Models, Modeling and Tools in Embedded Software Development
1 Motivation
2 What is a Model?
3 Characteristics of Models
4 Example Models Languages
5 Performance and Resources
6 Reliability
7 Language Workbenches
8 JetBrains MPS
9 mbeddr
10 Recap/Big Picture
Motivation
Complexity is killing us.
So?
(Software) Engineering to the Rescue!
Modularization

Procedures, Classes, Components, Services, User Stories
Encapsulatation

Private Members
Frameworks
Facade Pattern
Components
Layers/Rings/Levels
Packed Data Wrapper
Contracts

Interfaces
Pre/Post Conditions
Protocol State Machines
Message Exchange Patterns
Published APIs
Cross-Cutting Concerns

Aspect Orientation
Interceptors
Application Servers
Exception Handling
Parametrization

Function Arguments
Command-Line Args
Configuration Files
Types & Instances

Programming Languages
Components

Models & Metamodels
RDBMS/XML Schemas
Hierarchical Decomposition

Procedures/Methods
State Machines, Components
Specialization

Currying
Inheritance
State Machines
Viewpoints

Configuration Files
4+1 Model
Blackbox/Whitebox
Types/Instances/Deployment
Declaration

Implementation

App Servers (EJB), Plugin RT (Eclipse)
Models, Transactional Memory
<Reality Check>
typedef struct btrec
{
    ULONG funcnum;  /* Encode function number 0->none */
    ULONG key;      /* Key for function (N/A if no func) - max key value=64k-1 */
    ULONG length;   /* Length of following data */
    ULONG start;    /* start position of the encrypted data */
    ULONG cntrlblock[20]; /* control block */
    UCHAR data[500];   /* Data */
} BTREC;

typedef BTREC FAR * PBTREC;

#define ENCODE1(d,k) (UCHAR)(d ^ (UCHAR)k)
#define ENCODE2(d,k) (UCHAR)((UCHAR)(~d) ^ (UCHAR)k)
#define ENCODE3(d,k) (UCHAR)(d ^ (UCHAR)~k)
#define DECODE1(d,k) (UCHAR)(d ^ (UCHAR)k)
#define DECODE2(d,k) (UCHAR)((UCHAR)(d ^ (UCHAR)k))
#define DECODE3(d,k) (UCHAR)(d ^ (UCHAR)~k)

/* Name: NewKey */
/* Description: Returns a key to use in the translation */

USHORT NewKey(VOID)
{
    USHORT key;       /* Key */

    /* Work out the key from a random number */
    key=(USHORT)rand();
    /* return random number */
    return(key);
}

/* Name: NewFunc */
/* Description: Returns a function number to use in the translation */
ULONG NewFunc(VOID)
{
    ULONG func;       /* Function number */
</Reality Check>
THE QUESTION
How can we deal more effectively with...

Modules
Contracts
Types
Hier. Decomposition
Specialization
Viewpoints
Declarativeness
Cross-cutting Concerns
How can we also deal effectively with...

- Safety
- Security
- Performance
- Timing
- Reliability
- Maintainability

...<ComeUpWithYourOwn>-ility
THE ANSWER
Your own? Suitable Abstractions Suitable Notations (Good Tools)
Models
Modeling Languages
Modeling Tools

Your own?
What is a Model?
Trackpoint* makeTP(uint16 alt, int16 speed) {
    static int8 trackpointCounter = 0;
    trackpointCounter++;
    Trackpoint* tp = ((Trackpoint*) malloc(sizeof Trackpoint));
    tp->id = trackpointCounter;
    tp->timestamp = trackpointCounter;
    tp->alt = alt
    tp->speed = speed
    return tp;
}
beforeFlight

reset

next [alt > 0]

airborne

reset

next [alt == 0 && speed == 0]

flying

next [alt == 0 && speed > 0]

landing

next [alt == 0 && speed >= 0]

landed
```java
statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
   in next()
   in reset()
   out crashNotification() -> raiseAlarm
   state beforeFlight {
      on next [tp->alt > 0 m] -> airborne
   }
   composite state airborne initial = flying {
      on reset [ ] -> beforeFlight
      on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
      state flying {
         on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
         on next [tp->speed > 200 mps] -> airborne
         on next [tp->speed > 100 mps] -> airborne
      }
      state landing {
         on next [tp->speed == 0 mps] -> landed
         on next [ ] -> landing
      }
      state landed {
      }
   }
   state crashed {
   }
}
```
```c
statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
    in next(Trackpoint* tp)
    in reset()
    out crashNotification() -> raiseAlarm
    readable var int16 points = 0
    state beforeFlight {
        on next [tp->alt > 0 m] -> airborne
        exit { points += TAKEOFF; }
    }
    composite state airborne initial = flying {
        on reset { } -> beforeFlight { points = 0; }
        on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
        state flying {
            on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
            on next [tp->speed > 200 mps] -> airborne { points += VERY_HIGH_SPEED; }
            on next [tp->speed > 100 mps] -> airborne { points += HIGH_SPEED; }
        }
        state landing {
            on next [tp->speed == 0 mps] -> landed
            on next { } -> landing { points--; }
        }
        state landed {
            entry { points += LANDING; }
        }
    }
    state crashed {
        entry { send crashNotification(); }
    }
}
```
Does it really matter?
What is the difference?
Who cares?
[Notation]
One important aspect of languages.

Textual

Graphical

Tabular

Mathematical
[Abstraction]

The other important aspect.

- Application Domain
- Solution Domain
- Application Domain
- Solution Domain
- Application Domain
- Solution Domain
- Application Domain
- Solution Domain
- Application Domain
- Solution Domain

- Contracts
- State Machines
- Java Code
- Bytecode

Model?

Code?

???
The other important aspect.

- Contracts
- State Machines
- Java Code
- Bytecode

Artifacts, Models, Programs, Prodel
[Relationships]

representation of

model of

part of

view of
model of Abstraction
part of Hierarchy, Partitioning
view of Concern, Aspect, Viewpoint
representation of Presentation, Notation, Form
Characteristics of Models
Dimensions of Models

Abstractions
Notations
Purpose
Standardization
Tool Support

Aligned with Domain
Driven By Model Purpose
SE Best Practices
Meaningful to Users
Well-defined Semantics

One Size does NOT fit all.  
-> Multi-Paradigm!
Dimensions of Models

Abstractions

Notations

Purpose

Standardization

Tool Support

Text, Tables, Math, ...

Inspired by Existing N

Scalable!

Integrate Multiple

Switchable for same Abstr.

More than Diagrams
Dimensions of Models

Abstractions
Notations
Purpose
Standardization
Tool Support

Create-and-Think
Communicate
Stakeholder-Programming
Analysis
Interpretation/Generation
Effort
Platform
Complexity
Dimensions of Models

Abstractions
Notations
Purpose
Standardization
Tool Support

- Avoid Reinventing Wheels
- Enables Data Exchange
- Makes Hiring Simpler
- Limited Domain Match
- Potentially Large
- Standardize on Meta Level
- Adopt a Language
## Dimensions of Models

<table>
<thead>
<tr>
<th>Abstractions</th>
<th>Tool Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notations</td>
<td>Absolutely Crucial!</td>
</tr>
<tr>
<td>Purpose</td>
<td>As good as IDE!</td>
</tr>
<tr>
<td>Standardization</td>
<td>Standards help, but ...</td>
</tr>
<tr>
<td></td>
<td><strong>Language Workbenches</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Integrate L, not T</strong></td>
</tr>
</tbody>
</table>
Example Models
Languages
Requirements

1. Initially you have no points.
   InitialNoPoints /functional: tags
   When the game starts, you have no points.
   workpackage initial scope: 1 responsible: peter prio: 1 effort: 1 days

2. Once a flight lifts off, you get 100 points
   PointsForTakeoff /functional: tags
   Fusce vitae quam est, at facilisis
   requires also ExampleWithDependencies
   constant int8 POINTSFORTAKEOFF = 100
   workpackage impl1 scope: 1 responsible: peter prio: 1 effort: 10 days
   workpackage impl2 scope: 2 responsible: peter prio: 1 effort: 5 days

@node[ExternalFunctions]
1.2 | **Flight is Interpolated and Evaluated**

**FlightIsInterpolated** use case: tags


```java
use case FlightIsInterpolated {
    active participant UI
    passive participant DataStore
    passive participant Interpolator
    passive participant Judge
}
```

1.2.1 | **Describes the Interpolation**

Interpolation /scenario: tags

```java
scenario Interpolation
    UI {
        -> DataStore.getAFlight(): new Flight f
        -> DataStore.getAFlight() {
            return new Flight f2
        } DataStore.getAFlight
        -> Interpolator.process(received f2): ok
        loop over all the trackpoints in f {
        -> Judge.judge(new Trackpoint t)
        } loop
    }
```

Formal/Structured specs can be embedded and evaluated.
Implementation Models or Code can be trace to requirements to aid validation.

```
#define TAKEOFF 100; // implements PointsForTakeOff
#define HIGH_SPEED 10; // implements FasterThan100
#define VERY_HIGH_SPEED 20; // implements FasterThan200
#define LANDING 100; // implements FullStop

[checked]
exported statemachine FlightAnalyzer initial = beforeFlight {

    next(Trackpoint* tp) {
        Events
        reset()  

        beforeFlight
        { [tp->alt > 0 m] -> 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; }

        }

        airborne
        { [tp->speed > 0 mps] -> landing
          { points--; }

        }

        landed
        {------------------

        crashed
        { ------------------

        NextRound<j>

        You should land as short as possible

        [ ] -> beforeFlight

        }  

```
Behavior: State Machines

- State Machines

Useful for discrete behavior
By default, graphical notation
Potentially w/ embedded actions

Diagram of a state machine for a Bank ATM:
- States: Off, Self Test, Idle, Maintenance, Out of Service, Serving Customer
- Transitions:
  - Off to Self Test
  - Self Test to Maintenance
  - Maintenance to Out of Service
  - Idle to Service
  - Out of Service to Service
- Events:
  - turn on / startup
  - turn off / shutDown
  - cardInserted
  - cancel
  - entry / readCard
  - exit / ejectCard
  - Service: Customer Authentication, Selecting Transaction, Transaction
State Machines may also be represented as tables.
Behaviour: State Machines

```plaintext
statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
  in next()
  in reset()
  out crashNotification() -> raiseAlarm
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
  }
  composite state airborne initial = flying {
    on reset [ ] -> beforeFlight
    on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
    state flying {
      on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
      on next [tp->speed > 200 mps] -> airborne
      on next [tp->speed > 100 mps] -> airborne
    }
  state landing {
    on next [tp->speed == 0 mps] -> landed
    on next [ ] -> landing
  }
  state landed {
  }
  state crashed {
  }
}
```

... or as text. After all, that's just a representation! The model does not change!
Behaviour: State Machines

```c

statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
  in next(Trackpoint* tp)
  in reset()
  out crashNotification() -> raiseAlarm
  readable var int16 points = 0
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
  }

  composite state airborne initial = flying {
    on reset [ ] -> beforeFlight { points = 0; }
    on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
    state flying {
      on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
      on next [tp->speed > 200 mps] -> airborne { points += VERY_HIGH_SPEED; }
      on next [tp->speed > 100 mps] -> airborne { points += HIGH_SPEED; }
    }
    state landing {
      on next [tp->speed == 0 mps] -> landed
      on next [ ] -> landing { points--; }
    }
    state landed {
      entry { points += LANDING; }
    }
  }
  state crashed {
    entry { send crashNotification(); }
  }
}
```

The amount of detail may vary – depending on the purpose (analysis vs. generation)
Verification: State Machines

Temporal Logic provides a powerful formalism for verifying properties of state machines.
Useful for implementing continuous behavior (controls engineering). **NOT USEFUL** for graphical, low-level programming.

**Behavior: Dataflow Models**

- POSV1
- POSV2
- POSV3
- POSV4
- POSV5

**Diagram:**

- Minimum
- NOT
- Constant
- Fault
Behavior: Dataflow Models

Dataflow Models have a scalability problem. As behavior gets more complex, the models become big. Tools often don’t support modularization well.
Behavior: Mathematical

vector<int16, 3> aVector = [1, 2, 3] * 512;

vector<int16, 3> resultOfCrossProduct = aVector x aVector;

matrix<int16, 2x3> aMatrix = [1 + 2, 2 * 7, 42; 3, 51, 24];

matrix<int16, 3x2> transposedMatrix = aMatrix^T;

int32 averageIntArray(int32[][] arr, int32 size) {
    return \frac{\sum_{i = 0}^{\text{size}} \text{arr}[i]}{\text{size}};
}

Mathematical types, operators and notations can be used effectively to model behavior in some domains.
Enriched Type Systems and Contracts

```plaintext
section PowerUnits {
exported unit W := kg\cdot m^2\cdot s^{-2}\cdot s^{-1} for Active power
exported unit kW := <no spec> for Active power
exported unit Var := W for Reactive power
exported unit kVar := kW for Reactive power
exported unit VA := W for Apparent power
exported unit kVA := kW for Apparent power
exported conversion W -> kW {
  val as <no type> -> val / 1000
}
exported conversion kW -> W {
  val as <no type> -> val * 1000
}
} section PowerUnits

exported cs interface IRTCAlarm {
  void setAlarm(uint8 day, uint8 hour,
    pre(0) (day >= 1 && day <= 31) || day == 0,
    pre(1) hour <= 23 || hour == RTC_NO_HOUR_ALARM,
    pre(2) minute <= 59 || minute == RTC_NO_MINUTE_ALARM
  void setWeeklyAlarm(uint8 dayOfWeek, uint8 hour, uint8 minute, boolean repeat)
    pre(0) dayOfWeek <= 6
    pre(1) hour <= 23 || hour == RTC_NO_HOUR_ALARM
    pre(2) minute <= 59 || minute == RTC_NO_MINUTE_ALARM
  void clearAlarm()
}

int16/m/ dAlt = cur->alt - prev->alt;
int8/s/ dTime = cur->time - prev->time;

Error: type int16 /s m^(-1)/ is not a subtype of int16 /mps/
cur->vSpeed = dTime / dAlt;

The more semantics you put into models or code, the easier it is to implement meaningful checks.
```
The more semantics you put into models or code, the easier it is to implement meaningful checks.
Simple Better Abstractions

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

Even small, local embedded DSLs can solve real problems. Anyone remember goto FAIL?
Modularization and Interfaces

```c
exported component EraseFlashSegmentCommandHandlerImpl extends nothing {
  provides ICalibrationCommandHandler commandHandler
  requires IFlashMemoryController flashMemoryController

  uint8 commandHandler_getId() => op commandHandler.getId {
    return CALIBRATION_COMMAND_ID_ERASE_FLASH_SEGMENT;
  } runnable commandHandler_getId

  CalibrationCommandStatus commandHandler_commandInvoked(Datagram* pRequestFrame, Datagram* pResponseFrame) => op commandHandler.commandInvoked {
    uint16* pRequestData = ((uint16*)pRequestFrame->pVisiblePayload);

    // Read flash memory destination address from request message
    uint16* pFlashLocation = ((uint16*)*pRequestData);

    // Erase enclosing flash memory segment
    flashMemoryController.eraseSegment(pFlashLocation);

    // Set actual response data length
    pResponseFrame->visiblePayloadLength = 0;
    return CALIBRATION_COMMAND_STATUS_OK;
  } runnable commandHandler_commandInvoked
} component EraseFlashSegmentCommandHandlerImpl
```

Components, Ports and Interfaces are a well-established means of breaking down large systems.
Modularization and Interfaces

```java
exported composite component MetrologyRawSignalSimulatorTestHarnessImpl {
  provides IMetrologyRawSignalSimulationRunner metrologySimRunner

  internal instances {
    runner: SignalsSimRunnerImpl
    sim: RawSignalSimulatorImpl
    IGraphPlotter
    graphPlotterInst: GraphPlotterImpl
    IMetrologyRawSignalData
    IMetrologyRawSignalHandler
  }
}
```

Components, Ports and Interfaces are a well-established means of breaking down large systems.
Components, Ports and Interfaces are a well-established means of breaking down large systems.
Non-Functional: Timing

Timing and Scheduling Models are one of the ingredients for deployment onto hardware.
Non-Functional: Timing

OS Configuration and mapping to code made explicit – makes correct deployment simpler.

task Measurement
  prio = 1  wcet = 10 ms

memory layout {
  region ram:  0 .. 1024
  region eprom: startOf(ram) .. 2048
  region devices: endOf(eprom) .. startOf(devices) + sizeof(ram) * 2
}

task Kombi
  prio = 2  wcet = 100 ms

critical(data) {
  s = data.speed;
  r = data.rpm;
  showInUI(s, r);
}

critical(data) {
  data.rpm = measureRPM();
  data.speed = measureSpeed();
}

task Kombi {
  uint16 s = 0;
  uint16 r = 0;
  critical(data) {
    s = data.speed;
    r = data.rpm;
    showInUI(s, r);
  }
}
Combinations

Embedding Procedural C Code or Math in Components, Dataflow Blocks, State Machine Actions...

```
struct blockblock: Integrator
  exported atomic block Integrator
  double in
  double curTime
  optional boolean reset
  optional double resetIV
  parameters
    double initialValue;
    double minValue;
    double maxValue;
  state
    double integrated = initialValue;
    double prevTime = -1;
  on init
    integrated = initialValue;
    prevTime = 0;
  }
code {
  res = ((integrated < minValue)?(minValue):((integrated > maxValue)?(maxValue):(integrated));
  update {
    if (isConnected(reset)) {
      if (reset || prevTime < -0.5) {
        if (isConnected(resetIV)) {
          integrated = resetIV;
          } else { integrated = initialValue; }
      } else if (prevTime < -0.5) {
        integrated = initialValue;
      } else if (prevTime >= -0.5) {
        integrated += in * (curTime - prevTime);
      } if
      prevTime = curTime;
    };
```
Combinations

Embedding Procedural C Code or Math in Components, Dataflow Blocks, State Machine Actions...

Embedding a State Machine in a Dataflow Block

Using State Machines to Control Dataflow Models.

Using Expressions in Contracts (DF, SM, Comp)

All kinds of DSLs in other languages (DF, SM, Comp)
Performance and Resources
Generated Code is slow, ugly and bloated. Or is it?
SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms

Markus Püschel*      Bryan Singer      Jianxin Xiong      José M. F. Moura
Jeremy Johnson       David Padua       Manuela Veloso     Robert W. Johnson

Abstract

SPIRAL is a generator of libraries of fast software implementations of linear signal processing transforms. These libraries are adapted to the computing platform and can be re-optimized as the hardware is updated. SPIRAL uses a mathematical framework to generate the formula that results in a particular signal processing routine. It takes advantage of freedom when translating the computations to software, making possible the use of an intelligent search engine that finds within the large space of alternative formulas and implementations the “best” match to the given computing platform. We present empirical data that demonstrates the high performance of SPIRAL generated code.
Software Synthesis and Code Generation for Signal Processing Systems

Shuvra S. Bhattacharyya, Member, IEEE, Rainer Leupers, and Peter Marwedel, Member, IEEE

Abstract—The role of software is becoming increasingly important in the implementation of digital signal processing (DSP) applications. As this trend intensifies, and the complexity of applications escalates, we are seeing an increased need for automated tools to aid in the development of DSP software. This paper reviews the state-of-the-art in programming language and compiler technology for DSP software implementation. In particular, we review techniques for high-level block-diagram-based modeling of DSP applications; the translation of block-diagram specifications into efficient C programs using global target-independent optimization techniques; and the compilation of C programs into streamlined machine code for programmable DSP processors using architecture-specific and retargetable back-end optimizations. We also point out important directions for further investigation.

Index Terms—... into efficient C programs using global target-dependent optimizations ...
Using C Language Extensions for Developing Embedded Software: A Case Study

Markus Voelter
independent/itemis, Germany
voelter@acm.org

Arie van Deursen
Delft University of Technology, The Netherlands
Arie.vanDeursen@tudelft.nl

Bernd Kolb, Stephan Eberle
itemis AG, Germany
{kolb|eberle}@itemis.de

Abstract
We report on an industrial case study of using C language extensions for a smart embedded software for a smart home automation system. The using language and domain-specific components, physical units, state variables, and interrupts. We find that the extent in which the embedded software can be managed the complexity of the software, as well as the effort for its development.

Audience We target language engineering researchers (interested in empirical data justifying their work or looking to understand problems they may be able to solve) as well as embedded systems developers (seeking to understand how language extensions can help them in practice).

Categories and Subject Descriptors D.3.2 [Extensible languages]; D.3.4 [Code Generation]; D.2.3 [Program Editors]; C.3 [Real-time and embedded systems]

Keywords Embedded Software, Language Engineering, Language Extension, Domain-Specific Language, Case Study

... do not incur significant overhead regarding memory consumption and performance.
Generated Code is slow, ugly and bloated.

It is not, if you do it right.
Reliability
Errors over Time

reliability

t
Errors over Time

reliability

„perfect“

ideal and impossible
Errors over Time

use and fix

reliability

„perfect“
Errors over Time

reliability

„perfect“

test, verify, proof
Errors over Time

reliability

„perfect“

test, verify, proof

Test type system rules
Write example programs and run
DSL should also have test constructs
Feedback the result into the tool
coverage problem
Errors over Time

reliability

„perfect“
t

DSL should also have verification constructs
Feedback the result into the tool coverage problem (properties)

test, verify, proof
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], SATABAS [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].

```c
int t = 0;
bool makePace(Event e) {
  switch(e) { /* omitted */
  case Tick:
    if (t < LRI) {
      ++t;
      return false;
    } else {
      t = 0;
      return true;
    } /* omitted */
  case Sense:
    if (t < VRP) {
      return false;
    } else {
      t = 0;
      return false;
    }
  }
}
```

Listing 1: Sample pacing logic implementation in C
We've [...] implemented a huge amount of features optimizations [...] We were able to symbolically execute the mbeddr transformation completely [...] sliced for the [one] property that you want to prove [...] This was done in a supercomputer [...] a 1.7 Tb, 64 core machine [...] in under 3 hours. We don't have the proof yet that your transformation is correct, but...

Proving a non-trivial language/tool correct is MAJOR work.
Language Workbenches
Language Workbench
(Martin Fowler, 2004)

Freely define languages and integrate them
Language Workbench
(Martin Fowler, 2004)

powerful editing testing refactoring debugging groupware

language definition implies IDE definition
There's no difference!
A Language Workbench – a tool for defining, composing and using ecosystems of languages.
Open Source
Apache 2.0
http://jetbrains.com/mps
V 3.2.3 is current.
V 3.3 currently developed
Comprehensive Support for many aspects of Language Definition.

+ Refactorings, Find Usages, Syntax Coloring, Debugging, ...
For End Users and Language Developers

### Comprehensive IDE Features

**Data Received**
```c
#include "comitems/Utils/Platform/Implementation/intf/units"

section FractionUnits {
    exported unit % := <no spec> for Fraction
    exported unit ppm := <no spec> for Fraction

    exported conversion nunit -> % {
        val as <no type> -> val * 100
    }

    exported conversion % -> nunit {
        val as <no type> -> val / 100
    }

    exported conversion nunit -> % {
        val as <no type> -> val * 1000
    }
}
```

**States**
- idle
- awaking
- receivingFrame

**Events**
```c
dataReceived(uint8 data) {
    [data == DLT645_FRONT_LEADING_BYTE] -> awaking
    {
        send startFrame(idx, data);
        idx++;
    }
    [data + DLT645_FRAME_SIaddock FLAG] -> receivingFrame
    {
        send frameByte(idx, data);
        idx++;
    }
    [idx == DLT645_HEADER_DATALENGTH_OFFSET] -> receivingFrame
    {
        send dataLength(id, data);
        idx++;
    }
    length = DLT645.FRAME_END_FLAG
    + DLT645.FRAME_START_FLAG
    + DLT645.FRONT_LEADING_BYTE
    + DLT645.FRONT_LEADING_BYTE_COUNT
}
```

**Component**
```c
exported composite component DCFitTestHarnessImpl {
    provides IDCFilterTestRunner
}
```
An extensible set of integrated languages for embedded software engineering.
Open Source @ eclipse.org
Eclipse Public License 1.0
http://mbeddr.com
developers

itemis  7 developers, project management

fortiss  2 developers, verification stuff

SIEMENS  1 developer, verification stuff

SIoux  3 developers, C++

strategic collaboration with
Smart Meter

c.a. 60,000 SLOC
Great modularity
(due to components)
Great Test Coverage (80%)
Physical Units Detect Errors
Custom Extensions: Messages, Registers, Interrupts
Much Lower Integration Effort
Smart Meter
Key Drivers

Mastering complexity through well structured, modular architecture

Reuse through platform-based development approach

Hardware-independent testing

Incremental integration and commissioning
Embedded Software Designer

LMS INTERNATIONAL
Researchpark Haasrode 1237 | Interluevenlaan 58 | B-3001 Leuven [Belgium]
T +32 16 384 200 | F +32 16 384 350 | info@lmsintl.com | www.lmsintl.com

Worldwide
For the address of your local representative, please visit www.lmsintl.com/lmsworldwide

LMS is a leading provider of test and mechatronic simulation software and engineering services in the automotive, aerospace and other advanced manufacturing industries. As a business segment within Siemens PLM Software, LMS provides a unique portfolio of products and services for manufacturing companies to manage the complexities of tomorrow’s product development by incorporating model-based mechatronic simulation and advanced testing in the product development process. LMS tunes into mission-critical engineering attributes, ranging from system dynamics, structural integrity and sound quality to durability, safety and power consumption. With multi-domain and mechatronic simulation solutions, LMS addresses the complex engineering challenges associated with intelligent system design and model-based systems engineering. Thanks to its technology and more than 1250 dedicated people, LMS has become the partner of choice of more than 5000 manufacturing companies worldwide. LMS operates in more than 30 key locations around the world.
10 Recap/ Big Picture
Modeling and Model-Driven * is a diverse field.

Modeling is more than boxes + lines.

It can be a huge benefit!

BYO language is an option!
THANK YOU!