The Art of Building Tools

A Language Engineering Perspective

Markus Voelter
independent/itemis

http://voelter.de
voelter@acm.org
@markusvoelter
1. Tool Extension
2. Ex: mbeddr
3. GTSL
4. Ex: Requirements
5. Ex: Insurance
6. Wrap Up
Tool Extension
The majority of our interviewees were very successful with MDE but all of them either built their own modeling tools, made heavy adaptations of off-the-shelf tools, or spent a lot of time finding ways to work around tools. The only accounts of easy-to-use, intuitive tools came from those who had developed tools themselves for bespoke purposes. Indeed, this suggests that current tools are a barrier to success rather than an enabler.
Complexity problems are typically associated with off-the-shelf tools. Of particular note is accidental complexity – which can be introduced due to [..] [the] lack of flexibility to adapt the tools to a company’s own context [..]

Our interviews point to a strong need for tailoring of some sort: either tailor the tool to the process, tailor the process to the tool, or build your own tool that naturally fits your own process. Based on our data, it seems that, on balance, it is currently much easier to do the latter.
Tool Extensibility

Command-Line Tools

New File Formats
New Processors

Assemble Components (Pipes & Filters)
Tool Extensibility

UI Tools

Buttons  Views
Menus  Actions
(New Languages)
(New Editors)

Platform/Plugin Systems
An Example System

Language Engineering Embedded Software

specific Languages
A collection of integrated languages for embedded software engineering.
An Example System

Language Engineering Embedded Software

An extensible collection of integrated languages for embedded software engineering.

<table>
<thead>
<tr>
<th>User Extensions</th>
<th>Default Extensions</th>
<th>Core</th>
<th>Platform</th>
<th>Backend Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Support</td>
<td>Components</td>
<td>C core</td>
<td>JetBrains MPS</td>
<td>C Compiler, Debugger and Importer</td>
</tr>
<tr>
<td>Decision Tables</td>
<td>Physical Units</td>
<td>Model Checking</td>
<td>NuSMV</td>
<td>Implementation Concern</td>
</tr>
<tr>
<td></td>
<td>State Machines</td>
<td>SMT Solving</td>
<td>Yices</td>
<td>Analysis Concern</td>
</tr>
<tr>
<td></td>
<td>State Machine Verification</td>
<td>Dataflow Analysis</td>
<td>CBMC</td>
<td>Process Concern</td>
</tr>
<tr>
<td></td>
<td>Decision Tables</td>
<td>Visualization</td>
<td>PlantUML</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contracts</td>
<td>PLE Variability</td>
<td>Documentation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requirements &amp; Tracing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Glossaries | Use Cases & Scenarios
An Example System

Language Engineering Embedded Software

An IDE for all Of them

Specific Languages
An Example System

Language Engineering Embedded Software

Open Source
Eclipse Public License

http://mbeddr.com
About mbeddr

Built on JetBrains MPS

Open Source
Apache 2.0
http://jetbrains.com/mps

JetBrains

Meta Programming System
About MPS

Rich Set of Language Aspects

+ Refactorings, Find Usages, Syntax Coloring, Debugging, ...
About MPS

Projectional Editing

Parsing

Projection

Concrete Syntax

Abstract Syntax Tree

Concrete Syntax

Abstract Syntax Tree

Generic Tool
About MPS

Notational Flexibility

Regular Code/Text

Mathematical

Tables

Graphical

Generic Tool
About MPS

Language Composition

Separate Files

L2 → L1

Type System
Transformation
Constraints

In One File

Type System
Transformation
Constraints
Syntax
Editor/IDE

Generic Tool
An Example System
Built on JetBrains MPS
module HelloWorld {
    msgalist messages {
        INFO helloWorld() active: Hello, World
    }
    exported int32 main(int32 argc, string*[] argv) {
        report(0) messages.helloWorld();
        return 0;
    }
}

Messages abstract over IO. Report statements output them.
typedef (Trackpoint*)=>(Trackpoint*) as DataProcessorType;
DataProcessorType processor;

Trackpoint process_nullifyAlt(Trackpoint* tp) {
    tp->alt = 0;
    return e;
}

test case testProcessing {
    Trackpoint tp = {x = 0, y = 0, alt = 100 };  
    processor = :process_nullifyAlt;
    Trackpoint* res = processor(&tp);
    assert(1) res->alt == 0;
}
typedef (Trackpoint*)=>(Trackpoint*) as DataProcessorType;
DataProcessorType processor;

Trackpoint process_nullifyAlt(Trackpoint* tp) {
    tp->alt = 0;
    return e;
}

test case testLambdaProcessing {
    Trackpoint tp = {x = 0, y = 0, alt = 50 };
    processor = [p| p->alt = 100; p; ];
    assert(0) processor(tp)->alt == 100;
}

And yes, we have lambdas (it’s 2013 after all 😊)
void addToQueue(Trackpoint* tp) {
    report(0) messages.queueGettingFull()
        on pos >= QUEUE_SIZE * 3/4;
    pos++;
    if (pos >= QUEUE_SIZE) pos = 0;
    queue[pos] = tp;
}
Features

Test Cases

test case testProcessing {
    Trackpoint tp = {x = 0, y = 0, alt = 100 };
    processor = process_nullifyAlt;
    Trackpoint* res = processor(&tp);
    assert(1) res->alt == 0;
}

exported int32 main(int32 argc, string[] argv) {
    return test testAddToQueue, testQueueFilling;
}

Special expression to run test cases and collect failure count.
```
struct Trackpoint {
    int8  id;             // sequence ID of the trackpoint
    int8/s/ timestamp;   // timestamp as taken from GPS time
    int8/m/ x;           // longtitude, simplified as a number
    int8/m/ y;           // latitude, simplified as a number
    int8/m/ alt;         // altitude as of the GPS
    int8/mps/ speed;     // current speed, if available
};
```

-1

derived unit mps = m s for velocity

Types can have units; additional units can be defined.
Features

Physical Units II

Trackpoint tp = { id = 1, timestamp = 0 s,
    x = 0 m, y = 0 m, alt = 100 m }
assert(0) tp.id == 1 && tp.alt == 100 m;
assert(1) tp.id == 1 && tp.alt == 0 m;

int8 someInt = tp.x + tp.speed; // error, adding m and mps

int8/mps/ speed = (tp2.x - tp1.x) /
    (tp2.timestamp - tp1.timestamp);

Literals can have units; type system calculates w/ units.
convertible unit degC for temperature
convertible unit degF for temperature
conversion degC -> degF = val * 9 / 5 + 32
conversion degF -> degC = (val - 32) * 5 / 9

void storeTemperature(int8\text{/}degC/ temp) {
    // store temp in some data store
}

int8\text{/}degF/ aTempInF = 100 degF;
storeTemperature(convert[aTempInF -> degC]);
Features

Math

```c
int32 sumUpIntArray(int32[] arr, int32 size) {
    return \sum_{i=0}^{size} arr[i];
}
```

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

```c
double midnight1(int32 a, int32 b, int32 c) {
    return \frac{-b + \sqrt{b^2 - 4 * a * c}}{2 * a};
}
```

```c
double midnight2(int32 a, int32 b, int32 c) {
    return \frac{-b + \sqrt{b^2 - \sum_{i=1}^{4} a * c}}{2 * a};
}
```

```c
double sumOfProductsOfLogs(int32[] arr, int32 size) {
    return \sum_{k=0}^{size} \frac{\prod_{i=0}^{k} \log_2 arr[i]}{2};
}
```

Support for readable mathematical symbols.
Interfaces and Components I

```
module Components imports DataStructures {
  exported cs interface TrackpointProcessor {
    Trackpoint* process(Trackpoint* p);
  }
}

exported component Nuller extends nothing {
  provides TrackpointProcessor processor
  Trackpoint* process(Trackpoint* p) <- op processor.process {
    p->alt = 0 m;
    return p;
  }
}
```

Interfaces define operations. Components provide interfaces.
Components can be instantiated and wired.
exported cs interface TrackpointStore1 {
    void store(Trackpoint* tp)
        pre(0) isEmpty()
        pre(1) tp != null
    post(2) !isEmpty()
    post(3) size() == old(size()) + 1
    Trackpoint* get()
        pre(0) !isEmpty()
    Trackpoint* take()
        pre(0) !isEmpty()
        post(1) result != null
    post(2) isEmpty()
    post(3) size() == old(size()) - 1
    query int8 size()
    query boolean isEmpty()
}
In addition, interfaces can have protocol state machines.
exported component Interpolator extends nothing {
  provides TrackpointProcessor processor
  requires TrackpointStore store

  init int8 divident;
  Trackpoint* process(Trackpoint* p) <- op processor.process {
    if (store.is_empty()) {
      store.store(p);
      return p;
    } else {
      Trackpoint* old = store.take();
      p->speed = (p->speed + old->speed) / divident;
      store.store(p);
      return p;
    }
  }
}
Interfaces and components can be visualized.
Mock components specify expectations in context of a test.
exported test case testInterpolatorWithMock {
    initialize interpolatorInstancesWithMock;
    Trackpoint p1 = { id = 1, timestamp = 1 s, speed = 10 mps };
    Trackpoint p2 = { id = 2, timestamp = 2 s, speed = 20 mps };
    ipMock.process(&p1);
    ipMock.process(&p2);
    validateMock (0) interpolatorInstancesWithMock:storeIdMock;
}
Features

Interfaces and Components X

Interface contracts can be verified statically!
exported component Judge extends nothing {
    provides FlightJudger judger
    int16 points = 0;
    void judger_reset() <= op judger.reset {
        points = 0;
    } runnable judger_reset
    void judger_addTrackpoint(Trackpoint* tp) <= op judger.addTrackpoint {
        points += 0
        | tp->alt <= 2000 m | tp->alt >= 2000 m |
        | tp->speed < 150 mps | 0 | 10 |
        | tp->speed >= 150 mps | 5 | 20 |
    } runnable judger_addTrackpoint
    int16 judger_getResult() <= op judger.getResult {
        return points;
    } runnable judger_getResult
} component Judge

Decision tables nicely exploit the projectional editor.
exported component Judge extends nothing {
    provides FlightJudger judger
    int16 points = 0;
    void judger_reset() <= op judger.reset {
        points = 0;
    } runnable judger_reset
    void judger_addTrackpoint(Trackpoint* tp) <= op judger.addTrackpoint {
        points += 0
        tp->alt <= 2000 m  tp->alt >= 2000 m
        tp->speed < 150 mps 0  10
        tp->speed >= 150 mps 5  20
    } runnable judger_addTrackpoint
    int16 judger_getResult() <= op judger.getResult {
        return points;
    } runnable judger_getResult
} component Judge
SUCCESS: Table complete.
FAIL: cells (1, 1) and (2, 1) are inconsistent.
   tp.id : 0
   tp.timestamp : 0
   tp.x : 0
   tp.y : 0
   tp.speed : 0
   tp.alt : 2000
FAIL: cells (1, 2) and (2, 2) are inconsistent.
   tp.id : 0
   tp.timestamp : 0
   tp.x : 0
   tp.y : 0
   tp.speed : 150
   tp.alt : 2000

Decision Tables are analyzed for consistency and completeness
State machines fundamentally consist of states.

```plaintext
statemachinestate FlighAnalyzer initial = beforeFlight {
    state beforeFlight {
    }
    state airborne {
    }
    state landing {
    }
    state landed {
    }
    state crashed {
    }
}
```
Features

State Machines II

```javascript
state beforeFlight {
    entry { points = 0; }
    on next [tp->alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
}

state airborne {
    on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
    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; }
    on reset [ ] -> beforeFlight
}

state landing {
    on next [tp->speed == 0 mps] -> landed
    on next [ ] -> landing { points--; }
    on reset [ ] -> beforeFlight
}
```

States contain transitions with guards, and actions.
State machines can be instantiated; code can interact.
test case testFlightAnalyzer {
    FlightAnalyzer f;
    sminit(f);
}

test case testFlightAnalyzer {
    test statemachine f {
        next(makeTP(200, 100)) -> airborne
        next(makeTP(300, 150)) -> airborne
        next(makeTP(0, 90)) -> landing
        next(makeTP(0, 0)) -> landed
    }
}

+ special support for testing state machines.
State Machines V

```c
statemachine FlightAnalyzer initial = beforeFlight {
    ...
    state crashed {
        entry { raiseAlarm(); }<br>
    }<br>
}
...

void raiseAlarm() {}

statemachine FlightAnalyzer initial = beforeFlight {
    out crashNotification() => raiseAlarm<br>
    ...
    state crashed {
        entry { send crashNotification(); }<br>
    }
}
```

Outgoing interactions via function calls or out events.
Hierarchical state machines (composite states)
State Machines can be visualized in various ways.
Symbolic Model Checking for State Machines
// This state machine implements a way to grade flights. It has separate states for the important flight phases, such as @child(beforeFlight) or @child(airborne).

statemachine FlightAnalyzer initial = beforeFlight {
in next(Trackpoint* tp) <no binding>
readable var int16 points = 0
state beforeFlight {
   on next [tp->alt > 0 m] -> airborne
   exit { points += TAKEOFF; }
} state beforeFlight
section 1.2 existing.comps: Interfaces and Components {

[ Interfaces declare operations that can be provided or used by components. Each operation can also declare pre- and postconditions as well as protocols. These can be checked either at runtime or statically. The @cm(Components) module contains examples. Below is an interface:

embed as text Components.TrackpointStore1/

[ The interfaces, components and their relationships in a given module can also be rendered graphically. An example is shown in @fig(ci)

visualize Components.store.TrackpointStore1/
  components + interfaces (grouped) as ci
  location: vis:/
  scaling: width100

[ The components and their provided (solid lines) and required (dotted lines) ports.

Of course the visualizations are also not just images. In the source to the document, we embed references to \code(IVisualizable) instances. In the doc, one can select the visualization category, and then, during generation, PlantUML automatically rerenders the image.

}
mbeddr supports physical units. For example, \texttt{struct} members can have physical units in addition to their types. An example is the \texttt{Trackpoint/} in the \texttt{DataStructures} module. Here is the \texttt{struct}:

term: Vehicle
A vehicle is \texttt{->}(a special kind of \texttt{Car}).
A car typically has four \texttt{Wheel/Wheels}.
The Drake equation calculates the number of civilizations $N$ in the galaxy. As input, it uses the average rate of star formation $SF$, the fraction of those stars that have planets $fp$ and the average number of planets that have potentially support life $ne$. The number of civilizations can be calculated with $N = SF \times fp \times ne$. 

---

Error: type int8 is not a subtype of boolean
**Product Line Variability**

```
feature model FlightProcessor
  processing ? { 
    nullify
    normalizeSpeed xor { 
      maxCustom [int16/mps/ maxSpeed]
      max100
    }
  }
}
configuration model cfgNullifyMaxAt200 configures FlightProcessor
  processing { 
    nullify
    normalizeSpeed { 
      maxCustom [maxSpeed = 200 mps]
    }
  }
```
Trackpoint processTrackpoint(fmconfig<FlightProcessor> cfg, 
        Trackpoint tp) {

    Trackpoint result;
    variant<cfg> {
        case (nullify && maxCustom) {
            result = process_nullifyAlt(tp);
            if (tp.speed > maxCustom.maxSpeed) {
                result.speed = maxCustom.maxSpeed;
            }
        }
        case (nullify && max100) {
        }
        case (nullify) { result = process_nullifyAlt(tp); } 
        default { result = process_doNothing(tp); }
    }

    return result;
Features

Product Line Variability III

```c
Variability from FM: FlightProcessor
Rendering Mode: product line

module StaticVariability imports DataStructures {
    Trackpoint* process_trackpoint(Trackpoint* t) {
        t->alt = 0 m;
        return t;
    } process_trackpoint (function)

    exported test case testStaticVariability {
        Trackpoint tp = {
            id = 1
            alt = 2000 m
            speed = 150 mps
        };

        assert(0) process_trackpoint(&tp)->alt == 2000 m;
        assert(1) process_trackpoint(&tp)->alt == 0 m;
    } testStaticVariability(test case)
}
```

Static Variability for any Program w/ Variant Editing
Features

Tree Views I

Different Tree View Structures defined by language concepts.
Features

Tree Views II

Custom Commands are supported as well.
Debugging on the DSL Level (Extensible!)
Features

VCS Diff/Merge

Diff/Merge on the Projected Syntax
Features

CI Server Integration

Building Programs on Command Line/CI Server
3

Generic Tools GTSL Specific Languages
Thought Process

From Data Formats To Languages

Structure, Constraints, Semantics

Data Format + Syntax + IDE

Language
Thought Process

Language Engineering

Languages

Language Reuse
Language Modularization
Language Composition

Language Engineering
Thought Process
Language Engineering
Languages
Language Engineering
Text   Math   Graphics
Tables  Symbols  Forms
Syntactic Diversity
But does this really work?
Generic Tools, Specific Languages

Ingredients

Specific Languages | Languages
Language Engineering | Syntactic Diversity
Language Workbenches
Generic Tools, Specific Languages

Ingredients

Specific Languages

Language Engineering

Syntactic Diversity

Language Workbenches

(we don’t have to reimplement editors and synchronizers)
## Language Workbenches

### Typical Features

- Goto Definition/Find Usages
- Error Markup/Quick Fixes
- Syntax Highlighting
- Code Completion
- Search/Replace
- Refactoring
- Debugging
- Reporting
- Visualization
- Version Control
Language Workbenches

Typical Features

for any Language!
Language Workbenches act as the foundation for IDEs for any language.
The majority of our interviewees were very successful with MDE but all of them either built their own modeling tools, made heavy adaptations of off-the-shelf tools, or spent a lot of time finding ways to work around tools. The only accounts of easy-to-use, intuitive tools came from those who had developed tools themselves for bespoke purposes. Indeed, this suggests that current tools are a barrier to success rather than an enabler.
Complexity problems are typically associated with off-the-shelf tools. Of particular note is accidental complexity – which can be introduced due to poor consideration of other categories, such as lack of flexibility to adapt the tools to a company’s own context [...]

Language Workbenches

Typical Features

Used by the tool vendor to **build** the initial tool (languages).

Used by the end user to **adapt** the tool (lang extensions)!

Extensions are first-class!
Generic Tools, Specific Languages

Adaptability is built-in!

Extensions are first-class!

Fundamentally different from Today’s State-of-the-Art in Tools
Example II: Requirements
Features

Requirements

Requirements Architectural Components

1. **nullifies the altitude**
   - Nuller /participant: tags

2. **averages over the flights**
   - Interpolator /participant: tags

3. **stores flights in memory**
   - InMemoryStore /participant: tags

Structured and Hierarchical Requirements.
2. nullifies the altitude

Nuller /participant: tags

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent feugiat enim arcu, ut egestas velit. Suspendisse potenti. Etiam risus ante, bibendum ut mattis eget, convallis sit amet nunc. Ut nec justo sapien, vel condimentum velit. Quisque venenatis faucibus tellus consequat rhoncus. Vestibulum dapibus dictum vulputate. Phasellus rhoncus quam eu dui dictum sollicitudin. This requirements is a special case of $cfreq(Judger)$.
Features

Requirements Relationship Diagram

Relationships between Requirements (downstream, upstream)
Features

Requirements ext’d with Business Rules

4 Points you get for each trackpoint
InFlightPoints /functional: tags


calculation PointForATrackpoint: This rule computes the points awarded for a Trackpoint. It does so by taking into account the @alt and the @speed passed as arguments.

parameters: int16 alt: current altitude of the trackpoint
int16 speed: current speed of the trackpoint

result = (BASEPOINTS * 1) *

<table>
<thead>
<tr>
<th>speed &gt; 180</th>
<th>speed &gt; 130</th>
<th>otherwise 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt &gt; 2000</td>
<td>alt &gt; 1000</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>


test PointForATrackpoint(500, 100) == 0
PointForATrackpoint(500, 1200) == 0
PointForATrackpoint(1100, 165) == 210
PointForATrackpoint(2100, 140) == 100
PointForATrackpoint(2100, 200) == 300

Live (interpreted) Business Rules can be Embedded in Req.
Features

Debugging Business Rules („Live Program‘g“)

calculation PointForATrackpoint: This rule computes the points awarded for a Trackpoint. It does so by taking into account the @alt and the @speed passed as arguments.

parameters:

- int16 alt: current altitude of the trackpoint
- int16 speed: current speed of the trackpoint

\[
\text{result} = \left( \frac{10 \times \text{BASEPOINTS}}{10} \right) \times \begin{cases} 
  20, & \text{false} \\
  20, & \text{true} \\
  0, & \text{otherwise}
\end{cases}
\]

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

tests:

- PointForATrackpoint(500, 100) == 0
- PointForATrackpoint(500, 1200) == 0
- PointForATrackpoint(1100, 165) == 210
- PointForATrackpoint(2100, 140) == 100
- PointForATrackpoint(2100, 200) == 300

All intermediate expression values shown inline.
exported component Judge2 extends nothing {
    provides FlightJudger judger
    int16 points = 0;
    void judger_reset() ← op judger.reset {
        points = 0;
    } runnable judger_reset
    void judger_addTrackpoint(Trackpoint* tp) ← op judger.addTrackpoint {
        points += PointForATrackpoint(stripunit[tp->alt], stripunit[tp->speed]);
    } runnable judger_addTrackpoint
    int16 judger_getResult() ← op judger.getResult {
        return points;
    } } runnable judger_getResult

These Business Rules can be „called“ from C Code
1.2.1 Describes the Interpolation

Interpolation /scenario: tags


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 {
        -> Judger.judge(new Trackpoint t)
    } loop
}
Scenarios can be Visualized
An extension supports workpackages for requirements.
Features

Workpackage Assessments

Assessment: EffortsOfWorkPackages
query: workpackages for scope <no scope> responsible <no company> status any prio >= <no prio>
sorted: true
must be ok: false hide ok ones: false

FlightJudgementRules

<table>
<thead>
<tr>
<th>FasterThan100.impl (-)</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>FasterThan200.impl (-)</td>
<td>32</td>
</tr>
<tr>
<td>InFlightPoints.poc (-)</td>
<td>80</td>
</tr>
<tr>
<td>PointsFactor.prototype (1)</td>
<td>24</td>
</tr>
<tr>
<td>PointsForTakeoff.impl1 (1)</td>
<td>80</td>
</tr>
<tr>
<td>PointsForTakeoff.impl2 (1)</td>
<td>40</td>
</tr>
</tbody>
</table>

total 6, new 2, ok 0
total effort: 1 / 35 days

A report over the workpackages and the spent work
Features

Requirements Tracing

```
requirements modules: FlightJudgementRules
module StateMachines imports DataStructures, stdlib_stub, stdio_stub {

    [#define TAKEOFF = 100;]-> implements PointsForTakeoff
    [#define HIGH_SPEED = 10;]-> implements FasterThan100
    [#define VERY_HIGH_SPEED = 20;]-> implements FasterThan200

    statemachine FlightAnalyzer initial = beforeFlight {
        in next(Trackpoint* tp) <no binding>
        in reset() <no binding>
        state landed {
            [entry { points += LANDING; }]-> implements FullStop
                on reset [ ] -> beforeFlight
        } state landed
    }
```
Example III: Insurance
Insurance Workbench

More Form-Like Notation

**Rule Set Type** DemoRuleSetType

**Business objects**

- person : Person
- policy Policy :

**Variables:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMI</td>
<td>int</td>
</tr>
<tr>
<td>FR</td>
<td>int</td>
</tr>
<tr>
<td>NN</td>
<td>int</td>
</tr>
<tr>
<td>TT</td>
<td>int</td>
</tr>
<tr>
<td>J</td>
<td>int</td>
</tr>
<tr>
<td>A3</td>
<td>int</td>
</tr>
<tr>
<td>G3</td>
<td>int</td>
</tr>
<tr>
<td>ANUI</td>
<td>int</td>
</tr>
<tr>
<td>X</td>
<td>int</td>
</tr>
</tbody>
</table>

**Parent**

- <no parent>

**Libraries**

- Standard
- Extra

This workbench is to be used by insurance experts
Insurance Workbench

More Form-Like Notation

<table>
<thead>
<tr>
<th>Rule Set Type</th>
<th>DemoRuleSetType</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Business objects</strong></td>
<td></td>
</tr>
<tr>
<td>person : Person</td>
<td></td>
</tr>
<tr>
<td>policy Policy :</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variables:</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMI : int</td>
<td>&lt;no parent&gt;</td>
</tr>
<tr>
<td>FR : int</td>
<td></td>
</tr>
<tr>
<td>NN : int</td>
<td></td>
</tr>
<tr>
<td>TT : int</td>
<td>Libraries</td>
</tr>
<tr>
<td>J : int</td>
<td>Standard</td>
</tr>
<tr>
<td>A3 : int</td>
<td>Extra</td>
</tr>
<tr>
<td>G3 : int</td>
<td></td>
</tr>
<tr>
<td>ANUI : int</td>
<td></td>
</tr>
<tr>
<td>X : int</td>
<td></td>
</tr>
</tbody>
</table>

This workbench is to be used by insurance experts.
rule set DemoRuleSet2 is of type DemoRuleSetType

EU0 : int [ save false print false ]
CATEG : string [ save false print false ]
CATEG1 : double [ save true print true ]

Toggle Information

PREMIO =

\[ A1 > 10 \implies \text{EU0} \\
\text{<always>} \implies \text{FLAG} \]

FLAG =

\[ \begin{align*} 
\text{CATEG1 equals 60 or CATEG1 equals 63 or CATEG1 equals 64} & \implies 160 \\
\text{PREMIO equals 0} & \implies 162 \\
\text{CATEG1 > 0 or substr(inga[4], 1, 1) equals "V"} & \implies 163 \\
\text{<always>} & \implies \text{PREMIO + FLAG} 
\end{align*} \]

PREMIO = \[ \text{<always>} \implies \text{round(PREMIO \ast (1 + factacer), 0)} \]

Non-Programmers like Forms and Buttons - and need Lang's
Insurance Workbench

Mathematical Notation

\[
\text{int} \ \text{other}(a : \text{int}, b : \text{int}) \Rightarrow a + b + \sum_{i = 1}^{5} i + \prod_{p = 1}^{3} p
\]

\[
\text{local} = \left[ A1 \Rightarrow \sum_{i = 1}^{\text{NN}} \left( \frac{(D(X + \text{ANUI} + i - 1) - D(X + \text{ANUI} + i)) \times (1 - \frac{\text{TM18}[i]}{\text{TM17}})}{D(X + \text{ANUI})} \right) \right]
\]

\[
\text{int} \ \text{rate}(\text{age} : \text{int}) \Rightarrow 1 + \frac{\text{age}}{\text{AOPS} - 9} + \text{in01}
\]

\[
4 \times 5 + \sum_{i = 8}^{12} i \times 8
\]

This workbench was used by insurance domain experts
<table>
<thead>
<tr>
<th>sensorOmega</th>
<th>designOmega</th>
<th>curTime</th>
<th>torque</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0 s</td>
<td>-23 Nm</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.1 s</td>
<td>-38.5 Nm</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.2 s</td>
<td>-47.5 Nm</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.3 s</td>
<td>-47.5 Nm</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.4 s</td>
<td>-36 ±0.001</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.5 s</td>
<td>9 ±0.001</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.6 s</td>
<td>236.25 ±0.001</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.7 s</td>
<td>2023 ±0.001</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.8 s</td>
<td>22093 ±0.001</td>
</tr>
<tr>
<td>5 radps</td>
<td>10 radps</td>
<td>0.9 s</td>
<td>379457.5 ±0.001</td>
</tr>
</tbody>
</table>

A bit like „Excel“ with a real language behind it.
Insurance Workbench

Tables (taken from diff. Example)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Unit</th>
<th>Default</th>
<th>Description</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLB_Time</td>
<td>double</td>
<td>s</td>
<td>0.1</td>
<td>Time in seconds</td>
<td>range 0.00 .. 1.0E16</td>
</tr>
<tr>
<td>Temperature_K</td>
<td>double</td>
<td>K</td>
<td>300.0</td>
<td>Temperature in Kelvin</td>
<td>range 223.0 .. 1773.0</td>
</tr>
<tr>
<td>Temperature_C</td>
<td>double</td>
<td>degC</td>
<td>25.0</td>
<td>Temperature in Celsius</td>
<td>range -50.0 .. 1250.0</td>
</tr>
<tr>
<td>Torque</td>
<td>double</td>
<td>Nm</td>
<td>0.0</td>
<td>Torque in Nm</td>
<td>&lt;no elements&gt;</td>
</tr>
<tr>
<td>Inertia</td>
<td>double</td>
<td>kgm2</td>
<td>0.0</td>
<td>Inertia in kg m square</td>
<td>min 0.00</td>
</tr>
<tr>
<td>motor_speed</td>
<td>double</td>
<td>radps</td>
<td>&lt;none&gt;</td>
<td>Motor speed in rad per sec</td>
<td>range 0.00 .. 100000.0</td>
</tr>
<tr>
<td>shaft_speed</td>
<td>double</td>
<td>radps</td>
<td>&lt;none&gt;</td>
<td>Output Shaft Speed</td>
<td>range -20000.0 .. 20000.0</td>
</tr>
<tr>
<td>motor_power</td>
<td>double</td>
<td>W</td>
<td>&lt;none&gt;</td>
<td>Motor power in Watts</td>
<td>range -100000.0 .. 100000.0</td>
</tr>
<tr>
<td>coolant_flowrate</td>
<td>double</td>
<td>m3ps</td>
<td>&lt;none&gt;</td>
<td>Coolant volume flow rate</td>
<td>range 0.0 .. 3.0</td>
</tr>
</tbody>
</table>

A bit like „Excel“ with a real language behind it.
Summing up
Summing Up

Key Points

To build meaningful tools, the data must be extended.

Extending the tool (buttons, views, ...) is not enough!
Structured Data can be expressed with languages.

Languages are data formats plus syntax and IDE.
Language Engineering supports extension and composition. This supports adapting tools for specific domains easily.
IDE-style tools are very good for editing data/programs.

We've got a lot of experience from regular programming.
Summing Up

Key Points

Language Workbenches are the key enabling technology.

MPS is IMHO the most powerful, but it's not the only one!
Summing Up

Key Points

Let’s build new classes of tools!

... which make meaningful extensibility a reality!
The End.
The End.