Safer Software through Better Abstractions and Static Analysis
Introduction
Security

Safety

Robustness
Implementation

Does the system work properly?
Buffer Overruns, NPEs, Dead-Ends, Unhandled Cases/Values

Does it do the right things?
Are the behaviours expressed in requirements implemented faithfully? Are all use cases supported?

Does it only do the right things?
Additional, unspecified behaviors are often loopholes and can be exploited maliciously.
Correct-by-Construction
The language/framework/API/modeling tool doesn’t let devs make a particular class of mistakes.

Analysis-and-Fix
You analyze the code/model after the fact and try to find problems which devs then fix.
Domain-Specificity, Ability to detect interesting problems, (Development Effort)

- Use a reasonable language
- Lint & Co
- High Q Testing
- Advanced Analyses
- Extend Language (domain-specific)
- Extend Language (compensate crap)
- Analyze/Understand (domain-specific)
Verification: Correct by Construction and very simple checks
Additional Constraints

Prevent the use of insecure functions

Detect insecure C functions:

```c
char* strcpy ( char* dest, const char* src);
```

Mark vetted secure functions as `SECURE API` and only use those.

Check this with tool.
Same Syntax, Different Semantics

Timing Side-Channel Attacks

Guess based on execution time

```cpp
for (i = 0; i < length; ++i) {
    if (password[i] != input[i])
        send("bad password");
}
```

Ensure constant or random time

```cpp
/* Returns 1 if a and b are equal, in constant time */
int s2n_constant_time_equals(string, string, len);
```

Amazon’s s2n library
Same Syntax, Different Semantics

Timing Side-Channel Attacks

/* Returns 1 if a and b are equal, in constant time */
int s2n_constant_time_equals(string, string, len);

Naming conventions not checkable.
Semantic Annotations better.

Then insert busy wait at end (to randomize timing) through preprocessor.
char* encryptData(char* k_enc, char* data) {
    char k_clr[256];
    decryptKey(k_enc, k_clr);
    char* encryptedData = // encrypt with k_clr
    return encryptedData;
}

When leaving the function, k_clr still on the stack – only stack pointer moved.

Better: As variables leave scope, wipe their memory. Use Generator.
Language Extension

Apple’s Got Fail Bug

Apple’s Intention:

```c
if (validateStep1(data, ...) != 0) goto fail;
if (validateStep2(data, ...) != 0) goto fail;
if (validateStep3(data, ...) != 0) goto fail;
fail: handleFailedValidation(data, errorcode, ...);
```

Actual Code

```c
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
    goto fail;
    goto fail;
    goto fail;

    ... other checks ...

    fail:
    ... buffer frees (cleanups) ...
    return err;
```
Language Extension

Apple’s Got Fail Bug

Apple’s Intention:

```c
if (validateStep1(data, ...) != 0) goto fail;
if (validateStep2(data, ...) != 0) goto fail;
if (validateStep3(data, ...) != 0) goto fail;
fail: handleFailedValidation(data, errorcode, ...);
```

More robust notation:

```c
trysequentially {
    validateStep1(data, ...);
    validateStep2(data, ...);
    validateStep3(data, ...);
} on fail (errorcode) {
    handleFailedValidation(data, errorcode, ...);
}
```
Language Extension

Aren‘t Macros enough?

```
#define GUARD(x) if ((x)<0) return -1
```

Amazon‘s s2n library

Danger:

```
GUARD( do_something ) + 1;
```

Macros have no protection against wrong use; Poor man‘s LE. Real LE is more robust/safer.
Language Extension

State Machines (in C)

Typically implemented via switch, function pointer tables or macro-based libraries.

This is error-prone:

- Hard to read and understand
- Hard to analyze b/c of „hidden“ semantics
- Easy to forget a break;
- Hierarchical Machines a syntactic nightmare
Language Extension

State Machines in C

exported statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
    macro stopped(next) = tp->speed == 0 mps
    macro onTheGround(next) = tp->alt == 0 m
    in event next(Trackpoint* tp) <no binding>
    in event reset() <no binding>
    out event crashNotification() => raiseAlarm
    readable var int16 points = 0
    state beforeFlight {
        entry { points = 0; }
        on next [tp->alt > 0 m] -> airborne
        exit { points += TAKEDOFF; }
    } state beforeFlight
    composite state airborne initial = flying {
        on reset [ ] -> beforeFlight { points = 0; }
        on next [onTheGround & stopped] -> crashed
        state flying (airborne.flying) {
            on next [onTheGround & tp->speed > 0 mps] -> landing
            on next [tp->speed > 200 mps] -> flying { points += VERY_HIGH_SPEED; }
            on next [tp->speed > 100 mps] -> flying { points += HIGH_SPEED; }
        } state flying
        state landing (airborne.landing) {
            on next [stopped] -> landed
            on next [ ] -> landing { points--; }
        } state landing
        state landed (airborne.landed) {
            entry { points += LANDING; }
        } state landed
    } state airborne
    state crashed {
        entry { send crashNotification(); }
    } state crashed
}
Language Extension

State Machines in C

```c
exported statemachine FlightAnalyzer initial = beforeFlight {

    next(Trackpoint* tp)
    reset()

    | States          | Events                                                                 |
    |-----------------|------------------------------------------------------------------------|
    | beforeFlight    | [tp->alt > 0 m] -> airborne                                             |
    | airborne        | [tp->alt == 0 m && tp->speed == 0 mps] -> crashed                     |
    |                 | [tp->alt == 0 m && tp->speed > 0 mps] -> landing                      |
    |                 | [tp->speed > 200 mps && tp->alt == 0 m] -> airborne                   |
    |                 | { points += VERY_HIGH_SPEED; }                                        |
    |                 | [tp->speed > 100 mps && tp->speed <= 200 mps && tp->alt == 0 m] -> airborne |
    |                 | { points += HIGH_SPEED; }                                            |
    | landing         | [tp->speed == 0 mps] -> landed                                         |
    | landed          | [tp->speed < 0 mps] -> landing                                         |
    | crashed         | [tp->speed == 0 mps] -> crashed                                       |
    |                  | [tp->speed < 0 mps] -> landing                                         |
    |                  | { points--; }                                                        |
    | NextRound <j>    | [tp->speed == 0 mps] -> crashed                                       |
    |                  | [tp->speed < 0 mps] -> landing                                         |

`}

Various Table-based notations or even diagrams can make the situation even better.
A set of C extensions optimized for embedded software dev.

Research Project at itemis and Fortiss between 2011 and 2013. Since then open sourced, used in projects.
A set of C extensions optimized for embedded software development.

Many different extensions for embedded programming, program analysis and verification as well as process aspects. User-defined extensions seamlessly supported.

<table>
<thead>
<tr>
<th>User Extensions</th>
<th>User-defined Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>Visualizations</td>
</tr>
<tr>
<td>Logging</td>
<td>PLE Variability</td>
</tr>
<tr>
<td>Utilities</td>
<td>Requirements &amp; Tracing</td>
</tr>
<tr>
<td>Messaging</td>
<td>Documentation</td>
</tr>
<tr>
<td>Components</td>
<td>Reports &amp; Assessments</td>
</tr>
<tr>
<td>Physical Units</td>
<td>State Machine Verification</td>
</tr>
<tr>
<td>State Machines</td>
<td>Component Contracts Verification</td>
</tr>
<tr>
<td>Concurrency</td>
<td>C Verification</td>
</tr>
<tr>
<td>Importer</td>
<td>Decision Table Checking</td>
</tr>
<tr>
<td></td>
<td>PLE Variability Checking</td>
</tr>
</tbody>
</table>

Languages shipped with mbeddr

- C99

Platform

- Libraries for web server, node navigation, additional notations, pattern matching, palettes, XML processing, debugging...

MPS

- Syntax Highlighting, Code Completion, Goto Definition, Find Usages, Type Checking, Data Flow Analysis, Refactoring, Versioning, Debugging

Foundation

- C Compiler & Debugger
- PlantUML
- Latex
- HTML
- CBMC
- Z3
- Sat4J

Implementation |

Process |

Analysis
Verification: Type Systems
Advanced Type Checking

Preventing Overflow

```plaintext
// integer type, unlimited range
type number = number[-inf|inf]{0}
// positive integer
number[0|inf] = number[0|inf]{0}
// integer type, range as specified
number[10|20] = number[10|20]{0}
// decimal type with 2 decimal places, unlimited range
number{2} = number[-inf|inf]{2}
// range as specified, precision derived from range decimals
number[3.3|4.5] = number[3.3|4.5]{1}
```

```scala
type temperature: number[36|42]{1}
type measuredTemp: number[35|43]{2}

val T_measured: measuredTemp = 42.22
val T_calibrated: temperature = T_measured * 0.93
```

Error: type number[32.55|39.99]{4} is not a subtype of number[36|42]{1}
fun getData(url: string): string<tainted> { val v1 = taint[url] }
fun storeInDB(data: string<!tainted>): boolean { true }

fun createBlogPost(postID: number) {
    val inferred = getData("my/input/form")
    val explicit: string<tainted> = getData("my/input/form")
    val invalid: string<!tainted> = getData("my/input/form")

    // This does not work because
    val v2 = storeInDB(inferred)
    Error: type string<tainted> is not a subtype of string<!tainted>

    // tainted always wins
    val dirty: string<tainted> = sanitize[inferred] + "Hello"

    // works after sanitizing.
    val stored: string<!tainted> = sanitize[inferred]
    val success = storeInDB(stored)
}
val somethingUnclassified: string = "hello"
val somethingConfidential: string<confidential> = "hello"
val somethingSecret: string<secret> = "hello"
val somethingTopSecret: string<topsecret> = "hello"

fun publish(data: string) = data
val p1 = publish(somethingUnclassified)
val p2 = publish(somethingConfidential)
val p3 = publish(somethingSecret)
val p4 = publish(somethingTopSecret)

fun putIntoCIAArchive(data: string<confidential->) = data
val a2 = putIntoCIAArchive(somethingConfidential)
val a1 = putIntoCIAArchive(somethingUnclassified)
val a3 = putIntoCIAArchive(somethingTopSecret)
val a4 = putIntoCIAArchive(somethingSecret)

fun putIntoCIAArchive(data: string<confidential->) = data
val a2 = putIntoCIAArchive(somethingConfidential)
val a1 = putIntoCIAArchive(somethingUnclassified)
val a3 = putIntoCIAArchive(somethingTopSecret)
val a4 = putIntoCIAArchive(somethingSecret)

fun tellANavyGeneral(data: string<secret->) = data
val g1 = tellANavyGeneral(somethingConfidential)
val g2 = tellANavyGeneral(somethingSecret)
val g3 = tellANavyGeneral(somethingTopSecret)
val g4 = tellANavyGeneral(somethingUnclassified)
Verification: SMT Solving
For a set of equations $e_i \subseteq E$ with free variables $v_i$, is there an assignment of values to all $v_i$ such that all $e_i$ are true?

- No $\Rightarrow$ UNSAT
- Yes $\Rightarrow$ Value Assignments („Model“)

**E:**

$$2 \times x \equiv 3 \times y$$
$$x + 2 \equiv 5$$

**Model:**

$$x = 3, y = 2.$$ 

Solvers can do this quickly for large sets of equations with many free variables.

Many „Theories“: Booleans, Integers, Reals, Collections, ...
Example: Decision Table

<table>
<thead>
<tr>
<th>mode == MANUAL</th>
<th>mode == AUTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed &lt; 30</td>
<td>false</td>
</tr>
<tr>
<td>speed &gt; 30</td>
<td>false</td>
</tr>
<tr>
<td>speed &gt; 40</td>
<td>true</td>
</tr>
</tbody>
</table>

Overlap-Freedom:

\[
i \neq j \land \bigwedge_{i' = 1}^{n} i = i' \Rightarrow r_{i'} \land \bigwedge_{j' = 1}^{n} j = j' \Rightarrow r_{j'}
\]

Completeness:

\[-(speed < 30) \land -(speed > 30) \land -(speed > 40)\]

or: with \( n \) row headers \( r_1, ..., r_n \),

\[\bigwedge_{i=0}^{n} \neg r_i\]
Typical Solver Tasks

<table>
<thead>
<tr>
<th>Applicability: Is there a value assignment satisfying all $E_i$? Examples are any set of boolean expressions, or even a single complex one.</th>
<th>Equality: Are the $E_i$ semantically equivalent, even though they differ structurally (think: DeMorgan laws). Examples include refactorings that simplify expressions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completeness: For any combination of inputs, does at least one expression $E_i$ match? Examples include conditionals, switch statements, alt-expressions (see below), decision tables, or transition guards in state machines.</td>
<td>Subset: For any $i \in {1, \ldots, n}$, are the values satisfying $E_{i+1}$ a subset of those satisfying $E_i$? The canonical example is a list of ordered decisions, the earlier one must be narrow to not shadow later ones; any kind of subtyping through constraints such as chained typedefs; producer-consumer relationships where the consumer must be able to consume everything the producer creates, or possibly more.</td>
</tr>
</tbody>
</table>

**Decision Tables, Guards in SM**

Solvers can also be used to verify (large subsets of) programs wrt. arbitrary properties. But that’s outside our scope.
Verification: Correctness of switch

SMT Solvers can help find/fix non-trivial logic errors in programs.

```latex
\begin{verbatim}
fun example(x: number) = alt
  [x < 0 && x > 0 => 1]
  [x == 5 => 2]

Error: [MANUALLY CHECKED] This alternative can never be true.

fun example(x: number) = alt
  [x > 0 => 1]
  [x < 0 => 2]

Error: [MANUALLY CHECKED] Alternatives missing. For instance the following case is not covered: x = 0

fun example(x: number) = alt
  [x >= 0 => 1]
  [x <= 0 => 2]

Error: [MANUALLY CHECKED] Overlapping alternatives. The following case is covered by multiple alternatives: x = 0
\end{verbatim}
```
Verification: Correctness of Tables

SMT Solvers can help find/fix non-trivial logic errors in programs.
Verification: Order/Shadowing

Earlier rows must be more specific than later rows to avoid shadowing.

```
enum REGION { EU, ASIA, NA, ME }
enum COUNTRY { DE, FR, US, CA, JA }
type cur: number[0|∞]{2}
fun minutePrice(region: REGION, country: COUNTRY, rebated: boolean) =

<table>
<thead>
<tr>
<th>region</th>
<th>country</th>
<th>rebated</th>
<th>local: cur</th>
<th>longDis: cur</th>
</tr>
</thead>
<tbody>
<tr>
<td>EU–rebated</td>
<td>EU</td>
<td>true</td>
<td>0.80</td>
<td>1.00</td>
</tr>
<tr>
<td>EU–non-rebated</td>
<td>EU</td>
<td>false</td>
<td>0.85</td>
<td>1.10</td>
</tr>
<tr>
<td>DE</td>
<td>EU</td>
<td>DE, FR</td>
<td>false</td>
<td>0.82</td>
</tr>
<tr>
<td>US</td>
<td>NA</td>
<td>US</td>
<td>0.70</td>
<td>0.75</td>
</tr>
<tr>
<td>CA</td>
<td>NA</td>
<td>CA</td>
<td>0.75</td>
<td>0.80</td>
</tr>
<tr>
<td>REST</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.20</td>
</tr>
</tbody>
</table>
```
Solver Integration

General Pattern:

Specific Approach for Solver
KernelF core features

**Primitive Types**
Numbers, Booleans, Strings, Enums, Records

**Arithmetic Operations**
+ - * /

**Comparison Operations**
> >= < <= == !=

**Functions**
Definitions, Calls, Higher-Order

KernelF is a functional base language
KernelF Context: DSL Development

New Language

Analyze Domain to find Abstractions
Define suitable, new notations
Rely on existing behavioral paradigm
Reuse standard expression language
Interpret/Generate to one or more GPLs
KernelF Context: Three DSL Patterns

GPL Extension
Reuse GPL incl. Expressions and TS
Add/Embed DS-extensions
Compatible notational style
Reduce to GPL

New Language
Analyze Domain to find Abstractions
Define suitable, new notations
Rely on existing behavioral paradigm
Reuse standard expression language
Interpret/Generate to one or more GPLs

Formalization
Use existing notation from domain
Clean up and formalize
Generate/Interpret
Often import existing „models“
Verification: Model Checking
State machines express how a system’s state changes over time, as it is stimulated externally, and how the system reacts to stimuli depending on the state it is in.
State machines express how a system's state changes over time, as it is stimulated externally, and how the system reacts to stimuli depending on the state it is in.
Every imperative program can be seen as a (very complicated) state machine.
Every imperative program can be transformed into a (very complicated) state machine.
Temporal logic can express time (= state change) as part of logic expressions.

1. For all possible traces, after SM has been in state A, it will eventually move into state B.
2. For all possible traces, after SM has been in state A, it will move into state B or state C in the next step.
3. There exists an trace where, after SM has been in state A, it will eventually move into state B.
4. Before being in state B, SM has always been in A.
5. For all possible traces, SM will never reach state A after it was in state C.
6. Whenever SM is in state A, a variable v will never have a value greater than 10.
Temporal Quantifiers are at the core of TL.

1. $\text{AG}((\text{state} == \text{A}) \Rightarrow \text{AF}(\text{state} == \text{B})).$
2. $\text{AG}((\text{state} == \text{A}) \Rightarrow \text{AF}(\text{state} == \text{B} \lor \text{state} == \text{C})).$
3. $\text{EF}(\text{state} == \text{A}) \Rightarrow \text{EF}(\text{state} == \text{B}).$
4. $\text{EF}(\text{state} == \text{B}) \Rightarrow (\text{state} == \text{A}).$
5. $\text{AG}((\text{state} == \text{C}) \Rightarrow \neg \text{EF}(\text{state} == \text{A})).$
6. $\text{AG}(\text{state} == \text{A}) \Rightarrow (v > 10).$
Express a system as a state machine.

Transform a system into a state machine.

Express allowed and disallowed behaviors (state evolutions over time) as temporal logic properties.

Use a Model Checker to verify that a property always holds (or show a counter example).

Assuming (!) a complete set of properties, this proves correctness of the system.
Express properties as interface contract. Generate C, invoke CBMC to perform model check, lift results.
Heartbleed is a security bug in the OpenSSL cryptography library, which is a widely used implementation of the Transport Layer Security (TLS) protocol. It was introduced into the software in 2012 and publicly disclosed in April 2014. Heartbleed may be exploited regardless of whether the party is using a vulnerable OpenSSL instance for TLS as a server or a client. It results from improper input validation (due to a missing bounds check) in the implementation of the TLS heartbeat extension,[3] thus the bug's name derives from heartbeat.[4] The vulnerability is classified as a buffer over-read,[5] a situation where more data can be read than should be allowed.[6]

Heartbleed is registered in the Common Vulnerabilities and Exposures database as CVE-2014-0160.[5] The federal Canadian Cyber Incident Response Centre issued a security bulletin advising system administrators about the bug.[7] A fixed version of OpenSSL was released on April 7, 2014, on the same day Heartbleed was publicly disclosed.

As of May 20, 2014, 1.5% of the 800,000 most popular TLS-enabled websites were still vulnerable to Heartbleed.[8]

TLS implementations other than OpenSSL, such as GnuTLS, Mozilla's Network Security Services, and the Windows platform implementation of TLS, were not affected because the defect existed in the OpenSSL's implementation of TLS rather than in the protocol itself.[9]
Verification: Heartbleed Bug II

What they wanted to express:

```c
struct {
    uint16 payload_length;
    unsigned char payload[payload_length];
} HeartbeatMessage;
```

The above code is invalid in C
cannot „dynamically“ initialize length.

Instead „manual“ synchronization between the two values.

**Bug:** sizeof(payload) != payload_length
Led to Buffer Over-read.
Verification: Heartbleed Bug III

Find Problem via formal Verification

```c
HeartbeatMessage prepareUntrustedMessage() {
    HeartbeatMessage msg;
    assign nondet msg;
    return msg;
}
```

// Parsing the message
size_t length = pM->payload_length;
uint8* p = pM->payload;

// Just some memory to read into, allocate
void* dest = malloc(length);

// Here a problem happens, because
memcpy(dest, p, length);

// Verification (CBMC)

<table>
<thead>
<tr>
<th>Idx</th>
<th>Property</th>
<th>Status</th>
<th>Trac.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>dereference fail...</td>
<td>FAIL</td>
<td>59</td>
<td>0.44s</td>
</tr>
<tr>
<td>41</td>
<td>dereference fail...</td>
<td>SUCCESS</td>
<td>0.35s</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>arithmetic overfl</td>
<td>SUCCESS</td>
<td>0.29s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idx</th>
<th>Raw</th>
<th>Thres.</th>
<th>m.payload_le...</th>
<th>Kind</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>m.payload_le...</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>m.payload_le...</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
<td>m.payload[0]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>46</td>
<td>0</td>
<td>return</td>
<td></td>
<td>prepareUntrustedMessage()</td>
</tr>
</tbody>
</table>
```
Even better: **First-class message concept** that handles low level message stuff.

```c
uint8[6] f_time = {0x00A,  // field type identifier UNIT_TIME24,
                   3,  // 3 payload bytes follow
                     10, 20, 00};

message CurrentMeasuredValue:42 {
  int32 timestamp;  // time of measurement
  uint16/A/ value;  // measured value in Amps
  uint16 accuracy;  // accuracy in 1/100 %
}
message ... { ... }
...'''
```
Verification: Pacemaker Challenge I

Report from Dagstuhl Seminar 14062

The Pacemaker Challenge: Developing Certifiable Medical Devices

Edited by
Dominique Méry¹, Bernhard Schätz², and Alan Wassnyng³

¹ LORIA – Nancy, FR
² fortiss GmbH – München, DE
³ McMaster University – Hamilton, CA, wassnyng@mcmaster.ca

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 14062: "The Pacemaker Challenge: Developing Certifiable Medical Devices". The aim of the seminar was to bring together leading researchers and industrial partners of this field; the seminar edition 2014 gathered 28 participants from 8 countries: Canada, Denmark, France, The United States, Germany, United Kingdom, Brazil. Through a series of presentations, discussions, and working sessions, the seminar attempted to get a general view of the field of medical devices and certification, as well as the ecosystem around the pacemaker challenge. The seminar brought together, on the one hand, researchers from the different notations and various tools. The main outcome of the seminar is the exchange of information between different groups and the project of a book.

Seminar February 2–7, 2014 – http://www.dagstuhl.de/14062
1998 ACM Subject Classification D.2 Software Engineering, K.4 Computers and Society

Keywords and phrases Embedded systems, Real-time systems, Medical devices, Model-driven development, Software certification, Validation & verification, Formal methods

Digital Object Identifier 10.4230/DagRep.4.2.17
Verification: Pacemaker Challenge II

Better: extend C with a DSL to express safety properties as well as proof strategies.

Diagram:
- Refined Requirements
- Complying to Requirements
- Manual Encoding and Validation
- Manual Result Interpretation
- Reduced Abstraction Gap
- Model, Environment, Verification Conditions
- Counterexample in Model Terms
- Automatic Translation
- Verifiable C Implementation on C level
- C Verification
- Refined Requirements
- Complying to Requirements
- Manual Encoding, Validation and Result Interpretation on a Higher Abstraction Level
- Counterexample
Verification: Pacemaker Challenge II

Left: system is modeled as a state machine in C

Right: the representation in low-level C

```c
statemachine VVI initial = Start
in config(int lri, int vrp)
in s
in t
out p => doPace()
int c = 0
int LRI = 0
int VRP = 0
state Start {
on config -> Start {
  LRI = lri;
  VRP = vrp; }
on t -> Pace
}
state Wait {
on s [c <= VRP] -> Wait
  on s [c > VRP] -> Wait {
    c = 0; }
on t [c < LRI] -> Wait { ++c; }
on t [c == LRI] -> Pace
}
state Pace {
on entry {
  send p;
  c = 0; }
on t -> Wait {++c; }
}
typedef enum states {
  Start, Wait, Pace
} VVI_States;
void execute(Events_VVI e, int** args)
{
  switch (state) {
  case Start: {
    switch (e) {
    case config: {
      LRI = *args[0];
      VRP = *args[1];
      state = Start;
      return; }
    case t: {
      state = Pace;
doPace();
c = 0;
      return; }
    case s: {
      if (c <= VRP) {
        state = Wait;
      } // ...
    case Wait: {
  ```
Verification: Pacemaker Challenge III

Safety Properties (and their tx to C)

```c
after smlsInState(Wait) until smlsInState(Pace) exists c == LRI

static int q = 0;
static int e = 1;
if (state == Pace) {
    q = 0;
    assert(e);
    e = 1;
}

if (q) {
    e = (c == LRI) || e;
}
if (state == Wait && !q)
    { q = 1; e = 0; }
```

A

```c
nondet_choice {
    choice {
        smtrigger(s);
        assume(0 <= ch && ch < 2);
    }
    choice {
        // nothing
    }
    // nothing
}
```

B

```c
smStateSubset Initial:
smInState(Pace) && VRP < LRI;

smNonDetInit(Initial):
state = (VVI_States) nondet_int();
assume(0 <= state && state <= 2);
c = nondet_int();
LRI = nondet_int();
VRP = nondet_int();
assume(state == Pace && VRP < LRI);
```

C
Verification: Pacemaker Challenge IV

Inductive Proof

\[
\text{induction on } t \\
\text{from Initial} \\
\text{until MAX\_LRI} \{ \\
\quad \text{nondeterministically } \text{smtrigger}(s); \\
\quad \text{after } \text{smIsInState}(\text{Wait}) \\
\quad \text{until } \text{smIsInState}(\text{Pace}) \\
\quad \text{exists } c == \text{LRI} \} \\
\quad \text{smNonDetInit}(\text{Initial}); \\
\quad \text{for (int } \text{step} = 0; \text{ step} < \text{MAX\_LRI}; \text{ ++step) } \{ \\
\quad \quad \text{nondeterministically } \text{smtrigger}(s); \\
\quad \quad \text{after } \text{smIsInState}(\text{Wait}) \\
\quad \quad \text{until } \text{smIsInState}(\text{Pace}) \\
\quad \quad \text{exists } c == \text{LRI}; \\
\quad \quad \text{smtrigger}(t); \\
\quad \quad \text{after } \text{step} > 1 \text{ until } \text{step} == \text{MAX\_LRI} \\
\quad \quad \text{exists } \text{smIsInStateSubset}(\text{Initial}); \} 
\]
Generic tools don’t find specific problems

Guide the tools

Translate to their input language
Verification: Test Generation
Test Vector Generation

For everything that can be seen as taking a list of arguments, those can be synthesized.

*test case* TestFunctions [success] {  

- **vectors function add** -> producer: random 25
  - **results:** true
    - 0: valid
      - a: 3, a2: -3.06, b: 3, c: false, s: "BLUE", res: 3, status: ok
    - 1: valid
      - a: 2, a2: 2.74, b: 2, c: false, s: "BLUE", res: 2, status: ok
    - 2: valid
      - a: 1, a2: 0.22, b: 1, c: false, s: "BLUE", res: 1.22, status: ok
    - 3: invalid input
      - a: 4, a2: -0.45, b: 4, c: true, s: "BLUE", res: 7l:6h7sg!afLmULG UbwtI00H9", not executed
    - 4: invalid input
      - a: 1, a2: 1.38, b: 1, c: false, s: "RED", res: "Mtoa7J66uyTye2f2-fLhDSj8C2K", not executed
    - 5: invalid input
      - a: 1, a2: -2.63, b: 1, c: false, s: "BLUE", res: "n66r7E (f0J$aQMj$", not executed

- **vectors function plus** -> producer: eqclass
  - **results:** true
    - 0: valid
      - a: 0, b: -10, c: GREEN, res: -10, status: ok
    - 1: valid
      - a: 0, b: -10, c: BLUE, res: -10, status: ok
    - 2: valid
      - a: 0, b: 10, c: GREEN, res: 10, status: ok
    - 3: valid
      - a: 0, b: 10, c: BLUE, res: 10, status: ok
    - 4: valid
      - a: 0, b: -1, c: GREEN, res: -1, status: ok

*test case* TestFunctions [success] {  

- **vectors function add** -> producer: random 25
  - **results:** true
    - 0: valid
      - a: 7, b: 2, res: no expected value given; actual was 14
    - 1: valid
      - a: 1, b: 8, res: no expected value given; actual was 8
    - 2: valid
      - a: 5, b: 2, [POST] res == a * b
    - 3: valid
      - a: 7, b: 8, res: no expected value given; actual was 56
    - 4: valid
      - a: 5, b: 6, [POST] res == a * b
    - 5: valid
      - a: 8, b: 0, [PRE] b > 0
    - 6: valid
      - a: 5, b: 8, [POST] res == a * b
For a set of tests that all succeed, if after a change to the program they still do, this is a problem.
Coverage Analysis

Are all parts of the test code executed in the test?
Is every language construct used in the tests, with all relations, and in which degrees of complexity?
Coverage Analysis

Are all parts of the interpreter (incl. all branches) executed in the tests?

### org.iets3.core.expr.base

- TupleValue: Covered.
- SomeValExpr: Covered.
- LogicalImpliesExpression: Covered.
- ErrorExpression: Covered.
- RangeTarget: Covered.
- TupleAccessExpr: Covered.
- SomeExpression: Covered.
- ParenExpression: Covered.
- TryExpression: Partial. Missing: [success]
- IfExpression: Covered.

---

Total 99, new 1, ok 0

Value ranges:
- decimal.max = 340.00
- decimal.min = -80.01
- decimal.zero = true
- integer.zero = true
- integer.max = 99999
- decimal.minusOne = true
- integer.one = true
- integer.minusOne = true
- decimal.one = true
- integer.min = -99383

Coverage 99%
Validation: Review
This is better than a switch-case:

```csharp
statemachine TrafficLights initial = red {
    in event timer(int64 t) <no binding>
    in event buttonPressed() <no binding>
    var boolean button = false
    var int64 tEnter = 0
    state red {
        on buttonPressed [ ] -> red { button = true; }
        on timer [t - tEnter > 1000] -> green { tEnter = t; }
    } state red
    state green {
        on timer [t - tEnter > 500] -> red { tEnter = t; }
    } state green
}
```

<table>
<thead>
<tr>
<th>States</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>timer(int64 t)</td>
</tr>
<tr>
<td>red</td>
<td>[t - tEnter &gt; 1000] -&gt; green { tEnter = t; }</td>
</tr>
<tr>
<td>green</td>
<td>[t - tEnter &gt; 500] -&gt; red { tEnter = t; }</td>
</tr>
<tr>
<td></td>
<td>buttonPressed()</td>
</tr>
<tr>
<td></td>
<td>[ ] -&gt; red</td>
</tr>
<tr>
<td></td>
<td>{ button = true; }</td>
</tr>
</tbody>
</table>
```c
int32 averageIntArray(int32[] arr, int32 size) {
    size
    Σ arr[i]
    i = 0
    return ----- ;
} averageIntArray (function)
```
Code Review and Security Audits

Track review state in the code

```
exported component Judge2 extends nothing {
    int16 points = 0;
    void judger_reset() <= op judger.reset {
        points = 0;
    }
}
```

Assess review state over the system:

Assessment: ReviewOfComponentsStuff
query: code review summary for chunk Components

<table>
<thead>
<tr>
<th>reviewed</th>
<th>instancesJudging [InstanceConfiguration]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td>Judge2 [AtomicComponent]</td>
</tr>
<tr>
<td>raw</td>
<td>testJudging [TestCase]</td>
</tr>
<tr>
<td>not reviewed yet</td>
<td>ContractMessages [MessageDefinitionTable]</td>
</tr>
</tbody>
</table>
Validation: Simulation
fun circumference(r: number[0∞]) = \pi \times 2 \times r

fun hypotenuseLength(a: number, b: number) = \sqrt{a^2 + b^2}

fun cathetusLength(a: number, c: number) = \sqrt{c^2 - a^2}

---

test case testPythagoras [fail] {
  assert hypotenuseLength(3, 4) equals 5 [1 ms]
  assert cathetusLength(3, 6) equals 4 actual: 6.708203932499369415154433227144181728363037109375
}

test case testCircumference [fail] {
  assert circumference(-5) equals 14 actual: <constraint failed>
}
Simulation

State machine **AnnotatedSM1**

- **Inputs**
  - Submit inputs

- **Interact**
  - foo
  - bar
  - zum

- **Main**
  - Time in state: 0 days 00:00:00
  - Current state: BigA, A2

- **Log**

<table>
<thead>
<tr>
<th>Time</th>
<th>Component</th>
<th>State</th>
<th>Type</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 days 00:00:00</td>
<td>main</td>
<td>BigA, A1</td>
<td>COMMUNICATION</td>
<td>sending message &quot;in&quot;</td>
</tr>
<tr>
<td>0 days 00:00:00</td>
<td>main</td>
<td>BigA, A1</td>
<td>COMMUNICATION</td>
<td>s = in A1</td>
</tr>
<tr>
<td>0 days 01:00:00</td>
<td>-</td>
<td>-</td>
<td>CONTROL</td>
<td>Ticks 3600</td>
</tr>
<tr>
<td>0 days 02:00:00</td>
<td>-</td>
<td>-</td>
<td>CONTROL</td>
<td>Ticks 3600</td>
</tr>
<tr>
<td>0 days 03:00:00</td>
<td>-</td>
<td>-</td>
<td>CONTROL</td>
<td>Ticks 3600</td>
</tr>
<tr>
<td>0 days 04:00:00</td>
<td>main</td>
<td>BigA, A1</td>
<td>COMMUNICATION</td>
<td>Processing event bar</td>
</tr>
<tr>
<td>0 days 04:00:00</td>
<td>main</td>
<td>BigA, A2</td>
<td>STATE</td>
<td>Following transition on bar to A2</td>
</tr>
<tr>
<td>0 days 04:00:00</td>
<td>main</td>
<td>BigA, A2</td>
<td>STATE</td>
<td>Entering state BigA, A2</td>
</tr>
<tr>
<td>0 days 04:00:00</td>
<td>main</td>
<td>BigA, A2</td>
<td>COMMUNICATION</td>
<td>sending message &quot;in&quot;</td>
</tr>
<tr>
<td>0 days 04:00:00</td>
<td>main</td>
<td>BigA, A2</td>
<td>COMMUNICATION</td>
<td>s = in A2</td>
</tr>
</tbody>
</table>
Simulation
Simulation

Diarrhea

Have you had any of these symptoms?

Severe Cramping
- Yes
- No

Fever
- Yes
- No

Nausea/Vomiting
- Yes
- No

Have you been confined to your home as a result of your diarrhea?
- Yes
- No

Next
Impact Analysis
Safer Software through Better Abstractions and Static Analysis

Validation: Tracing
Requirements

6.1 You should land as short as possible
ShortLandingRoll /functional: tags
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Suspendisse potenti. Etiam risus ante, bibendum ut mattis
vel condimentum velit. Quisque vehicula faucibus tellus Phasellus
rhoncus quam eu dui dictum sollicitudin

6.2 Once you land successfully, you get another
FullStop /functional: tags
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Suspendisse potenti. Etiam risus ante, bibendum ut mattis
purposes, this one references @req(InFlightPoints)

Rules

calculation PointForATrackpoint: This rule computes the points a
It does so by taking into accounted as arguments.

parameters: [int16 alt: current altitude of the trackpoint]
result = (BASEPOINTS * 1) *

<table>
<thead>
<tr>
<th>speed &gt; 180</th>
<th>speed &gt; 130</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

tests:
PointForATrackpoint(500, 100) == 0
PointForATrackpoint(500, 1200) == 0
PointForATrackpoint(1100, 165) == 200
PointForATrackpoint(2100, 140) == 100
PointForATrackpoint(2100, 200) == 300

Tracing

[constant TAKEOFF = 100;] => implements PointsForTakeoff
[constant HIGH_SPEED = 10;] => implements FasterThan100
[constant VERY_HIGH_SPEED = 20;] => implements FasterThan200
[constant LANDING = 100;] => implements FullStop

[checked]
exported statemachine FlightAnalyzer initial = beforeFli
state beforeFlight {
    entry { points = 0; }
    on next [tp->alt > 0 m] -> airborne
    [exit { points += TAKEOFF; }] => implements PointsForTakeoff
} state beforeFlight

Visualisations
Trace from **every** program element to requirements (internal or external)

Analyze and find untraced program fragments.

Different kinds of traces to express semantics.
Assuring the Transformation
What good is all the abstraction if we cannot trust the translation to the implementation?

A question of tool qualification.
Baseline Tool Architecture

Risk Analysis

+ Risk Analysis
+ Mitigations
Risk Analysis

- **Compensating**
  - Requirements
  - Model $M_S$
  - Model $M_T$
  - Language
  - Language

- **Related**
  - Trafo $T_S$
  - Trafo $T_T$

- **Random Error**
  - Systematic Error
  - Error

- **Transformation Engine**
  - CSC Program
  - CSC Tests

- **Runtime System**
  - Error
Safe Tool Architecture

Validation
- Review
- Walkthrough
- Simulation

System Model $M_S$
- ensures
- $T$

T&V Model $M_T$
- $T$

Different DEVS
- $T$
- $T_S$
- $T_T$

RTE
- $T$
- System Implementation $I_S$
- ensures
- $T$

T&V Implementation $I_T$
- $T$

RTE II
- $T$

Fuzzing

Redundant Execution

Review Subset

Ensure Coverage

Static Analysis

Pen Testing
Used in Medical System
Configuration
for visualizations, simulations and documentation

State Machines
components, parameters, instantiation, states, transitions, events

Test and Scenarios
test vector generation, algo instantiation, user interaction/events

KernelF Extensions
decision trees and tables, dealing with time

KernelF Expressions
binary ops, if-then-else, primitive types and literals, collections
Wrap Up
Domain-Specificity, Ability to detect interesting problems, (Development Effort)

Advanced Analyses

High Q Testing

Lint & Co

Use a reasonable language

Analyze/Understand (domain-specific)

Extend Language (domain-specific)

Extend Language (compensate crap)

High Q Testing

Use a reasonable language

Language Analysis Tools

Wrap Up

Try sequentially {
  validateStep1(data, ...);
  validateStep2(data, ...);
  validateStep3(data, ...);
  on fail (errorCode) {
    handleFailedValidation(data, errorCode, ...);
  }
}
• Verify the VM
• Clean up Solidity (eg., reentrancy)
• Integrate Solver/Model Checker into Solidity for require/assert clauses
• Higher-level language constructs that express state machines and declarative „contract“ patterns (eg., decisions)
• DSLs for particular app domains
An Overview of Program Analysis using Formal Methods
With a Particular Focus on their Relevance for DSLs

Markus Voelter
with Tamás Szabó and Björn Engelmann

An addendum to the book DSL Engineering

Static program analysis refers to determining properties of programs without executing it, relying on a range of formal methods. While these methods have been around for a long time, over the last couple of years, some of these methods started to scale to solve problems of interesting size. We have used advanced type systems, abstract interpretation, SMT solving and model checking to answer relevant questions about programs written with various DSLs. In this booklet we introduce the methods, illustrate what we have done with them, and describe how we have integrated the analysis method and existing tools with languages and IDEs.

Note that this booklet documents the authors’ experience. This is not a scientific paper. There is no contribution. The aim is to explain and illustrate.

Version 1.0 | July 18, 2017

Automated Testing of DSL Implementations
Experiences from Building mbeddr

Daniel Ratiu · Markus Voelter · Domenik Pavletic

Received: date / Accepted: date

Abstract Domain specific languages promise to improve productivity and quality of software development by providing problem-adequate abstractions to developers. Projectional language workbenches, in turn, allow the definition of modular and extensible domain specific languages, generators and development environments. While recent advances in language engineering have enabled the definition of DSLs and tooling in a modular and cost-effective way, the quality assurance of their implementation is still challenging. In this paper we discuss our work on testing different aspects of the implementation of domain specific languages and associated tools, and present several approaches to increase the automation of language testing. We illustrate these approaches with the JetBrains MPS language workbench and our experience with testing mbeddr, a set of domain specific languages and tools on top of C tailored to embedded software development. Based on the experience gained from the mbeddr project, we extract generic lessons for practitioners as well as challenges which need more research.

Keywords domain specific languages, testing, quality assurance, automation
Using Language Workbenches and Domain-Specific Languages for Safety-Critical Software Development

Markus Voelter
independent / itemis AG
voelter@acm.org

Bernd Kolb, Klaus Birken
itemis AG
{kolb|birken}@itemis.de

Federico Tomassetti
independent
federico@tomassetti.me

Patrick Alff, Laurent Wiart
Voluntis
{Patrick.Alff|Laurent.Wiart}@voluntis.com

Andreas Wortmann
OHB System AG
andreas.wortmann@ohb.de

Arne Nordmann
Bosch Corporate Research
arne.nordmann@de.bosch.com

Language workbenches allow developers to create, integrate and efficiently use domain-specific languages, typically by generating programming language code from models expressed with domain-specific languages. This leads to increased productivity and higher quality. However, in safety-/mission-critical environments, such generated code may not be considered trustworthy, because of the lack of trust in how the code was obtained. This makes it harder to use language workbenches in such an environment. In this paper, we demonstrate an approach to use such tools in critical environments. We argue that models created with domain-specific languages are easier to validate, and that the additional risk resulting from the transformation to code can be mitigated by an suitably designed transformation and verification architecture. We validate the approach with an industrial case study from the medical domain. We also discuss the degree to which the approach is appropriate for critical software in space, automotive and robotics systems.