On the Difficulty of Drawing the Line

Steve Boßelmann, Stefan Naujokat(✉), and Bernhard Steffen

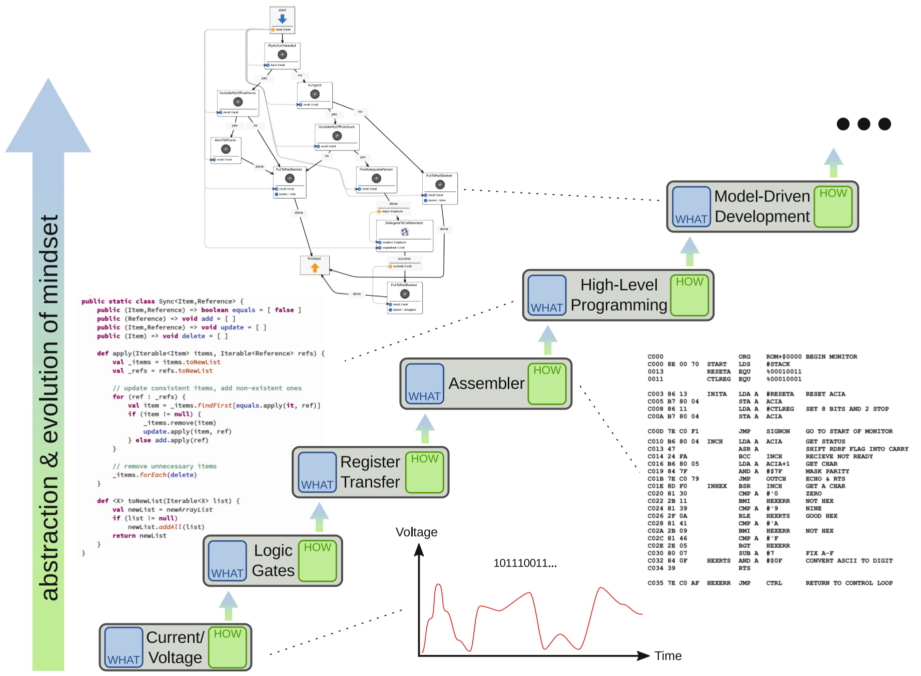
Chair for Programming Systems, TU Dortmund University, Dortmund, Germany
{steve.bosselmann,stefan.naujokat,steffen}@cs.tu-dortmund.de

Abstract. The paper considers domain-specific tool support as a means to turn descriptive into prescriptive models, and to blur the difference between models and programs, and even between developers and users. Conceptual underlying key is to view the system development as a decision process which increasingly constraints the range of possible system implementations and Domain-Specific Languages (DSLs) as a means to freeze taken decisions on the way towards a concrete realization. This way naturally comprises both programming and modeling aspects. In fact, considering all interactions that influence the behaviour of the system as development turns GUIs into DSLs and makes it even hardly possible to draw the line between developers and users. We will illustrate this viewpoint in the light of the development of the Equinocs system, Springer's new editorial service.

Keywords: Modeling · Metamodeling · Programming
Domain-specific languages · Language-oriented programming
Language-driven engineering · Code generation · Model transformation

1 Introduction

There is a general tendency that structures and categorizations considered obvious in the past often get blurred in the course of deeper investigation. E.g., the separating line between control and data path, traditionally clearly defined, is today often profitably moved by changing the level of interpretation, and even the gender classification has recently moved from a binary to a continuous spectrum. A similar trend can also be observed when considering the role, structure, and pragmatics of modeling/specification languages on the one, and programming languages on the other side: Originally, there was a quite clear distinction between the typically very abstract loose and descriptive modeling and the much more concrete and prescriptive programming. This distinction was also clearly reflected in their role: Modeling aimed at the description of WHAT a system is supposed to do (in particular, this description should not force the programmer to overspecify), and programming should aim at the description of an efficient solution, the HOW. Technically, this was reflected by modeling languages to support certain logics, whereas programming tended to be more imperative. With the evolution of more and more involved compilation and synthesis techniques



```
public static class Sync<Item,Reference> {
    public (Item,Reference) => boolean equals = [ false ]
    public (Reference) => void add = [ ]
    public (Item,Reference) => void update = [ ]
    public (Item) => void delete = [ ]

    def apply(Iterable<Item> items, Iterable<Reference> refs) {
        val _items = items.toNewList
        val _refs = refs.toNewList

        // update consistent items, add non-existent ones
        for (ref : _refs) {
            val item = _items.findFirst[equals.apply(it, ref)]
            if (item != null) {
                _items.remove(item)
                update.apply(item, ref)
            } else add.apply(ref)
        }

        // remove unnecessary items
        _items.forEach(delete)
    }

    def <X> toNewList(Iterable<X> list) {
        val newList = newArrayList
        if (list != null)
            newList.addAll(list)
        return newList
    }
}
```

```
C000                      ORG     ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START      LDS     #STACK
0013           RESETA     EQU     %00010011
0011           CTLREG     EQU     %00010001

C003 86 13     INITA      LDA A   #RESETA   RESET ACIA
C005 B7 80 04             STA A   ACIA
C008 86 11                LDA A   #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04             STA A   ACIA

C00D 7E C0 F1             JMP     SIGNON    GO TO START OF MONITOR

C010 B6 80 04  INCH       LDA A   ACIA      GET STATUS
C013 47                   ASR A             SHIFT RDRF FLAG INTO CARRY
C014 24 FA                BCC     INCH      RECIEVE NOT READY
C016 B6 80 05             LDA A   ACIA+1    GET CHAR
C019 84 7F                AND A   #$7F      MASK PARITY
C01B 7E C0 79             JMP     OUTCH     ECHO & RTS
C01E 8D F0     INHEX      BSR     INCH      GET A CHAR
C020 81 30                CMP A   #'0       ZERO
C022 2B 11                BMI     HEXERR    NOT HEX
C024 81 39                CMP A   #'9       NINE
C026 2F 0A                BLE     HEXRTS    GOOD HEX
C028 81 41                CMP A   #'A
C02A 2B 09                BMI     HEXERR    NOT HEX
C02C 81 46                CMP A   #'F
C02E 2E 05                BGT     HEXERR
C030 80 07                SUB A   #7        FIX A-F
C032 84 0F     HEXRTS     AND A   #$0F      CONVERT ASCII TO DIGIT
C034 39                   RTS

C035 7E C0 AF  HEXERR     JMP     CTRL      RETURN TO CONTROL LOOP
```

Fig. 1. Evolution of mindset in the past decades: from HOW to WHAT

this clear categorization got blurred.[1] Figure 1 sketches this evolution, which indicates the steady move towards more and more abstract system description organized in a HOW/WHAT cascade.

This cascaded abstraction influenced the development of increasingly higher-level modeling and programming languages as well as their support frameworks and infrastructures: defining a system's behavior at the level of electric current and voltage is almost impossible, leading to the introduction of numerous well-defined abstractions in the design of hardware components: transistor level, gate level, register transfer level, up to the instruction set level, which was the first abstraction offered to humans to "interact" with the machine by programming it. The instruction set of a CPU was in fact the basis for assembler languages, which in the fifties and sixties were regarded as extremely high-level abstractions of the underlying hardware. The successive raise from one level to the next built a systematic discipline to map the higher-level concepts stepwise down to the electrical level, where each program, each instruction, each clock cycle level operation are eventually executed. Assembler languages introduced the "thinking in commands" perspective, and thereby the imperative programming paradigm.

[1] In the hardware domain, synthesizing circuits from logical descriptions is actually standard since many decades! [11].

Today, they typically form the lowest abstraction level modern computer scientists are willing to consider.

The enormous development of programming and modeling languages [3,4] of the last 50 years equipped programming languages with concepts, metaphors, and mechanisms that allow programmers to focus more and more on describing the intended functionality (the WHAT) directly at the upper level, rather than having to deal with implementation details like inner structures, specific machine characteristics or the economic use of storage (the HOW). The more complex the addressed platform, the greater is the benefit of this approach. A good example here are modern Web applications which require the configuration of a complicated stack of technologies [1]. Another example are cross-cutting concerns like security and real time behaviour [2,18]. Enhanced development frameworks allow developers also here to focus on the required functionality, while the other concerns are taken care of during code generation and deployment, e.g., in an aspect-oriented fashion [13]. This ongoing development is, in fact, not so different from the well-established steps that decoupled programming from electrical level considerations.

In this paper we discuss a, in a sense, more radical trend and its impact: moving from universal languages to domain-specific or even purpose-specific languages. We illustrate that adequate specialization combined with tailored tooling allows one to simplify the required languages to a point that even application experts without any programming knowledge can express their intents in a way that corresponding solutions can be automatically generated without sacrificing non-functional properties like dependability, efficiency, and security. Conceptual backbone is our Language-Driven Engineering (LDE) [9,19] approach, which can be explained as a discipline of DSL orchestration.

After sketching the LDE background and philosophy in Sect. 2, Sect. 3 discusses LDE-oriented language refinement along the development of Springer's Equinocs system, before we conclude with a discussion of the approach and directions for future research in Sect. 4

2 LDE: Background and Philosophy

Modeling, specification, programming: drawing clear lines between these notions seems impossible. As clearly pointed out by Völter [23], usual criteria like executability, looseness, textual, graphical, and abstraction, give tendencies, but do not lead to accurate classification. We go even a step further and see the distinction between prescriptive and descriptive as a matter of purpose and perspective. Consider climate models, mentioned by Völter as clearly out of scope of prescriptive model-driven approaches: Good descriptive models for climate are naturally ideal prescriptive models for corresponding simulators, which, in a sense, make the original climate models even executable!

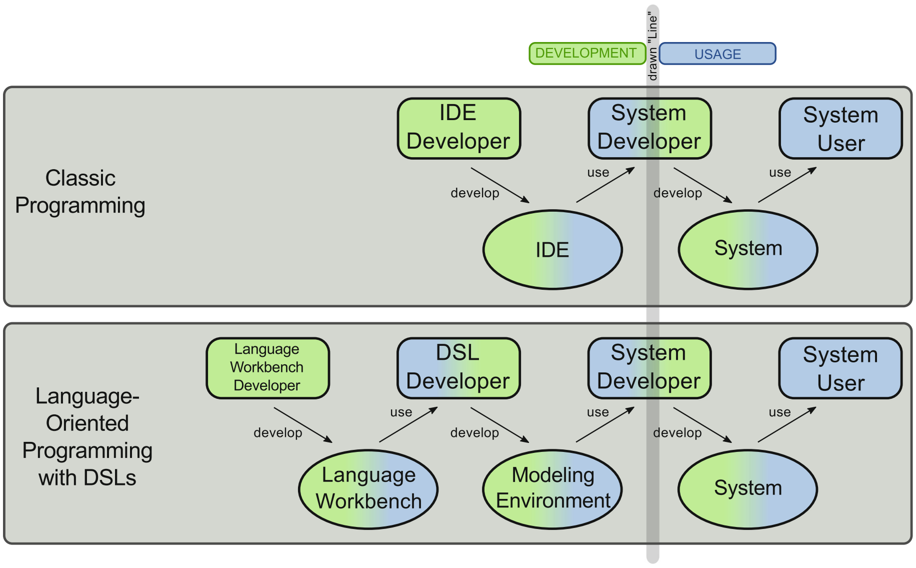


Fig. 2. Comparison: meta levels in classic programming vs. domain-specific approaches

In principle, also the most abstract descriptions[2] can become executable when regarded in a specific context: the graphical model of a calender turns into a prescriptive specification when, e.g., defining travel schedules, and typical (product) configurators use numerous similarly abstract and domain-specific descriptions as prescriptive modeling components for search and filtering.[3] In fact, the whole idea of domain-specific (modeling) languages (DSLs) [8,16] can be regarded as a way to automatically make abstract description *operational*.[4] Of course, this requires to establish adequate corresponding (execution) landscapes, powerful enough to adequately interpret the abstract input description.[5]

Language-Oriented Programming (LOP) [5,6,24] exploits this observation by offering to develop DSLs within appropriate meta environments – often called Language Workbenches [7] – in order to also address stakeholders without (explicit) programming expertise (cf. Fig. 2). Impressive is here the projectional editing functionality of JetBrains MPS [12], which allows, e.g., business people

[2] here to be understood as descriptive models, as any form of description can be interpreted that way.

[3] For a car configurator, (descriptive) images of a car, are often used to support the adequate selection (prescription).

[4] Please understand 'operational' as 'lives up to its purpose in a corresponding realization'.

[5] Traditional correctness-by-construction methods fit in here as well. In fact, there is no clear conceptual difference between this DSL-'enactment' and a compiler. Only the level of application is different.

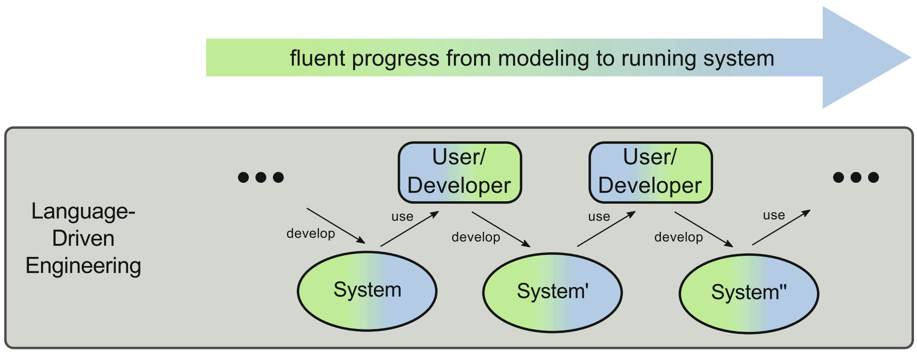


Fig. 3. Continuous system development with language-driven engineering

to contribute tabular data as part of a larger system in the 'Excel'-like fashion they are used to. LOP typically results in a waterfall-like 'meta workflow' where required additional DSLs are provided before the actual systems development starts.

Language-driven engineering (LDE) [9,19] goes a step further by considering the development of tailored DSLs as essential continuous part of the system development and evolution process (cf. Fig. 3). E.g., our LDE-based development of the Equinocs system[6] proceeds by co-evolution of the metamodeling environment Cinco [17] for easier generation of (graphical) domain-specific languages, of DIME [1], our dedicated (graphical) DSL-specific[7] integrated development environment (IDE),[8] and of Equinocs itself.

This co-evolution requires elaborate means for maintaining coherence, e.g., to allow one to (easily) migrate existing Equinocs versions into new versions of DIME and existing DIME versions into new versions of Cinco in a way that exploits the new features of DIME, an Eldorado for generative programming and model-to-model transformations.[9] In this context, it is important to distinguish three flavours of language evolution:

- In an upgrading form (e.g. Java 7 to Java 8). Often, at least when this happen as natural extension, migration is not an issue at all.
- As DSL specialization. In this case it makes a lot of sense to define the semantics of the specialized language via translation to the original language,

[6] The next version of Springers Online Conference Service.
[7] In [19] we introduced the term mindset-specific IDE, and the acronym mIDE to indicate that DSLs are more than just means for purpose-specific simplification. New mindsets are powerful means to reach 'out of the (traditional) box'.
[8] DIME comprises currently four graphical DSLs.
[9] This requirement may remind of the situation of frameworks that have to explicitly deal with round-trip engineering, a problem that is overcome by frameworks following a full (code) generation policy.

with the advantage that reusing the available code generator for that language immediately provides a code generator for the specialized language [21]. On the other side, migration is typically quite constraint and complicated, but typically also only required in a very restrictive fashion as part of a (major) refactoring activity.

– As service oriented extension with new dedicated languages in the way discussed in [19], which can be regarded as a light-weight language specialization technology generalizing the idea of component libraries, with an intent reminiscent of the ideal of projectional editing of MPS.

LDE considers all these forms of evolution as part of the product development/system life cycle. In a sense, this generalizes the idea of ‘a software system is never finished’ to its entire development and evolution scenario. Everything co-evolves, the meta-metamodels for defining tailored DSLs, the DSLs, the system itself, as well as the infrastructure the system is supposed to run on. In this highly agile approach, the roles of metamodels, models, specifications, and programs converge, however, as part of an overall scenario that is characterized by diversification: We envisage the number of languages involved in individual projects to radically grow. In particular, we envisage that the use of graphical notation will lead to a tighter integration of domain experts into the system/application development process, as it is typically more intuitive and helps supporting the intended mindsets and purpose. We will discuss our corresponding visions, options, and the state of the art along the Equinocs development in the remainder of this paper.

3 Application Example: The Equinocs Development

In the following, we sketch LDE-oriented language refinement in the context of the development of Equinocs – Springer’s new online conference management system that handles all phases of a conference, from setup over submission to review and proceedings production. Equinocs is entirely developed in a model-driven way with DIME, a tool that provides dedicated modeling languages for the development of complex multi-user Web applications.

Modeling Web applications with DIME is done by defining user interactions, the presentation and organization of the GUI, as well as eventually required permissions to access data or perform actions. DIME provides specific modeling languages for the creation of these aspect-specific models:

– **Interaction processes** manage the user interactions with the application, like page transitions including the data flow between different pages.
– **GUI models** define the appearance of pages in the Web application. They comprise basic UI components to display various types of content (text fields, images, etc.) as well as structural components to define panels, blocks, grid layouts, etc. Furthermore, data input via forms can be modeled with respective components for content-specific form fields.

– **Data models** are used to model the actual domain by means of various data types with attributes as well as (bidirectional) associations between them.

The basic building blocks of models in DIME are based on a component model called Service-Independent Building Blocks (SIBs) [20] that either are basic components or reference hierarchically to other models in the workspace. While the so-called Native SIBs wrap native implementations like service calls to an API, those SIBs that hold references to other models in the workspace enable the reuse of models. In DIME, process models and GUI models can be nested inside each other in a hierarchical fashion, both mutually as well as within models of the same type. In the context of GUI models, this enables the easy creation and encapsulation of recurrent complex components, like forms. Regarding process models, a clean process architecture arises from the use of sub-processes via Process SIBs and the integration of user interfaces happens likewise via GUI SIBs.

From DIME models, the running application is fully generated by a powerful code generator. Beyond that, DIME is itself generated in a model-driven way from models of the domain 'modeling language development'. The tool that provides editors for those models, as well as the required code generator, is our Cinco framework, which provides specific modeling languages, the Meta Graph Language (MGL) and the Meta Style Language (MSL), for the definition of both the structure as well as the visualization of graph models. Finally, in this line of modeling and code generation, Cinco is built upon the frameworks provided by the Eclipse Modeling Project [10,22]. Already here, we see two things:

1. Modeling is used – like programming – for the development of a program, tool or system.
2. The user of one level can be a developer of another: users of DIME are developers of the Equinocs system and the developers of DIME are users of the Cinco framework.

Both observations show that trying to draw a clear line seems impossible. Should this even be tried? Or should we strive for more levels of language development, so that with each level, the domain of the target system is more and more narrowed until we reach the finally desired system? Of course, such a more fluent approach only works efficiently with a powerful support framework. Most of the challenges (some of which we will discuss in the following) are not yet solved with any of the available language workbench systems. We already address some with the Cinco ecosystem, but believe that the modeling/programming community needs to team up for this ambitious aim.

In the following, three examples from the context of Equinocs, which motivate the provision of new UIs as DSLs, will be presented.

3.1 Report Form Language

In the Equinocs system, reviewers fill out report forms to provide their assessment for each paper they have been assigned to. These reports can be accessed

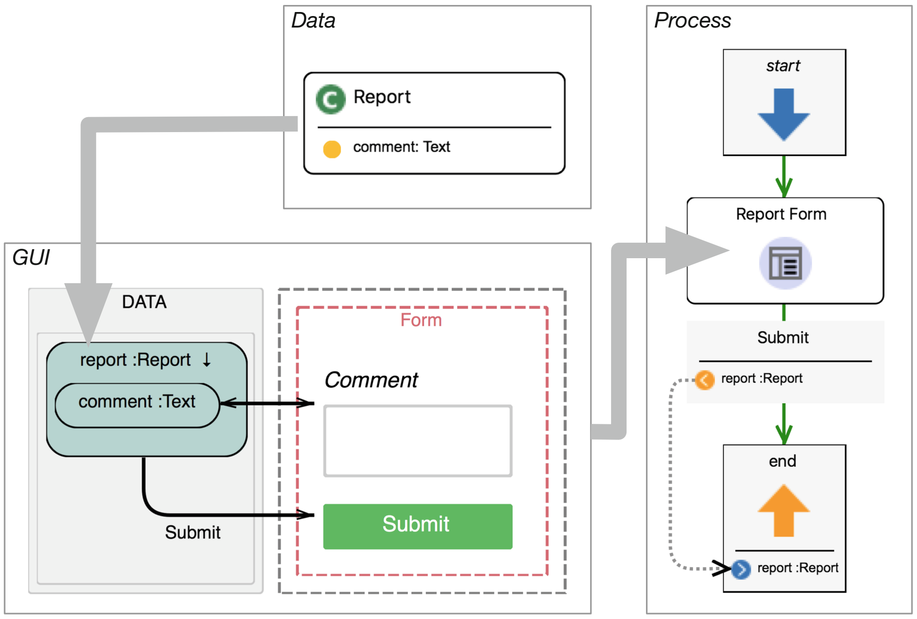


Fig. 4. DIME models for report creation

by the members of the conference committee and make up the basis for decision making. Hence, the UI created by the developers comprises a page for creating and editing a report (report form) as well as a page for presenting the actual result (report details). To keep the following example simple, let us assume that this report form consists of only a single field *Comment* to hold a free text. Figure 4 shows the required models in DIME for the realization of this report form in Equinocs. The GUI model holding the form is embedded in the interaction process. Both models reference the type *Report* from the data model. While the process model requires this reference for the definition of the data flow, the GUI model uses it to link the form field *Comment* with the data type's attribute *comment* to express where the field's value should be saved.

Let us now assume that we endeavour to extend the report form by introducing an additional field *assessment* to hold one of the discrete values *reject*, *accept* or *indifferent*. This would be helpful to provide a condensed overview for the conference committee. Based on the models so far, this is realized by the Equinocs developers through the following steps:

S1 Extend the data model of the application by means of adding a new attribute *assessment* to the data type *Report*.
S2 Add an additional input field to the UI model of the report form and link it with the newly created attribute *assessment*.
S3 Add an additional display field to the UI model of the report detail page.

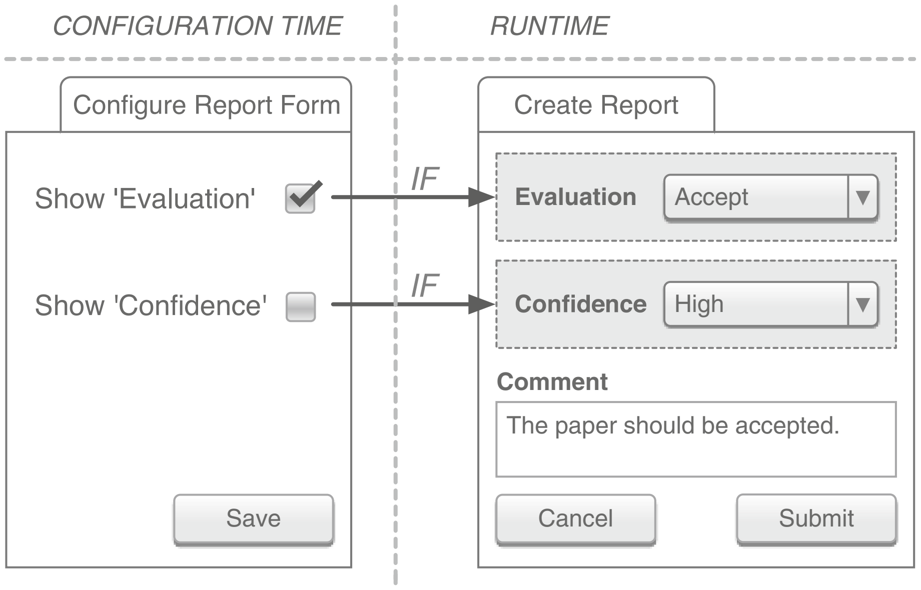


Fig. 5. Report form configuration

From a language-centric point of view, what we create and maintain this way, is a very restricted DSL for reports to be used by reviewers in the running Equinocs system. In particular, any request for new form fields actually is a change request for the report DSL communicated from system users to system programmers, i.e., a language management issue spanning the two layers of system runtime and system design. We will discuss some of the implications derived from this still simple example in more detail. In particular, we will illustrate how a user role morphs more and more into a developer role.

Implications Towards the System Level. The prominent disadvantage of the depicted approach of changing the report form on the system design level is that this would have immediate effect on all managed conferences after the re-deployment of the system, although not all organizers of all conferences would consider the changes as a welcome improvement. Typical management systems for conferences or journals have more complex forms, with often more than 10 fields like ‘confidence of the reviewer’, ‘in scope for the topic of the conference’, ‘novelty in the field’, or even ‘quality of language’. The perception of these feature-rich solutions totally differs, as some may find it absolutely necessary, while others consider it highly annoying. However, it is almost natural to introduce conference settings to hold options for the organizers to turn these features on or off, either at initial configuration time or even on demand at runtime.

In the UI, these settings can be realized with a simple check box for each of the report form fields, named with something like ‘Show <field.name>’ (cf. Fig. 5).

From a language-centric point of view, what we just introduced, is a DSL for report forms to be used by system users like the conference organizers. Though we have already introduced a report DSL for reviewers above, the abstraction layer of this report form DSL differs, as the latter can be used to change the former. The ability to change the report DSL has so far been reserved for the system developers. But we shifted parts of the definition of the report DSL from programming level to the system level by introducing a new restricted DSL for that purpose and handing it over to the system users.

This is an excellent example to stress that switching the abstraction layer from the modeling language (report DSL) to the metamodeling language (report form DSL) does not necessarily mean to switch from system level to programming level. The language shift is more fluid and different parts of a language's definition might even be allocated on separate levels, as illustrated by this specific example.

Providing a DSL for report forms to be used by conference organizers adds flexibility for the latter at the cost of clarity and control on the programming level. While so far, the developers of the system had known exactly which fields exist in the report form, they now have to deal with the fact that some may not exist. Any business logic of the system that relies on values from one of the fields needs to be rebuilt in a more robust manner to not fail if they are missing. As an example, consider the logic of the system asserting that papers can only be accepted if at least one report exists whose value for *assessment* is other than *reject*. In general, if the system so far behaved differently depending on the current value of a specific field, it now has to first check whether such a value exists and behave predictable otherwise. The possible absence of a value imposes constraints for defining the corresponding business logic.

What if we continue to shift control over the report DSL from programming level to system level? Although the conference organizers can decide which fields are shown in the report forms, which fields are available at all is still predefined by the system developers. In particular, the workflow to add a completely new field on demand has not changed much. It still would require a change request addressed to the developers and the latter to run through the transformation steps listed above. As an alternative, we might transfer even more power to the system users by allowing them to create new form fields on demand, thereby extending the report DSL used by the reviewers. The required UI for the conference configuration (i.e., the report form DSL) would then need to be extended by a form for the organizers to change the list of available report form fields, including the feature to remove existing fields or create completely new ones (cf. Fig. 6). The latter can be achieved in multiple ways based on different restrictions. We might allow to only create fields of a specific type, e.g. those holding a pre-defined set of possible values (e.g., enum literals). In this case, providing a name for the field as well as the actual list of possible values is enough. Alternatively, we might hand over the definition of the value type of a form field, thus adding the definition options for text, boolean or integer values.

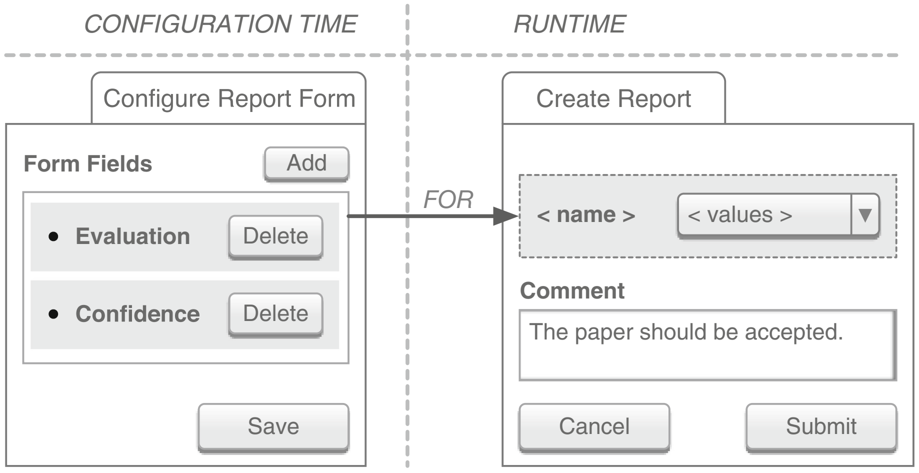


Fig. 6. Report form field creation

Again, these design decisions have so far been reserved for the system developers. And just like we discussed above, this transformation means more flexibility for the system users, but even less clarity and control for the system developers. While so far the developers had to respect that some fields of the report form may not be used at runtime, they now have to deal with the fact that (a part of) the report form is not even known at design time. What remains is kind of a meta-level perspective that tells them there may exist fields, the types they may have as well as the types of possible values. However, the remaining control over these features is hardly enough to be used as a basis to define meaningful business logic of the system. It is impossible to define system behavior based on the value of fields from which the developers do not know whether they exist. However, we might counteract this development by means of providing another specific DSL for the system users to define such behavior that depends on the actual values of report form fields. We will pick up this idea again in Sect. 3.3 where we exemplarily discuss the introduction of such a language.

From an abstract language-centric perspective, we have outlined a stepwise top-down transfer of control over a DSL from system design level to system level (cf. Fig. 7). It is easy to see that with increasing control comes increasing flexibility, in exchange for complexity. We have achieved that the report DSL can now be manipulated by the system users. But what would be an adequate way to do so? We argue that whether the increased complexity is manageable for the system users depends on the language they use. Using the instruments of the system developers requires knowledge and skills that span the handling of types and attribute definitions, i.e., knowledge that is traditionally associated with the programming domain. On the contrary, a DSL specialized to the definition of forms can provide an intuitive user interface and at the same time hide most of the complexity (type definitions, etc.) underneath. We might either create a DSL

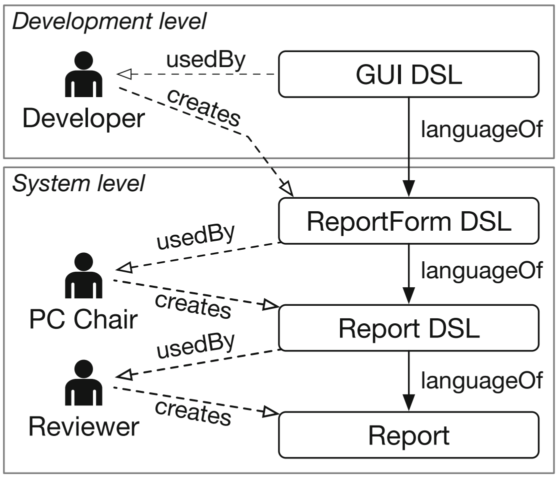


Fig. 7. Top-down DSL creation

by means of a Web form with distinct input fields, or a graphical DSL to build a visual model of the report form. However, both languages can be understood as restricted versions of the DSLs at system design level, as DIME's GUI models already comprise components for this very task. But the restricted versions focus solely on those components that are necessary to define a form by means of only including the different types of form fields that the conference organizers should be able to create.

Please note, the definition of data types must not be part of the report form DSL as there is no need to change the data model explicitly. Required data types as well as the relations between form fields and associated type attributes can be generated, as each form field matches a unique attribute of the already existing data type *Report*. New form fields can trigger the creation of new attributes automatically. However, these semantics have to be defined and assured by the Equinocs developers along with the definition of the form DSL, as the mapping of its language constructs on the applications' data model is still their responsibility.

Increase vs. Decrease of Expressive Power. The report form example describes a stepwise increase of the expressive power of the DSL, reaching from no control at all over configuration-based activation/deactivation of pre-defined form fields up to fully customized field types with respective sets of possible values. The driving motivation is restricting the DSL to ensure that the outcome, i.e., the model created by the user on the next level, represents a manageable artifact that can be integrated into the system. However, we could very well have chosen an opposite approach, i.e., starting with few restrictions and decreasing the expressive power by means of adding constraints on demand. Although we might hand over the full power of a general purpose programming language like Java, in practice, we would strive for a language that is more tailored to the application domain. For the report form example, the GUI DSL in DIME would have been a suitable starting point. With this, users on the system level can create components to extend the report form[10]. However, without additional constraints, they might as well create other UI components apart from those that fit into forms. Hence, it is advisable to restrict the GUI DSL to elements that actually can be integrated into forms, i.e., form fields and other content-related components like texts, images, etc. as well as components to add structure.

The difference between both approaches to create a DSL is apparent. A stepwise increase of expressive power aims at restricting the users of the language as much as possible, initially starting with only a few alternatives to choose from. In contrast, a stepwise decrease strives for providing as much freedom as possible, thereby trying to assure with appropriate constraints that the outcome is still valid by means of being integratable into the system context.

Implications Towards the Meta Level. We have just discussed the creation of a form DSL as a solution for the conference organizers to build their own report forms for reviewers. In this setting, we investigated the Equinocs application on the system level in relation to the modeling environment DIME on the development level. In the next subsection, we switch the perspective to investigate DIME on the system level in relation to CINCO on the development level.

3.2 Automatic Generation of Forms

To keep the example simple, let us consider the initial state of the Equinocs application as described at the beginning of the previous section. Here, changes to the report forms can only be done by the developers of Equinocs, i.e., within the DIME modeling environment. It is easy to see that in this scenario the transformation steps S1 to S3 listed above (changes to the data model, report form and report detail) would have to be repeated (or reverted) for any change requests regarding the extension (or reduction) of the report form fields, i.e., the report DSL. Although the development effort is relatively small, this approach means repetitive work for the developers. Reusing model elements like the UI components for the form fields would only reduce the effort but cannot eliminate the necessity of running through these steps over and over again. This is due to a mismatch between the intent of the Equinocs developers and what can be expressed in the DIME languages available for designing the Equinocs system. These languages let them define a *Report* data type with attributes (Data DSL) and form fields (GUI DSL), as well as *which* form field is related to *which* of these attributes.

However, what they actually need to express is that certain attributes of the *Report* data type are relevant for creating or editing reports and that respective form fields should be displayed at runtime for this exact purpose. But

[10] As Equinocs is a Web application, here we assume that a suitable Web editor for the GUI DSL exists to be easily integrated.

this requires addressing metamodel concepts of DIME's Data language, e.g., by means of a for-each construct in GUI models to iterate over attributes of data types. Additionally, as some attributes might be irrelevant in this specific context, some kind of filter mechanism would have to be established, along with a feature to enhance the Data model with some sort of 'UI cues'. Without going into further detail, we realize that a solution for this issue would definitely require changes to the modeling languages in DIME, i.e., changes on the metamodel level from the perspective of the Equinocs developers. Along with these additional syntactical elements, a model generator that generates GUI models for default forms might save the developers a great deal of work.

From this example, we can see that although the affected levels of abstraction differ, this constellation is very similar to the one depicted above regarding change requests for the report DSL. The only difference is the perspective, as now DIME is on the system level in relation to Cinco on the development level.

3.3 Conference Flow Language

The previous examples both show how the transfer of power over (parts of) a specific DSL can blur the line between development level and system level, as along with control over the DSL responsibilities are shifted from one level to the other. To illustrate this point further, we now introduce a new user role along with a new DSL into the system design process.

In the current implementation of Equinocs, the flow of a conference – i.e., which phases exist, when they are triggered, and what happens in each of them – is determined by the Equinocs developers in those process models that define the application's business logic. Additionally, some variation points are added into these processes, to reflect that the flow slightly differs depending on configuration parameters set by the conference organizers, like the minimum number of reviewers per paper or the decision for allowing a rebuttal phase.

As already discussed, the GUI for setting these configuration parameters can be understood as a DSL for the conference organizers to 'program' the behavior of the conference system. However, this DSL limits conference organizers to the options foreseen by the Equinocs developers and new demands regarding more expressive power result in increased complexity on both the system level as well as the development level. In fact, with more configuration parameters, the conference flow, which is spread across various process models already, becomes increasingly cluttered due to conditional branching to cover all possible conference flow alternatives.

The LDE way to address this problem is to introduce a dedicated DSL that is particularly tailored towards the definition of conference flows together with a code generator that guarantees the requirements on both the system level as well as the development level. In particular, this DSL may comprise constructs to decide on per-paper phase transitions based on the actual form field values of associated reports, as introduced in Sect. 3.1.

This scenario directly leads to the idea of also adding an additional role to the system design process: experts that know the domain of 'running conferences' and thus are suitable to define conference flows. This qualification profile, however, is neither matched by the typical conference organizer – as learning a dedicated DSL to run a conference is something that very few, if any, conference organizers are willing to do – nor by the typical developer, which in our scenario is expert for modeling Web applications with DIME.

Actually, for the future of Equinocs, we foresee that Springer's support team is capable of realizing the conference organizers' needs using such a dedicated flow DSL. But in which stage of the system design process would the definition of the conference flow be integrated? We can find good arguments for integrating this task into the development process, just as we do for shifting it to the system runtime. The best solution might even be to allow it into both levels. This illustrates that drawing a line between developers and users is as difficult as between programming and modeling, and that a corresponding distinction very much depends on the observer's perspective.

4 Conclusion and Perspectives

We have argued that domain-specific tool support has the potential to serve a means to turn descriptive into prescriptive models, and to blur the difference between models and programs, and even between developers and users. Underlying mindset is to view the system development as a decision process which increasingly constraints the range of possible system implementations and Domain-Specific Languages (DSLs) as a means to freeze taken decisions for future development and evolution. The corresponding development paradigm LDE (for Language-Driven Engineering), introduces a novel cooperative development paradigm that is tailored to specifically address all stakeholders, and it emphasizes that what is to be considered a model or a program is a matter of perspective.

We have illustrated the pragmatics of this viewpoint in the light of the development of the Equinocs system, Springer's new editorial service, by showing how the power of a user can incrementally be enhanced to turn her stepwise into a more and more powerful developer. In addition, we have sketched how the development of Equinocs could profit from the evolution of DIME, Equinocs' underlying development environment.

LDE and the imposed viewpoint on specifications, (meta) models, and programs has also reached (even undergraduate) teaching [15]. Experience shows that the traditional distinctions do not at all show up when confronting students directly with our perspective-driven viewpoint.

Currently, we are working on making LDE widely accessible, e.g., by providing a corresponding Web-based development environment [14], and by successively strengthening the domain-specific development support. Following the open-source philosophy, we hope to attract users and co-developers in order to establish an LDE community.

References

1. Boßelmann, S., et al.: DIME: a programming-less modeling environment for web applications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 809–832. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_60
2. 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, M.G.A. (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)
3. Broy, M., Havelund, K., Kumar, R.: Towards a unified view of modeling and programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 238–257. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_17
4. Chatley, R., Donaldson, A., Mycroft, A.: The next 7000 programming languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives, LNCS, vol. 10000. Springer (2018, to appear)
5. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. JetBrains onBoard Online Magazine 1 (2004). http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
6. Felleisen, M. et al.: A programmable programming language. Commun. ACM **61**(3), 62–71 (2018)
7. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages?, June 2005. http://martinfowler.com/articles/languageWorkbench.html. Accessed 10 Apr 2018
8. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley/ACM Press, New York (2011)
9. Gossen, F., Margaria, T., Murtovi, A., Naujokat, S., Steffen, B.: DSLs for decision services: a tutorial introduction to language-driven engineering. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018, LNCS 11244, pp. 546–564. Springer, Cham (2018)
10. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Boston (2008)
11. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms, 1st edn. Springer, Boston (1996). https://doi.org/10.1007/0-387-31005-3
12. JetBrains: Meta Programming System. https://www.jetbrains.com/mps/. Accessed 10 Apr 2018
13. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
14. Lybecait, M., Kopetzki, D., Zweihoff, P., Fuhge, A., Naujokat, S., Steffen, B.: A tutorial introduction to graphical modeling and metamodeling with cinco. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018, LNCS 11244, pp. 519–538. Springer, Cham (2018)
15. Margaria, T.: From computational thinking to constructive design with simple models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018, LNCS 11244, pp. 261–278. Springer, Cham (2018)
16. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
17. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. Softw. Tools Technol. Transf. **20**(3), 327–354 (2017)

18. Naujokat, S., Neubauer, J., Margaria, T., Steffen, B.: Meta-level reuse for mastering domain specialization. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 218–237. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_16
19. Steffen, B., Gossen, F., Naujokat, S., Margaria, T.: Language-driven engineering: from general-purpose to purpose-specific languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives, LNCS, vol. 10000. Springer (2018, to appear)
20. Steffen, B., Margaria, T.: METAFrame in practice: design of intelligent network services. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 390–415. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48092-7_17
21. Steffen, B., Naujokat, S.: Archimedean points: the essence for mastering change. LNCS Trans. Found. Mastering Chang. (FoMaC) **1**(1), 22–46 (2016)
22. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
23. Voelter, M.: Fusing modeling and programming into language-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018, LNCS 11244, pp. 309–339. Springer, Cham (2018)
24. Ward, M.P.: Language oriented programming. Softw. Concepts Tools **15**(4), 147–161 (1994)