Requirements as First-Class Citizens: Integrating Requirements closely with Implementation Artifacts

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

Federico Tomassetti
Politecnico di Torino
Torino, Italy
federico.tomassetti@polito.it

***Abstract*—Requirements often play second fiddle in software development projects. The tools for managing requirements often just support "numbered lists of prose paragraphs", and they don't integrate well with the tools used for implementing the system. This leads to all kinds of challenges in terms of versioning and traceability. Moreover, because they are mainly prose text, they cannot easily be checked for consistency and completeness, limiting their usefulness. In this paper we describe an alternative approach, where requirements are (at least partially) formalized to support consistency checking, where parts of requirements can be used directly as the implementation, and where requirements are managed with the same tools that are used for system development. The approach is illustrated with the mbeddr system, a comprehensive IDE for embedded software development based on an extensible version of C and domain-specific languages.**

I. Introduction

Collecting, organizing and managing requirements is mandatory for life-critical systems, essential for mission-critical ones and it can still be very useful in the development of other kinds of systems. Still, it is a cumbersome activity which either is relentelessly executed with major effort (typically if the customer or a certification standard/agency requires it) or it is mainly overlooked, leading to poorly structured and maintained requirements.

CURRENT STATE In our opinion, one main problem with the traditional ways to collect and maintain requirements is the inadequacy of the supporting tools (see also the study in [5]; we discuss it in Related Work). In some fields, such as aerospace, automotive or telecoms, requirements are often collected and managed using MS Office documents or tools like DOORS, which basically gather paragraphs of text with no or very limited structure. The relation between requirements and other artifacts (i.e., implementation code, tests, etc.) is collected in other documents, or with comments in the code, requiring manual synchronization with the actual system implementation. It is not surprising that, when possible, practitioners try to escape this situation by either using simpler approaches for requirements elucidation (such as CRC cards[1] and user stories) or completely avoiding it. While agile approaches to requirements engineering limit the burden of managing them, they often do not provide a maintenance strategy for requirements; instead they are considered transient artifacts. This is not acceptable in many domains, where standards require a more structured approach to requirements management.

OUR APPROACH The current state of the art can be improved by the seamless integration of requirements engineering concerns into software development tools, developed with language workbenches. This would lead to four main benefits:

- *reuse of well known tooling:* both for editing of requirements and for managing and versioning them. This limits the accidental complexity introduced by requirements engineering.
- *ease of creating requirement traces:* creating links between requirements and artifacts can become much simpler. More importantly, the tool can help to keep traces consistent,
- *ability to query the requirements model:* getting some practical benefits from the requirements not only during the validation phase but also during development. The practitioner (either a developer or a business analyst) will be able to navigate relations between requirements and artifacts and see, for examples, which requirements are related to failing tests or are not connected to implementing artifacts.
- *extensibility:* by using the language modularization and composition features supported by language workbenches, domain-specific extensions to (generic) requirements documents can be seamlessly integrated.

In this paper we illustrate an approach that permits the reduction of the cost of maintaining requirements and helps to better leverage them, with immediate benefits to practicioners. Our approach is based on the mbeddr technology stack, which, in turn, is based on JetBrains MPS (discussed below). This tool stack allows us to scale the level of sophistication of the language used to express requirements to the needs of the project. This way, agile projects can use a basic version of the language to express requirements in

[1] `http://en.wikipedia.org/wiki/CRC_Cards`

a very lightweight form. For more demanding contexts, for example the development of complex embedded systems, the tooling permits to plug-in domain-specific extensions of the requirements language, supporting a more sophisticated approach to requirements management.

MBEDDR mbeddr[2] is an open source project supporting embedded software development based on incremental, modular domain-specific extension of C. It also supports other languages, which is what we exploit in this paper. Figure 1 shows an overview, details are in [3] and [4].

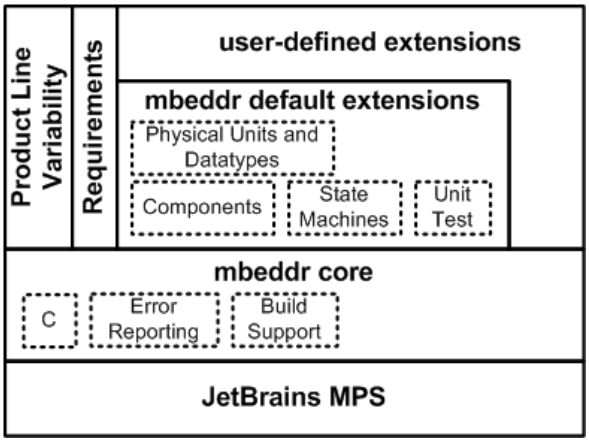

Figure 1. The mbeddr technology stack rests on the MPS language workbench. Above it, the first language layer contains an extensible version of the C programming language plus special support for logging/error reporting and build system integration. On top of that, mbeddr comes with a set of default C extensions plus cross-cutting support for requirements, traceability and product line variability.

mbeddr builds on the JetBrains MPS language workbench[3], a tool that supports the definition, composition and use of general purpose or domain-specific languages. MPS uses a projectional editor, which means that, although a syntax may look textual, it is not represented as a sequence of characters which are transformed into an abstract syntax tree (AST) by a parser. Instead, a user's editing actions lead *directly* to changes in the AST. Projection rules render a concrete syntax *from* the AST. Consequently, MPS supports non-textual notations such as tables, and it also supports unconstrained language composition and extension – no parser ambiguities can ever result from combining languages.

The next layer in mbeddr is an extensible implementation of the C99 programming language in MPS. On top of that, mbeddr ships with a library of reusable extensions relevant to embedded software. As a user writes a program, he can import language extensions from the library them into his program. The main extensions include test cases, interfaces and components, state machines, decision tables and data types with physical units. For many of these extensions, mbeddr provides an integration with static verification tools (model checking state machines, verifying interface contracts or checking decision tables for consistency and completeness; see also [2]).

Finally, mbeddr supports two important aspects of the software engineering process: requirements and product line variability. Both are implemented in a generic way that makes them reusable with any mbeddr-based language. We discuss requirements in detail in the remainder of this paper.

[2]http://mbeddr.com
[3]http://jetbrains.com/mps

II. Challenges and Solutions

In this section we describe a set of current challenges in requirements engineering as well as our approach to solving them in mbeddr.

A. Requirements Versioned with Code

Traditionally, requirements are stored in a tool-specific database. Artifacts are instead typically stored in version control systems (VCS) such as git, SVN or ClearCase. This situation leads to synchronization problems when keeping requirements in sync with the implementation. The natural solution would be to store requirements and implementation artifacts in the same VCS. Since most of today's VCS work with an update-and-merge strategy (as opposed to pessimistic locking), the requirements tool would need to support diff and merge for requirements as well.

In mbeddr, requirements are collected with a special *requirements* language. Each requirement has an ID, a short description, an optional longer prose, a priority and any number of additional attributes. Requirements can also be nested. Figure 2 shows an example.

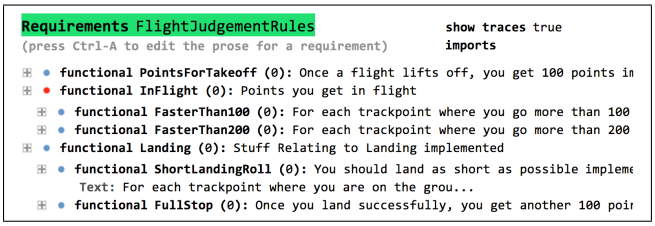

Figure 2. Requirements in mbeddr are arranged as a tree. The colored dots on the left reflect the trace status of a requirement (discussed below).

Importantly, since mbeddr is based on MPS, and MPS comes with a generic XML-based storage, all requirements are stored in XML files, *along with any other implementation artifacts*. MPS also supports diff and merge for any arbitrary language (based on the projected concrete syntax of each particular language), so we get support for diffing and merging requirements for free.

mbeddr's requirements tooling also has an importer to import requirements via XML or CSV files. This way, data migration from traditional requirements management tools is supported.

B. Traceability into Code

When we talk about the integration between code and requirements, we first have to define what we mean by code. In the context of mbeddr, code is any program (or model) expressed with any MPS-based (programming or modeling) language. In particular, C, all extensions of C (default and user-defined) are considered code in the context of this discussion.

The simplest kind of integration between code and requirements is tracing: a program element has a pointer to one or more requirements. Such a trace pointer essentially expresses that this particular element is somehow related to a set of requirements. By using different trace kinds, the nature of "somehow related" can be qualified. Trace kinds typically include `implements` or `tests`.

```
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>
```

Figure 3. A C module with a set of constants that have a trace to a single requirement each. The tracing facility in mbeddr can add traces to any program element expressed in any language.

Figure 3 shows a piece of mbeddr program code. The root element is a `module`, and it has an annotation that specifies to which requirements modules we may want to trace from within that module. We can then add a trace to any program element in that module, tracing to any requirement in the referenced requirements module. There are four important characteristics of this implementation.

1) The requirements trace is not just a comment. It is a well-typed program element that can be used for all kinds of analyses. For example, it is possible to select the requirement, open the Find Usages dialog, and retrieve all program elements that have traces attached to the current requirement.
2) The trace is not an independent program element that is just "geographically close" to the program element it traces. Instead, the trace is a child element of the traced element. This means that, if you move, copy, cut or paste the element, the trace moves with it.
3) Since MPS is a projectional editor, the program can also be shown *without* the traces, if the user so desires. The traces are still there and can be turned back on again at any time.
4) The tracing facility is *completely independent* of the traced language. Program elements defined in any (MPS-based) language can be traced. Users can define new languages, and the tracing mechanism will work with the new language automatically.

While our tracing framework cannot remove the burden of users to manually establish and maintain the traces according to the actual relationship between the code and the requirements, the approach does solve all technical challenges in providing universally-applicable tracing support. However, the fact that referential integrity is automatically checked and that arbitrary analyses can be built on top of the program/requirement/trace data, can be used to ease the work of the developer: requirments and traces are "real code", and not just second-class meta data.

C. From Requirements to a Functional Architecture

Many projects start out by collecting requirements in a tool such as DOORS, or in other prose-based "databases" like the one discussed for mbeddr above. However, using prose only, it is very hard to keep things consistent – after all there is no type checker or compiler for prose text. One problem in this context is the definition of (functional) components, their responsibilities and their collaborations with other components, which express the high-level, functional structuring of the to-be-built system. One way to get to such components is to play through collaboration scenarios. From these scenarios we can derive which data a components owns, which other components it collaborates with, and which services one component uses from another component as part of such collaborations.

However, if we do this only with pen and paper (cf. CRC cards), it can be tough to keep things consistent (this is the prose-only problem in a different guise). At some point, we have to become (somewhat) more formal.

In mbeddr, this is realized as follows. Requirements, as introduced above, have a requirement kind (such as `functional`, `operational` or `usability`). Requirements also have additional data. Since MPS supports arbitrary language extension and composition, it is possible to define additional DSLs that can be plugged into a requirement. To express the functional architecture, we have defined three new requirement kinds: `actor` (an actor outside the system boundary), `component` (a functional building block of the to-be-built system) and `scenario` (an examplary collaboration scenario between actors and components, not unlike sequence diagrams).

```
component Nuller (0): nullifies the altitute
component Interpolator (0): averages over the flights
component InMemoryStore (0): stores flights in memory
component InMemoryStore {
  data Flights
  capability store(data aFlight):
  capability setup(): data SetupStatus
}
```

Figure 4. A functional component. It owns a piece of data and provides two capabilities. It does not collaborate with any other component.

Figure 4 shows an example of a functional component. Note how it lives inside a requirement, even though the original

requirements language was not invasively changed. Figure 5 shows a scenario, expressed with a textual language, with all the usual IDE support. In particular, if component A uses a capability ("calls an operation on") a component B, and B is not defined as a collaborator of A, this results in an error in the IDE. A quick fix can then add B as a collaborator for A. Silimarly, one can only use capabilities that are actually defined on the components. Finally, arguments to capabilities can only be taken from the data that is owned by the client component, or has been received from another capability call during that same scenario. As a consequence, after defining a set of scenarios, the components accumulate data, capabilities and collaborators that are necessary to execute the scenarios: a functional architecture arises, "enforced" by the underlying language and its constraints.

```
scenario StoreFlight (0): A flight is stored in the store
scenario StoreFlight
  Driver {
    == Setup ==
    -> InMemoryStore.setup(): SetupStatus
    alt setup status ok? {
      == Operation ==
      # Any number of store calls possible now
      -> InMemoryStore.store(SomeFlightData)
      -> InMemoryStore.store(SomeFlightData)
      else in case setup failed {
        ! system startup failed
  } } }
```

Figure 5. A scenario that describes exemplary interactions between collaborating components (and external actors).

VISUALIZATION Scenarios are reminiscent of sequence diagrams, so they can also be visualized this way (see Figure 6). To make this possible, we have integrated PlantUML[4] into mbeddr – the diagram is rendered directly in the tool, and a double-click on a diagram element selects the underlying element in the editor. Additional diagrams show the components, their data items and capabilities and the collaborations. Another language extension supports defining use cases, and use case diagrams can be rendered from these. The use case attributes are extensible.

EXTENSIBILITY This language for expressing the functional architecture does not have expressions, sophisticated data types or a type checker. At this level of abstraction, these would be distractions – the goal of this language is the allocation of data, responsibilities and collaborations to high-level functional building blocks of an application.

However, the language is extensible: new entities (in addition to components or actors) can be defined; components can own additional things (in addition to data items and capabilities) and scenarios can contain additional steps (in addition to capability calls, headings, or alternatives). For example, a component may contain a wireframe mockup (which would have to be drawn outside of MPS) to represent

[4]http::://plantuml.sourceforge.net

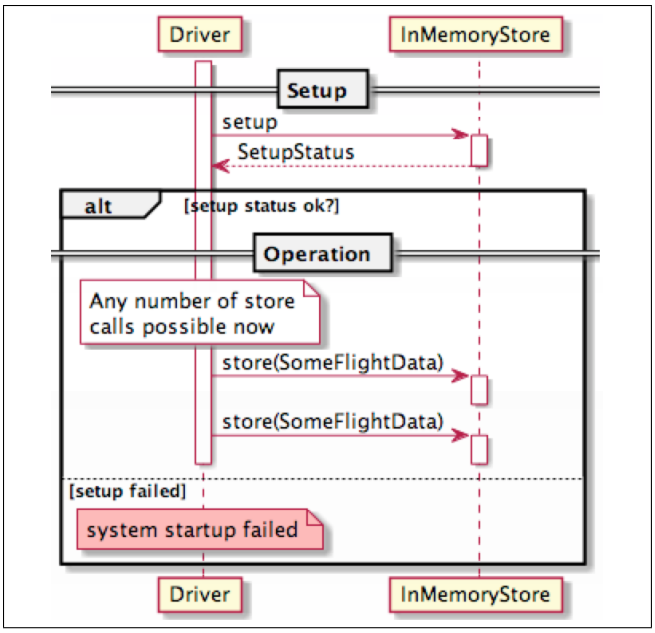

Figure 6. A scenario that describes exemplary interactions between collaborating components (and external actors).

UI aspects. It is also possible to add additional properties and then check constraints based on these. For example, components may be allocated to layers, and constraints can be used to check whether collaborations and capability usages respect layer constraints (e.g., you can call from the `business` layer into the `persistence` layer, but not vice versa). These additional data and constraints can be added without invasively changing the basic scenario language, and can also be added *after* the initial set of components and scenarios have been defined, supporting incremental refinement of the language as we incrementally refine our understanding of the system. For example, systems engineers may first define the components and the scenarios. Then, in a second step, software architects may add the layer markup and the associated constraints, and then, if some of the constraints fail, split up or reallocate components to make them fit with the layer structure. Refactorings can be added to make such changes to the component structure simpler.

D. Tracing into other Artifacts

In many projects, requirements are not the last step before coding, and the functional architecture discussed in the previous section is too simplistic to describe the functionality of the system. Instead, other artifacts are developed, including system engineering models, function models or physical models. Often these models are built with tools such as Matlab/Simulink[5] or Modelica[6], or use formalisms such as EAST-ADL[7]. It is usually not possible to automatically derive software artifacts from such models, since they are

[5]http://www.mathworks.com
[6]https://www.modelica.org
[7]http://www.east-adl.info/

too abstract. However, as software artifacts are developed, it is necessary to relate the software artifacts to these models.

To make this possible, mbeddr's tracing framework is extensible: Other artifacts can be used as requirements targets as well (as long as the respective language constructs implement an mbeddr-provided interface). This way, arbitrary descriptions or models (such as system models, functional models or component models) can be traced to. By adding an import facility, models created with other engineering tools can be integrated reasonably well with mbeddr-based artifacts. For example, we are currently implementing an importer for Matlab/Simulink models to support tracing to simulink blocks from mbeddr program nodes.

E. Formal Business Logic in Requirements

The previous two subsections have addressed the challenge of becoming "more formal" with the goal of narrowing down the functional architecture of a system. Another way of getting incrementally closer to the implementation is to embed important parts of the business logic into requirements, and then use those in the implementation code.

```
functional PointsFactor (0): The factor of points
constant int8 BASE_POINTS = 10
functional InFlightPoints (0): Points you get for each trackpoint
calculation PointForATrackpoint (int8): Points for each Trackpoints
    params int16 alt: current altitude of the trackpoint
           int16 speed: current speed of the trackpoint
  = BASE_POINTS * 0
```

	alt > 2000	alt > 1000
speed > 180	30	15
speed > 130	10	20

```
tests: PointForATrackpoint(500, 100) == 0
       PointForATrackpoint(500, 1200) == 0
       PointForATrackpoint(1100, 140) == 200
       PointForATrackpoint(2100, 140) == 110
```

Error: failed; expected 110, but was 100

Figure 7. A calculation is a function embedded into a requirement. They include test cases that allow "business people" to play with the calculations. An interpreter evaluates tests directly in the IDE for quick turnaround.

Figure 7 shows two requirements. The first one defines a constant `BASE_POINTS` with the type `int8` and the value `10`. The second requirement defines a calculation `PointsForATrackpoint`. A calculation has a name, a list of parameters, and a result expression, which, in this case, uses decision table (a two-dimensional representation of nested `if`-statements[8]). The calculation also references the `BASE_POINTS` constant. Using constants and calculations, business users can formally specify some important business data and rules, while not having to deal with the actual implementation of the overall system. To help with getting these data and rules correct, calculations also include test cases. These are evaluated directly in the IDE, using an interpreter: users can directly "play" with the calculations.

CONNECTING TO CODE If the constants and calculations that business users specify in the requirements were *only* used in requirements, this would be only partially useful. In the end, these calculations should make their way into the code directly, without manual re-coding.

```
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
```

Figure 8. Implementation code can directly call calculation functions defined in requirements. In this case, a calculation is called from a component, expressed in the mbeddr's components C extension.

Figure 8 shows a component, expressed in mbeddr's component extension to C. Inside the component we directly invoke a calculation (the green code), using function call syntax. When this code is translated to C, the expression in the calculation is translated into C and inlined.

The constant and the calculation are just examples of possible "plug in" languages into mbeddr's requirements system. Any DSL, using a wide range of business user-friendly notations, can be plugged in and made available to C-based implementations.

III. Discussion

The tooling described in this paper solves some important challenges in requirements engineering. However, it is not a complete solution (yet). For example, some contexts require information security or multi-client capability for the requirements. This is currently not addressed. Also, it is assumed that all artifacts reside in MPS, which limits the applicability of the approach. However, it is our opinion – and the core message of this paper – that *any* engineering tool should *always* be based on a language workbench like MPS that supports approaches like the one discussed in this paper. Another limitation is that no integration with current trends such as OSLC[9] (Open Services for Lifecycle Collaboration) is provided. Finally, this paper only addresses the overall paradigm and tooling, it does not discuss a methodology for requirements management. In our opinion these two concerns are largely orthogonal: once a tool is as powerful and extensible as mbeddr/MPS, it can be adapted to many differnet methodologies or processes.

It could be argued that the high-level components and scenarios discussed in Section II-C, as well as the formal business logic discussed in Section II-E are not requirements anymore, but rather architecture or design. However, we think that this distinction is arbitrary and not very helpful, especially in the context of a tool such as the one described in this paper: there has to be some consistent and integrated

[8]Projectional editors like MPS can deal with non-textual notations such as tables, vectors, matrices or fraction bars or "big sum" symbols.

[9]`http://open-services.net/`

path from prose requirements to the implementation code. The tooling discussed in this paper provides such a path. It is not important at which point is this continuum we stop calling the activity "requirements engineering".

IV. Related Work

As mentioned in the introduction, Winkler and von Pilgrim [5] performed a literature review on traceability, considering it both for MDD and requirements. They conclude that tracing is rarely used in practice and the most prominent problem leading to this is the lack of proper tool support. Our approach provides a possible solution to this dilemma and could therefore contribute to helping practitioners in adopting requirements traceability, particularly, in contexts where the process requires it.

DSLs have traditionally not seen much use in requirements engineering, they are typically associated more with the implementation phase or with software architecture. However, as we demonstrate in this paper, DSLs, especially extensible DSLs, can be very useful in requirements engineering. Other tools, for example, itemis' Yakindu Requirements[10] also implement this idea: it also uses (mostly) textual DSLs plus visualization. In contrast to our approach, however, extensibility is more limited, since the underlying language workbench (Eclipse Xtext[11]) supports only limited forms of language extension.

Favaro et al. [1] present an approach to requirements engineering that has some commonalities with ours. Like us, they have the goal to introduce structured, model-based requirements. Their approach relies on the use of a wiki enriched by semantic links, and they also provide a requirements browser inside the IDE (Eclipse) supporting some navigation capabilities from the requirement to the artifact (but not vice versa). They underline two points with which we strongly agree: a) the importance of having an adaptable mechanism for requirements, depending not only on the nature of the project but also on the kind of the requirement, with a lighter process for "non-technical" requirements; b) the fact that requirements and implementation artifacts are intrinsically integrated. We think, however, that our approach offers: a) better integration between requirements and artifacts, and b) the possibility to have both a flexible approach but also specific IDE support for any particular kind of formal language embedded into the requirements (thanks to the projectional editor).

V. Future Work

There are three main areas for future work. First, we will add reporting functionalilty, targetting requirements documents in HTML and Latex. The reports will include the diagrams, as well as trace reports. Another target is Excel, which is often the preferred way to get "numbers" by management. Second, a colleague of ours is currently working on an MPS editor component that supports mixing free text (with text-like editing support) and instances of language concepts. Integrating this editor with the requirements management tooling discussed in this paper will be extremely useful: for example, one could reference other requirements from within the prose description of a particular requirement, while making sure that this reference would take part in refactorings.

[10] `http://www.yakindu.de/requirements/`

[11] `http://eclipse.org/xtext`

VI. Summary

mbeddr's core idea is discussed in [3]: building domain-specific tools is not just about adapting a *tool* to a particular domain (windows, buttons, tool bars). It is rather more important to adapt to the domain the languages, formalisms and data formats that underlie the tool. If you do this based on a language workbench, you get the *tool* adaptation essentially for free. This is because the actual tool, JetBrains MPS, is essentially a very powerful editor for any kind of language. By adapting the language, you get the adapted tool automatically. In this paper, we have demonstrated this idea for requirements management. All the benefits discussed in this paper involve only *language* engineering. No *tool* aspects have been customized.

ACKNOWLEDGEMENTS We thank the mbeddr and MPS development teams for creating an incredibly powerful platform that can easily accomodate the features described in this paper. We would also like to thank Christoph Becker for making us aware of PlantUML and inspiring the scenario extension to requirements. We also want to thank Andreas Graf and Nora Ludewig for their feedback to the paper.

References

[1] J. Favaro, H.-P. de Koning, R. Schreiner, and X. Olive. Next generation requirements engineering. In *Proc. 22nd Annual INCOSE International Symposium (Rome, Italy, July 2012)*, 2012.

[2] D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb. Language Engineering as Enabler for Incrementally Defined Formal Analyses. In *FORMSERA'12*, 2012.

[3] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Journal of Automated Software Engineering*, 2013.

[4] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.

[5] S. Winkler and J. Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9:529–565, 2010.