Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages

Bernhard Steffen[1(✉)], Frederik Gossen[1,2], Stefan Naujokat[1], and Tiziana Margaria[2]

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

Abstract. In this paper, we present the paradigm of Language-Driven Engineering (LDE), which is characterized by its unique support for division of labour on the basis of Domain-Specific Languages (DSLs) targeting different stakeholders. LDE allows the involved stakeholders, including the application experts, to participate in the system development and evolution process using dedicated DSLs, while at the same time establishing new levels of reuse that are enabled by powerful model transformations and code generation. Technically, the interplay between the involved DSLs is realized in a service-oriented fashion. This eases a product line approach and system evolution by allowing to introduce and exchange entire DSLs within corresponding Mindset-Supporting Integrated Development Environments (mIDEs). The impact of this approach is illustrated along the development and evolution of a profile-based email distribution system. Here we do not want to emphasize the precise choice of DSLs, but rather the flexible DSL-based modularization of the development process, which allows one to freely introduce and exchange DSLs as needed to optimally capture the mindsets of the involved stakeholders.

Keywords: Service orientation · Domain-specific languages
Mindset · DSLs as a service · Software development environments
Software evolution · Product lines · Code generation
Decision diagrams

1 Introduction

"Programming language research is short of its ultimate goal–provide software developers tools for formulating solutions in the languages of problem domains.": This quote appeared in CACM [31] shortly before our final version deadline. It ideally paves the way for establishing our vision and approach. We therefore decided to use the *Language-Oriented Programming* (LOP) approach of the

B. Steffen and G. Woeginger (Eds.): Computing and Software Science, LNCS 10000, pp. 311–344, 2019.
https://doi.org/10.1007/978-3-319-91908-9_17

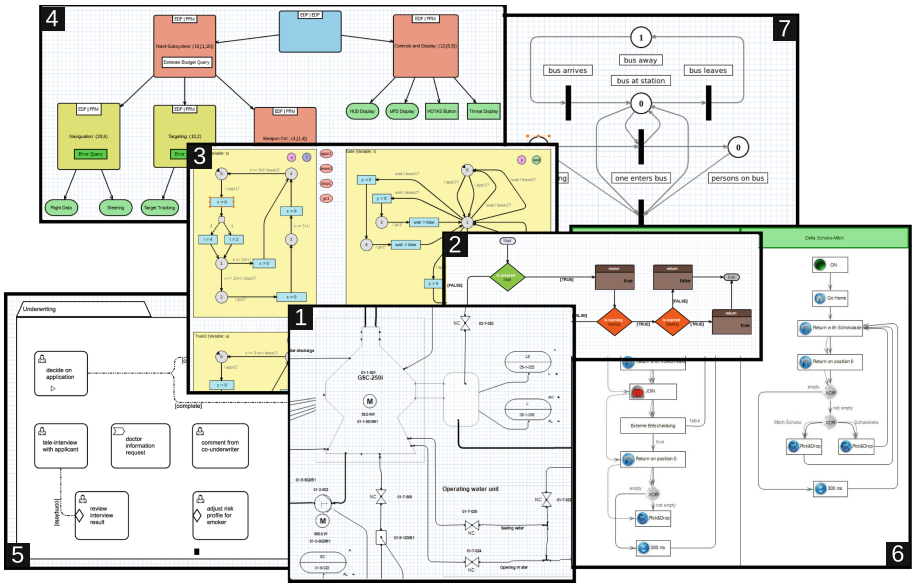


Fig. 1. (1) Piping & instrumentation diagram [107] (2) flow graph [107] (3) probabilistic timed automata [81] (4) hierarchical scheduling systems [26] (5) OMG's case management CMMN [105] (6) EasyDelta pick and place DSL [21] (7) place/transition net [79]

Racket team presented there as a means to highlight some essential features of our *Language-Driven Engineering* (LDE) approach.

It is surprising how different these approaches are despite their similar naming and guiding vision: While our LDE approach aims at enriching typically graphical domain languages[1] like the ones shown in Fig. 1 in order to define an external DSL for which full code can be generated, LOP aims at capturing domain-specific features by establishing tailored internal domain-specific languages (there called *embedded DSLs* or *eDSLs*) on top of LISP/Racket (see, e.g., Fig. 2).[2]

As a consequence, in the LOP approach the addressed software developers are clearly programmers,[3] while it is the goal of LDE to provide tailored (graphical) languages that allow application experts without programming knowledge to act themselves as software developers.

In order to set the scene for our LDE development, we structure the introduction in four parts: a sketch of the vision, followed by a description of the

[1] Which are very popular in practice, as *"pictures are (often) worth a thousand words"*.

[2] The difference between internal and external DSLs can be sketched as follows: an internal DSL is added (e.g. via API functionality) to a host language, which is usually a general-purpose programming language, while an external DSL comes with an own syntax that is completely independent of already existing languages.

[3] Cf. Fig. 2 (reprinted from [16]) for an exemplary Racket eDSL, accompanied by a simple graphical representation for communication with the reader.

```
#lang video

(image "splash.png" #:length 100)

(fade-transition #:length 50)

(multitrack (blank #f)
            (composite-transition 0 0 1/4 1/4)
            slides
            (composite-transition 1/4 0 3/4 1)
            presentation
            (composite-transition 0 1/4 1/4 3/4)
            (image "logo.png" #:length (producer-length talk)))

; where
(define slides
  (clip "slides05.MTS" #:start 2900 #:end 80000))

(define presentation
  (playlist (clip "vid01.mp4")
            (clip "vid02.mp4")
            #:start 3900 #:end 36850))

(fade-transition #:length 50)

(image "splash.png" #:length 100)
```

Fig. 2. A script in the Racket-based *Video language* (reprinted from [16]).

background, the discussed application scenario, and a summary of the contributions of this paper.

1.1 Vision and Approach

We envisage a development paradigm for bridging the *(semantic) gap* [82] of software (system) engineering by allowing all stakeholders[4] to solve their respective tasks using *domain/purpose-specific* languages (DSLs) supporting their established *mindsets*.[5] Technically this requires Mindset-Supporting Integrated Development Environments (mIDEs) that orchestrate the individual stakeholder-specific artifacts and aggregate them to a whole from which entire software systems are automatically generated.

The different mindsets of the involved stakeholders are the main reason for the semantic gap. Making these mindsets precise and allowing orchestration and aggregation of mindset-specific artifacts is a major challenge. Key to the

[4] The various application experts involved, domain modelers, platform and GUI experts, software architects, programmers, etc.

[5] In the following, we assume the established notion of *domain-specific* languages to also comprise the even more specific flavor of *purpose-specific* languages.

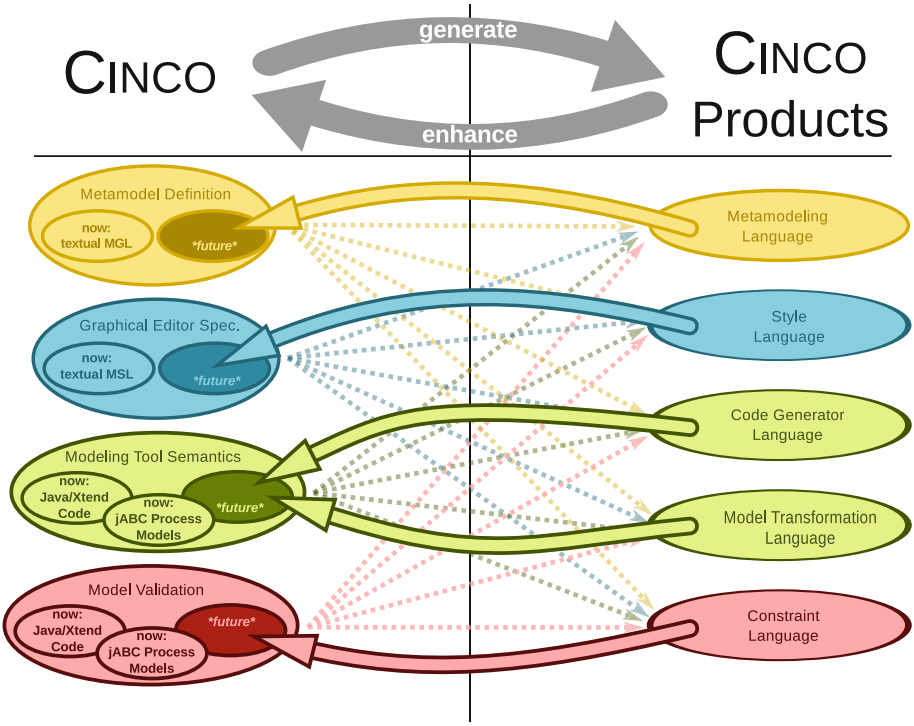


Fig. 3. Enhancing the CINCO framework with CINCO-developed graphical languages in a bootstrapping fashion (reprinted from [80]).

proposed Language-Driven Engineering (LDE) approach is therefore a new class of stakeholders whose task is to generate and maintain the required mIDEs. This task comprises language and code generator design as well as the aggregation of all aspects required to obtain mIDEs that support full code generation.

That multi-DSL design is very promising is also emphasized in [31] where the authors state: "*Large projects often employ a tower involving a few dozen languages, all helping manage the daunting complexity in modern software systems.*" In fact, we believe that the number of graphical DSLs used in the various fields of application easily outnumbers their textual counterparts, and that it is possible to enrich many of these DSLs to satisfy the LDE requirements (cf. Fig. 1). Moreover, we envisage that bootstrapping (cf. [36,50]) will help overcoming the major hurdle for LDE currently perceived: the construction of the required mIDEs. Considering mIDE development as the domain of interest and using dedicated mIDEs for developing and refining mIDEs imposes a natural continuous improvement cycle. Figure 3 sketches this cycle for CINCO, our mIDE for generating mIDEs [80]: the meta-level family of DSLs used to develop the essential aspects of mIDEs are used to create new DSLs for DSL development,

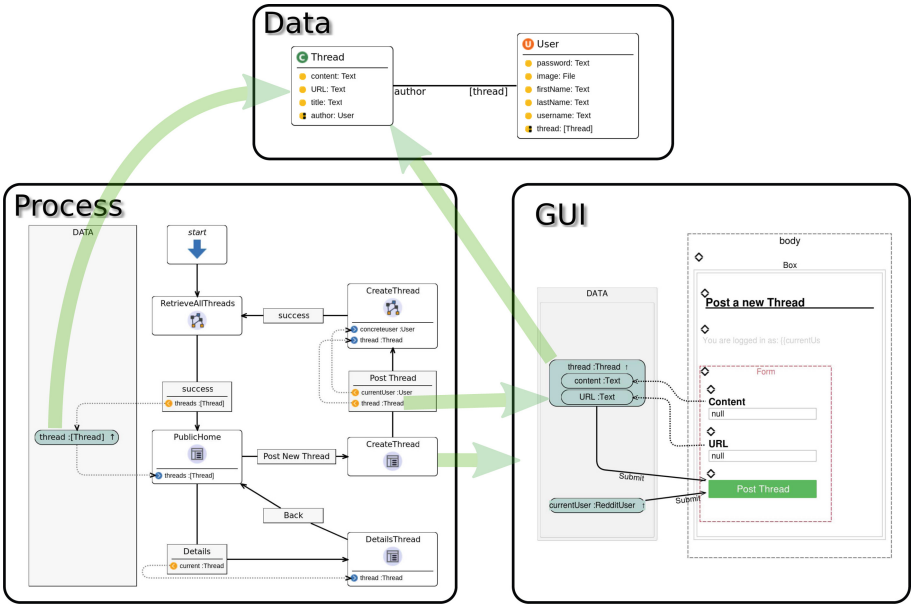


Fig. 4. Horizontal composition of specialized modeling languages for data, processes and GUI as provided by the DIME framework (reprinted from [79]).

which in turn can be fed back into the overall ecosystem of languages in a bootstrapping fashion. Our experience with Cinco [79] is very encouraging.

The semantic requirements for allowing the orchestration and aggregation of the artifacts written in stakeholder-specific languages are in general very complex. This paper therefore concentrates on a *service-oriented* version of orchestration and aggregation of language artifacts as well as, at the meta-level, of entire DSLs.

1.2 Background

There is consensus that modular system development should ideally support *horizontal composition*, so that for example the composition of modules/procedures should not depend on implementation details of the promised functionality, as well as *vertical refinement*, so that for example refinement should be possible without considering the global usage scenario, and also that *evolution* should be decomposed into steps of local impact. The underlying motivation is a general design principle: the more one can rely on things that do not change – called *Archimedean points* in [102] – the better one can control in a separation of concerns manner the change a development or evolution step imposes. In practice, service-oriented development proved practical to support this goal [73,74]: it does not even require a common implementation language.

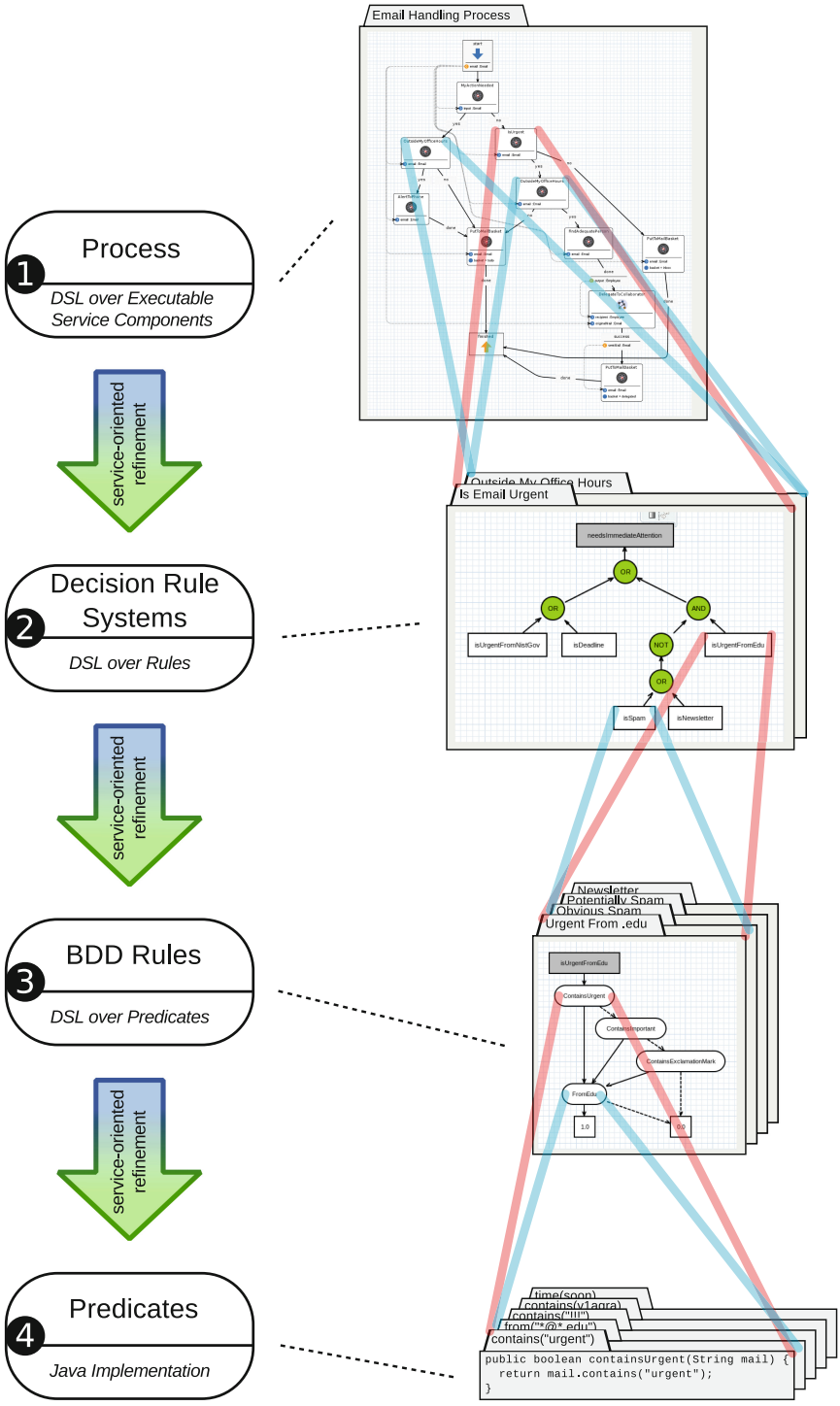


Fig. 5. Vertical DSL-based refinement

The Language-Driven Engineering (LDE) approach presented in this paper hinges on the observation that a similar picture is found also at the meta level, when developing mIDEs, for example using language workbenches [32]:

In a language engineering setting, *horizontal composition* refers to the development of complementary modeling languages, e.g., for processes, data, and GUI. As common practice for the implementation of different (same-level) procedures of a program, the DSLs and mIDEs for modeling data or processes should only be required to support the required meta-level interface, like for example the create, read, update and delete operations that define the interplay between data and process models. For instance, Fig. 4 illustrates how DIME [22,23], our mIDE for graphical modeling of Web applications, uses in each model type dedicated model elements that *represent* entities (in this case the thread entity) of other (horizontal) same-level models, this way guaranteeing the type-correct relation between models of the different languages.

Vertical refinement, on the other hand, addresses cascading domain-specific languages that become increasingly more specific and therefore more powerful and safer/easier to use for the corresponding stakeholder. This is illustrated in Fig. 5, where for example the language for decision rule systems is used for service-oriented refinement of the process language.

Finally, the *evolution* of mIDEs should – like in modular system design – follow the locality principle. The consideration of DSLs as units of evolution and exchange establishes a level of locality that is new and characteristic for LDE. For example, in Sect. 4.2 a BDD-based DSL for modeling binary decisions is (simply) replaced by variants of DSLs for fuzzy logic.

1.3 LDE Application Example

The essence and impact of the proposed *service-oriented* form of DSL composition, refinement, and evolution will be illustrated along the stepwise development and evolution of a profile-based email distribution system (Sect. 2). Service-oriented means here that DSLs are treated themselves as units of evolution and exchange during a hierarchical mIDE development, i.e., DSLs are treated as services themselves. For vertical refinement and evolution, the focus of this paper, the situation is as follows:

- Vertical DSL refinement is illustrated in Fig. 5: the decision nodes of the process model are refined in three hierarchical steps:
  1. the decision nodes of the process model are refined using a DSL for Decision Rule Systems: a decision is expressed as rules in this DSL,
  2. the leaf nodes of a Decision Rule System are refined by decision diagrams: the domain concepts are expressed as a BDD over primitive concepts that are domain-specific predicates, and
  3. the internal nodes of the decision diagrams are refined by native calls implementing the required predicates: a predicate is “evaluated” by executing native Java code.

– Evolution is illustrated in Fig. 9 (cf. Sect. 4): it shows the extensive reuse when changing the DSL for decision rules from Binary Decision Diagrams (BDDs) to Algebraic Decision Diagrams (ADDs) [18] for fuzzy logic and to n-ary decisions.

The setup is chosen to illustrate the role of the diverse DSLs addressing different concerns, ranging from ease of use for the involved stakeholders to efficiency. In fact, graphical process modeling languages have proved to be successful in involving non-programmers, the graphical syntax diagram notation is convenient for specifying logical combinations of arbitrary predicates, and decision diagrams are a de facto standard for the efficient treatment of Boolean functions.

These quite diverse DSLs address a wide range of intents and mindsets of professionals. E.g., (business) process experts are meant to directly 'draw' their intended 'workflows', and rule designers should be able to combine elementary decision rules to more complex ones without worrying about implementation details or efficiency. Moreover, decision diagrams are an intuitive representation for small predicates/rules, and also a scalable means to generate highly efficient code via their underlying well-known minimization technology (cf. [1,90,91]).

Our goal is to provide all stakeholders with a means to express their desires in terms of what they want to achieve (WHAT/Requirement level), without worrying about possible ways of realization (HOW/Implementation level). It should be clear that the WHAT level of one stakeholder may well be a HOW level for another stakeholder. E.g., BDDs are certainly at the HOW level for process and rule experts, while they are at the WHAT level for someone who is responsible for efficient rule implementation. Likewise, Java may well be the HOW level for the rule implementors, whereas it is typically WHAT level for someone who is responsible for code generation.

The case study focuses on vertical refinement and DSL-based system evolution during the development of a basic email distribution system and highlights the role of the service-oriented interplay between the various DSLs thus illustrating DSL-based refinement. Along Fig. 5, the provided dedicated DSLs address

1. process experts, here through a simple flow chart language to model under which circumstances which email handling criteria should be applied (see Sect. 3.1),
2. rule engineers, here through a simple predicate logic for modeling the required email distribution criteria according to a given email profile (see Sects. 3.2 and 3.3),
3. algorithmic experts, for guaranteeing performance on the basis of BDD/ADD technology,
4. programmers, to realize the profile extraction from incoming mails, e.g. in Java (see Sect. 3.4), and also
5. meta-modeling experts, for providing the required mIDEs including their required code generators, e.g. using the CINCO framework [79,80].

Subsequently, Sect. 4 illustrates the DSL-driven evolution process which is designed to clearly separate concerns and to control potential feature interaction in a two-dimensional fashion: At the meta level to generate the required

new mIDEs and at the object level to use these mIDEs for modeling the new functionality (cf. Fig. 9).

1.4 Contribution

The LDE paradigm proposed in this paper supports division of labour among different stakeholders on the basis of stakeholder-specific languages. Key to this approach is to accompany system development with the development of dedicated mIDEs: the conceptual decompositions typical for system development are coherently matched by the provision of adequate mIDEs for the stakeholders, so that they can contribute to the actual system development in their mindset. The required parallel maintenance of DSLs and system models is particularly elegant as long as a *service-oriented* style of DSL composition, refinement, and evolution is adopted.

LDE allows all the involved stakeholders, including the application experts, to directly contribute to the system development and evolution using dedicated DSLs for their level of expertise. This approach establishes three levels of reuse, each supporting a specific kind of LDE stakeholder:

- mIDE-based development provides an intrinsic simplification that manifests itself as a powerful form of IDE support that goes beyond the usual IDE support for generic programming. Being domain-specific, it can exploit domain knowledge like "this has to run on a certain platform" to generate the entire running system from a purely functional description developed with a (stakeholder-specific) DSL. The task "get it to run on the platform" is materialized as an mIDE artifact using another, dedicated DSL, and thus can be simply stored and factored out for future reuse.
- DSLs are considered as units of reuse during mIDE construction. For example, a DSL for GUI design may be reused independently of the platform (e.g., Web, mobile phone, etc.) where the actual user application runs.
- mIDE development environments can be reused for the construction of DSLs and further mIDEs. For example, our meta tooling suite Cinco [79] can be (re)used to generate graphical domain-specific development tools, or even entire mIDE. In particular, the DSL for DSL specification underlying Cinco and its corresponding code generator have been reused over and over again. This kind of reuse is particularly interesting for its bootstrapping effect: an mIDE for code generation generated with Cinco may well become part of Cinco itself, as described in [80] and illustrated in Fig. 3.

LDE is based on a very general notion of service: a service is a means to bridge the various HOW/WHAT gaps in system development, and it stands simply for any form of aggregation of technical detail to a new (behavioral) unit that serves a specific (user-defined) purpose or concern at the higher level. Typical examples of such services are the decision functions of the email process, which are implemented using DSLs for rule composition and predicate definition (cf. Sect. 3).

Such services allow for a more abstract level of system composition, where certain lower level concerns are already taken care of inside the service components.[6]

LDE goes a step further by considering DSLs as services themselves, namely as meta-level services that can simply be used for mIDE design in order to provide stakeholders with their mindset-specific mIDE. This way, the interplay between different mindsets during system development and evolution can be supported in a *DSL as a Service* fashion as illustrated in Sect. 4. The essence of the corresponding system refinement and evolution happens indeed at the meta level. For example, the change of mindset required when moving from Boolean to fuzzy logic is entirely taken care of by a corresponding (service-oriented) exchange of DSLs of the corresponding mIDE (cf. Sect. 4.2).

Whereas the elegance of achieving user-intended solutions is a consequence of the purpose-specific WHAT perspective, efficiency hinges on additional HOW knowledge expressed in a language whose purpose is to support efficient implementation. BDDs and ADDs are good examples for such efficiency-oriented DSLs with high potential to be frequently reused as a service.

Language-Driven Engineering (LDE) aims at enabling a new level of cooperative system development whose support does not end with the deployment but continues throughout the systems' life cycle by continuously providing all the involved stakeholders with languages tailored to their own task and its corresponding mindset. The goal is a consequent separation of concerns that allows developers to focus on required functionality, while trusting that the remaining issues, like performance, security, or platform-specific anomalies, are already taken care of or delegated to dedicated experts. As discussed in [79,107], providing dedicated DSLs allows the involved stakeholders, including the application experts, to directly participate in the development process. For example, in the project described in [107], experts in industrial fluid processing could be involved thanks to a specialized DSL resembling piping and instrumentation diagrams, and electrical engineers thanks to another specialized DSL expressing connectivity on the basis of cabinet layouts familiar to them.

All the mIDEs of all DSLs used in our example are generated with the CINCO meta-modeling framework [79], which is open source and available on GitLab[7]. Also the ADD-Lib framework for dealing with decision diagrams is open source, so readers may replicate entirely the development described in this paper[8].

In the remainder of the paper, Sect. 2 sketches LDE along the above introduced application scenario that we will use to illustrate the pragmatics and impact of LDE, before Sect. 3 presents the pragmatics of system development via DSL-based decomposition and Sect. 4 discusses the potential of system evolution via DSL introduction and exchange. The paper closes after a discussion

[6] This notion of service is more general than the very constrained notion proposed by the Web service community, which is typically directly linked to the use of certain specialized technologies and protocols, nor is it necessarily linked to the component view of today's popular service-oriented architectures.

[7] https://gitlab.com/scce/cinco

[8] Please see the LDE case study at [2].

of related work and potential application fields in Sect. 5 with our conclusions and future perspectives in Sect. 6.

2 Example-Based Sketch of LDE

We will show, how a simple service for the automatic extraction of urgent emails from a stream of incoming emails using binary logic can be evolved in an LDE-fashion towards an efficient distribution service based on fuzzy logic. Here, vertical DSL-based decomposition will be used to improve the scalability and performance of the basic email distribution system (cf. Sect. 3), and DSL-based evolution will be useful for treating different variants of fuzzy rules (cf. Sect. 4). Figure 5 summarizes the vertical decomposition using four DSLs:

1. At the top level, processes allow for a user-level definition of the business logic. Such process descriptions are comparable to languages like BPMN [86] and UML's activity diagrams [85], but additionally provide full code generation. We use here a specialized variant of the process language from the DIME framework [22] which provides fully model-based development of all aspects of a multi-user single-page web application[9]. As the DIME processes are the technological successors of the *Service Logic Graphs* (SLGs) of the jABC framework [84,100], they comprise building blocks for the inclusion of executable services, and these building blocks are connected according to their flow of control. Included services provide functionality for, e.g., putting emails into (named) baskets and email forwarding, as well as profile-based decision services – the starting points for the vertical decomposition described next.
2. Decision services are components 'simply used' within the process layer. Of course they could be directly implemented in code, however we aim for a user-level definition of the decisions and introduce a domain-specific language that allows to define *Decision Rule Systems* as compositions of *Decision Rules*. These rules are again provided in a service-oriented fashion: Without needing to know how exactly such rules are realized/implemented, users can combine them with simple logic operators in a graphical language.
3. Binary decision diagrams [30] (BDDs), the level 3 components of Fig. 5, are a common graphical language for decision modeling. They are intuitively defined and understood, and form a research field on their own. Thus, algorithms for optimization, minimization, etc. are widely available.
4. Rules (and their compositions) are based on a *profile* of the processed email, which comprises various predicates resulting from the analysis of the email's body, header, and other metadata. We decide to break the chain of graphical user-level languages at this point, and allow for the inclusion of arbitrary predicate implementations given in Java: 'general-purpose programming' seems to be now an adequate 'domain'. Of course, we could have also included further DSLs, e.g. using regular expressions to model text matching.

[9] The full DIME framework comprises various languages for the web domain, all spanning the horizontal dimension of our LDE approach (cf. Fig. 4).

In addition to this vertical dimension, each level may as well have its own dimension of in-language hierarchy. For example, an executable service component in a process can be a process itself; a rule in a rule system can be itself a composite rule system; a predicate/variable decision in a BDD can be another BDD, and the implemented Java method has access to the full language potential of structuring the program: method calls, classes, libraries, etc.

The second dimension, the DSL-based evolution, is characterized by a generalization of BDDs to Algebraic Decision Diagrams (ADDs) in order to adequately treat fuzzy rules. This generalization can easily be extended to also comprise multi-basket capability. Our corresponding two-step definition of decision functions defined as composite rules made of BDDs (cf. Sect. 3.2), which might look a bit artificial in isolation, is ideal to illustrate the DSL-based WHAT/HOW interplay outlined above:

- It establishes a logical layer for hierarchical WHAT-level reuse in a service-oriented way: whatever library of domain-specific rules are available, this layer supports their combination with the typical logical operators. In the refined settings arising during the system evolution (cf. Sect. 4), this layer will comprise more general algebraic operators.
- It establishes a HOW-level for performance optimization: logical combinations of BDDs can be 'partially evaluated' to obtain a redundancy-free representation in terms of BDDs that guarantees that every embodied predicate is evaluated at most once at runtime. The impact of this optimization is particularly striking for the refined setting where 'fuzzy' domain-specific predicates (cf. Sect. 4.1) are represented as ADDs.

We will see that the service-oriented interplay between these two layers eases the system evolution process by keeping the changes at the different modeling layers and at the code generators to a minimum.

3 Vertical DSL-Based Decomposition

The pragmatics of vertical DSL-based decomposition concern in particular the support of the service-oriented interplay between the involved DSLs in order to establish a clear separation of concerns and of the cooperative development using stakeholder-specific mIDEs. We discuss the 4 level stepwise refinement from the top-level perspective of the global (distribution) process, via two DSL-supported layers for decision rule definition to a programming layer for implementing so-called *native services*. Besides easing the involvement of stakeholders with different backgrounds, these four layers are meant to decouple/modularize the system in such a way that the impact of later evolution steps is localized, an effect discussed in Sect. 4.

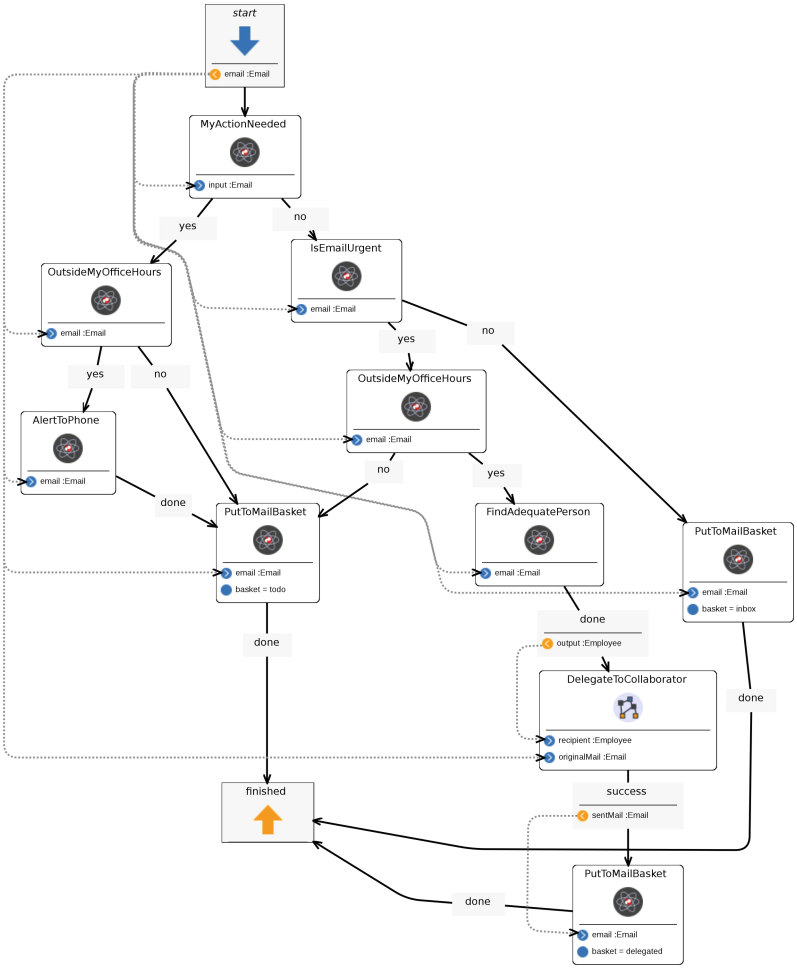


Fig. 6. Email handling process in DIME

3.1 The Global Email Distribution Process

The global process layer is conceived to allow the customers of the envisioned email distribution system to describe their desires at a WHAT level: the language refers to notions from the user perspective and, in particular, it does not require any programming knowledge. Process descriptions must also be precise enough to enable full code generation once the service-oriented refinement at the other three layers is complete. Figure 6 shows the DIME process model of the email handling process. This model is self-explaining, and it can be even constructed by non-programmers using DIME's easy-to-use component libraries with little training.

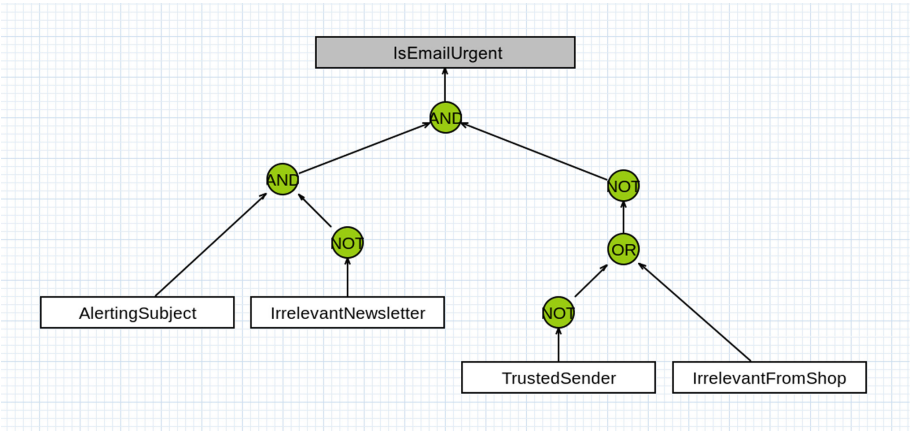


Fig. 7. IsEmailUrgent: a composed rule in the DSL for the composition of Binary Decision Diagrams.

Customers are usually presented with palettes of ready building blocks and use them in their process definition. A customer may also customize or extend the DSL by 'inventing' new components, name them to reflect their domain-related intention and perhaps add some documentation. Such atomic building blocks can be directly used in a process model, and be refined and implemented later on. In practice, process modelers start with the available palette of building blocks and propose new building blocks only at need, a typical case during system evolution. Such top-down/bottom-up interplay is characteristic for service-oriented refinement and it occurs at all DSL layers.

Service-oriented refinement substitutes the building blocks definition with one or more corresponding implementations, the key enabling mechanism of full code generation. The following three subsections illustrate this refinement for the decision services, in particular the building block that decides whether an email is urgent. This refinement is organized in three levels of DSLs:

- Syntax diagrams for the (propositional) logical combination of elementary decision rules,
- BDDs for the definition of elementary decision rules on the basis of elementary predicates, and
- Java for the implementation of elementary predicates.

We will show how this service-oriented decomposition enables cooperative development, supports reuse, and eases evolution.

3.2 A DSL for Rule-Based Composition of Decision Services

The decision services occurring in the process model are defined by the corresponding stakeholder as logical compositions of decision rules[10] that are already available or to be implemented later. Figure 7 shows the inner structure of the "IsEmailUrgent" decision service as a logical composition of five decision rules.[11] The syntax tree-like look of the DSL is easy to handle and to read, even for hierarchical decision service definitions. In the corresponding propositional logic-based mIDE, users specify their intents simply by drag and drop from component libraries. If a new rule is needed, users may introduce a placeholder and directly use them. Such placeholders are requests for new rules, to be handled by the responsible experts who turn the 'partial' specification into a complete specification for which full code can be generated.[12]

Using this mIDE to compose rules is quite easy after a short introduction. Understanding the logical impact of the rule compositions on the other hand is by no means trivial and requires expertise, or an adequate mindset. Our mIDE helps building up this mindset by easing experimentation and providing immediate feedback for the specifier, to check whether the intents were met. The decision services can be simulated and logical inconsistencies can be automatically detected. In particular, the consistency check is a great help, as inconsistencies in decision service definitions can be just flaws, but may also point to a major misconceptions.

DSLs and their corresponding mIDEs are powerful means to factor out tasks for a specific domain and provide support to solve them once and for all. This support can be the stronger the more specific the DSL. E.g., while for a generic programming language it is difficult to characterize inconsistency, for appropriately defined DSLs, like the BDDs introduced in the next subsection for rule definitions, this is very easy.

3.3 A Language for Efficient Decision Rule Implementation

Intuitively, decision rules are 'if-then' specifications[13] that describe under which circumstances which action/decision has to be taken. In the binary case decision rules are formally Boolean predicates. In the evolution steps we will use more

[10] In this basic setting, the decision rules are (predefined) predicates and the logical combination considered in this section is just a simple means for a hierarchical specification of more complex predicates. Why we chose to call these predicates rules will become clearer at the lower level and in view of evolution, in Sect. 4.

[11] The composition model is obviously not minimal. Instead, it reflects the individual user's mindset. It will be optimized automatically during code generation.

[12] In a pure top-down development from scratch, no library components are available: They must be introduced as part of the specification and subsequently refined. In practice, after some time most of the required components can be drawn from libraries and only a few need to be introduced.

[13] Popular representatives of decision rules are Event Condition Action Rules [14,29] or weighted rules [37].

general rules: Fuzzy rules, tailored to deal with uncertainty, and n-ary rules supporting decisions with more than two outcomes.

In our example, we use as DSLs for rules definition Binary Decision Diagrams (BDDs, [24])[14] and corresponding generalizations. BDDs represent decision trees as minimal directed acyclic graphs (DAGs) whose nodes are associated with Boolean variables or predicates, whose two outgoing edges encode the outcome of the predicate evaluation, and whose leaves are the Boolean constants TRUE and FALSE, denoted by 1 and 0, respectively. Given a fixed order of the variables resp. predicates, BDDs are canonical normal forms. Formal definitions and details are available in [30].

While huge BDDs have been used for decades as boolean encodings for hardware verification, SAT solving, and similar machine-managed representation domains, small BDDs as those shown in Fig. 8 are well suited to establish domain-specific libraries of (elementary) rules at the WHAT level: the meaning of these BDDs is intuitively clear also for unexperienced users.[15] Once the implementation code for the involved elementary predicates is available, the BDDs shown in Fig. 8 are sufficient to generate fully executable code for the composition of Fig. 7.

In the seminal paper [24], BDDs were established as an efficient data structure for Boolean functions $\mathbb{B}^n \rightarrow \mathbb{B}$, with efficient logical operators to evaluate a complex formula to a single result BDD. The formula corresponding to the syntax tree of Fig. 7 evaluates to the BDD shown in Fig. 11(a) using the BDD definitions of the single predicates shown in Fig. 8. This evaluation technology allows one to generate code whose performance can be hardly achieved via manual coding. The canonical nature of the BDDs eases many frequent analyses: e.g., checking functional equivalence of expanded BDDs reduces to rooted DAG isomorphy, and inconsistent formulas are reduced to the one-node BDD FALSE, a particularly handy property when dealing with large rule compositions.

3.4 Implementation of Elementary Predicates

Elementary predicates like those extracting certain characteristics from incoming emails, may well be implemented in Java, here considered the 'generic DSL' for everything where there is no specific DSL support. An important feature of service-oriented refinement is in fact that it allows one to link to programming languages at any point during the refinement process without harm. In facts, refinements typically end with implementations of elementary services in a generic programming language. The level at which one decides to turn to generic programming may change in the course of a larger project. For example, one could later decide to introduce regular expressions for text matching as a new DSL layer. Service-oriented refinement is ideal to support this form of evolution.

[14] BDDs earned their fame more on the HOW level, where they support amazing optimizations, at a small scale they are quite intuitive even for unexperienced users.

[15] We adopt the de facto standard graphical representation for BDDs where a solid edge represents a node's then-successor, and a dashed edge its else-successor.

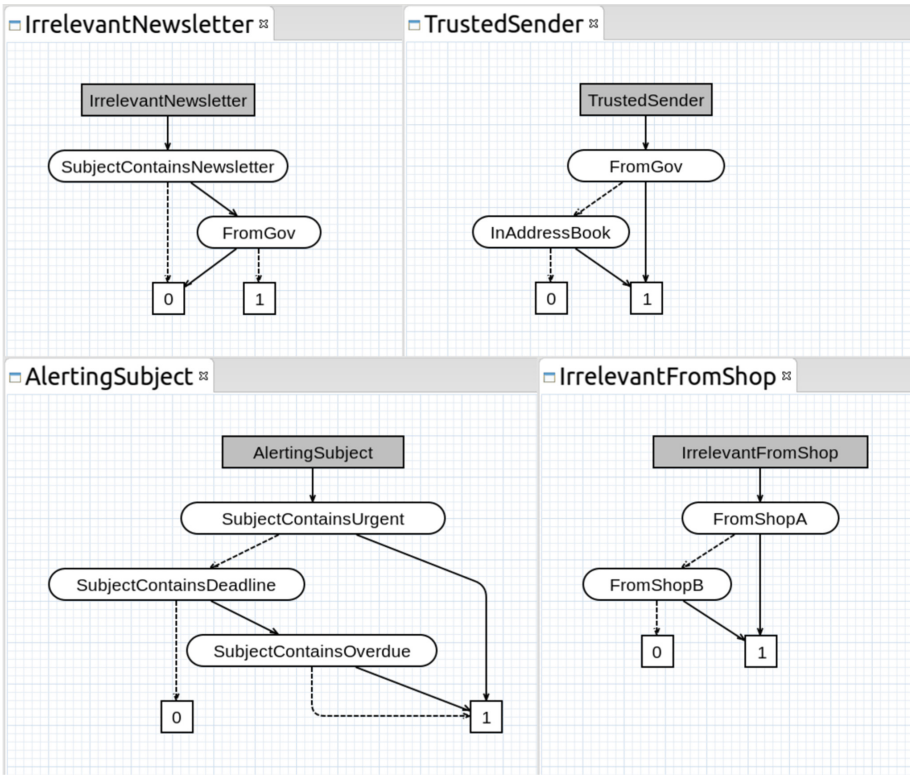


Fig. 8. Elementary classification services for urgent emails specified in the DSL for Binary Decision Diagrams as BDD rules

4 DSL-Based Evolution

We already saw one DSL-based evolution step: the evolution to optimized BDDs sketched in the transformation indicated by Fig. 9(a) illustrates that the HOW-knowledge about the BDD-based decision rule realization can be used for optimization purposes, by partially evaluating the expression that defines the logical composition of the decision rules. The realization of this optimization is almost for free: the rule partial evaluation step needs to be implemented as a new code generator[16], but the entire grey part remains unchanged, constituting an Archimedean point.

[16] This is quite simple, using available open source libraries [1,90,91].

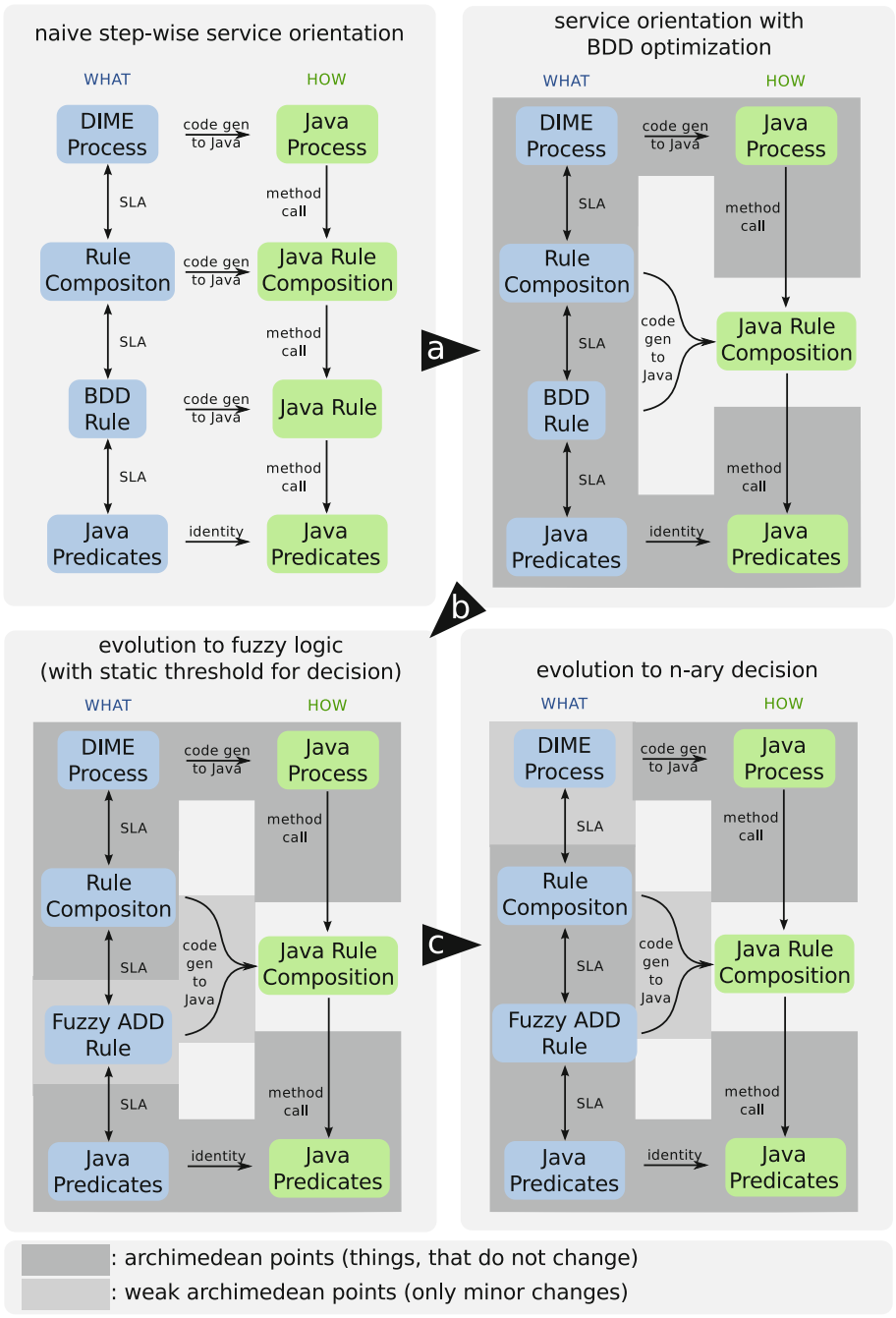


Fig. 9. Three steps of DSL-based evolution: from the initial implementation to optimized and fuzzy n-ary decisions. In this figure, SLA refers to a meta-level variant of service-level agreement.

Two restrictions of the modeling with propositional logic (and BDDs) that can be quite severe in practice are addressable with evolution to new DSL specializations:

- Decision rules for our example are often not strict: criteria like keywords, subjects, or origin are typically only indicators for urgency, and the more of such indications apply, the stronger becomes the indication and the corresponding decision support. Fuzzy rules are an adequate technique to address this issue, and ADDs provide an efficient realization technology (HOW level) that only requires very local changes (cf. Fig. 9(b)).
- Many applications mandate to decide between more than two alternatives. In our example, to distinguish more levels of urgency of incoming email requests requires generalization, for which ADDs turn out to provide the technology of choice. In contrast to the previous two evolution steps, this generalization also requires a change at the process level, as the decision service has now more than two outgoing edges that need to be adequately connected in the process graph (cf. Fig. 9(c)).

We will now sketch how ADDs generalize BDDs and then describe the two evolution steps displayed in Fig. 9(b) and (c). They illustrate the principle of service-oriented refinement with its corresponding high potential for reuse, inter stakeholder cooperation, and Archimedean point-oriented evolution [102].

4.1 ADDs for Dealing with Uncertainty

BDDs are compact representations of decision structures based on Boolean algebra:

$$\mathcal{A}_{bool} := (\mathbb{B}, \{\wedge, \vee\}, \{\neg\})$$

yielding optimized evaluation structures for expressions over a set $\mathbb{B}$ that use the binary operators $\wedge, \vee$ and the unary operator $\neg$.

It is straightforward to lift this pattern of evaluation structure to any algebraic structure consisting of a set $\mathcal{S}$ together with a carrier set of binary operators $\mathcal{O}_b$ and a set of unary operators $\mathcal{O}_u$, resulting in the Algebraic Decision Diagrams (ADDs). The CUDD package [90,91] is a prominent C library that includes ADD support. ADDs are mainly used for arithmetics, i.e., for algebraic structures supporting integer computation $(\mathbb{Z}, \{+, *\}, \{-\})$ or floating point computation $(\mathbb{Q}, \{+, *\}, \{-\})$.

In contrast, our evolution steps use two simple fuzzy logics given by the following two algebras: a min - max algebra

$$\mathcal{A}_{fuzzy} := ([0,1], \{\wedge_f, \vee_f\}, \{\neg_f\}) \tag{1}$$

$$\text{with} \quad a \wedge_f b := \min(a, b) \tag{2}$$

$$a \vee_f b := \max(a, b) \tag{3}$$

$$\neg_f a := 1 - a \tag{4}$$

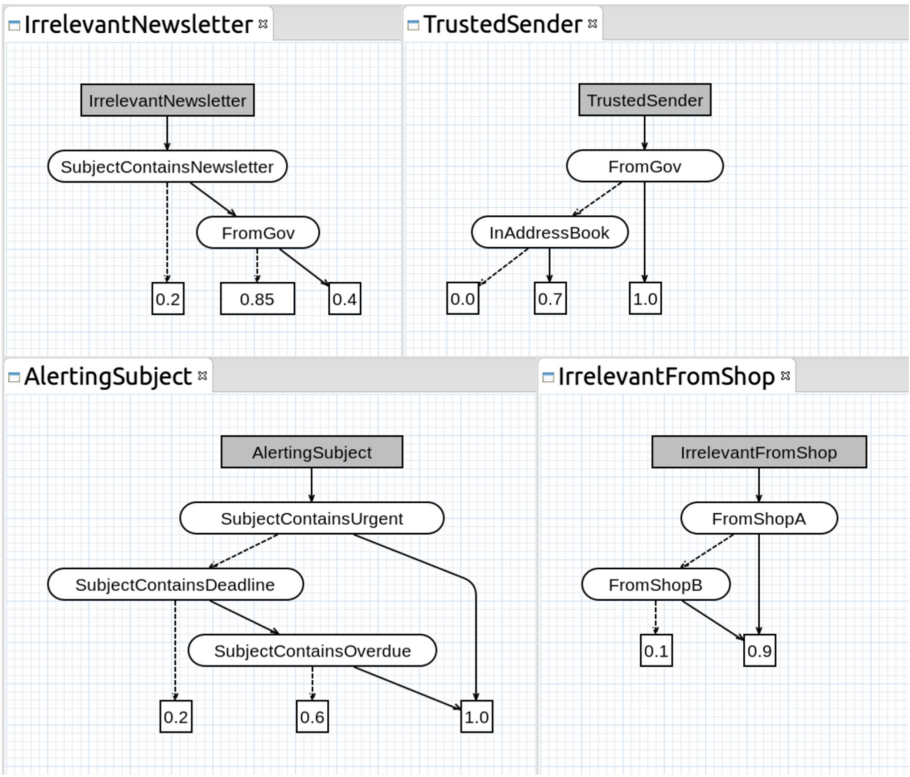


Fig. 10. Classification services for urgent emails specified in a Fuzzy purpose-specific language for Algebraic Decision Diagrams

and an algebra with a probabilistic interpretation of $\wedge$, $\vee$, and $\neg$

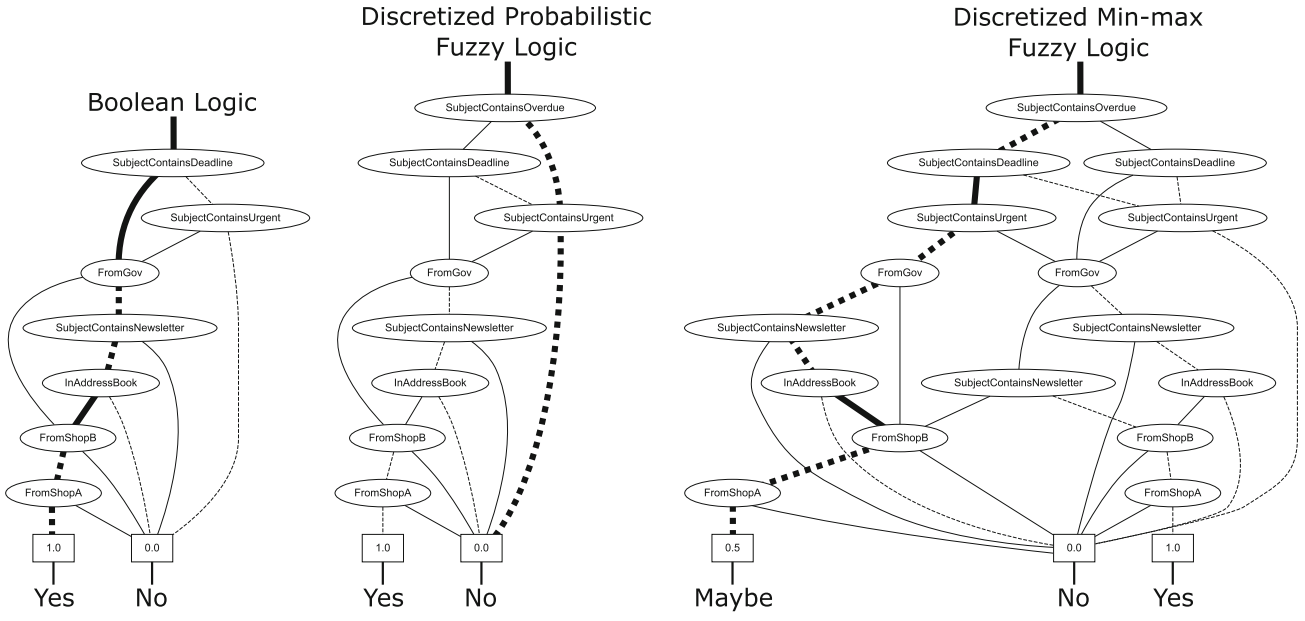
$$\mathcal{A}_{prob} := ([0,1], \{\wedge_p, \vee_p\}, \{\neg_p\}) \tag{5}$$

$$\text{with} \quad a \wedge_p b := ab \tag{6}$$

$$a \vee_p b := 1 - (1-a)(1-b) \tag{7}$$

$$\neg_p a := 1 - a. \tag{8}$$

Several other variants can deal with uncertainty, all with their specific strength and weaknesses, and we do not claim this choice to be optimal. Instead, we want to show that service-oriented refinement is an ideal means to switch between such options depending on which variant is most adequate. Such a switch is not just a matter of easing the development: each choice comes with a specific mindset, and it is the role of DSLs to provide mindset-specific support.



(a) hard Boolean interpretation

(b) probabilistic interpretation

(c) min/max interpretation

Fig. 11. Generated Algebraic Decision Diagrams for an email classification service as composed by the model in Fig. 7. From the left to the right: Standard Boolean Logic, Probabilistic Fuzzy Logic (Thresholded at $\frac{1}{2}$), and min/max Fuzzy Logic (Thresholded at $\frac{1}{3}$ and $\frac{2}{3}$) with thresholded min/max logic compared to standard Boolean logic. The highlighted path defines the decision made for an email that was sent from a sender in the address book and that mentions "deadline" but not "overdue".

4.2 Dealing with Uncertainty

Rules for decision support typically provide recommendations rather than strict knowledge: certain senders or subjects of an email are often only indications for, e.g., urgency. By allowing fuzzy rules as in Fig. 10, the values at the leaves indicate the level of certainty. The corresponding evolution of our email handling process is described in Fig. 9(c) and requires only little effort:

- the DSL for the rules is generalized to allow for more than two leaf nodes and floating point values in $[0, 1]$. This is an easy task using Cinco.
- we do not intend to touch the structure for rule composition, so only the semantics of the operators needs to be adapted to reflect the chosen variant of fuzzy logic. In our current implementation this is done in Java, consistent with our decision to end the DSL refinement at this level. However, we intend to also provide a corresponding DSL as part of the ADD-Lib [1], our framework for ADD-based modeling.
- the partial evaluation of the composition structure connecting the individual rules is implemented using the ADD-Lib [1] based on the CUDD library. After discretization into two, respectively three categories it results in the three kinds of ADDs shown in Fig. 11: (a) for the hard Boolean interpretation discussed earlier, (b) for the probabilistic interpretation, and (c) for the min/max interpretation. The only required change to the code generator is the threshold-based discretization of the ADDs, in order to allow n-ary branching in the process models.
- the decision to place the threshold-based interpretation of the leaf values in the code generator requires a change of the process model only in case the discretization distinguishes more than two categories.

Actually, we did not just evolve the email handling system, but also its corresponding mIDEs: the described changes, which concern the rule DSL and the threshold-based interpretation only, are, in fact, at the meta level. Using Cinco, we are able to fully automatically generate the two new mIDEs for the min/max and the probabilistic interpretation after slight variations of the meta model for BDDs.

The Gain of Threshold-Based Decision. To illustrate the nature of explicit uncertainty modeling for both the min/max and the probabilistic interpretation we consider two scenarios: binary decisions with a threshold of $1/2$ and a ternary decision with thresholds $1/3$ and $2/3$ which models a separate treatment of unclear cases. Figure 11 illustrates the impact for our example:

- The min/max interpretation does not add anything new to the binary case. In fact, its aggregated decision diagram coincides with the BDD shown in Fig. 11(a). In contrast, the probabilistic interpretation shows a difference (cf. Fig. 11(b)), as, e.g., small uncertainties are amplified in conjunctions.
- The min/max interpretation makes a difference in the ternary case as shown in Fig. 11(c). In fact, in the ternary case, our example path distinguishes all

three interpretations, as the probabilistic interpretation would also result in 0.0 in this case.

Adequacy and mindset of these interpretations are quite different, and none is universally superior to the others. Thus easing the context-specific choice is important.

5 The LDE Landscape

Considering the history and context of the LDE approach, we sketch the roots of LDE (Sect. 5.1), then discuss its related work (Sect. 5.2) and its connections to the work presented in the other contributions of this volume (Sect. 5.3).

5.1 LDE: The Roots

The first direct experience with the power of DSL-based mindsets came with the attempt to prove the optimality of an algorithm for partial redundancy elimination [77]. Thinking in terms of temporal logic and thereby directly in terms of the desired (temporal) properties rather than in terms of fixpoint computations as it was common at that time, radically changed the mindset. It led to a drastically shortened proofs and later allowed us to elegantly solve two important open related problems: the optimal reduction of register pressure [56–58,87],[17] and an algorithm for eliminating all partial redundancies [94]. Essentially, this was due to the compositionality of temporal logic specifications, in particular concerning conjunction.

This context also bore the idea of introducing corresponding code generators [89,92,93]. Similar to our choice to use the well established CUDD tool for the accompanying example of this paper, that code generator was based on a pre-existing model checker. In fact, full code generation became a central objective throughout all the further developments.

The idea of using components, called Service-Independent Building Blocks (SIBs) by the ITU-T Standard [46,47], which can easily be recombined due to the simplicity of their interfaces, was motived by the fast growing library of special commands for the Concurrency Workbench [28]. SIBs, semantically characterized using simple taxonomies for classification could profitably be used to automatically synthesize tailored command sequences from small temporal logic specifications [68,99], a technique later also used for automatic mediator synthesis [67].

The SIB concept combined with taxonomic classification and model checking also became the heart of a very successful industrial cooperation resulting in the Siemens Nixdorf INXpress Service Definition Environment for Intelligent Network value added services [97,98]. Their evaluation of our technology revealed

[17] This algorithm, which can be generated from a four line CTL specification, is now a standard for optimizing compilers, as it is both more efficient and more powerful than its competitors.

a time to market reduction of the services of a factor 5! This success drove us to transform our corresponding development environment, the METAFrame tool [96], stepwise into a more general IDE for application development called Application Building Center (ABC) and later jABC [100] when we moved from the C++ implementation to Java.

The easy service-oriented definition and exchange of functionality turned out to be a good way of communicating between the stakeholders [74]: with all the stakeholders working on the same artifact (the *One Thing* as we called it [70, 101]) but at their dedicated level of abstraction defined by the underlying service hierarchy[18]. jABC's full code generation philosophy, which avoids typical round-trip problems, maintained controllability during the entire life-cycle of a system at the modeling level [62,71]. jABC's model checking and model synthesis facility, in addition, provided dedicated support for logically controlling evolution and establishing product lines via their behavioural (temporal) properties [52,63].

The final enabling step for LDE was the move from DSLs defined via taxonomically organized service libraries to CINCO, our framework for meta-model-based generation of graphical mIDEs [78,79]. CINCO allows one to generate entire mIDEs on the basis of enriched modeling languages used, e.g., in industry. Figure 1 already sketched a few of such languages we have adopted and supported in the past. The resulting mIDEs may be combined to form more complex mIDEs supporting the cooperative development of all stakeholders involved using the One Thing Approach.

5.2 Related Work

Two properties are characteristic for LDE and its corresponding mIDEs:

- It aims at enabling all the stakeholders (in particular the application experts) to co-develop software without programming.
- It explicitly supports the multi-DSL-based cooperation of the individual stakeholders.

To our knowledge, the related work can be partitioned into approaches that address the first or the second characteristic.

Languages for Non-Programmers. Fowler, who coined the popular term *Language Workbenches* [32], characterizes in [33, p. 34] the role of, in his case textual, DSLs: "it's not that domain experts will write the DSLs themselves; but they can read them and thus understand what the system thinks it's doing". On the other hand, several graphical languages became very successful in dedicated application domains, like MatLab/Simulink [75], ladder diagrams [49], and Modelica [20,34]. The understanding that one should address application experts with graphical notations is also shared by the developers of the KIELER framework [35]. They provide means to automatically generate domain-specific

[18] The most recent version of the jABC supported even higher-order services [83,84].

graphical views for textual DSLs realized in the Eclipse modeling context.[19] We therefore concentrate on graphical DSLs in this subsection.

Prominent frameworks for the development of graphical modeling languages are MetaEdit+ [9,55], GME [12,64,65], Pounamu/Marama [8,39,108] or DeVIL [54,88]. These powerful frameworks are designed for generating graphical IDEs, including corresponding code generators, for a specified graphical DSL. The aspect of coordinating stakeholders with different mindsets in a cooperative fashion is not addressed. The same also applies to the Eclipse modeling ecosystem [38] with the Rich Client Platform (RCP) [76] and the Eclipse Modeling Framework (EMF) [103]. However, while there is good support for textual DSLs in Eclipse (e.g. using the Xtext [13] framework), building graphical DSLs with GMF [6], Graphiti [7], or even the Epsilon [4,5,59,60] project is very tedious.

In general, applying LDE is independent of any frameworks for the development of domain-specific languages. However, as designing the LDE languages and mIDEs required for a project is already a difficult task, CINCO explicitly aims at maximum for a higher simplicity for their technical realization.

Language-Driven Development. The *Language-Oriented Programming* approach (LOP) of the Racket team [31] is very similar to LDE concerning the second property. In fact, *Language-Oriented Programming*:

- advocates multiple cooperating languages for a project,
- has a feature called FFI (foreign-function interface) similar to our notion of native services, and
- uses a meta language 'syntax parse' to define languages

However, there are clear conceptual differences which limit the cooperation with non-programmers: LOP is based on internal DSLs (called embedded DSLs, eDSLs) based on a single base language (Racket [10], one of the successors to Lisp [106]). In [16], the exemplary DSL code is accompanied with some simple graphical notation for readability (cf. Fig. 2), which suggests that also the members of the Racket teams do not consider their DSLs as a vehicle for communication with non-programmers.

Another approach to language-driven development is projectional editing [104] as most prominently provided by JetBrains' Meta Programming Systems (MPS) [48]. However, also this approach clearly addresses programmers, or even super-programmers, capable of mastering various (programming) languages.

[19] While the KIELER framework is indeed mature and powerful – so that it is by now generalized as the Eclipse incubation project *Eclipse Layout Kernel* (ELK) [3] – its primary goal is to provide views to better communicate with non-programmers, while the actual (textual) models still require programmers or highly technical experienced domain experts.

5.3 Volume-Related Interrelations

LDE has the potential to enter many disciplines. In this section, we briefly sketch the interrelationship between LDE and the topics addressed in the other papers of the *Methods, Languages and Tools for Future System Development* part of this volume.

[20] is a good example for an approach based on its own elaborate DSL, Modelica, and [27] clearly indicates that one language is not enough, even for generic programming. The architecture presented in [61] which specifically addresses the need for dealing with multiple (domain-specific) languages is quite close in spirit to the LDE approach. It could profit from LDE-based language organization presented in [81] as well as from dedicated languages for orchestrating different analysis methods or abstractions along the lines presented in [69,95].

Also the approaches presented in the remaining papers could profit from DSLs, e.g., as follows: [41] for specifying certain assertions or contracts, [43] for specifying data flow analyses[20], [25] for specifying test models, [42] for defining learning alphabets or representing the learning result, [40] for modularly specifying the required code instrumentation, e.g. in an aspect-oriented fashion, and [15,19,66] for conveniently specifying their enriched system structures. Corresponding mIDEs (could) then guide the development by exploiting the DSL's specifics, e.g., the interpretation of assertions, security predicates, time, or probabilities. The corresponding code generators would transform the domain-specific specification directly into input for the target tool or platform.

On the other hand, LDE could also profit from the approaches presented in the other papers. In particular, all the involved analysis, verification and validation methods of [15,19,25,40–43,61,66] are good candidates for inclusion in mIDEs in order to improve the development support and/or to control non-functional constraints. Finally, [27] provides a wealth of observations and techniques with potential to impact the future mIDE development.

6 Conclusions and Perspectives

We have presented Language-Driven Engineering (LDE) as a paradigm for supporting division of labour on the basis of stakeholder-specific domain-specific languages. LDE is unique in allowing all the involved stakeholders, including the application experts, to directly participate in the system development and evolution process, while at the same time establishing new levels of reuse enabled via powerful transformations and code generation. We have illustrated how the service-oriented interplay between the involved DSLs eases product lining and system evolution through the introduction and exchange of entire DSLs together with their corresponding mIDEs.

Conceptually, LDE follows the One Thing Approach [70,101] which is reminiscent of the model-view-control pattern in that it

[20] In [92,93] this has profitably been done in temporal logic, cf. also Sect. 5.1.

– provides stakeholders with simplicity-oriented [72] individual views that are expressive enough to
– control their part of responsibility and aggregates all these views to a
– global, consistent model from which full code can be generated.

The striking new aspect of LDE is that the DSLs become first class citizens of the system development, which establishes a new level for reuse, refinement and evolution by evolving the underling mIDEs in order to resemble the domain and purpose-specific structure currently of interest. As the mIDEs for all DSLs are specifically generated, each stakeholder can get maximum support for his tasks, while (accidental) misuse is reduced to a minimum. In a sense, the mIDE functions here both as tool for maximum purpose-specific support, and as sandbox that prohibits damage. In particular, purpose-specific support can be easily enhanced by, e.g., integrating corresponding analysis and verification tools in a service-oriented fashion. Figure 1 summarizes some of the graphical DSLs from our recent industrial cooperations and student projects.

The practicality of this approach depends on the ease of DSL and mIDE development guaranteed in our context by the Cinco framework which also exploits itself service-oriented refinement. This allows, e.g., to exchange the variants of fuzzy logic simply by adapting the algebraic structure of the representing ADDs. Everything else can remain unchanged as it is captured by ADD-Lib, our ADD framework [1] which, in fact, has also been developed using the Cinco framework. The major remaining hurdle is the development of the code generators for the various mIDEs. We are currently developing a dedicated Cinco tool (an mIDE) for this purpose, which generalizes the approach presented in [51,53].

We are convinced that LDE with its growing tool stack has the potential to radically change the way software will be written in the future, as it enables the involved stakeholders to directly participate in the development process using dedicated tools matching their mindset, and it also increases the mere development performance due to its generative nature. Recommending dedicated DSL development even for individual projects sounds unintuitive at first, but we experienced an enormous leverage due to the bootstrapping effect, which steadily improves the mIDE development performance: the meta-level libraries of reusable components grow, and the Cinco-based mIDEs for e.g. program/DSL analysis and code generation become directly part of the meta-level support. This in turn increases the performance of Cinco-based mIDE and system development. Even in cases where developing the first running version takes a little bit longer with the LDE approach, this price has been paid off very early along the system's life-cycle.

With its new dimension of system development and evolution, LDE is an exciting research area with yet to be explored potential and enormous practical impact. It comprises and harmonizes many fields, like program and analysis and verification, constraint-based synthesis, meta-modeling, code generation, test- and learning-based validation, software product lines, system evolution, etc. In fact, the holistic nature of LDE radically changed our own way of system development, as it supports and motivates us to take the medicine we propose to

others. Essentially, the development of all our projects and tools, even the code generation framework, follows the LDE paradigm. We invite everybody to share this exciting experience with us in an open source platform [11].

As a new and encompassing modeling paradigm, LDE requires its own pragmatics and expertise, e.g., to avoid a drift to excessive, ad hoc DSL generation. We envisage instead new DSLs to resemble, instrument, and refine modeling languages already used in established fields of application, leveraging their own established mindsets. Examples are BNF for syntax definition [17], SQL for data querying [45], or piping and instrumentation diagrams for e.g., modeling the flow within fluid-processing machinery [44]. New DSLs are foreseen too, but they should be developed with care, with a clear vision of their potential impact in mind. As part of the continuous improvement cycle, we envision taxonomically classified libraries of DSLs ready to be reused for the construction of new mIDEs. The DSLs for decision diagrams presented in this paper are good candidates for such a library.

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. ADD-Lib. http://add-lib.scce.info
2. ADD-Lib LDE Case Study. http://add-lib.scce.info/language-driven-engineering-case-study
3. Eclipse Layout Kernel. http://www.eclipse.org/elk/. Accessed 23 Mar 2018
4. Epsilon. http://www.eclipse.org/epsilon/. Accessed 10 Apr 2018
5. Epsilon EuGENia. http://www.eclipse.org/epsilon/doc/eugenia/. Accessed 10 Apr 2018
6. Graphical Modeling Framework (GMF) Tooling. http://eclipse.org/gmf-tooling/. Accessed 10 Apr 2018
7. Graphiti - A Graphical Tooling Infrastructure. http://www.eclipse.org/graphiti/. Accessed 10 Apr 2018
8. Marama. https://wiki.auckland.ac.nz/display/csidst/Welcome. Accessed 10 Apr 2018
9. MetaCase - Domain-Specific Modeling with MetaEdit+. http://www.metacase.com. Accessed 10 Apr 2018
10. Racket. https://racket-lang.org/. Accessed 23 Mar 2018
11. SCCE - Service Centered Continuous Engineering. http://scce.info
12. WebGME. https://webgme.org/. Accessed 10 Apr 2018
13. Xtext - Language Engineering Made Easy! http://www.eclipse.org/Xtext/. Accessed 10 Apr 2018
14. Almeida, E.E., Luntz, J.E., Tilbury, D.M.: Event-condition-action systems for reconfigurable logic control. IEEE Trans. Autom. Sci. Eng. **4**(2), 167–181 (2007)
15. Alur, R., Giacobbe, M., Henzinger, T., Larsen, K., Mikučionis, M.: Continuous-time models for system design and analysis. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 452–477. Springer, Cham (2018)

16. Andersen, L., Chang, S., Felleisen, M.: Super 8 languages for making movies (functional pearl). In: Proceedings of the ACM on Programming Languages 1 (ICFP) (2017)
17. 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)
18. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebric decision diagrams and their applications. Formal Methods Syst. Des. **10**(2), 171–206 (1997). https://doi.org/10.1023/A:1008699807402
19. Baier, C., Hermanns, H., Katoen, J.P.: The 10,000 facets of MDP model checking. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 420–451. Springer, Cham (2018)
20. Benveniste, A., Caillaud, B., Elmqvist, H., Ghorbal, K., Otter, M., Pouzet, M.: Multi-mode DAE models - challenges, theory and implementation. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 283–310. Springer, Cham (2018)
21. Berg, A., Donfack, C.P., Gaedecke, J., Ogkler, E., Plate, S., Schamber, K., Schmidt, D., Sönmez, Y., Treinat, F., Weckwerth, J., Wolf, P., Zweihoff, P.: PG 582 - industrial programming by example. Technical report, TU Dortmund (2015). http://hdl.handle.net/2003/34106
22. 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
23. 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)
24. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
25. Candea, G., Godefroid, P.: Automated software test generation: some challenges, solutions, and recent advances. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 505–531. Springer, Cham (2018)
26. Chadli, M., Kim, J.H., Larsen, K.G., Legay, A., Naujokat, S., Steffen, B., Traonouez, L.M.: High-level frameworks for the specification and verification of scheduling problems. Softw. Tools Technol. Transf. (2017)
27. Chatley, R., Donaldson, A., Mycroft, A.: The next 7000 programming languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 250–282. Springer, Cham (2018)
28. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: a semantics-based tool for the verification of concurrent systems. ACM Trans. Program. Lang. Syst. **15**(1), 36–72 (1993). https://doi.org/10.1145/151646.151648
29. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: a rulebase of ADBMS features. In: Sellis, T. (ed.) RIDS 1995. LNCS, vol. 985, pp. 1–17. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60365-4_116
30. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Softw. Tools Technol. Transf. (STTT) **3**(2), 112–136 (2001)

31. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. Commun. ACM **61**(3), 62–71 (2018)
32. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? June 2005. http://martinfowler.com/articles/languageWorkbench.html. Accessed 10 Apr 2018
33. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley/ACM Press (2011). http://books.google.de/books?id=ri1muolw_YwC
34. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley, Hoboken (2004)
35. Fuhrmann, H., von Hanxleden, R.: Taming graphical modeling. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 196–210. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_14
36. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Syst. Comput. Controls **2**(5), 45–50 (1971)
37. Gossen, F., Margaria, T.: Generating optimal decision functions from rule specifications. Electron. Commun. EASST (to appear)
38. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Boston (2008)
39. Grundy, J., Hosking, J., Li, K.N., Ali, N.M., Huh, J., Li, R.L.: Generating domain-specific visual language tools from abstract visual specifications. IEEE Trans. Softw. Eng. **39**(4), 487–515 (2013)
40. Havelund, K., Rosu, G., Reger, G.: Runtime verification - past experiences and future projections. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 532–562. Springer, Cham (2018)
41. Hähnle, R., Huisman, M.: Deductive software verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 345–373. Springer, Cham (2018)
42. Howar, F., Jonsson, B., Vaandrager, F.: Combining black-box and white-box techniques for learning register automata. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 563–588. Springer, Cham (2018)
43. Huth, M., Nielson, F.: Static analysis for proactive security. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 374–392. Springer, Cham (2018)
44. International Organization for Standardization: Diagrams for the chemical and petrochemical industry - Part 1: Specification of diagrams. ISO 10628-1:2014, September 2014. https://www.iso.org/standard/51840.html
45. International Organization for Standardization: Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework). ISO 9075-1:2016, December 2016. https://www.iso.org/standard/63555.html
46. International Telecommunication Union: CCITT Recommendation I.312 / Q.1201 - Principles of Intelligent Network Architecture, October 1992. https://www.itu.int/rec/T-REC-I.312-199210-I/en
47. International Telecommunication Union: ITU-T Recommendation Q.1211 - Introduction to Intelligent Network Capability Set 1, March 1993. https://www.itu.int/rec/T-REC-Q.1211-199303-I/en
48. JetBrains: Meta Programming System. https://www.jetbrains.com/mps/. Accessed 10 Apr 2018
49. John, K.H., Tiegelkamp, M.: IEC 61131–3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming

Systems, Decision-Making Aids, 2nd edn. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12015-2
50. Jones, N.D., Sestoft, P., Søndergaard, H.: Mix: a self-applicable partial evaluator for experiments in compiler generation. LISP Symb. Comput. **2**(1), 9–50 (1989)
51. Jörges, S.: Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach. LNCS, vol. 7747. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36127-2
52. Jörges, S., Lamprecht, A.L., Margaria, T., Schaefer, I., Steffen, B.: A constraint-based variability modeling framework. Int. J. Softw. Tools Technol. Transf. (STTT) **14**(5), 511–530 (2012)
53. Jörges, S., Margaria, T., Steffen, B.: Genesys: service-oriented construction of property conform code generators. Innov. Syst. Softw. Eng. **4**(4), 361–384 (2008)
54. Kastens, U., Pfahler, P., Jung, M.: The Eli system. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383, pp. 294–297. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0026439
55. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In: Constantopoulos, P., Mylopoulos, J., Vassiliou, Y. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61292-0_1
56. 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)
57. Knoop, J., Rüthing, O., Steffen, B.: Lazy strength reduction. J. Program. Lang. **1**, 71–91 (1993)
58. Knoop, J., Rüthing, O., Steffen, B.: Optimal code motion: theory and practice. ACM Trans. Program. Lang. Syst. **16**(4), 1117–1155 (1994)
59. Kolovos, D.S., Rose, L.M., Abid, S.B., Paige, R.F., Polack, F.A.C., Botterweck, G.: Taming EMF and GMF using model transformation. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 211–225. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_15
60. Kolovos, D., Rose, L., García-Domínguez, A., Paige, R.: The Epsilon Book (2015). http://eclipse.org/epsilon/doc/book/. Accessed 4 Feb 2015
61. Kordon, F., Leuschel, M., van de Pol, J., Thierry-Mieg, Y.: Software architecture of modern model-checkers. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 393–419. Springer, Cham (2018)
62. Kubczak, C., Jörges, S., Margaria, T., Steffen, B.: eXtreme model-driven design with jABC. In: CTIT Proceedings of the Tools and Consultancy Track of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA), vol. WP09-12, pp. 78–99 (2009)
63. Lamprecht, A.L., Naujokat, S., Schaefer, I.: Variability management beyond feature models. IEEE Comput. **46**(11), 48–54 (2013)
64. Lédeczi, A., Maróti, M., Völgyesi, P.: The generic modeling environment. Technical report, Institute for Software Integrated Systems, Vanderbilt University, Nashville (2003). http://www.isis.vanderbilt.edu/sites/default/files/GMEReport.pdf
65. 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)
66. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S., Grosu, R.: Statistical model checking. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 478–504. Springer, Cham (2018)

67. Margaria, T., Bakera, M., Kubczak, C., Naujokat, S., Steffen, B.: Automatic generation of the SWS-challenge mediator with jABC/ABC. In: Petrie, C., Margaria, T., Zaremba, M., Lausen, H. (eds.) Semantic Web Services Challenge, pp. 119–138. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-72496-6_7
68. Margaria, T., Meyer, D., Kubczak, C., Isberner, M., Steffen, B.: Synthesizing semantic web service compositions with jMosel and Golog. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 392–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04930-9_25
69. Margaria, T., Nagel, R., Steffen, B.: jETI: a tool for remote tool integration. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 557–562. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_38. http://www.springerlink.com/content/h9x6m1x21g5lknkx
70. 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)
71. Margaria, T., Steffen, B.: Continuous model-driven engineering. IEEE Comput. **42**(10), 106–109 (2009)
72. Margaria, T., Steffen, B.: Simplicity as a driver for agile innovation. Computer **43**(6), 90–92 (2010)
73. Margaria, T., Steffen, B.: Service-orientation: conquering complexity with XMDD. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 217–236. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2297-5_10
74. 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). https://doi.org/10.1007/11596141_34
75. MathWorks: Simulink. http://www.mathworks.com/products/simulink. Accessed 3 Apr 2018
76. McAffer, J., Lemieux, J.M., Aniszczyk, C.: Eclipse Rich Client Platform, 2nd edn. Addison-Wesley Professional, Boston (2010)
77. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM **22**(2), 96–103 (1979)
78. Naujokat, S.: Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools. Dissertation, TU Dortmund, Dortmund, August 2017. http://hdl.handle.net/2003/36060
79. 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. (2017)
80. 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
81. Naujokat, S., Traonouez, L.-M., Isberner, M., Steffen, B., Legay, A.: Domain-specific code generator modeling: a case study for multi-faceted concurrent systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 481–498. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_33
82. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968. Scientific Affairs Division, NATO, Brussels 39 Belgium (1969)
83. Neubauer, J., Steffen, B.: Plug-and-play higher-order process integration. IEEE Comput. **46**(11), 56–62 (2013)

84. Neubauer, J., Steffen, B., Margaria, T.: Higher-order process modeling: product-lining, variability modeling and beyond. Electron. Proc. Theor. Comput. Sci. **129**, 259–283 (2013)
85. Object Management Group: Unified Modeling Language. http://www.uml.org/. Accessed 14 Mar 2018
86. Object Management Group (OMG): Documents Associated with BPMN Version 2.0.1, September 2013. http://www.omg.org/spec/BPMN/2.0.1/. Accessed 10 Apr 2018
87. Rüthing, O., Knoop, J., Steffen, B.: Sparse code motion. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), pp. 170–183. ACM (2000)
88. Schmidt, C., Cramer, B., Kastens, U.: Generating visual structure editors from high-level specifications. Technical report, University of Paderborn, Germany (2008)
89. 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). https://doi.org/10.1007/3-540-49727-7_22. http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=760066
90. Somenzi, F.: Efficient manipulation of decision diagrams. Int. J. Softw. Tools Technol. Transf. **3**(2), 171–181 (2001). https://doi.org/10.1007/s100090100042
91. Somenzi, F.: CUDD: CU Decision Diagram Package Release 3.0.0. University of Colorado at Boulder, December 2015
92. 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). https://doi.org/10.1007/3-540-54415-1_54. http://www.springerlink.com/content/y5p607674g6q1482/
93. Steffen, B.: Generating data flow analysis algorithms from modal specifications. Sci. Comput. Program. **21**(2), 115–139 (1993)
94. Steffen, B.: Property-oriented expansion. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 22–41. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61739-6_31
95. Steffen, B., Margaria, T., Braun, V.: The electronic tool integration platform: concepts and design. Int. J. Softw. Tools Technol. Transf. (STTT) **1**(1–2), 9–30 (1997)
96. Steffen, B., Margaria, T., Claßen, A., Braun, V.: The METAFrame'95 environment. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 450–453. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_100
97. Steffen, B., Margaria, T., Claßen, A., Braun, V., Reitenspieß, M.: An environment for the creation of intelligent network services. In: Intelligent Networks: IN/AIN Technologies, Operations, Services and Applications - A Comprehensive Report, pp. 287–300. IEC: International Engineering Consortium (1996)
98. Steffen, B., Margaria, T., Claßen, A., Braun, V., Nisius, R., Reitenspieß, M.: A constraint-oriented service creation environment. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 418–421. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61042-1_63
99. Steffen, B., Margaria, T., Freitag, B.: Module configuration by minimal model construction. Technical report, Fakultät für Mathematik und Informatik, Universität Passau (1993)

100. 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). https://doi.org/10.1007/978-3-540-70889-6_7
101. Steffen, B., Narayan, P.: Full life-cycle support for end-to-end processes. IEEE Comput. **40**(11), 64–73 (2007)
102. Steffen, B., Naujokat, S.: Archimedean points: the essence for mastering change. In: Steffen, B. (ed.) Transactions on Foundations for Mastering Change I. LNCS, vol. 9960, pp. 22–46. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46508-1_3
103. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
104. Voelter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 41–61. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_3
105. Weckwerth, J.: Cinco Evaluation: CMMN-Modellierung und -Ausführung in der Praxis. Master's thesis, TU Dortmund (2016)
106. Weissman, C.: LISP 1.5 Primer. Dickenson Publishing Company Inc., Belmont (1967)
107. Wortmann, N., Michel, M., Naujokat, S.: A fully model-based approach to software development for industrial centrifuges. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 774–783. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_58
108. Zhu, N., Grundy, J., Hosking, J.: Pounamu: a meta-tool for multi-view visual language environment construction. In: 2004 IEEE Symposium on Visual Languages and Human Centric Computing (2004)