Return to the Language Forrest: the Case for DSL Oriented Software Engineering

András Vajda  
Ericsson Software Research, Finland  
andras.vajda@ericsson.com

Johan Eker  
Ericsson Research, Sweden  
johan.eker@ericsson.com

ABSTRACT
In this position paper we argue for a radical departure from today’s approach to software engineering, centered on target-specific tools and general processes such as modeling and agile practices. The old principle of right tool for each problem shall be applied to software engineering as well: to address ever higher complexity, we need to raise dramatically the level of abstraction; as general solutions all largely failed, we need to focus on problem domain specific approaches. Instead of focusing on the platform paradigm (von Neumann and derived architectures), we need to focus on each of the problem domains, by creating a language or modeling environment specific to each domain that can be used by domain experts with very limited software or hardware competence, while putting the platform mapping in the target-specific compiler, developed by platform experts. This approach to software engineering will require a new approach to ways of working as well as research into new technologies.

Categories and Subject Descriptors
D.3.2 [Programming Languages]: Language Classifications – specialized application languages, very high level languages.

General Terms
Design, Languages, Theory.

Keywords
Domain specific languages, software engineering methods.

1. INTRODUCTION: STATE OF THE PROBLEM
There have always been two dimensions to software development: one was the platform (HW and OS) dimension and the other was the problem domain dimension. For most of the past fifty years, software engineering focused almost exclusively on the first one, creating languages that allowed description of programs in a way that was easily translated to Neumann types of (sequential) machines. The problem domain, whatever it was, had to be squeezed into this paradigm.

During the 80s it became painfully clear that this approach is a big bottleneck, so the community attempted to raise the abstraction for available high level assembler languages (C, Fortran etc), pushing these towards more genericity and somewhat higher level of abstraction (e.g. object orientation), while leaving some of the obvious tasks for the tool/run-time system domain to handle. The underlying assumption remained though the same: follow the platform – any attempt to raise abstraction level in general led to inefficiency when deployed on HW.

While this fixed some of the issues around actually writing code, the rift between domain knowledge and platform knowledge remained wide open – which led to the invention of UML, agile ways of working etc. These methods masked some of the discrepancies but didn’t fix the real problem, as the community hit the same wall: generic, high level of abstraction meant inefficient code and the issue of communication between domain and platform experts was still not fully resolved. Graphics were just aiding understanding – shortening getting up to speed times – but weren’t giving the order of magnitude boost that was hoped for.

Another important development recently was the emergence of multi-core systems, as the hardware industry’s solution for increasing the performance of processors. Software engineering for single core systems is dominated by the imperative style, which quickly becomes the bottleneck when moving applications to multi-core processors: parallelism in programs written in imperative style is always explicit and achieved by means of threads and processes, defined and carefully implemented by the programmer; any potential for automatic parallelization is prevented by the very tools used to implement the algorithms. The control over low-level detail, once considered a merit, tends to over-specify programs: not only the algorithms are specified, but also how inherently parallel computations are sequenced. Hence we believe successful utilization of many-core hardware will require new paradigms, tools and languages. In [5] the authors explicitly state that speed-up of legacy software should not even be a measure of success for research on parallel computing. What would constitute success, is tools that would help us build applications that scale with the available parallelism on the target system and that are portable between various target systems.

2. DOMAIN SPECIFIC APPROACH
The key insight of the past few years was that while generic, high abstraction level, Neumann-paradigm focused design leads to inefficiency in deployed code and does not really bridge the gap
between domain expertise and programming expertise, there may be a middle way that could mitigate these tensions: what cannot be solved in generic terms, becomes much simpler if you narrow the domain down so that the range of choices is greatly reduced.

Therefore, we believe the community shall focus on specific, limited, well-defined domains (for example, DSP programming, communication stack development etc), create a language/modeling environment / infrastructure that allows domain experts to express what the software shall do (not how) and then focus on making the transformation within this limited, well-defined domain to efficient software executable on Neumann architectures automatic. The key is being restricted to a well defined domain: efficient automatic transformations become possible and the result is an eco-system where you design in the domain level – what, not how: possible by domain experts – and then can generate efficient target code through specific, limited, targeted transformations (created by platform experts). Figure 1 illustrates this flow on a high level.

![Figure 1 Relationship between domain and platform specific engineering](image)

3. LANGUAGE AND COMPILER LINES

The fundamental principle we promote is the right domain specific tool for each problem, instead of some universal tool coupled with a way of working that tries to wrap it so that it becomes usable in various contexts. The primary meta-tool we promote is usage of high level, strictly domain specific languages, based on formal concepts used and widely understood by domain experts who may have limited or no software engineering knowledge.

By ways of example, consider chess: there’s a very well defined language to describe moves well understood by any chess player, but hardly usable to describe e.g. the architecture of a building. Since the language is well defined, it’s easy to create a translation of it into e.g. displaying moves on a monitor, instructing a robot to make the right moves etc. How this is done is irrelevant and hidden from the chess expert – all he needs to know is the language itself – coupled, obviously, with her chess competence.

Such an approach will inevitably lead to a proliferation of languages and domain specific compilers, potentially even within the same domain. The challenges raised by such an approach are twofold: first, how to manage the ecosystem of a large number of languages; second, how to mitigate the threat of one language for each product syndrome.

There have been recently proposals on creating language or modeling workbenches (such as [1], [2], [3]) that can provide a unified meta-environment in which to create new language or modeling environments as well as supporting compilers. Irrespective of how it will be achieved, we believe that the answer is to create language lines, in analogy to product lines: groups of languages that share the same foundations and exhibit controlled variations to accommodate specific needs of specific domains. Language and modeling workbenches are one way to achieve this; embedding in host languages – such as Haskell – is another.

A related problem is the problem of transforming domain specific applications written in domain specific languages or modeling environments into target specific, efficient software (expressed in assembler or traditional imperative languages). As opposed to the language lines, compilers for domain specific languages / modeling environments need to mitigate two constraints: support domain specific constructs and target specific, architectural features. While on the language side there’s one dimension – domain type – for compilers there will be two: domain type and target type; creating separate compilers for each target and each domain is economically hard to sustain – hence a different approach is needed.

We believe there are two key technologies that will help us manage the complexity of a compiler forest: first, finding a commonly usable abstraction model to which programs written in all or most domain specific languages can be transformed; second, creating compiler product lines, that can efficiently transform programs represented in the abstraction model into quality target code. Compiler product lines are likely to center on specific targets with variability points for domain specific constructs.

The grand challenge of this model will be finding the right abstraction model. As the work of Intentional Software [2] or MetaCase [3] has shown, such abstract models and repositories can indeed be designed – but lot of issues remain to be addressed, related primarily to handling specific variability for different problem domains. Host languages provide another interesting alternative that is yet to be thoroughly explored.

Figure 2 summarizes our view: the concepts of language/modeling workbenches and host languages, common abstract models and compiler product lines.

![Figure 2 Basic concepts](image)
4. **LOSE: LANGUAGE ORIENTED SOFTWARE ENGINEERING**

Traditionally, there has been a pretty sharp divide between domain experts and software experts: domain experts are supposed to specify to as great detail as possible – in one shot or iteratively – what needs to be supported; software experts are expected to design, implement and test these requirements as faithfully as possible – providing the how component of software system design.

The key problems we are struggling with are related to the ever-increasing complexity of target systems that put too much emphasis on target specific issues as opposed to domain specific problems and the extreme difficulty to comply with the requirement of as faithfully as possible. This difficulty is primarily related to the handover that has to happen between domain experts and software experts ‘supported’ by the inexact nature of human communication.

Language oriented software engineering (LOSE) aims at tackling both of these problems by putting the responsibility of designing the system in the hands of domain experts, hence removing the problem area of requirement handover; inherently, requiring domain experts to design systems mandates the usage of very high level languages and design environments in order to avoid the need for having non-software experts deal with software and hardware specific details.

Language oriented programming has been proposed before (see [7]). However, the focus was still on software engineers and programmers – we believe there’s a need to put the right tools in the hands of domain experts, whereas software experts will focus on the translation techniques only.

![Figure 3 Steps in Language Oriented Software Engineering](image)

The phases of software engineering will be markedly different in LOSE, illustrated in **Figure 3**. The first step is to define the problem domain and the concepts and constructs that are needed to express solutions for the specific domain. Essentially, this step replaces from the software engineer’s point of view, the requirement management phase, as it defines what problem domain rather than requirements needs to be supported.

The second step is the selection or the definition and implementation of a domain specific language that can support the concepts and constructs defined in the first step. This work is highly interactive between software and domain experts and by its own nature is condemned to succeed: either the language will meet the requirements or it will be unusable which will be found out as soon as prototypes are being implemented.

Once the domain specific language definition is sufficient for the task at hand, the software and domain experts can continue working largely independently of each other: domain experts will focus on the implementation of the system itself – which may require multiple iterations on the language itself – while software experts will focus on selecting the right target platform and creating the compiler, debug and profiling infrastructure for the domain specific language. There is no need for software experts to understand the problem itself – what is needed is a precise definition of the expected behavior of the concepts and constructs in order to be able to build efficient transformations towards the target environment. This step may go through multiple iterations, as the need for more or different constructs is discovered.

The final step, as soon as the system itself and the compiler are in place, is the automatic generation of target code, profiling, which may result in potential refinement of both source code and compiler in order to reach the desired performance characteristics.

LOSE has several specific characteristics: it shifts away from the software experts the task of designing the actual system towards the domain experts, allowing the software experts to focus primarily on target specific issues rather than the problem domain. In many ways, it is the implementation of the work division principle in software engineering: allow everyone to focus on their best skills, provided with the right tools. This is a significant departure from today’s approach of having software experts deal with a significant amount of problem domain issues.

There’s an interesting connection between product lines and language-oriented software engineering. Software product lines are characteristic to specific problem domains; as the goal of LOSE is to create domain specific high level development environments usable by domain experts, LOSE actually creates the domain specific infrastructure for software product lines, providing a common foundation based on which multiple products within the same product line can be built. It’s a different approach from traditional software product line engineering: it emphasizes the domain specific tools needed to create the product line, as foundation for the components that will actually contribute to the family of products.

5. **RESEARCH CHALLENGES**

Language oriented software engineering raises a number of research challenges that need to be addressed in order to fully exploit the benefits of such an approach.

First, we need to define and develop reliable workbenches, frameworks and/or host languages that enable fast, reliable and easy definition of new domain specific languages or modeling environments with a backend infrastructure that can support a formal representation of programs written using these languages. There are several attempts in this direction, but scaling such approaches to large scale, distributed development and – equally important – making it easy to use by non-software experts is a challenge that still needs to be addressed.

The compilation infrastructure is of equal importance. We need to understand how we can create large families of language and model compilers that can deal with potentially thousands of domain specific variations and large numbers of target platforms. Support for automatic verification of consistency, validation on
the domain level etc all need to be revisited and expanded. Support for automatic transformation into software that can execute on modern multi- and many-core hardware architectures is a challenge in its own right.

Beside the technology aspects, we need to reshape the way we develop systems. LOSE is a radical departure from today’s approaches focusing on the software engineer, with domain experts playing just a supporting role – we have to accommodate two types of main actors, with different visions on the problem, solution and ways to achieve this. Traditional cycles of development are not valid anymore, the task division happens along different lines (platform-specific versus domain-specific).

6. EXISTING EXAMPLES
There are several well-established domain specific languages that have shown great potential in delivering higher abstraction level without compromising on performance: examples include SQL, Erlang (specifically developed for designing massively parallel telecommunications systems but has shown excellent performance and efficiency in various settings, e.g. Web 2.0 applications); the activities at Stanford's Pervasive Parallelism Laboratory, aiming at creating a DSL framework for parallel targets based on the SCALA language, have also delivered efficient DSLs for various domains.

In this chapter we highlight two existing examples of domain specific high level languages where the authors of this paper have been involved and which have shown good productivity gains, even when used by domain experts rather than software experts.

6.1 Feldspar: Functional Embedded Language for DSP and Parallelism
Feldspar (available open source at http://feldspar.sourceforge.net ) is an embedded functional language targeting DSP algorithm design. It is made up of two layers: the high level, abstract layer is where domain experts operate, using familiar constructs such as vectors, filters, bit manipulations; however, what they actually produce are code generators that generate an internal, functional, format suitable for target-specific compilation. The representation in this intermediate format is transformed into target specific C code, which exhibits comparable behavior with handcrafted C code (4).

6.2 CAL: An Actor Language for Parallelism
The CAL actor language is a data flow language designed for streaming applications. It is part of the MPEG/ISO RVC standard as the means for specifying decoders. By describing the applications as a dataflow network the available parallelism is explicitly exposed. The details of relating the application parallelism to the target hardware is not part of the actual application, which only specifies how data flow, not where and when the execution occurs. CAL has been used for a number of applications showing a dramatic increase in development time. In [6] an MPEG 4 decoder is implemented in CAL for FPGA achieving result that was both faster and smaller (both in silicon area and lines of code) and developed in quarter of the time compared to hand coded VHDL; the higher abstraction level allowed the authors to work faster and make more radical architecture modification and more easily trying out different solutions, in the end leading to a more efficient implementation. Ongoing work at Ericsson Research using CAL for multimedia applications shows good scale-up with the number of cores, without the application written with particular targets in mind.

7. SUMMARY
In this paper we argue for a novel approach to engineering software systems, where the task of designing the actual system is put in the hands of domain experts with no or limited software engineering expertise, while software experts’ role is transformed into providing the needed tools, transformation infrastructure and target support, rather than developing the system itself. We believe the main tool is domain specific, high level, intuitive, potentially graphical languages, hence we call this approach LOSE (Language Oriented Software Engineering).

8. REFERENCES