Formale Methoden in der Praxis
SMT Solver und Model Checker
Introduction
Robustness/Optimization
Development Speed
Security

Safety

Robustness/Optimization Development Speed
Process
- Education
- Requirements
- Reviews
- Pen Testing

Architecture
- Redundancy
- Fail-Safe
- Runtime Monitoring

Implementation
- Good Languages
- Guidelines
- Testing
- Formal Analysis
<table>
<thead>
<tr>
<th>Requirements Specification</th>
<th>Functional Design</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are the modeled behaviors correct?</td>
<td>Are all other behaviors prohibited?</td>
<td>Are there additional risks introduced by the impl?</td>
</tr>
<tr>
<td>Is the spec complete and consistent?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Actual Impl Code</td>
<td>Libs/Frameworks</td>
</tr>
<tr>
<td></td>
<td>Language/Compiler</td>
<td>Hardware</td>
</tr>
</tbody>
</table>
Verification
Ensure that the program performs correctly the things the program text tells it to do.

Validation
Ensure that the program does the correct things, wrt. to the requirements.
Execution

The program processes a particular set of inputs, covers one of potentially many paths, and produces particular outputs. “Real” data.

Analysis

Conceptually “execute” all possible executions for all valid input data, and understand all possible outputs.
Testing

Execute the program for a particular set of inputs, asserting over the observable outputs.

Formal Analysis

Specify properties that hold for all possible executions and then analyze whether they always hold.
Derivation

\[
\text{var } x = 2 + 3.33
\]

What is the type of \( x \)?

\[
\text{var } x = 0;
\quad \text{for (int } i=0; i<10; i++) x++; 
\]

What values can \( x \) have?

Checking

\[
\text{int } x = 2 + 3.33 
\]

Is this correct?

\[
\text{int[0..10] } x = 0;
\quad \text{for (int } i=0; i<10; i++) x++; 
\]

Will \( x \) stay in range?

Synthesis

\[
\text{int[0..10] } x = ??;
\quad \text{for (int } i=0;??; i++) x++; 
\]

Synthesize a program that respects all the specifications.
Synthesis ...

```kotlin
fun sortAsc(l: list<int>): list<int>
    post l.size == res.size
    post forall e in l | e in res
    post i, j: int[0..l.size] | j > i => res[j] >= res[i]
{
    ??
}
```

... vs. Code Gen

```kotlin
fun sortAsc(l: list<int>): list<int> {
    sort l, ASC;
}
```
Specification

Express an expectation of what the program should do, so the checker can verify that the implementation delivers.

- redundant wrt. to the implementation
- abstract over the “how”
- “simple”: easier to understand and less error prone than the implementation
- Separate from the code (“model”) or part of the code
Specification

```haskell
fun divBy2(x: int) post res = x * 2 {
  x * 2
}
```

```haskell
fun sortAsc(l: list<int>): list<int>
  post l.size == res.size
  post forall e in l { e in res }
  post forall i, j: int[0..l.size]
    { j > i => res[j] >= res[i] } 

  .. 20 lines of intricate quicksort ...
}
```

**Bug:** number of occurrences of each value in the result is not specified.
Hierarchical Specification

All analysis approaches have limits to scalability. Working around those requires modularizing the system.

For level $L_i$, you check that $L_i$’s implementation conforms to the contract specified by $L_i$, assuming that all contracts of $L_{i+1}$ hold.

(Requires spec to be compositional; not always true)
Spec-based

Describe program and spec, check one against the other.

Automated

Fully automated checking, internal consistency, spec is “implicit” in the lang construct.

Interactive

After each analysis result, the developer writes a bit more spec to “guide” the verification.
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.
Correct-by-Construction

Languages

Analysis-and-Fix

Analysis Tools
## Correct-by-Construction Languages

<table>
<thead>
<tr>
<th>multi-party-decision</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial parties:</td>
<td>bernd, klaus</td>
</tr>
<tr>
<td>dynamic?</td>
<td>✗</td>
</tr>
<tr>
<td>sealable?</td>
<td>☐</td>
</tr>
<tr>
<td>procedure:</td>
<td>unanimous</td>
</tr>
<tr>
<td>time limit:</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>turnout:</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>revokable?</td>
<td>✗</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>multi-party-decision</th>
<th>AccessControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial parties:</td>
<td>bernd, klaus</td>
</tr>
<tr>
<td>dynamic?</td>
<td>✗</td>
</tr>
<tr>
<td>sealable?</td>
<td>☐</td>
</tr>
<tr>
<td>procedure:</td>
<td>majority</td>
</tr>
<tr>
<td>time limit:</td>
<td>20000</td>
</tr>
<tr>
<td>turnout:</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>revokable?</td>
<td>☐</td>
</tr>
</tbody>
</table>
Languages
Analysis Tools
Program lines must **always** be reachable.

State machines can **always** be checked for dead states and unused transitions.

Decision tables must **always** be overlap-free and complete.
Model Checking for Code
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 system can be seen as a (complicated) state machine. Existing model checkers allow us to model systems as state machines.
 MODULE traffic_lights_controller(cross_request) {

 VAR {
     traffic : { Green, Yellow, Red }; 
     pedestrian : { Walk, DontWalk }; 
     timer : 0..10; 
 }

 ASSIGN {
     init(traffic) := Green; 
     next(traffic) := case 
         traffic = Green & cross_request : Yellow; 
         traffic = Yellow & timer = 0 : Red; 
         traffic = Red & timer = 0 : Green; 
         TRUE : traffic; 
     esac;
     init(timer) := 0; 
     next(timer) := case 
         traffic = Green & cross_request : 3; 
         traffic = Yellow & timer > 0 : timer - 1; 
         traffic = Yellow & timer = 0 : 10; 
         traffic = Red & timer > 0 : timer - 1; 
         TRUE : timer; 
     esac;
     init(pedestrian) := DontWalk; 
     next(pedestrian) := case 
         traffic = Red & timer > 0 : Walk; 
         TRUE : DontWalk; 
     esac;
  
}
Traffic Lights Controller

MODULE traffic_lights_controller(cross_request) {
    VAR {
        traffic : { Green, Yellow, Red };  
        pedestrian : { Walk, Don'tWalk };  
        timer : 0..10;
    }
}


STATE-MACHINE {
    green2yellow  
    cross_request / 
    timer = 3 ;

    yellow2yellow  
    timer > 0 / 
    timer = timer - 1 ;

    yellow2red  
    timer = 0 / 
    timer = 10 ; pedestrian = Walk ;

    red2green  
    timer = 0 / 
    pedestrian = Don'tWalk ;

    red2red  
    timer > 0 / 
    timer = timer - 1 ;

}
Every imperative program can be seen as a (very complicated) state machine.
What are the possible states of this program?

Cross-product of possible states of the memory managed by this code:

- variables – e.g. `a, b`
- program counter - `PC`

A state is given by the values of the tuple `(PC, var_i)`
char a;
char b;

void main() {
  label:
  a = 1;
  b = a + 1;
  goto label;
}
Cross-product of possible states of the memory managed by this code:

- variables (a)
- program counter (PC)
- stack
- ressources (file descriptors, libraries internal state, etc)
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. $\text{AG}(\text{state} == \text{A}) \Rightarrow (v > 10).$
Traffic Lights Controller Properties

```haskell
MODULE traffic_lights_controller(cross_request) {
  VAR {
    traffic : { Green, Yellow, Red };  
    pedestrian : { Walk, Don'tWalk }; 
    timer : 0..10; 
  }
}
```

Never allow both pedestrians and cars to cross at the same time

```haskell
SPEC AG !(pedestrian = Walk & traffic = Green);
```

After the request button is pushed, pedestrians are eventually allowed to cross

```haskell
SPEC AG (cross_request -> AF pedestrian = Walk);
```
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.
Bounded Model Checking

Explore the traces up to a given depth $N$
- i.e. perform all possible first $N$ steps

$N = 2$
Programs can have very big (or infinite) loops. Bounded model checkers unroll loops and recursion up to a given depth.
Verification as Reachability Problem

Many properties can be expressed as reachability

```
... assert (n != 0)
...
```

```
... if ( n== 0 ) {
    ERROR_LABEL_ASSERT: 0;
}
...
```

```
... speed = dist / time;
...
```

```
... if ( time == 0 ) {
    ERROR_LABEL_DIV_0: 0;
}
speed = dist / time;
...
```

Software verification tools automatically instrument the code for checking robustness properties (overflows, div-by-zero, NAN, ...)
Ingredients

System under verification
software component (e.g. one or more functions)

Environment
Other inputs and program parts
NOT belonging to the system under verification

Properties
are what we check for (e.g. overflow, assertions)

Verification Result
can be pass/fail or timeout

Witness
is a trace through the system illustrating the failure scenario
Model Checking Tools

Verification tools are not magic but rather advanced engineering

Numerous powerful model-checkers exist
How to Use the Model-Checkers?

... we consider verification tools as black-boxes

... and use DSLs for defining the environment, properties or software components
Software Verification Competition

https://sv-comp.sosy-lab.org/2018/

15+ open source C-level verification tools
Compete in 6 categories of benchmarks
Verification Cases
Verification of „Simple Components“

Environment: $\text{ENV}^{\text{(SWC)}}$

System Under Verification (e.g. one or more functions)

Inputs:
- parameters,
- global variables,

Outputs:
- returned values,
- global variables

Does property $X$ hold?
From Testing to Verification

How to choose test vectors?

- Boundary values
- Partition the input space
- Invalid values
- ...

```c
void heap_sort(int32 [] arr, uint32 N) {
    if (N == 0) {
        return;
    }
    if

    int32 t;
    uint32 n = N;
    uint32 parent = N / 2;
    uint32 index;
    uint32 child;

    while (true) {
        if (parent > 0) {
            t = arr[--parent];
        }
```
Choosing values for \( N \):
-1
0, 1, MAX_LEN
2, 5, 10, 100

Choosing values for \( arr \):
{}
{42}
{11, 12, 10}
{12, 12, 10, 5}
{2, 11, 10, 4, 6, 32, 31, 11, 12}
{12, 1, 10, 14, 36, 31, 71, 12, ...}
How Much Testing is Enough?

How many relevant cases have we covered?
Where to stop?

How about checking ALL possible inputs up to a given size?
Intuition: Test vs. Verification Cases

Test Cases: Points in Input Space

Verification Cases: Regions in Input Space
Verification Cases

void heap_sort(int32[] arr, uint32 N) {
    if (N == 0) {
        return;
    } if
    int32 t;
    uint32 n = N;
    uint32 parent = N / 2;
    uint32 index;
    uint32 child;
    while (true) {
        if (parent > 0) {
            t = arr[--parent];
        }
        while (true) {
            if (parent > 0) {
                t = arr[--parent];
            }
            parent = n / 2;
            index = parent;
            child = index * 2 + 1;
            if (child >= n) {
                break;
            }
            if (child + 1 < n) {
                if (arr[child] < arr[child + 1]) {
                    t = arr[child];
                    arr[child] = arr[child + 1];
                    arr[child + 1] = t;
                    child = child + 1;
                } else {
                    break;
                }
            } else {
                break;
            }
        }
    }
}

void heapsort_verificationCase() {
    int32 [MAX_ARR_LENGTH] my_arr = {0};
    uint8 my_N;

    harness {
        nondet assign my_N; constraints {
            my_N in [0..MAX_ARR_LENGTH]
        }
        for (i ++ in [0..my_N]) {
            nondet assign my_arr[i]; constraints {
                my_arr[i] in [-MAX_VALUE..MAX_VALUE]
            }
        }
    } for
    heap_sort(my_arr, my_N);
    for (i ++ in [1..my_N - 1]) {
        assert(my_arr[i - 1] <= my_arr[i]);
    }
}
Much Higher Coverage of Input Space

Usually \( \text{ENV}^{\text{BMC}} \ll \text{ENV}^{\text{SWC}} \)

But still, a **HUGE** input space can be covered:

- \( \text{MAX\_ARR\_LENGTH} = 7 \)
- \( \text{MAX\_VALUE} = 50 \)

10\(^{14}\) test vectors

Exhaustive testing – completely infeasible!

CBMC needs < 30 seconds to complete

```c
void heapsort_verificationCase() {
    int32 [MAX_ARR_LENGTH] my_arr = {0};
    uint8 my_N;

    harness {
        nondet assign my_N; constraints {
            my_N in [0..MAX_ARR_LENGTH[
        }

        for (i ++ in [0..my_N]) {
            nondet assign my_arr[i]; constraints {
                my_arr[i] in [-MAX_VALUE..MAX_VALUE]
            }
        } for

        heap_sort(my_arr, my_N);

        for (i ++ in [1..my_N - 1]) {
            assert(my_arr[i - 1] <= my_arr[i]);
        } for
    }
    } heapsort_verificationCase (function)
```
Verification of CRCs

CRC is an error detection code used to detect random errors on a noisy channel

Properties of CRC algorithms:
• **Detection of error bursts**: contiguous errors up to a certain length
• **Hamming distance**: the maximum number of bit changes which can be detected

![Diagram of sender and receiver with payload and CRC]

Payload + CRC
Testing of CRC Algorithms?

Input space is HUGE!

- e.g. for 20 bytes payload and detection of error bursts with len. 16
- $2^{160}$ possible payload messages
- $2^{16}$ possible errors starting from position X
- 154 possible locations for starting the error burst

$2^{160} \times 2^{16} \times 154 \approx 2^{183}$ possible inputs to be tested

Very low input coverage can be achieved by classical testing
Easy to reach a high structural (e.g. line, condition) code coverage
Verification Case

Nondeterministically initialize the input

Compute the initial CRC (on the sender side)

Nondeterministically choose the starting point for the error burst

Nondeterministically choose the erroneous bits

Compute the new CRC (on the receiver side)

Check that the CRCs on the sender and receiver sides are different
**Verification Case Example**

```c
for (i += in [0..20]) {
    nondet assign payload[i];
} for

uint16 initialCRC = compute_crc(((uint8*) payload), length);

nondet assign errorPosition; constraints {
    errorPosition < 154
}

for (idx += in [errorPosition..errorPosition + 16]) {
    uint8 erroneousByteIndex = idx / 8;
    uint8 erroneousBitIndex = idx % 8;
    nondeterministic_choice: {
        choice (0): {
            payload[erroneousByteIndex] ^= 1 << (7 - erroneousBitIndex);
            errorInjected = true;
        }
        choice (1): {
            // do not change this bit
        }
    }
} for

uint16 modifiedCRC = compute_crc(((uint8*) payload), length);

assert(errorInjected -> (initialCRC != modifiedCRC));
```

- Initialize the payload
- Compute CRC at sender
- Inject an error burst
- Compute CRC at receiver
- Check that CRCs are different
What Properties are Checked?

Robustness
- div-by-zero, NaNs, overflows,
- buffer overflow, concurrency

Unreachable code

Correct use of APIs
- memory leaks

Users-defined Assertions
Users-defined Assertions

Temporal Properties

before buttonPressed must outputLight == RED;

can be expressed in C as ...

```c
{
    static bool ___p = false;

    ___p = ___p || buttonPressed;
    if ( !___p )
    {
        assert(outputLight == RED);
    }
}
```
Specify contracts at C-level

- Requires
- Ensures
- Behaviors

```c
/*@ requires: array_size > 0;
@ behavior: empty_array {
@   assumes: array_size == 0
@   ensures: \result == -1;
@ }
behavior empty_array
@ ensures: \result >= -1;
@*/

int16 search(int16* array, int16 array_size, int16 elem) {
```
Model Checking with DSLs
Spec-based

Program ↔ Spec

Describe program and spec, check one against the other.

Automated

Program ↔ Spec

Fully automated checking, internal consistency, spec is “implicit” in the language construct.
Express properties as interface contract. Generate C, invoke CBMC to perform model check, lift results.
Heartbleed

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;
}
```

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

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

// [ Here a problem happens, because we read more than we allocate ]
memcpy(dest, p, length);
```
Verification: Heartbleed Bug IV

Even better: First-class message concept that handles low level stuff, such as buffer sizes

```c
// a field representing a timestamp
uint8[6] f_time = {0x00A, // f_time
                   UNIT_TIME24,
                   3, // 3 payload bytes
                   10, 20, 00 /}

// a field representing a measured value
uint8[4] f_value = {0x04D, // field type identifier
                    UNIT_QDOT, // unit used: mass flow
                    1, // 1 payload byte follows
                    &dataField // addr of variable }

// a message that uses the two fields
uint8[5] message = {0xAEE, // message type identifier
                    ID, // unique running message ID
                    2, // two fields following
                    f_time, // embed the time field
                    f_value// embed the value field }
```

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:

```
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:

```
trysequentially {
    validateStep1(data, ...);
    validateStep2(data, ...);
    validateStep3(data, ...);
} on fail (errorcode) {
    handleFailedValidation(data, errorcode, ...);
}
```
Report from Dagstuhl Seminar 14062

The Pacemaker Challenge: Developing Certifiable Medical Devices

Edited by
Dominique Méry¹, Bernhard Schätz², and Alan Wassnyng

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, “Pacemaker Challenge: Developing Certifiable Medical Devices”. The aim of the seminar was to bring together leading researchers and industrial partners of this field; the seminary experienced participants from 8 countries: Canada, Denmark, France, The United States, Germany, UK, and Brazil. Through a series of presentations, discussions, and working groups, the seminar attempted to get a general view of the field of medical devices and certification through the pacemaker challenge. The seminar brought together, on the one hand, researchers from 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
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 Wait: {
    switch (e) {
      case t: {
        if (c < LRI) { ++c;
        return; }
        if (c == LRI) { state = Pace;
        doPace();
        c = 0;
        return;
      }
    }
  }
  case s: {
    if (c <= VRP) {
        state = Wait;
      }  
    }  
```
Verification: Pacemaker Challenge III

Safety Properties (and their tx to C)

```plaintext
after smIsInState(Wait)
until smIsInState(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

nondet_choice {
    choice {
        smtrigger(s);
        assume(0 <= ch
            && ch < 2);
    } 
    choice {
        // nothing
    } 
    if (ch == 0) {
        execute(s, 0);
    } 
    if (ch == 1) {
        // nothing
    }
}

int ch = nondet_int();

B

smStateSubset Initial:
smInState(Pace)
&& VRP < LRI;
state = (VVI_States)
    nondet_int();
assume(0<= state
    && state <= 2);
smNonDetInit(Initial);
c = nondet_int();
LRI = nondet_int();
VRP = nondet_int();
assume(state == Pace
    && VRP < LRI);
```
Verification: Pacemaker Challenge IV

Inductive Proof

\[
\text{induction on } t \\
\text{from } \text{Initial} \\
\text{until } \text{MAX\_LRI} \{ \\
\quad \text{nondeterministically smtrigger}(s); \\
\quad \text{after } \text{smIsInState}(\text{Wait}) \\
\quad \text{until } \text{smIsInState}(\text{Pace}) \\
\quad \text{exists } c == \text{LRI} \}
\]

\[
\text{smNonDetInit}((\text{Initial}); \\
\text{for } (\text{int } \text{step} = 0; \text{step} < \text{MAX\_LRI}; ++\text{step}) \{ \\
\quad \text{nondeterministically smtrigger}(s); \\
\quad \text{after } \text{smIsInState}(\text{Wait}) \\
\quad \text{until } \text{smIsInState}(\text{Pace}) \\
\quad \quad \text{exists } c == \text{LRI}; \\
\quad \text{smtrigger}(t); \\
\quad \text{after } \text{step} > 1 \text{ until } \text{step} == \text{MAX\_LRI} \\
\quad \quad \text{exists } \text{smIsInStateSubset}(\text{Initial}); \}
\]
Verification: Read the Paper

Automated Domain-Specific C Verification with mbeddr

Zaur Molotnikov
Fortiss Institute
Guerickestraße 25
Munich, Germany
molotnikov@fortiss.org

Markus Völter
independent/itemis
Oetztaler Straße 38
Stuttgart, Germany
voelter@acm.org

Daniel Ratiu
Fortiss Institute
Guerickestraße 25
Munich, Germany
ratiu@fortiss.org

ABSTRACT

When verifying C code, two major problems must be addressed. One is the specification of the verified systems properties, the other one is the construction of the verification environment. Neither C itself, nor existing C verification tools, offer the means to efficiently specify application domain-level properties and environments for verification. These two shortcomings hamper the usability of C verification, and limit its adoption in practice. In this paper we introduce an approach that addresses both problems and results in user-friendly and practically usable C verification. The novelty of the approach is the combination of domain-specific language engineering and C verification. We apply the approach in the domain of state-based software, using mbeddr and CBMC. We validate the implementation with an example from the Pacemaker Challenge, developing a functionally verified, lightweight, and deployable cardiac pulse generator. The approach itself is domain-independent.

Requirements such as this one describe the properties the system should have. To make them verifiable by tools, they have to be expressed as formal verification conditions. A (not necessarily correct) implementation of this pacing logic is given in Listing 1. This code could be verified using C verification tools such as CBMC [10], SATABS [11] or CPAchecker [7]. Working at the abstraction level of C, they can be used to, for example, check assertions or error-label reachability. This makes it impractical to directly represent the application domain-level semantics implied by requirements such as those described above [13, 30].

int t = 0;
bool makePace(Event e) {
    switch(e) {
        case Tick:
            if (t < LRI) {
                ++t;
                return false;
            } else {
                t = 0;
                return true;
            }
    }
}
SMT Solving
What Solvers Do

For a set of equations $e_i \subset E$ 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: \[ 2 \times x = 3 \times y \]
\[ x + 2 = 5 \]

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:

for $i, j \in \{1, \ldots, n\}$,

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

Completeness:

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

or: with $n$ row headers $r_1, \ldots, r_n$, \[
\bigwedge_{i=0}^{n} \neg r_i
\]
Typical Solver Tasks

Any

**Applicability:** Is there a value assignment satisfying all $E_i$? Examples are any set of boolean expressions, or even a single complex one.

**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.

**Overlap:** For any combination of inputs, does at most one expression $E_i$ match? Examples include any set of Boolean expressions that are not ordered, so no two can match any given set of inputs. Often used together with completeness, hence the same examples apply.

Refactorings

**Equality:** Are the $E_i$ semantically equivalent, even though they differ structurally (think: DeMorgan laws). Examples include refactorings that simplify expressions.

**Subset:** For any $i \in \{1, ..., n\}$, 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.

Solvers can also be used to verify (large subsets of) programs wrt. arbitrary properties. But that’s outside our scope.
SMT Solvers can help find/fix non-trivial logic errors in programs.

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

\[
\text{lt} \begin{cases} 
  x < 0 \land x > 0 & \Rightarrow 1 \\
  x = 5 & \Rightarrow 2 
\end{cases}
\]

**Error:** [MANUALLY CHECKED] Overlapping alternatives. The following case is covered by multiple alternatives: 
\( x = 0 \)

\[
\text{alt} \begin{cases} 
  x \geq 0 & \Rightarrow 1 \\
  x \leq 0 & \Rightarrow 2 
\end{cases}
\]

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

\[
\text{alt} \begin{cases} 
  x > 0 & \Rightarrow 1 \\
  x < 0 & \Rightarrow 2 
\end{cases}
\]
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
A solver-integrated functional language to be used at the core of DSLs based on MPS.
Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.

The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, dynamic allocation, and inductive datatypes, and builds in specification constructs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics.

To further support specifications, the language also offers updatable ghost variables, recursive functions, and types like sets and sequences. Specifications and ghost constructs are used only during verification; the compiler omits them from the executable code.

The Dafny verifier is run as part of the compiler. As such, a programmer interacts with it much in the same way as with the static type checker—when the tool produces errors, the programmer responds by changing the program’s type declarations, specifications, and statements.
Wrap Up
A Word of Warning

Verification tools can have bugs themselves
Missalignements with the compiler
Missalignements with the target computer

Bad properties specifications
Wrong model of the system

Confirm that the verification tool properly
„understands“ your code
Complement formal verification with testing !
Other formal Methods

Data Flow Analysis + Abstract Interpretation
Property-based Testing
Advanced Type Systems
Proof Assistants (Coq, Isabelle)
Discrete Event Simulation ...

Very lively research community, lots of progress, increased (not enough) focus on real-world tools.
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
Verification Tools can play a role in practical, real-world projects and systems.

SMT Solving and Model Checking are the most approachable two formal methods.

You have to allocate time to learn, to be able to use the tools productively.

Use it for your most critical parts of app, where you will get the biggest bang for your buck.

Defining your own languages on top helps a lot with managing the tool complexity.

You still want to use testing, the formal approaches are “just another tool“ in your kit.