Meta-Level Reuse for Mastering Domain Specialization

Stefan Naujokat[1](✉), Johannes Neubauer[1], Tiziana Margaria[2], and Bernhard Steffen[1]

[1] Chair for Programming Systems, TU Dortmund University, Dortmund, Germany
{stefan.naujokat,johannes.neubauer,steffen}@cs.tu-dortmund.de
[2] Chair of Software Systems, University of Limerick and Lero, The Irish Software Research Centre, Limerick, Ireland
tiziana.margaria@lero.ie

Abstract. We reflect on the distinction between modeling and programming in terms of WHAT and HOW and emphasize the importance of perspectives: what is a model (a WHAT) for the one, may well be a program (a HOW) for the other. In fact, attempts to pinpoint technical criteria like executability or abstraction for clearly separating modeling from programming seem not to survive modern technical developments. Rather, the underlying conceptual cores continuously converge. What remains is the distinction of WHAT and HOW separating true purpose from its realization, i.e. providing the possibility of formulating the primary intent without being forced to over-specify. We argue that no unified general-purpose language can adequately support this distinction in general, and propose a meta-level framework for mastering the wealth of required domain-specific languages in a bootstrapping fashion.

Keywords: Simplicity · Abstract tool specification · Full code generation · Metamodeling · Domain-specific tools · Hierarchy · Service-orientation · Modularity

1 Motivation and Background

At a conceptual level, modeling and programming can be regarded as two sides of the same medal: the WHAT and the HOW descriptions of a certain artefact. This duality of WHAT and HOW has a long tradition in engineering, where models were built to predict certain WHATs, like the aerodynamics of an envisioned car or its visual appearance, in order to optimize vital aspects, before entering the costly HOW-driven production phase, where modifications become extremely expensive. Because of their purpose-specific nature, there are usually many WHAT descriptions that together describe one artefact with only one HOW description.

In classical engineering, there is typically a very clear and agreed upon distinction between a model (a WHAT) and an implementation (the HOW), frequently connected to distinct abstraction layers and different natures of the respective description means. For example, in hardware design there are agreed

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part II, LNCS 9953, pp. 218–237, 2016.
DOI: 10.1007/978-3-319-47169-3_16

and standardized abstractions in terms of chip layout, transistor level, gate level, register transfer level, etc. This clarity of distinction is, however, lost in computer science, where the viewpoints changed quite a bit over time: 60 years ago, assembler was considered a WHAT for the HOW descriptions at the processor's instruction set level and this has been considered in turn a WHAT for lower levels. Assembler then became itself the HOW for WHAT descriptions in terms of 'higher' programming languages like Fortran and ALGOL and so on. In fact, the understanding of what is a HOW (an implementation or a program) and what a WHAT (a model or a specification) in software becomes quite situation dependent. This distinction hinges on the purpose as well as the community, and it steadily changes over time. How can it be that the same language, e.g. ML by Robin Milner [19], was designed as a modeling language[1] and later on considered a programming language by its inventor? In fact, today few will remember that ML was originally not intended to be a programming language!

This development suggests that this phenomenon has to do with a certain understanding of maturity: a modeling language becomes a programming language as soon as one can 'program' with it. This self-referential definition requires a convention of what it means to program, or what is the 'essence' of programming. Are, for instance, executable specifications (models) already programs, or do we have certain performance requirements to the 'program' or 'program-like' artefacts? In the ML case, the growing quality of the ML compiler was certainly important for its change of status. However, we are very distant from reaching a global agreement about the distinction between modeling and programming. For example, many would consider writing a class diagram in UML as modeling, whereas they would consider the same as programming if done directly in Java.

Independently of this discussion, there is no doubt that modeling and programming converge at the conceptual level [7]. A lot of concepts and techniques have been transferred between them, making e.g. modeling languages executable or adding powerful concepts of abstraction to the programming level. They have also been sharing numerous concerns for a long time, like modularity, comprehensibility, variability, or versioning. Further on, in numerous scenarios unifying efforts in terms of integrating features into a single language seems to have the intended effect. For example, general-purpose languages like *Kotlin*[2] and frameworks like *GWT*[3] (Google Web Toolkit) offer to transpile to JavaScript (another general-purpose language), in order to lower the overall complexity of the software stack as well as the learning curve for developers. However, this does not (necessarily) mean that there is a convergence in the direction of a concrete universal language. In most scenarios, integrating additional abstractions into programming languages via *internal DSLs* [17], libraries, or frameworks seems to be at first sight a good tradeoff. It leads however to WHAT-descriptions embedded in a more or less hidden fashion in the syntax of HOW-descriptions in the universal host language. Another path is to add more and more *native language*

[1] Originally ML was designed to describe proof tactics of the LCF theorem prover [34].
[2] http://kotlinlang.org.
[3] http://www.gwtproject.org.

constructs, resulting in increasingly complex *multi-paradigm languages*. Such heterogeneity makes it hard to reason about their WHAT, and raises the knowledge bar for developers: adopters must now learn a multi-paradigm general-purpose language, the semantics of internal DSLs and *APIs* (the WHAT), and how to use them in the host language (the HOW).

Due to these observations, we do not believe in a single universal language, and consider rather the opposite path as viable: in addition to powerful general-purpose languages, there will be a plethora of domain-specific programming and modeling languages, all conceptually based on a *growing common conceptual core*. Accordingly, we envision development frameworks that allow one to master the inherent diversity of modeling and programming, and support the conceptual common core via meta-level functionality. The common core and the sharing establish a new kind of invariants (called Archimedean Points in [51]) spanning whole landscapes of domain-specific modeling languages and tools. The envisioned metamodeling-based software development paradigm aims at simplifying the adopter's experience by strongly exploiting domain-specific characteristics after a rapid model-driven development of the corresponding domain-specific modeling tools. The success of this approach depends on the ease of this development process, which is envisioned to already pay off even for one time use via meta-level reuse.

In the following, we will first argue in Sect. 2 whether a universal language comprising both modeling and programming is desirable. Then Sect. 3 discusses domain-specific modeling and sketches an approach aimed at mastering or even exploiting heterogeneity. Key to this approach is continuous improvement in a bootstrapping-like modeling style, where generated artefacts are fed back into the generation framework itself, and thereby enable a new level of reuse. We then present in Sect. 4 how to achieve this meta-level reuse using the CINCO Meta Tooling Suite. The paper closes with our conclusions and some concrete proposal for future work.

2 Inherent Limitations of Universality

History provides some evidence supporting both the doubts concerning unified approaches to programming and modeling, and the hopes concerning the usefulness of meta-level approaches to master a growing landscape of domain-specific solutions in a unified and holistic fashion.

Originally, the formal methods community started developing universal proposals for modeling and specification languages, to serve as a means for documentation and manual reasoning. Later on such languages became a target of automated analysis tools. In the nineties, Pierre Wolper coined the term *strong formal methods* to classify this new tool-oriented direction [53]. Whereas Wolper and others focused on behavioral models and technologies like model checking [11], the software community originally elaborated on Entity/Relation models so successful in the database community [10] and thereby on static aspects of software. This resulted, in particular, in the static diagrams core of UML, the arguably most popular modeling landscape, which claims to comprise or even unify essentially all aspects of software. In particular, UML covers also behaviors, typically modeled in terms of state diagrams, activity diagrams or message sequence charts [43].

The formal methods community embraced the challenges posed by UML and provided various approaches to its semantic foundation [13], consistency checking [41], and (partial) code generation [48]. Despite all these efforts, the common usage of UML and most of its impact concerns static models: they provide the basis for generating code stubs to be subsequently manually refined. They also provide the foundation for the EMF [52] and MOF [40] metamodeling frameworks. UML therefore clearly establishes a level of description and modeling above the programming level, and requires modelers to pay special attention to keep models and the corresponding programs aligned and consistent. Most popular here is the round trip engineering approach [50], which, however, hardly lives up to its promises, especially when including behavioral aspects and not only classes and packages, and therefore found only marginal attention in practice [16].

In practice, accordingly, UML seems far from being a good candidate for unifying modeling and programming. This impression is also supported by its conceptual heterogeneity which clearly indicates that the intended meaning of ‘unified’ in its name is ‘comprehensive’ rather than ‘holistic’ or ‘consistent’.

The remainder of this section sketches recent developments for enhancing the classical concept of programming language in order to provide a background for the subsequent discussion of inherent limitations of universality.

2.1 Extensions, Internal DSLs, Libraries, and Frameworks

Over the last decades significant effort was poured into integrating complex functionality into general-purpose languages via language extensions, internal DSLs, libraries, and frameworks. Prominent examples are *graphical user interface* (GUI) frameworks, *Java EE* (Java Enterprise Edition) and application servers, the *Document Object Model* (DOM) for representing *Extended Markup Language* (XML) documents in memory, and *object relational mapper* (ORM). These approaches offer powerful abstractions and in some cases even seem to be valid WHAT descriptions. In the following we will take a deeper look at examples of the different variants and their pros and cons.

The most invasive approach is to extend the syntax of a language to comprise another (domain-specific) language. Scala, e.g., allows writing XML code directly in the Scala code: the Scala language designer assumed that XML would be the long term standard way to represent structured data. However, the growing significance of *JavaScript Object Notation* (JSON) challenges this early decision. A major disadvantage of such hardwired language extensions is that it is hard to change them. Because universal languages should have a long life cycle, they must give hard guarantees regarding backward compatibility also in the long term. If removing support for XML at some point is unlikely, since it would break existing code, one could instead add more and more language extensions at need. However, this universality path leads to an overly complex and hard to learn language, and makes its compiler and auxiliary tools harder to maintain.

The least invasive approach is to integrate text blobs of a domain-specific language (in form of string literals or files loaded from disc) into the host language, and interpret them at runtime. Script engines allow to load code of scripting languages into the host language and execute them. For example the *Oracle Nashorn* project enables integrating JavaScript into Java via an interpreter. However, the interaction between host and guest language is very generic, similar to spawning new processes and collecting their results after termination. Further on, *SQL* queries – or, in case of an ORM language like *JPQL* (Java Persistence Query Language) – are often represented as string literals. They are hard to validate, since the compiler cannot distinguish literals that are queries from others of different nature. Some *IDEs* (Integrated Development Environments) try to guess whether a string is a query or not and validate it. But these approaches are stretched to their limits if the string (i.e. the DSL code) is constructed dynamically in the host language via string interpolation or template processing. Hence, this approach towards a universal language inherently lacks referential integrity. Because whatever is not captured by the language itself has to be captured via language constructs, integrations tend to be too loose.

The issue of striking the 'right' balance is a central motivation behind the emergence of internal DSLs. For example, *Criteria* facilitates the type-safe implementation of JPQL queries directly in Java by making heavy use of *Generics*, i.e., parametric polymorphy. As a result, the internal DSL *Criteria* is closely related to the external DSL *JPQL* by modeling basic components like `SELECT`, `FROM`, `JOIN`, and `WHERE` clauses via corresponding generic classes, so that they appear to be WHAT descriptions. But a Criteria query is an object tree of these generic components, constructed via slotting the objects together in Java code. At runtime, the object tree is constructed and used to generate a database query. So, the WHAT description of JPQL is hidden in a HOW description in the host language Java. In contrast to using a text blob, the Java compiler is now able to check whether the internal DSL has been used syntactically correctly. The 'knowledge' of the semantics is however very limited.

Languages like *Ruby* or *Python* try to compensate this limitation by allowing a high degree of language adaptation, so that internal DSLs can express WHAT descriptions more naturally. This is realized by relaxing the type system to be dynamic, i.e., checked at runtime. Runtime typing makes it much harder to reason about types, and prevent errors, so it ends up generating the need for highly skilled and disciplined developers. The increased level of programming discipline has two reasons. Firstly, many problems a type checker would identify and prevent in a statically typed language are now left under the responsibility of the developer. Secondly, the more a language changes, the less predictability a developer can expect.

2.2 The Power of Domain-Specificity

Modern programming languages free programmers from memory management; automatic clustering software takes care of scalability; version management systems support the development process; application servers ease the mastering of the web stack; security frameworks deal with authentication and authorization; technologies like *SSL/TLS* (secure socket layer/transport layer security) provide transparent encryption and decryption of network connections etc. In turn, e.g., the development of version management systems is certainly a very specific domain and could benefit from a domain-specific development framework. However, future challenges will not be limited to this kind of *horizontal* separation of concerns, which is typically addressed with technologies like aspect-oriented programming, but also *vertical* separation of concerns as classically provided by compilation (or transformation) technologies.

The example of program analysis is a good illustration. Dataflow analysis (DFA) frameworks provide a domain-specific language for specifying program analysis problems in terms of minimal or maximal solutions of (boolean) equation systems. The corresponding solutions can be typically computed via fixpoint iteration, so the user's task is essentially reduced to the specification of an equation system. Compared to a traditional program for the analysis algorithm, equation systems can be certainly regarded as WHAT-style descriptions. However, this WHAT is the implied fixpoint computation and not the original analysis problem. In the implied fixpoint computation, for example, live variable analysis amounts to a backward propagation of information about variable usage: a variable is considered live at each program point where such usage information can be propagated. This means that one has to understand the fixpoint computation, which itself is a HOW, in order to understand what the equation system means. The situation dramatically improves when one specifies the program analysis in terms of temporal logic properties. The property of liveness of a variable or, as one could say, the "true" WHAT specification becomes

there is a path that passes through a variable `use` *before its* `modification` *or* `termination`

which is a simple *unless* property in temporal logic.

This gain in abstraction may not seem very impressive. However, it is crucial when it comes to verifying properties about the program analyses. This impact became apparent during our construction of the lazy code motion algorithm [27,28]. The possibility to refine the WHAT specification by conjunction of other WHAT specifications, which is typically impossible for HOW specifications, let us elegantly and efficiently solve a 15 year old problem (see e.g. [14]) in dataflow analysis. In fact, our corresponding temporal specification runs faster on a classical iterative model checker [47] than the weaker original handwritten algorithm. Conceptually more intriguing is, however, the elegance of the corresponding correctness and optimality proofs: this comparison is particularly striking with respect to the required argumentation in the original paper on partial redundancy elimination [35]. There was yet another benefit: the algorithms specified in terms of temporal formulas worked directly also for an interprocedural setting when using a model checker for context-free systems [8,9,47], demonstrating this way the superiority of WHAT descriptions when it comes to adaptation and migration.

Another striking example are BNF grammars [4], which form an (even reflexive) (meta)modeling language for extremely concise definition of the syntax of languages. E.g., the BNF

$$N ::= 0 \mid succ(N)$$

defines a language that syntactically represents the natural numbers and reflects faithfully all five Peano Axioms.

How can this be? The first Peano Axiom requires 0 to be a natural number and is explicitely covered. So is the required existence of a (unique) successor $succ(N)$ for each natural number (the second Peano Axiom). The other three Peano Axioms are consequences of two essential conventions of BNFs:

– the syntactic (free) interpretation[4] of terms or two different strings also means different things, and
– the minimality requirement of the sets defined via BFN, i.e., everything must be constructible in finitely many steps by applying the BNF rules.

In particular the fifth Peano Axiom, the foundation for natural induction, is nothing more than an elegant formulation of the minimality requirement.

Thus, in contrast to the Peano Axioms, which specify natural numbers from scratch, the BNF formulation is based on two powerful conventions: the term interpretation and the minimality requirement.

It is the power of such conventions that imposes the lever of the resulting domain-specific scenario. E.g., if we are interested in parsing, BNF specifications are sufficient to entirely generate the corresponding parser code[5]. This impressively shows the leverage of the distinction between WHAT and HOW: the BNF describes only the syntax of the envisioned language, whereas the parser generated from it is a complex program that automatically reads a string from a file, tokenizes it, and builds an abstract syntax tree (AST), all along checking for syntax correctness. This lever impact depends on domain knowledge about parser generation, and reaches far beyond what is reachable with what we traditionally would call code generation (cf. also [22]).

Many striking examples work along these lines, like (hardware and software) synthesis environments, planners, the generation of language interpreters via SOS rules [42], the generation of dataflow analysis algorithms from temporal logic specifications [44–46], or even the (interactive) theorem-prover-based proof generation directly from problem descriptions, even to the point that it comprises program and hardware synthesis [12,20].

[4] One sometimes speaks of term or Herbrand interpretation.
[5] ANTLR: http://www.antlr.org/
Yacc: http://dinosaur.compilertools.net/yacc/
JavaCC: https://javacc.java.net/.

In particular the last example illustrates the extreme power of domain specificity: the entire theorem prover is considered 'domain knowledge' allowing to reduce a proof construction language to simply describe the problem (a WHAT) and not the solution (a HOW). This way, the hard part is moved to *the few* designers of the theorem prover, while making life easy for *the many* users. As a rule of thumb, the more specific is the knowledge about a domain, the more tool support can be given.

The described domain-specific scenarios are clearly far beyond what can be adequately covered by traditional programming. Of course, one may argue that the enhancements discussed in Sect. 2.1 are well capable of treating each of these individual domains in some satisfactory way. However, the approaches described in Sect. 2.1 do not scale to support a significant number of domain-specific settings. In particular, it does not scale to the envisioned scenario where the support for the developer should not only be domain-specific, but problem-specific, or even specific to a particular new requirement for a system already in operation [51].

3 Mastering Domain-Specific Diversity

A number of approaches aim at trading generality for systematic development support and, in particular, full code generation [5,6,21,26,29]. In essence, they advocate domain-specificity as a key for turning generic modeling environments into so-called domain-specific modeling (DSM) frameworks[6] where traditional programming becomes obsolete. In contrast to common UML frameworks, these approaches constrain the addressed (domain-specific) modeling scenario so much that all the running code can be generated fully automatically. Manually filling gaps in generated code stubs is not required, avoiding the need for round trip engineering.

In a comprehensive framework, modeling of a system splits into a number of modeling activities to address individual aspects. These many (aspect) models need to be aggregated during code generation in a consistent fashion. This is a change of mindset from usual programming: instead of taking source files of the same type and generate from each a single artefact of the same target format, here many source files of different types specify different aspects of the target artefacts, which can be themselves of multiple types (cf. Fig. 1).

This multi-dimensional approach is similar to classical mechanical engineering design where, e.g., models for evaluating the wind resistance and models used in crash tests are completely different in nature. On the one hand this tendency to heterogeneity (also) explains the wealth of model types in UML. On the other hand it emphasizes the impact of the *One-Thing Approach* (OTA) [30], whose consistency requirement is a key prerequisite for enabling full code generation. In the following, we first sketch these aspects along a concrete case study: a tool for modeling and fully generating web applications. We then discuss the concepts

[6] The term DSM is often correlated to Kelly and Tolvanen's book [26] and the corresponding MetaEdit framework. However, we broaden the term to all approaches aiming at a similar purpose.

that make us confident to master the challenge of developing and maintaining the wealth of such domain-specific tools.

3.1 Case Study: Full Modeling of Web Applications

The *DyWA Integrated Modeling Environment* (DIME) [5,6] is a model-driven development framework for web applications that puts the application expert (potentially, non-programmers) in the center of the web application development process. DIME is developed with Cinco (cf. Sect. 4) and follows the *One-Thing Approach*. In OTA, multiple models of different types, specialized to certain areas of development, are interdependently connected yielding by construction a much higher traceability than what is common in today's model-driven approaches. This model collective consistently shapes the *one thing*, to the extent of completeness that the described artefact (e.g., a tool, or a web application) can be one-click-generated from that model collective and deployed as a running application. This way, the user is provided with an early prototype of an up-and-running web application right from the beginning. DIME generates entire web applications which run within the Dynamic Web Application (DyWA) [38], a framework that fosters prototype-driven development of web applications throughout the whole application life-cycle in a service-oriented manner [32].

A web application is specified in DIME using three different modeling languages: for data, processes, and GUI. While *data models* define the target domain model in terms of types (including inheritance, attributes, and relations), the business logic is modeled with *processes*. Processes are conceptually based on the service logic graphs (SLGs) already used in jABC4 [39] and its predecessors [31,49], but provide different – more specialized yet similarly structured and handled – types for dedicated behavioral aspects of the application[7]. Finally, *GUI models* reflect the structure of the individual web pages and are primarily used as interaction points within sitemap processes.

In combination, those three model types allow to specify the complete application. As introduced before, a model influences multiple generated artefacts. For example, domain concepts defined in a data model are represented by corresponding types on all layers of the running application (cf. Fig. 1):

– At the lowest layer, data is persisted using the *Java Persistence API* (JPA).
– Processes executed within the DyWA (backend business logic) use dedicated DyWA types implemented in Java.
– During communication between frontend and backend via REST [15], data is represented with JSON objects.
– Finally, for use in the interaction processes of the frontend business logic and in the GUI models for the user interface, DIME data models are generated to dedicated Dart types [1].

[7] A detailed introduction to the available process model types is given in [5,6] and DIME's web site: http://dime.scce.info.

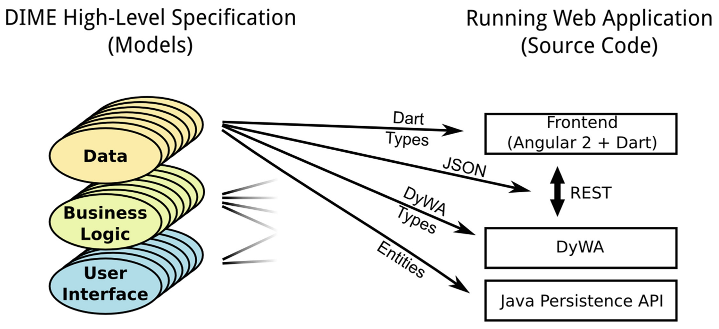

Fig. 1. Examples for Data Model targets

The required management happens in the corresponding code generators and in the running application, without any need for the modeling user, i.e. the application expert who develops the system, to actually know this structure. A more detailed explanation can be found in [5].

3.2 The Continuous Improvement Process

A major challenge and clear bottleneck for DSM approaches is how to provide the required code generators. Today this is mainly treated manually, while the DSM approach we envisage addresses this problem in a framework-bootstrapping fashion. Starting from simple core capabilities, framework bootstrapping enriches this core by successively integrating and then improving tools for modeling very specific kinds of code generators dedicated to specific scenarios like process modeling, parser generation, theorem proving, model checking, planning, synthesis, and SMT solving.

The idea is to use state-of-the-art functionalities and integrate them into dedicated domain-specific modeling environments for enhancing, adapting, and combining these functionalities into increasingly sophisticated solutions. These solutions are then themselves integrated into the overall framework,

- as basic functional building blocks to be (re-)used during subsequent modeling, or
- as extensions enhancing the framework's conceptual support for the development of more or less specialized modeling environments itself.

Whereas the integration of functional building blocks simply supports some higher-level concept of hierarchical design, the extensions introduce a continuous improvement process in a bootstrapping fashion for the entire framework. First results of this approach have been presented in [21,23,24,37] with the Genesys framework. Being a generator of code generators, Genesys' required building blocks concern the basic functionality for writing code generators. These blocks were automatically generated from metamodels of the considered (source) modeling languages, turning the actual code generator development into a modeling discipline. This approach frees the code generator developers from dealing with tedious syntax, and allows for model checking-based consistency proofs [23] of the properties of the resulting code generators. In addition, Genesys provides a model-driven testing framework for back-to-back testing [25]. This technology is based on Genesys' model interpreter, which, in addition, was also the basis for the bootstrapping-based realization of the first Java code generator [21]. Experience showed that writing the first code generator for a certain family of scenarios is still quite complex, but the task becomes increasingly simple for new variants or languages due to strong reuse effects [21] that make it behave similarly to a product line for code generators.

Of course, establishing a product line for a new family of (generalized code) generators, like parser generators, theorem provers, planners etc., is a non-trivial effort and it requires to establish dedicated domain-specific modeling languages. Such languages may require their own analysis and generation technologies, but they profit from a common conceptual core for model checking, simulation, constraint solving, abstraction, view generation etc. Many of these technologies can be applied and reused elsewhere as long as the domain-specific modeling obeys certain rules. Important is here that these rules can be enforced, if required, already at the metamodeling level, in order to guide the domain expert at domain definition time [51]. Many such rules are part of today's implicitly existing common conceptual core. They concern

- The use of BNF for syntax definition as a basis for inductive definition and reasoning
- The use of some kind of typing to enforce consistency at some abstract level
- The use of relational modeling (taxonomies and ontologies) as a basis for defining domains
- The use of structured operational semantics [42] for behavioral semantic definition as the basis for simulation, code generation, and the generation of transition graphs
- The use of transition graphs as a basis for some kind of abstract model checking

However, specific domains allow for stronger constraints and therefore provide better support. E.g., in the context of DIME a lot more is set up upfront for the development of web applications:

- The use of a browser as the GUI technology
- The use of databases for persisting data
- The treatment of events
- User log-in and session handling
- Asynchronous communication between frontend and backend
- Suspending and resuming long-running processes

In fact, in a typical DIME application scenario, the entire web technology stack consisting of database servers, application servers, etc. will be already installed and set up, taking away the burden of technology choice and installation.

Altogether we envision a future design and development technology landscape where hierarchies of (application) domains will be directly linked to product lines of corresponding modeling frameworks, and these hierarchies will be mutually supporting each other in a bootstrapping fashion, inheriting corresponding common conceptual cores. We are therefore convinced that what will increasingly be unified are meta-level patterns rather than concrete languages, and that future tools will be fit to directly deal with these patterns and not just with specific instances.

4 Meta-Level Reuse with the Cinco Framework

The Cinco Meta Tooling Suite [36] provides an initial implementation framework designed to serve as a platform for adoption and use of the concepts on continuous meta-level improvement envisioned in this paper. Cinco is a metamodeling-based tool for creating domain-specific modeling environments. It follows a fully generative approach insofar as it generates complete modeling solutions (which we call Cinco Products) from high-level specifications. Cinco is built upon the Eclipse ecosystem, using the metamodeling framework EMF [52] and the *Rich Client Platform* [33]. Basically, Cinco and all modeling tools it generates comprise of a set of bundles[8] added to the standard Eclipse Modeling Tools release [2].

Framework enhancements leading towards our envisioned unified conceptual core can happen on two levels:

1. Cinco is built to ease the development of highly specialized modeling tools. We thus intend to use Cinco to build modeling tools for the target domain "modeling tool development", and then integrate these tools into Cinco itself in a bootstrapping fashion. This way, certain tool development tasks are designed the first time, but are incorporated in the platform and ready to use from then on. Tools that lend themselves may concern, e.g., checking the syntax of some input string, and obtain their specific DSL, perhaps BNFs. They can arise at the level of the individual modeling tools, the Cinco products, to ease the domain-specific modeling task (in the DIME example, a web application), which may comprise parsing a certain string, and be successively lifted to the Cinco-level, to ease the development of modeling tools (in the DIME example, DIME itself).
2. More general concepts required in many domains (and thus in many tools developed with Cinco) will be 'lifted to the meta level', i.e. they are adequately generalized and abstracted to be integrated as a meta plug-in into Cinco. This way, they can be configured on the meta level with WHAT-driven specifications, resulting in complete sophisticated realizations in the generated modeling tool. Examples range from commonly required 'flavors' of model types (such as data, processes, etc.) to various features found in programming languages (execution semantics, type systems, error handling, scoping, higher order, etc.).

[8] Bundle is the term used by Eclipse's underlying OSGi architecture. The term plug-in is probably more commonly understood for non-Eclipse developers.

The following two subsections individually sketch each of these levels along the example of specifying model semantics, in particular with code generation. A detailed introduction of Cinco can be found in [36] and on the Cinco website[9].

4.1 Framework Evolution in a Bootstrapping Fashion

The formalisms used by Cinco to fully specify and automatically generate a modeling tool can be regarded under four orthogonal aspects (cf. Fig. 2 (left)):

Metamodels of a Cinco product are defined in the Meta Graph Language (MGL), a specialized textual meta-level DSL for the definition of graph structures built from *nodes* and *edges*. The metamodel of each modeling language in a Cinco Product is defined by its own MGL specification.

The visual appearance of nodes and edges is defined with a Meta Style Language (MSL) model, which is also a Cinco-specific textual DSL. It allows for the simple definition of rendering styles in form of shapes and their appearance and is designed to specifically support metamodels defined in MGL.

The semantics in a modeling tool is often defined in a translational way, i.e. the semantics of a model is given by a translation (i.e. code generator or model to model transformation), and the inherent semantics of the target structure. The semantics of a Cinco product's model type can be defined either with modeled code generators based on the jABC and Genesys frameworks, or be implemented programmatically with Java or Xtend [3].

Validation covers aspects of static semantics, i.e. properties of models that can not directly be reflected by the metamodel defined with MGL. It requires similar constructs as translational semantics, e.g. regarding model traversal, but it checks for properties instead of generating a target artefact. Thus, validation can also be realized with jABC models, or implemented programmatically.

Cinco already simplifies the development of modeling tools by providing strong domain-specific support, but improvements are still possible: MGL and MSL are specifically designed for Cinco as textual formats, but some users might prefer graphical representations. Moreover, modeling code generators, transformations, and validation checks as supported by the Genesys framework is still based on jABC, which is not specialized to any of those tasks. As Cinco is developed for defining modeling languages, we intend to enhance all the aspects of this specification activity with more specialized variants realized with Cinco itself. This does, of course, not necessarily mean that there will be exactly four new languages, as certain parts of aspects might be better supported with even more specialized model types. For instance, separate formalisms for semantics definitions – one specialized on code generation, the other on transformations – would further focus the development.

Figure 2 illustrates this idea. The Cinco side (left) shows the four aspects of modeling tool specification, each of which is required for the generation of

[9] http://cinco.scce.info.

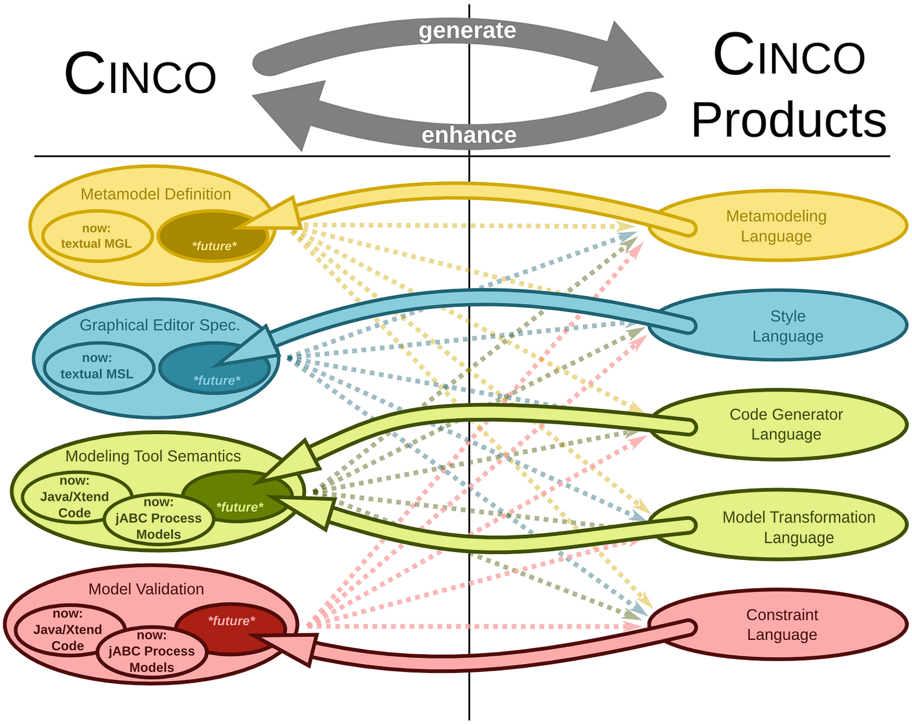

Fig. 2. Extending Cinco's specification formats in a bootstrapping fashion with specialized modeling tools generated with Cinco.

each Cinco Product on the right side (depicted with the dashed arrows in the background). The Cinco Products on the right side specialize on individual aspects of modeling tools and are integrated into future versions of Cinco to enhance the pool of available meta-level languages.

4.2 Enhancing the Conceptual Core with Meta Plug-Ins

A first step in the evolution of Cinco will be to develop a successor of the Genesys framework to free the Cinco ecosystem from jABC's legacy technology[10], to replace the jABC-based definition of code generators for Cinco Products. Such a new Cinco-Genesys will have considerable similarities with process models in DIME, as both are spiritual successors of jABC-based processes[11]. However, they will come with certain characteristics of their domains.

[10] This is the main reason why we developed DIME's initial code generators using Xtend.

[11] Prior to DIME, processes for DyWA-based web applications were modeled in jABC with dedicated components generated from the application's data schema, in turn modeled in DyWA.

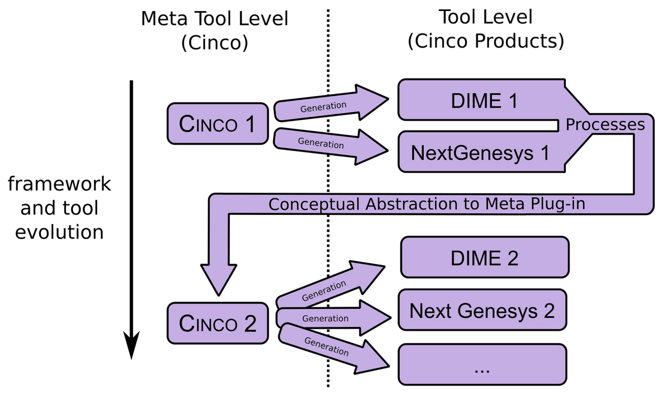

Fig. 3. Enhancing the framework by abstracting the concept of 'tools for process modeling' to the meta level (i.e. realize as a meta plug-in).

We plan to lift the concept of 'executable processes' to the meta level, i.e. provide a corresponding meta plug-in for Cinco. This conceptual uplift will take the realization of modeling tools that support process modeling from the current HOW to a corresponding WHAT level. In turn, the WHAT-level configurations of the resulting meta plug-ins will need dedicated specification formats. Figure 3 illustrates how the iterative evolution of akin modeling tools could look like. The conceptual abstraction to a meta plug-in comprises the following steps:

– Certain structural and visual design decisions (i.e. portions of MGL and MSL specifications) will be shared.
– Parts of the semantics will essentially be the same, only with several basic modeling components then specialized to the modeling of code generators (for instance, the efficient inclusion of templates, and structures supporting the traversal of models), and of web applications (e.g., long-running processes, and interaction with the database).
– Other future tools realizing more specialized variants of jABC will benefit from them too, if they require similar (structural as well as semantical) aspects.

The general structure of processes just served as one example. We envision other concepts to be abstracted as meta plug-ins that handle how to design, manipulate and check them, for instance type systems, error handling, scoping, and higher order.

Integrating those into a common conceptual core aims at evolving language creation to a "shopping experience", where one just selects what aspects the domain-specific language requires, finds predesigned off-the-shelf specialized and well-fitting tools to handle them, and just needs to apply some configuration and fine-tuning in order to tailor the concept and its tooling annex to the specific domain.

5 Summary and Discussion

We have sketched a scenario where application programming gradually evolves into the discipline of using highly specialized domain-specific (modeling) languages, and where the art of mastering the required construction of languages and development tools becomes a commodity. In our terminology, this means that there will be an increasing number of dedicated WHAT-style languages, whose corresponding tool frameworks profit from a growing unified conceptual core on the HOW level. It does, however, not mean that there will be a uniform general-purpose HOW language. Rather, because the distinction between WHAT and HOW very much depends on the beholder's perspective, there will also be domain-specific HOW languages. For example, BNF grammars are very domain-specific and they are certainly considered to be at the HOW level by many people. In fact, we envision a bootstrapping effect where the results of dedicated WHAT-level developments (e.g. for certain analyses) are integrated into development frameworks, rendering these tasks from then on for the bulk of the enhanced framework's users. With this change, we believe that domain-specific tool development will evolve and be simplified to a point where domain-specific frameworks are designed even for individual projects as discussed in [51].

Domain-specific languages do not necessarily describe an application[12] entirely. Therefore, some programming languages already offer an interoperability or bridging layer. Java needs this capabiliy, e.g., in order to call system dependent functions. Although Java processes live in a virtual machine (i.e., the JVM) they have to interact with the concrete system when it accesses devices, e.g., for reading files from disc or communicating over the network. Java offers JNI in order to bind constructs from the underlying system directly to Java components with a well-defined and configurable transformation of data for parameters and return value. Apple's language *Swift* has a sophisticated interoperation layer to *Objective-C* and *C* code, too. The focus in Swift does not lie on accessing system dependent functions, but in code reuse for existing frameworks and libraries, as the underlying *LLVM* compiler framework is not based on a virtual machine.

These interoperability layers offer a well-defined mapping from language constructs between the participating languages, so they allow to introduce referential integrity. Until now this has been used for enabling platform independence (Java) and reuse (Swift), only. We believe that the trend will be to transfer this pattern to realize interdependent families of domain-specific languages quite similar to the envisioned scenario proposed in this paper.

This trend will not help to overcome the difference between modeling and programming. Rather, what we call programming today will appear in special sub-disciplines in the future landscape of system development: a special art, mastered by a few experts, who are, in particular, required to evolve the overall scenario. They will e.g. be responsible for all the required meta tooling and development frameworks, apply bootstrapping technology, aggregate purpose-specific models to a whole, provide automatic deployment and quality assurance,

[12] This can be very different artefacts, not just classic desktop applications.

and guarantee security. The bulk of the development, however, will concern application development, and be in the hands of the application experts who do not need to have any dedicated programming knowledge, just as today one does not need to be a web designer with special knowledge in HTML, CSS, or JavaScript to set up a website [18]. Thus programming experts will turn into a kind of generalized infrastructure providers, enabling the application experts to solve their customer-specific tasks themselves.

From a wider perspective, programming and modeling will be quite similar in this new setting. Both will serve very specific purposes while abstracting from many other issues. For example, the purpose of programming may, depending on the actual sub-discipline, just concern the code generation, security aspects, performance issues, scalability, etc., i.e. issues that can be treated independently of the actual primary customer concern, while the application experts can fully focus on the functionality of the application. The underlying domain-specific frameworks are intended to support a clean separation of concerns by providing required but purpose-specific functionality as built-in commodity.

This future scenario illustrates the impact an underlying framework or domain-specific setting can have on the mindset. Rather than trying to establish a universal language, we consider the identification, design, realization, and the evolution of conceptually new domain-specific languages as a driver for innovation. Of course, the unification of modeling and programming, and in particular the steadily growing underlying conceptual common core, are essential for mastering this challenge.

Acknowledgments. This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

References

1. Dart programming language. https://www.dartlang.org/. Online; last accessed 26 Jul 2016
2. Eclipse Modeling Tools. http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/lunasr2. Online; last accessed 30 Jul 2016
3. Xtend - Modernized Java. https://www.eclipse.org/xtend/. Online; last accessed 30 Jul 2016
4. Backus, J.W.: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In: IFIP Congress, pp. 125–131 (1959)
5. Boßelmann, S., Frohme, M., Kopetzki, D., Lybecait, M., Naujokat, S., Neubauer, J., Wirkner, D., Zweihoff, P., Steffen, B.: DIME: a programming-less modeling environment for web applications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part II. LNCS, vol. 9953, pp. 809–832. Springer, Cham (2016)

6. Boßelmann, S., Neubauer, J., Naujokat, S., Steffen, B.: Model-driven design of secure high assurance systems: an introduction to the open platform from the user perspective. In: Margaria, T., Solo, A.M.G. (eds.) The 2016 International Conference on Security and Management (SAM 2016). Special Track "End-to-end Security and Cybersecurity: from the Hardware to Application", pp. 145–151. CREA Press (2016)
7. Broy, M., Havelund, K., Kumar, R.: Towards a unified view of modeling and programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part II. LNCS, vol. 9953, pp. 238–257. Springer, Cham (2016)
8. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)
9. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theoret. Comput. Sci. **221**(1–2), 251–270 (1999)
10. Chen, P.P.S.: The entity-relationship model - toward a unified view of data. Trans. Database Syst. (TODS) **1**(1), 9–36 (1975)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
12. Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J., Smith, S.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, Upper Saddle River (1986)
13. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Formal Methods Syst. Des. **19**(1), 45–80 (2001)
14. Dhamdhere, D.M.: A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). Int. J. Comp. Math. **27**, 1–14 (1989)
15. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
16. Filev, A., Loton, T., McNeish, K., Schoellmann, B., Slater, J., Wu, C.G.: Professional UML Using Visual Studio .Net. Wiley Publishing Inc., Indianapolis (2003)
17. Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley/ACM Press (2011)
18. Gelbmann, M.: WordPress powers 25% of all websites (2015). https://w3techs.com/blog/entry/wordpress-powers-25-percent-of-all-websites. Online; last accessed 19 Jul 2016
19. Gordon, M., Milner, R., Morris, L., Newey, M., Wadsworth, C.: A metalanguage for interactive proof in LCF. In: Proceedings of the 5th Symposium on Principles of Programming Languages (POPL 1978) (1978)
20. Jackson, P.B.: Nuprl and its use in circuit design. In: Stavridou, V., Melham, T., Boute, R. (eds.) Proceedings of the IFIP TC10/WG10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, pp. 311–336 (1992)
21. Jörges, S. (ed.): Construction and Evolution of Code Generators. LNCS, vol. 7747. Springer, Heidelberg (2013)
22. Jörges, S., Lamprecht, A.L., Margaria, T., Naujokat, S., Steffen, B.: Synthesis from a practical perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I. LNCS, vol. 9952, pp. 282–302. Springer, Cham (2016)
23. Jörges, S., Margaria, T., Steffen, B.: Genesys: service-oriented construction of property conform code generators. Innov. Syst. Softw. Eng. **4**(4), 361–384 (2008)
24. Jörges, S., Steffen, B.: Exploiting ecore's reflexivity for bootstrapping domain-specific code-generators. In: Proceedings of 35th Software Engineering Workshop (SEW 2012), pp. 72–81. IEEE (2012)

25. Jörges, S., Steffen, B.: Back-to-back testing of model-based code generators. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 425–444. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45234-9_30
26. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press, Hoboken (2008)
27. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI), pp. 224–234. ACM (1992)
28. Knoop, J., Rüthing, O., Steffen, B.: Optimal Code Motion: Theory and Practice. ACM Trans. Program. Lang. Syst. **16**(4), 1117–1155 (1994)
29. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomasson, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing (WISP 2001) (2001)
30. Margaria, T., Steffen, B.: Business process modelling in the jABC: the one-thing-approach. In: Cardoso, J., van der Aalst, W. (eds.) Handbook of Research on Business Process Modeling. IGI Global (2009)
31. Margaria, T., Steffen, B., Reitenspieš, M.: Service-oriented design: the jABC approach. In: Cubera, F., Krämer, B.J., Papazoglou, M.P.(eds.) Service Oriented Computing (SOC). No. 05462 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
32. Margaria, T., Steffen, B., Reitenspieß, M.: Service-oriented design: the roots. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 450–464. Springer, Heidelberg (2005). doi:10.1007/11596141_34
33. McAffer, J., Lemieux, J.M., Aniszczyk, C.: Eclipse Rich Client Platform, 2nd edn. Addison-Wesley Professional (2010)
34. Milner, R.: LCF: a way of doing proofs with a machine. In: Bečvář, J. (ed.) MFCS 1979. LNCS, vol. 74, pp. 146–159. Springer, Heidelberg (1979). doi:10.1007/3-540-09526-8_11
35. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Comm. ACM **22**(2), 96–103 (1979)
36. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools (to appear, 2016)
37. Naujokat, S., Neubauer, J., Lamprecht, A.L., Steffen, B., Jörges, S., Margaria, T.: Simplicity-first model-based plug-in development. Softw. Pract. Exp. **44**(3), 277–297 (2013)
38. Neubauer, J., Frohme, M., Steffen, B., Margaria, T.: Prototype-driven development of web applications with DyWA. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 56–72. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45234-9_5
39. Neubauer, J., Steffen, B., Margaria, T.: Higher-order process modeling: product-lining, variability modeling and beyond. Electron. Proc. Theoret. Comput. Sci. **129**, 259–283 (2013)
40. Object Management Group (OMG): OMG Meta Object Facility (MOF) Core Specification Version 2.4.1, http://www.omg.org/spec/MOF/2.4.1/PDF. Online; last accessed 23 Apr 2014
41. Object Management Group (OMG): Documents associated with Object Constraint Language (OCL), Version 2.4, February 2014. http://www.omg.org/spec/OCL/2.4/

42. Plotkin, G.D.: A Structural Approach to Operational Semantics. Tech. rep., University of Aarhus, dAIMI FN-19 (1981)
43. Rumbaugh, J., Jacobsen, I., Booch, G.: The Unified Modeling Language Reference Manual. The Addison-Wesley Object Technology Series, 2 edn. Addison-Wesley Professional, July 2004
44. Schmidt, D., Steffen, B.: Program analysis *as* model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998). doi:10.1007/3-540-49727-7_22
45. Steffen, B.: Data flow analysis as model checking. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 346–364. Springer, Heidelberg (1991). doi:10.1007/3-540-54415-1_54
46. Steffen, B.: Generating data flow analysis algorithms from modal specifications. Sci. Comput. Program. **21**(2), 115–139 (1993)
47. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: Lee, I., Smolka, S. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 72–87. Springer, Berlin Heidelberg (1995)
48. Steffen, B., Jörges, S., Wagner, C., Margaria, T.: Maintenance, or the 3rd dimension of eXtreme model-driven design. In: IEEE International Conference on Software Maintenance 2009 (ICSM 2009), pp. 483–486 (2009)
49. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-driven development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)
50. Steffen, B., Margaria, T., Wagner, C.: Round-Trip Engineering, chap. 94, pp. 1044–1055. Taylor & Francis (2010)
51. Steffen, B., Naujokat, S.: Archimedean points: the essence for mastering change. LNCS Trans. Found. for Mastering Change (FoMaC) **1**(1) (2016)
52. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
53. Wolper, P.: The meaning of "formal": from weak to strong formal methods. Int. J. Softw. Tools Technol. Transf. (STTT) **1**(1), 6–8 (1997)