, and Trevor Jennings.
Towards the formalization of SPARK 2014 semantics with explicit run-time checks using Coq.
In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, pages 21–22, New York, NY, USA, 2013. ACM.

References III

[CBW11] Roderick Chapman, Eric Botcazou, and Angela Wallenburg.
SPARKSkein: a formal and fast reference implementation of Skein.
In *Formal Methods, Foundations and Applications*, pages 16–27. Springer, 2011.
http://www.skein-hash.info/SPARKSkein-release.

[CF12] Martin C Carlisle and Barry S Fagin.
IRONSIDES: DNS with no single-packet denial of service or remote code execution vulnerabilities.
In *GLOBECOM*, pages 839–844, 2012.

References IV

[Cha10] Patrice Chalin.
Engineering a sound assertion semantics for the verifying compiler.
IEEE Trans. Software Eng., 36(2):275–287, 2010.

[CKM12] Cyrille Comar, Johannes Kanig, and Yannick Moy.
Integrating formal program verification with testing.
In *Proc. Embedded Real Time Software and Systems*, Toulouse, February 2012.

[HLL$^+$12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson.
Behavioral interface specification languages.
ACM Comput. Surv., 44(3):16, 2012.

References V

[Hoa03] Tony Hoare.
The verifying compiler: A grand challenge for computing research.
Journal of the ACM, 50:2003, 2003.

[JP09] Paul B Jackson and Grant Olney Passmore.
Proving SPARK verification conditions with SMT solvers.
Technical report, Technical Report, University of Edinburgh, 2009.

[JSW13] Paul Jackson, Florian Schanda, and Angela Wallenburg.
Auditing user-provided axioms in software verification conditions.
In *FMICS*, pages 154–168, 2013.

References VI

[KCC$^+$14] Johannes Kanig, Roderick Chapman, Cyrille Comar, Jerôme Guitton, Yannick Moy, and Emyr Rees.
Explicit assumptions - a prenup for marrying static and dynamic program verification.
In *8th International Conference on Tests & Proofs*, July 2014.

[KHCP00] Steve King, Jonathan Hammond, Roderick Chapman, and Andy Pryor.
Is proof more cost-effective than testing?
Software Engineering, IEEE Transactions on, 26(8):675–686, 2000.

References VII

[KSD12] Johannes Kanig, Edmond Schonberg, and Claire Dross.
Hi-Lite: the convergence of compiler technology and program verification.
In *Proceedings of the 2012 ACM conference on High integrity language technology*, HILT '12, pages 27–34, New York, NY, USA, 2012. ACM.

[O'N12] Ian O'Neill.
SPARK – a language and tool-set for high-integrity software development.
In Jean-Louis Boulanger, editor, *Industrial Use of Formal Methods: Formal Verification*. Wiley, 2012.

References VIII

[Rol] Martin Rolfe.
How technology is transforming air traffic management.

http://nats.aero/blog/2013/07/
how-technology-is-transforming-air-traffic-manage

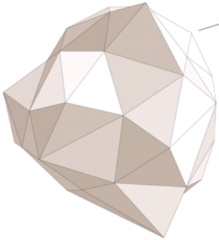
[RTC11a] RTCA.
DO-178C: Software considerations in airborne systems and equipment certification, 2011.

[RTC11b] RTCA.
DO-333: Formal methods supplement to do-178c and do-278a, 2011.

References IX

[SB12] Florian Schanda and Martin Brain.
Using answer set programming in the development of verified software.
In *ICLP (Technical Communications)*, pages 72–85, 2012.

[TE14] Piotr Trojanek and Kerstin Eder.
Verification and testing of mobile robot navigation algorithms: A case study in SPARK.
In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 1489–1494, 2014.



INNOVATION MAKERS