Domain Specific - a Binary Decision ?

Markus Voelter
independent/itemis
Ötztaler Strasse 38
70327 Stuttgart, Germany
voelter@acm.org

Bernhard Merkle
SICK AG
R&D Software Engineering
Erwin-Sick-Str.1
79183 Waldkirch, Germany
bernhard.merkle@gmail.com

ABSTRACT
It is often considered a binary decision whether something is domain specific or not. Consequently, there are domain specific languages (DSL) and general purpose languages (GPL), there are domain specific and non-domain specific modeling tools, there are domain specific and non-domain specific methodologies etc. In this paper we argue, that domain specificity is not a hard decision, but rather one extreme on a continuum. We also argue that many systems can be more efficiently described with a mix of domain specific and non-domain specific abstractions. This view of the world has consequences for languages, tools and methodologies, specifically the ability to modularize and compose languages. Additionally we outline these consequences and provide an extensive example based on embedded systems.

Keywords
domain specific language, programming, modeling, projectional editor, language development, language composition, modular language, language workbench, embedded systems.

1. INTRODUCTION
Traditionally, domain specific modeling (DSM) (or more generally, the use of domain specific languages plus code generation or interpretation) is considered the opposite of general purpose based approaches. We all know the discussions and "religious battles" we and others are continuously having with people who advocate the use of general purpose languages for describing software. However, we all agree that some general purpose languages are extremely useful. We don't call them modeling languages, we rather call them programming languages. In many methodologies that proclaim to use domain specific modeling, there is still the need to write code in general purpose programming languages to express certain aspects of the system. On the other hand, notations such as state machines, hierarchical components, or data flow diagrams can hardly be called domain specific. They are not as general purpose as a programming language, but they are still very general compared to any reasonable definition of "domain specific".

In this paper we argue that it is useful to be able to mix domain-specific and general purpose aspects in a cojoint environment. Specifically DSLs and GPLs can be seen as collaborating languages in one tool called language workbench. Also, programming and modeling should not be considered different. This leads to the notion of modular modeling and programming languages and the corresponding consequences for tools.

This paper is structured as follows. An overview about the domain terminology is given in Chapter 2. In Chapter 3 we challenge the question about "modeling vs. programming" and whether there should be a distinction at all between the two. Chapter 4 describes the idea of modular languages as well as various ways of combining them. A real world case study is presented in Chapter 5: modular languages applied for developing embedded applications. Chapter 6 shows consequences for tools and finally Chapter 7 summarizes and concludes the paper.

2. WHAT IS A DOMAIN
To be able to discuss about the domain specificity of something, we should first define what a domain is. We have taken the following definitions:

- Domain (software engineering): a field of study that defines a set of common requirements, terminology, and functionality for any software program constructed to solve a problem in that field
- Domain engineering: the reusing of domain knowledge in the production of new software
- Domain knowledge: a specific expert knowledge valid for a pre-selected area of activity, such as surgery

As programmers we often distinguish between business domains and technical domains, with the rough distinction being that business domains are what non-programmers care about and technical domains are what programmers care about. However, if you consider electrical engineers or scientists as non-programmers, then the distinction blurs: in embedded systems for example, state machines and block diagrams are what these "domain experts" use for designing their systems, although developers probably would not consider these languages to be business-domain specific.

So let us consider a domain to be anything for which a specific set of abstractions and notations is advantageous when describing structure or behavior in that domain. It can be wide, narrow, deep, shallow, business-oriented or technical.

Domains can also be hierarchical in that one domain is a specialized (or: narrower) variant of another one. For example, robot control is specialized compared to embedded systems in general. The abstractions from embedded systems are also useful in robot control, but there are more specialized abstractions that make sense in addition. This is an important observation, because this kind of relationship between domains should be represented by the DSLs for domains.

3. MODELING VS. PROGRAMMING
Historically there is (still) a somewhat "clear" distinction between modeling and programming. Modeling is often associated with graphical languages, domain specific (Entity-Relational for DBs)

or general purpose (e.g. UML). A code generator then produces most of the artifacts automatically, yet the developer often has also to manually implement certain artifacts. Sometimes he even has to extend or complete the generated code, e.g. because the modeling language does not support the concepts the developer needs. Protected regions are a classical example of such a undesirable situation. The scenario would in some sense be comparable to manually extending or modifying generated assembler code, produced from a high level programming language. It will simply not scale.

With the advent of textual DSLs [11] and the associated IDE support via textual language workbenches [10], the modeling community is increasingly becoming aware of the benefits of textual modeling. However the gap between modeling and programming still exists in practice, especially regarding the tools used for the two activities.

We do not think that it is really necessary (or even useful) to distinguish between modeling and programming in the end. Essentially modeling and programming should be the same. One could even come up with the provoking statement that "we don't want to model; we want to program"…

- at various levels of abstraction
- from different viewpoints
- all of these integrated
- with different degrees of domain specificity
- with suitable notations
- and with suitable expressiveness
- and always precise, tool processable and with IDE support.

Some of the viewpoints for describing a system might be suitable for usage by nonprogrammers, if we have these kinds of people in our group of stakeholders. So in essence, we want a development environment that allows us to mix and integrate arbitrary languages and notations, independent of whether they are domain specific or not. This explicitly includes programming languages. The next paragraph elaborates on the meaning of the term "integrating".

4. MODULAR LANGUAGES

Modular languages are languages that can be incrementally extended. They integrate with other modules in the following ways:

- **referencing:** language concepts from module A can refer to language concepts from module B. Languages come with their own notations and the programs written in these languages are stored in their own partitions/models/diagrams. However, references to program elements expressed with other languages are possible and tool supported. Example (see case study): the system model (a kind of makefile) references modules defined as part of the "normal" C programming language.
- **cascading:** concepts from module A are translated into concepts from module B as a means of implementing A's concepts. This way, more abstract languages can be cascaded on top of more general languages, supporting incremental addition of domain specific abstractions. An interesting base case is where things are cascaded on top of existing programming languages -- this is conceptually similar to what's today typically called code generation. Example (see case study): the robot control language being translated down to a general purpose embedded systems language
- **extension:** module B extends module A in the sense that it provides additional language concepts. B is a superset of A. Example (see case study): tasks and state machines as additional contents of modules, in addition to procedures
- **embedding:** concepts from module A can be embedded in concepts from module B. Notations defined in A and B are reused, even if the language modules are embedded. To make this feasible, it must be possible to extend the languages as well, since additional concepts that connect the embedded concepts to the surrounding modules are required. Example (see case study): boolean C expressions can be embedded as guard conditions in state machines
- **annotation:** language A contributes additional properties to existing concepts of a language B without invasively modifying B. This is similar to introductions in AOP. Example (see case study): the traceability and variability expressions are implemented in this way.
- **cross-cutting by translation:** this last kind of integration is a little bit different in that it does not add any language concepts. Instead a module A (transparently) changes the translation of concepts defined in a language B without invasively changing B. Example (see case study): the *safety* feature translates assignments in a way so that a runtime value range check is performed.

Language modules can either be custom-built for a project, platform or business domain (a.k.a. domain specific) or they can be reused from a library of existing language modules (these are most likely more general, hence applicable in many contexts).

5. CASE STUDY

This section contains an extensive case study that illustrates the various ways of integrating language modules; these include what's traditionally considered "programming" and what's traditionally considered "modeling". The case study is from the embedded systems domain. Let's start with some background.

5.1 Embedded Software Development

Traditional embedded system development approaches use a variety of tools for various aspects of the system, making tool integration a major headache. One of these tools is the C programming language. Some of the specific problems of embedded software development include the limited capability for meaningful abstraction in C and "dangerous" features of the language (leading to coding conventions such as Misra-C [1]), the proprietary and closed nature of modeling tools, the integration of models and code, traceability to requirements [2], long build times as well as the consistent implementation of product line variability [3].

To address these issues, we propose a modular modeling and programming language based on C that supports higher-level abstractions and system-specific extensions supported via a projectional language workbench. The proposed language uses C as its core and adds several useful extensions, including a module

system with visibility rules, physical quantities (as opposed to just *ints* and *floats*), first-class state machines, dataflow ports, mathematical notations, memory mapping and bit fields, as well as first-class support for various inter-process communication approaches. These additional abstractions are transformed down to C for eventual compilation. Our proof-of-concept is implemented with JetBrains MPS for the language parts and the Lego Minstorms kit for embedded systems..

5.2 JetBrains MPS

JetBrains' Meta Programming System is an open source projectional language workbench [4]. This means that users don't edit ASCII text that is then parsed, but each editing operation directly changes the tree structure of the underlying program. Since we deal with a projectional editor there is no parser involved and different projections (textual, graphical, and tabular) are possible. Lastly the generator is defined to emit text (for a low-level language) or it transforms higher-level code into code expressed in lower level languages..

Editing the tree as opposed to "real text" needs some getting used to. Without specific adaptations, every program element has to be selected from a drop-down list and "instantiated". However, MPS provides editor customizations to enable editing that resembles modern IDEs that use automatically expanding code templates. In some cases though, the tree shines through: Consider changing a statement like *int i = j+k*; to *int i = (j+k)*2*; you cannot simply move the cursor to the left of *j* and insert a left parenthesis. Rather, you have to select the + operator (the root node of the expression on the right) and use a *Surround with Parens* refactoring.

We now briefly illustrate how a language is defined in MPS. A more extensive description can be found in [5]. MPS, like other language workbenches, comes with a set of DSLs for language definition, a separate DSL for each language aspect. Language aspects include structure, editor, type system, generator as well as support for features such as quick fixes or refactorings.

Defining a new language starts with the language structure (aka meta model). This is very much like object oriented programming since language elements are represented as concepts that have properties, children and references to other concepts. The second step is the editor for the language concepts. An editor defines how the syntax for the concepts should look like - it constitutes the projection rules. Figure 1 is an example.

Next is the definition of the type system. For example, the type property of a *LocalVariableDeclaration* must be compatible with the type of its *init* expression. For the type system definition as well as further customization the reader is referred to [5].

We already alluded to the relationship between object oriented programming and language definition in MPS. This analogy also holds for language extension and specialization. Concepts can extend other concepts, and subconcepts can be used polymorphically.

Languages also define translation rules to lower-level languages or to text. MPS includes an incremental translation engine that reduces program code as long as translation rules are available for the program elements. At the end, text generators output regular program text that can be fed into a compiler.

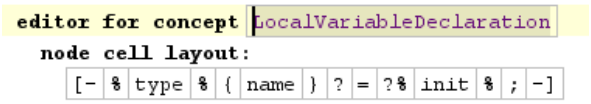

Figure 1. Defining an editor for a local variable declaration statement (as in int i = 2*2;)

5.3 The Modular Embedded Language

As a proof of concept, we are currently building a first cut of a modular embedded language (MEL) based on JetBrains MPS. We use Lego Mindstorms [6] as the target platform together with the OSEK [7] operating system. C and OSEK are widely used in automotive systems, so the technologies used in the proof-of-concept are relevant in real systems. The current baseline showcase is a simple line follower robot. It uses a single light sensor to follow a thick black line and keeps track of that line by changing the left and right motor speeds to turn along with the line. The following sections describe some of the features of the current MEL.

Core Language. The core of the MEL is a implementation of the C programming language. It supports variables, constants, enums, structs, functions, most of C's statements and expressions as well as the type system including pointers. Instead of header files, the language provides the concept of a module. Modules are like namespaces and contain variables, typedefs, structs and functions - all the elements that can be declared on top level in C programs. Module contents can be exported. Modules can declare dependencies to other modules which makes their exported contents visible to depending module.

Figure 2 shows a screenshot of the basic line follower program implemented with MEL. It contains a module *main* that uses three external API modules. It contains constants as well as a *statemachine* which manages the two phases of the program: *initialization* and *running*. The cyclic *run* task is responsible for reading the sensor and adjusting motor speeds. It is called every two system ticks. What the *run* task actually does is state-dependent by virtue of the *stateswitch*; if the *linefollower* state machine is in the *running* state it reads the light sensor and updates the motor settings. This state is entered by signalling *linefollower:initialized* at the end of the *initialize* block. Finally, the module contains the *updateMotorSettings* function which actually drives the motors.

Let us look at some of the features available in the MEL and relate them to the various language composition mechanisms outlined above.

Tasks. Tasks capture behaviour that should be executed at specific times, currently *at startup* and *cyclic* are supported. Tasks are like functions, but they have no arguments and no return type.

Tasks are an example of extension in that new kinds of *IModuleContents* are defined. *IModuleContent* is an interface defined by the core language to type everything that can be embedded into modules. Tasks are also an example of cascading in that the tasks are translated to C functions; in the rest of this paper we will not mention cascading explicitly if we "simple generate C code".

Statemachines. The MEL contains a language module for state machines. It supports the declaration of state machines (with

states, events, transitions, guard conditions as well as entry and exit actions), a statement to fire events into a state machine as well as a *stateswitch* statement to create a *switch*-like construct for implementing state-dependent behavior in tasks or functions.

```
doc This module represents the code for the line follower lego robot. It has a coupl
module main safe imports OsekKernel, EcAPI, BitLevelUtilies {

  constant int WHITE = 500;
  constant int BLACK = 700;
  constant int SLOW = 20;
  constant int FAST = 40;

  doc Statemachine to manage the transition between the initialization and operation p
  statemachine linefollower {
    event initialized;
    initial state initializing {
      initialized [true] -> running
    }
    state running {

    }
  }

  initialize {
    ecrobot_set_light_sensor_active(SENSOR_PORT_T::NXT_PORT_S1);
    event linefollower:initialized
  }

  terminate {
    (bumper &&!debugOutput) ecrobot_set_light_sensor_inactive(SENSOR_PORT_T::NXT_PORT_S1);
  }

  doc This is the cyclic task that is called every 1ms to do the actual control of the
  task run cyclic prio = 1 every = 2 {
    stateswitch linefollower
      state running
        int32 light = 0;
        light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
        if ( light < ( WHITE + BLACK ) / 2 ) {
          updateMotorSettings(SLOW, FAST);
        } else {
          updateMotorSettings(FAST, SLOW);
        }
      default
        <noop>;
        <<hidden, only visible in projection level 'all'>>
  }

  doc This procedure actually configures the motors based on the speed values passed i
  void updateMotorSettings( int left, int right ) {
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, left, 1);
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, right, 1);
  }
```

Figure 2: Code for the simple line follower robot expressed with the MEL

State machines are another example of extension: State machines implement the *IModuleContent* interface so they can be put into modules. The *event…* keyword and the *stateswitch* are subtypes of the core language's *Statement* concepts, making sure they can be used in statement context. State machines are also an example of embedding, since core's *Expressions* can be embedded into guards in transitions. Extension is used to provide additional kinds of expressions to refer to the arguments of events.

Note that by the end of 2010 we will most likely support a graphical notation for state machines. However, as of now, we also support tabular notations as shown in Figure 3.

linefollower	initializing	paused	running	crash
initialized	true running			
bumped			true crash	
blocked			true paused	
unblocked		true running	true crash	

Figure 3: Editing the state machine as a table embedded in the program code

A special kind of integer. Working with the sonar sensor to detect obstructions in the robot's path requires averaging over repeated measurements because the sensor is very sensitive. This is useful for many sensors, so an extension of the type system to provide averaging variables is included. Figure 4 shows the declaration of a variable with an *avg* type: the base type is *int*, the number of elements over which to average is 10 and the initialization value is 250. From this declaration the transformation produces two more variables: a rolling buffer that remembers the last 10 measurements and an index into that buffer to determine where the next value is stored. The =/ operator (see B) inserts a new measurement into to the buffer, calculating the new average and assigning it to the variable of the left of the operator. The variable can be read just like any other variable (see C). This is another example of extension. We've included this example to illustrate that we not only can extend "big" things such as tasks, but also more intricate aspects such as the type system.

A

```
{sonar} var avg(int, 10) currentSonar = 250;
```

B

```
{sonar} task sonartask cyclic prio = 2 every = 100 {
          currentSonar =/ ecrobot_get_sonar_sensor(SENSOR_PO
          {debugOutput} debugInt(2, "sonar:", currentSonar);
        }
```

C

```
{sonar} if ( currentSonar < 150 ) {
  event linefollower:blocked
  terminate;
}
```

Figure 4: Averaging Variables

Safety. One aspect of safety is making sure that values are not assigned to variables that cannot hold these values. For example, if a programmer assigns a value to an *uint8*, the value must not be larger than 255. To enforce this, coding guidelines of many organizations require the use of safe utility functions (such as *mul32(…)*) instead of the built-in operators. The MEL supports such an approach transparently. A module can simply be marked as *safe*: all assignments and initializations are wrapped with a *checkSizeNN(…)* function call that logs an error if the value is outside the permitted range. An alternative implementation would transparently replace built-in operators with safe library functions such as *mul32*.

This is an example of cross-cutting by translation. No change in the notation is necessary, but when translating the models to low-level C code, the translation of existing concepts is performed differently.

Components. Component-based development is widespread in software engineering in general, and in embedded systems in particular. Components are the "smallest architecturally relevant building block". They encapsulate behavior and expose all system-relevant characteristics declaratively. In the proof-of-concept, a language module is available to express interfaces, components and component implementations

This is another example of extension of course, since components and interfaces can be embedded into modules. However, embedding is also used since existing statements can be used in the body of component "methods".

A domain specific language. All the MEL facilities described so far address embedded software development in general. There was

nothing in the languages that is specific to mobile robots that can drive and turn, such as our line follower.

Consider now the role of a robot vendor who sells robots with two wheels that can drive a predefined route. Each customer wants a different predefined route. The vendor has to develop a different route-driving program for each customer. Of course this can be achieved with tasks, state machines, variables and functions. But it would be better if a domain specific language for defining routes was available. Figure 5 shows an example program written in such a language.

```
module impl imports <<imports>> {

  int speed( int val ) {
    return 2 * val;
  }

  robot script stopAndGo
    block main
      accelerate to 12 + speed(12) within 3000
      block newBlock on bump stop
        drive on for 2000
        turn left for 200
      {long} block driveMore
               accelerate to 80  within 2000
               turn right for 3000
      decelerate to 0  within 3000
      stop

}
```

Figure 5: A robot routing script embedded in a module

This is an example of cascading languages: on top of a general purpose language (Embedded/C/OSEK/Mindstorms), a domain specific language (robot routing) is cascaded. The domain specific constructs are translated down to the more general purpose constructs for execution. Embedding is also used, however, since existing core expressions can be still used within robot scripts, for example to calculate the target speed of the accelerate command.

Requirements traceability and product line variability. MEL also supports traceability to requirements as well as feature annotations to express product line variability. Arbitrary program elements can be annotated with traces to requirements or expressions that determine the dependency of this element to configuration features. Figure 6 shows this: the (green) *trace* annotations are the requirements traces, the (blue) expressions in curly braces are feature dependencies.

These are examples of annotation, where a language (the feature dependencies language or the tracing language) can annotate additional data to elements from other languages, without changing these other languages. Note how figure 5 contains a feature dependency annotation for DSL code.

6. CONSEQUENCES FOR TOOLS

The distinction into programming and modeling tools, and further, into domain specific and non-domain specific modeling tools should be questioned. Sure, we should be able to define our own domain specific languages since this ability is crucial for effective software development. However, general-purpose notations, be they programming languages, simple expression languages, component languages or state machines should also be available. This also implies that the tools must be flexible enough to be able to use both graphical and textual notations and mixed them since textual state machines or graphical programming languages are unwieldy in general.

```
trace Cyclic, Efficient
doc This is the cyclic task that is called every 1ms to do the actual control of the
task run cyclic prio = 1 every = 2 {
  trace TwoPhases
  stateswitch linefollower
    state running
      {bumper} int8 bump = 0;
      {bumper} bump = ecrobot_get_touch_sensor(SENSOR_PORT_T::NXT_PORT_S3);
      {bumper} if ( bump == 1 ) {
        {debugOutput &&!bumper} { debugString(3, "bump:", "BUMP!"); }
        event linefollower:bumped
        terminate;
      }
      {sonar && debugOutput} if ( currentSonar < 150 ) {
        event linefollower:blocked
        terminate;
      }
      trace Init
      int32 light = 0;
      light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        trace ConsistentSetting        ;
        updateMotorSettings(SLOW, FAST)
      } else {
        trace ConsistentSetting        ;
        updateMotorSettings(FAST, SLOW)
      }
      {debugOutput} trace OptionalOutputlkjlkjljlkj
                    { debugInt(4, "light:", light); }
    {sonar} state paused
              trace Calibration
              updateMotorSettings(0, 0);
              trace Init
              if ( currentSonar < 255 ) {
                event linefollower:unblocked
              }
    {bumper} state crash
               updateMotorSettings(0, 0);
    default
      <noop>;
```

Figure 6: a program with trace (green) and feature dependency (blue) annotations

In the rest of this section we briefly mention a few other tools (in addition to MPS, which we've illustrated in the case study) that are able to implement this approach.

6.1 Intentional Software Domain Workbench

Charles Simonyi has been working for Microsoft Research on a project called Intentional Programming. His company Intentional Software is now continuing this research and is productizing the system as the Intentional Domain Workbench. Intentional has not published a lot about what they are doing, but a number of things are known based on several publications such as [8].

Like JetBrains, Intentional also uses a projectional approach. It is similar in concept, but quite different in detail. The layouting and rendering engine it more powerful than the one used by MPS. The authors have seen examples where for example circuit diagrams or fraction bars are used as part of (otherwise normal) C programs. Other examples include insurance mathematics mixed with "normal" programs. So the ability to mix and match notations seems to be quite sophisticated.

6.2 SDF, Stratego and Spoofax

These tools are developed by Eelco Visser and his group at the TU Delft. SDF is a way to define grammars and languages, Stratego is a term rewriting tool used for translation, and Spoofax is an IDE framework based on SDF [9].

Traditional parsers use two phases: in phase one a character stream is broken into the tokens defined by the language. In phase two the parser consumes the tokens, checks the token sequence

for conformance to the grammar and builds and AST. Since tokens are defined without any context, ambiguities can arise if grammars are combined that define different tokens for the same sequence of characters

SDF in contrast has no separate tokenization phase. The parser directly consumes the character stream, everything is context-aware. If language modules are combined, there can never be a problem with overlapping token definitions. Language composition is therefore no problem.

Stratego is a term rewriting framework based on SDF. It maps terms (think: tree fragments) of one tree to terms of an output tree. As a consequence of how Stratego is built, it is possible to use the concrete syntax of the source and target languages when defining term rewriting rules: a rewriting rule looks like "text pattern mapping". However, what really happens is that a model to model transformation is executed, where source and target model are written down in their respective concrete syntaxes.

SDF is scalable and can handle non-trivial languages e.g. Java, XML, and HTML have been implemented based on SDF. Additionally a set of languages called WebDSL[12] was created that showcase the idea of using different language modules to address different aspects of developing web applications.

Eelco's group now develops Eclipse-based tooling for SDF and Stratego (Spoofax), providing editor support for building and using SDF-based languages and Stratego-based transformations.

6.3 Eclipse Modeling and Xtext

The Eclipse Modeling Project provides a wide range of tools for developing domain specific languages, generators and transformations. As part of Eclipse Modeling, the TMF Xtext [10] project supports textual domain specific languages. It is easily possible to define textual DSL including the necessary tooling (scanner, parser, model read/write/emit support, syntax highlighting, code completion, and constraint checks, but currently no debugger). Because of the underlying parser technology language modularization and composition is limited - a language can inherit (and reuse and redefine concepts from) one base language. Direct integration with common GPL (e.g. Java, C or C++) is not supported. However, using the so-called *JavaVMMetamodel*, it is easily possible to reference and navigate to Java types in the Eclipse workspace. Several other Eclipse-based textual modeling frameworks are available, examples include TEF, TCS, EMFText . Xtext has created a lot of buzz and is used widely. It is very mature and scales beyond trivial languages. Xtext and TCS are self hosted and used in other Eclipse projects (e.g. B3 builds on Xtext and ATL on TCS).

7. SUMMARY

An environment that unifies "programming" and "modeling", while allowing us to modularize, compose, mix and extend languages at different abstraction levels and with different notations is very promising. Cross-cutting concerns only have to be implemented once and can be handled consistently in all languages. Tool integration issues, and the challenge of integrating models and code will be a thing of the past. The discussion about "domain specific or not" becomes irrelevant, we'll just talk about suitable and not-so-suitable abstractions and notations. If abstractions are not suitable, we can build and integrate additional language abstractions. We can build our own, or use languages from a library. Finally domain experts can be integrated into the process where it makes sense.

We (the authors) really want to work in such a world! Let's make it happen!

8. REFERENCES

[1] MISRA Group, Misra-C, http://www.misra-c2.com/

[2] Jarke. M. , Requirements tracing, Communication of ACM Volume 41, Issue 12, 1998, pages 32 - 36

[3] Eisenecker, U., Czarnecki, K., Generative Programming, Addison-Wesley, 2000

[4] JetBrains, Meta Programming System, http://jetbrains.com/mps

[5] Voelter, M., Solomatov, K., Language Modularization and Composition with Projectional Language Workbenches illustrated w/ MPS, submitted to SLE 2010

[6] Lego, Mindstorms, http://mindstorms.lego.com

[7] Sourceforge.net, nxtOSEK, http://lejos-OSEK.sourceforge.net/

[8] Intentional Software, Intentional Domain Workbench, http://intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf

[9] http://strategoxt.org/Spoofax

[10] http://www.eclipse.org/Xtext/

[11] F.Jouault, J.Bezevin, TCS: a DSL for the specification of textual concrete syntaxes in model engineering, GPCE 2006

[12] http://webdsl.org