DSLs for Decision Services: A Tutorial Introduction to Language-Driven Engineering

Frederik Gossen[1,2], Tiziana Margaria[2], Alnis Murtovi[1], Stefan Naujokat[1], and Bernhard Steffen[1]

[1] Chair for Programming Systems, TU Dortmund University, Germany
firstname.lastname@tu-dortmund.de
[2] CSIS, University of Limerick, Ireland
firstname.lastname@ul.ie

Abstract. Language-Driven Engineering (LDE) is a new paradigm that aims at involving stakeholders, including the application experts, in the system development and evolution process using dedicated domains-specific languages (DSLs) tailored to match the stakeholders' mindsets. The interplay between the involved DSLs is realized in a service-oriented fashion, with corresponding Mindset-Supporting Integrated Development Environments (mIDEs). This organization eases product line and system evolution, because one can introduce and exchange entire DSLs as if they were services. Using as example a smart email classification system that highlights important emails in the inbox, we model its decision procedure in a tailored graphical domain-specific language based on Binary Decision Diagrams. BDDs are a compact form of the popular decision trees and thus a mindset natural to many application experts. We then evolve this language and its mIDE to meet the new users' wish to model some uncertainty in the classification. To evolve the language, we first manually adapt its metamodel and code generator. Subsequently we show, how this step can be automated by refining the BDD DSL with a dedicated DSL for defining algebraic structures. As this exchange happens in a service-oriented fashion, it does not impair the optimization potential and nicely follows the successive refinement of the users' mindset.

Keywords: domain-specific languages, language-driven engineering, language evolution, code generation, abstract tool specification, binary decision diagrams, algebraic decision diagrams

1 Introduction

The *Language-Driven Engineering* (LDE) [17] paradigm aims at bridging the semantic gap [15] between the various stakeholders of a software project by means of tailored *Domain-Specific Languages* (DSLs) [11, 6] that capture each stakeholder's preferred mindset. These languages are provided via language-specific *Mindset-Supporting Integrated Development Environments* (mIDEs). mIDEs are comfortable state-of-the-art development environments that allow stakeholders

to think in their chosen mindset. Once various DSLs are available, a new class of stakeholders provides, maintains, and evolves the needed mIDEs.

In this paper, we introduce the LDE paradigm using as example a smart email selection/classification system that highlights important emails in the inbox. We apply the LDE paradigm to the decision services that classify, categorize, and label the incoming emails as they are received. Initially, the application classifies email in two categories (urgent, not urgent). After a refinement, it also ranks them according to their importance, and in a further refinement the emails are assigned colours.

Each evolution step improves the user's overview of the inbox, making the e-mails organisation system smarter. The interesting evolution, however, concerns the underlying domain-specific languages and their mIDEs, which allow application experts to model the appropriate decision service in their chosen mindset – in this case, the initial mindset of decision diagrams. We show how easily the language is then adapted to new users' needs, introducing a means to rank emails instead of classifying them into just two categories. We illustrate that the evolution of domain-specific languages and mIDEs can be fast enough to be applied iteratively, in a successive refinement fashion during system development and evolution. The possibility to refine DSLs in a service-oriented fashion, here by introducing a dedicated DSL for the specification of algebraic structures, support the LDE vision.

In a user-friendly world, we wish the user to concentrate on the own needs, and enabled to express those needs in a possibly easy and natural way, for example, in terms of predicates as in this paper, or predicates and then rules for a rule-based approach like miAamics [7]. We wish then the technical environment to be able to ingest that description and deliver a most efficient and runtime optimal decision structure, that is computed, maintained and executed outside of the care by the user. The reuse and the meta-level reuse advocated in [14] and demonstrated in [3] come here into play.

In Section 2, we consider the binary classification case, where we classify emails along urgency in two categories and introduce domain-specific languages for predicate abstraction (in Sect. 2.1), decision diagrams (in Sect. 2.2), and their composition (in Sect. 2.3). In Section 3, we demonstrate how the language evolves along changing user needs. We first move from binary decision to fuzzy min-max logics. Subsequently, we show how easily the LDE approach allows the replacement of the underlying algebraic structure. Section 4 concludes this paper.

2 Email Classification with Binary Decision Diagrams

Organizing the stream of incoming emails in the inbox can be very challenging: choosing what emails to read first and what are not worth reading at all takes time and individual evaluation. The automation of this classification is common practice in mail management tools. For example, spam filters are typically configured with a set of rules. A rule-based mindset yields great potential for op-

timization of decision functions [7], but it is only one of many possible mindsets. A binary classification can be easily generalized to a classification into many categories, or to ranking emails along some adequate criteria. Email classification is a prime example of a decision service and serves as an ongoing example in this paper.

We consider the simple case of a binary email classification like spam detection or the selection of particularly important emails. The yes/no outcome makes it a Boolean decision for each individual email. The criteria for the decision can be fairly complex, and so is the Boolean predicate that encapsulates the entire decision process for the yes/no evaluation. We focus instead on a set of simple predicates that express individual relevant traits of the email under consideration, and that together characterize the email in its entirety for the purpose of this classification. We call this set of predicates an *email profile*. This abstraction from the concrete email to its profile allows to focus on the important characteristics and discarding irrelevant information early in the process.

Using rules to define a decision service, as many spam filters do, is only one of the many mindsets that foot on this abstraction. We focus here instead on decision trees, a popular tool for representing decision making knowledge that is a familiar mindset to many users, and represent the decision trees in a canonic minimal form as *Binary Decision Diagrams* (BDDs) [4, 5]. The intuitive graphical representation of the BDDs links well with the predicates of the email profile. At the same time their minimality and optimality is operationally fully hidden from the users, who do not need to think about how to compose and minimize the decision diagram when they specify the criteria for the profile of their decision service. This property nicely separates an easy specification (the WHAT-level) by the user, from an optimal implementation (the HOW-level) as advocated in [17] and initially in [10].

2.1 The WHAT level: Defining Email Profiles

The DSL for email profiles is a domain-specific modelling language for binary decision services consisting of a finite set of predicates that characterize properties of the emails. Given an email, the Boolean decision service computes the predicates and takes a decision. They can be regarded either as a set of individual functions with Boolean co-domain, or collectively as a single function that characterizes the input by a Boolean vector. Each predicate is defined by a name and the implementation of its characteristic function. Already a symbolic representation of predicates is sufficient to use the full power of our modelling language, because both the optimization and the code generation work with predicate symbols, independently of their concrete implementation. For this reason, predicates are a key element of the decision models, but their implementation is not needed until the generated decision service is executed.

Among the many ways to specify a set of predicate symbols, e.g., by a list of their names, we represent predicate models in a graphical form (cf. Fig. 1). For the user, it has the advantage that predicates can be arranged and organized on a two-dimensional canvas – an interface preferred by many non-programmer

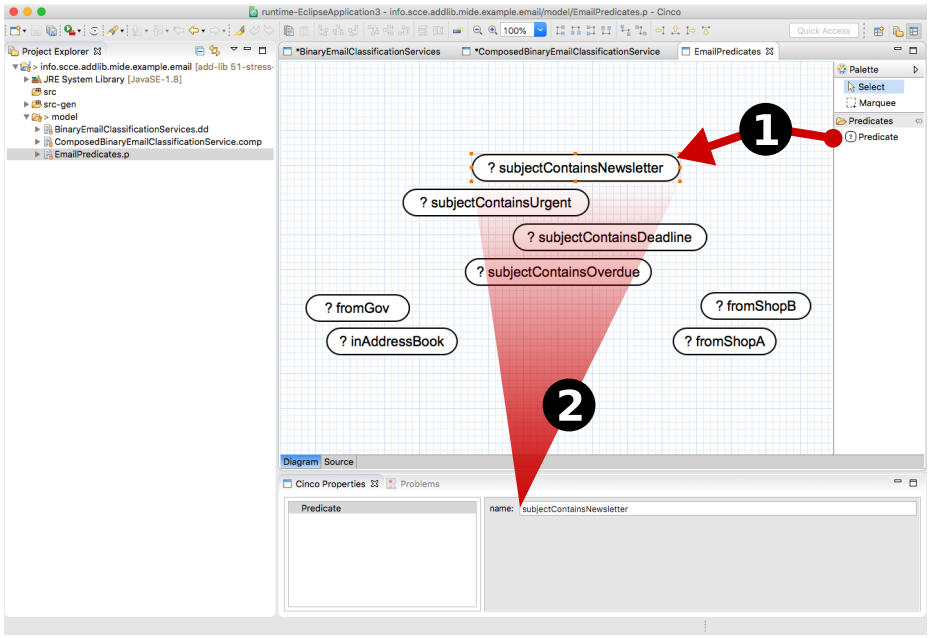


Fig. 1: Screenshot of a predicates model in its dedicated mIDE for the email domain.

users – rather than in a linear textual format. From the technical support point of view, it has the decisive advantage that we can use meta tooling frameworks, like Cinco [13], providing full mIDE support for the graphical representation without additional effort. Another advantage is that this specific DSL is also seamlessly integrable into other graphical languages that build up on it.

In this paper, we regard the predicates as mutually independent, a criterion for the optimality of the corresponding BDD optimization. In future evolutions of the predicate DSL, however, we aim at covering also implications between predicates or even predicate groups. With our graphical DSL for predicates this extension is easy: just add a new type of edge to represent implication. This extension would not impact validity: as this extension only adds model elements, previous predicate models are guaranteed to remain valid.

In our email example, predicates convey one bit of information about the email. Information expressed through predicates can be, for example, whether or not an email was sent from a contact in the address book, or presence/absence of keywords in the email's subject or in its content. The set of predicates collectively characterizes essential traits of the email and discards unnecessary information, namely the exact textual content of the email. The complete characterization is captured in a Boolean vector (the email profile) where each component represents the evaluation of one individual predicate.

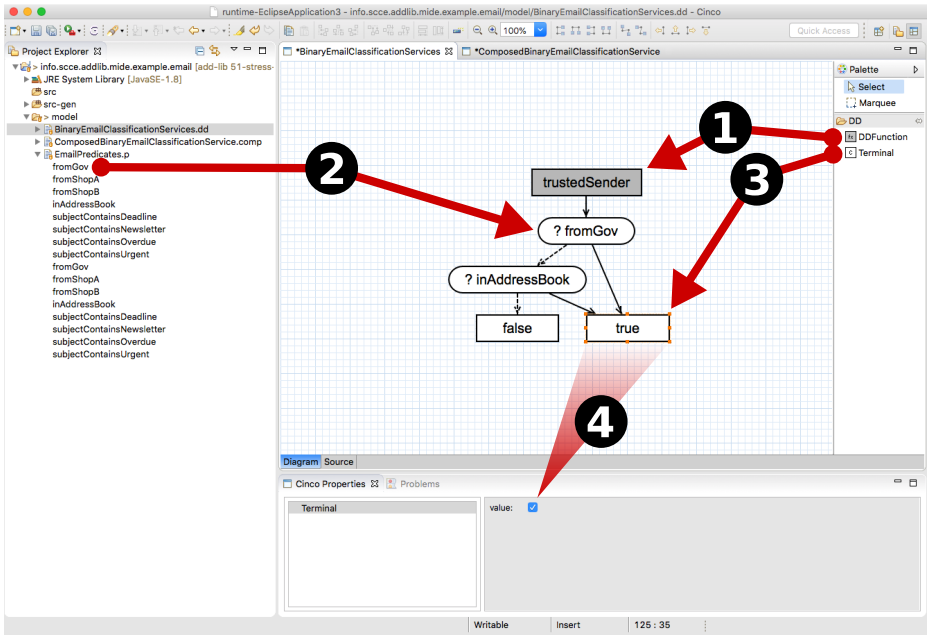


Fig. 2: Screenshot of a decision diagram model in its dedicated mIDE. The model specifies a strategy to identify trustworthy senders.

Figure 1 shows the small exemplary set of predicates we use for the email DSL. They are arranged in the two-dimensional canvas and grouped by similarity. This set is relatively small, yet it characterizes some key characteristics of emails with regard to their urgency. Because this predicate language is realized with an mIDE, users can drag and drop new predicate nodes from the tool's palette onto the canvas (cf. 1 in Fig. 1), whose only attribute is the name of a predicate that can be edited through the properties menu (cf. 2 in Fig. 1).

While some predicates may be cheap to compute, e.g., checking for a word in the subject, others can be very expensive, e.g., requiring full text search. The email profile is therefore not computed in its entirety, but predicates are evaluated only when actually needed.

2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are widely popular in computer science. In fact, the paper introducing their ordered variant (OBDDs) [4] was for a long time one of the most cited papers in computer science. Even outside of computer science, decision diagrams are a popular representation of decision processes. Their popularity indicates that describing decision procedures this way is a pretty natural mindset for large and diverse user groups: one simply visually follows the path from a starting point through the various decision points in the diagram

down to the (here binary) result. The exemplary diagram in Figure 2 determines whether an email was sent from a trustworthy person or not. Given an e-mail, simple yes or no questions are answered by evaluating the corresponding predicates until either *true* is reached, indicating that the email was sent from a trustworthy source, or *false* otherwise.

Formally, Binary Decision Diagrams are a rooted, directed and acyclic graph structure. The graph's internal nodes are associated with a predicate, and the terminals are associated with the Boolean values *true* and *false*. Every internal node has exactly two successors, one then-successor and one else-successor, while terminals, as their name suggests, have no successor. The diagram's evaluation starts at its root. For every internal node the evaluation of the associated predicate determines which successor to choose. The evaluation's result in the given situation is the Boolean value of the reached terminal.

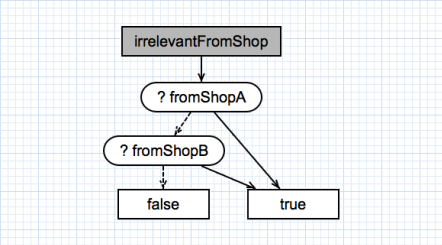
As for the predicates model, we provide full mIDE support for Binary Decision Diagrams, too. Elements of the model can be created per drag and drop from the tool's node palette, allowing users to rapidly create and modify decision diagrams and to experiment with variations. The modelling language for BDDs comprises three node types: predicate and terminal nodes correspond to the standard node types of decision diagrams, function nodes are new and serve to select and label root nodes.

Function nodes are introduced in our modelling language to assign a name to a BDD: their only successor is the initial node of the decision diagram. This way, users identify decision services by name and set the entry point to the body of the decision diagram, which is either a predicate node or a terminal node. Function nodes are created by drag and drop from a palette in the mIDE (cf. 1 in Fig. 2) and are labelled in the tool's properties menu.

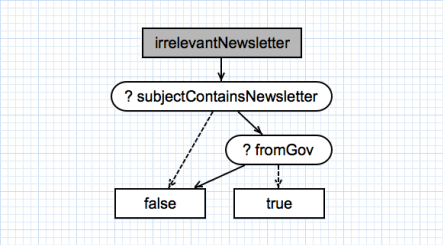
Predicate nodes correspond to the internal nodes of BDDs, which are usually associated with a variable. Here, they are associated to previously defined predicates of the predicate model: The available predicate symbols are shown in the mIDE's project explorer, and users build the decision structure by drag and drop of the predicate symbols onto the canvas, where they become predicate nodes (cf. 2 in Fig. 2). For each predicate node, one solid then-edge and one dashed else-edge is drawn to the desired successor nodes, which are themselves either another predicate node or a terminal node.

Terminal nodes hold the (here Boolean) result of the decision process. Terminal nodes have no successor, are taken directly from the mIDE's palette (cf. 3 in Fig. 2), and their value is edited through the mIDE's properties menu (cf. 4 in Fig. 2).

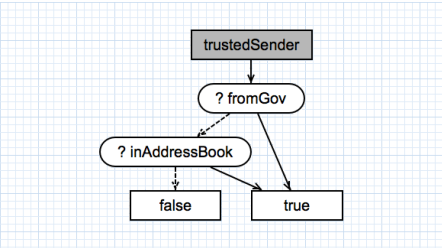
Figure 2 shows a screenshot of a decision service modelled in the mIDE for Binary Decision Diagrams. Based on the predicate abstraction of emails, this service determines whether or not the email was sent from a trustworthy source, whereby here trustworthiness is based on a governmental address or an entry in the receiver's address book. Together with the decision diagrams shown in Figure 3, this decision service forms the rule base for the case study presented in [17].



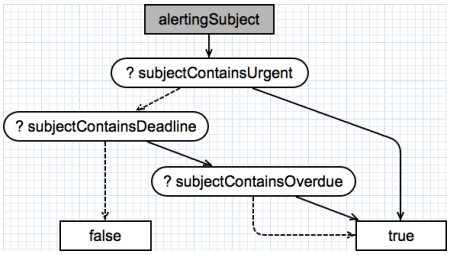
(a) Identifies as unimportant emails from one of two online shops.



(b) Identifies as irrelevant newsletters those that were not sent from a government email address.



(c) Indicates emails from known contacts and trusted senders.



(d) Indicates emails with alerting subjects based on the occurrence of certain keywords.

Fig. 3: Four exemplary decision diagram models as realized in our DSL for decision diagrams (Examples reused from [17]).

2.3 Composition of Binary Decision Diagrams

Decision diagrams are great to model small decision services, like those in the previous examples. As soon as the decision services becomes more complex and concern a variety of different aspects, like those displayed in Figure 3, scalability becomes an issue. Modularity can help: As individually modeled decision services simply represent Boolean functions, they can be combined using logical operators $\wedge$, $\vee$ $\neg$. Our corresponding graphical mIDE resembles typical abstract syntax trees, as shown in Figure 4. Yet, this user-centric modelling does not impair the performance of later realizations, as we see next.

2.4 Fully Automatic Optimization and Code Generation

In mIDE-based development, efficient realization is a clearly separate issue from modelling, and it is delegated to the code generator. Typically, this domain-specific code generator has a much bigger optimization potential than compilers for general-purpose programming languages, because it takes advantage of the knowledge about the specific domain. This effect is particularly striking for

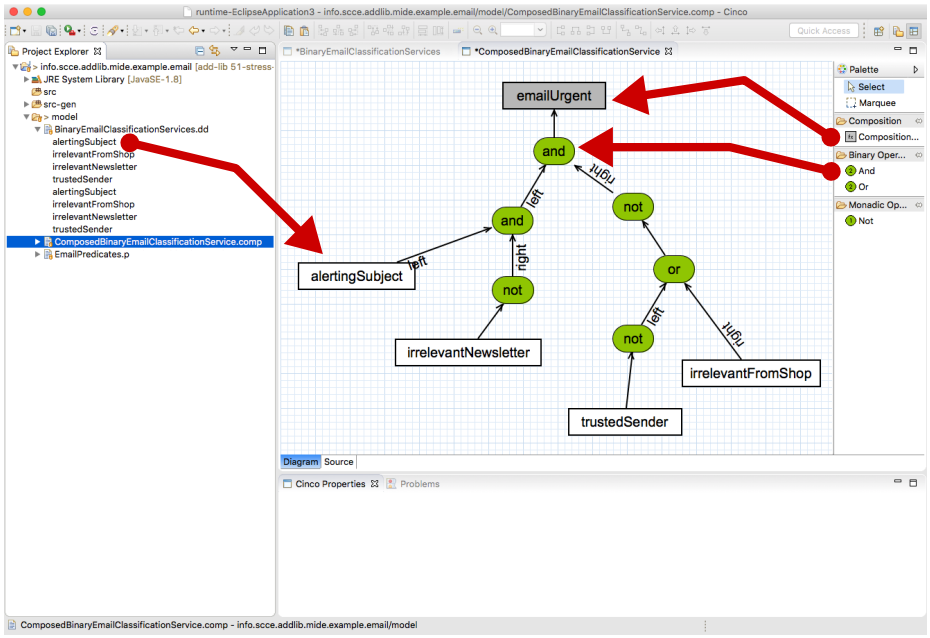


Fig. 4: Screenshot of a composition model in its dedicated mIDE. The model composes the four decision diagrams shown in Figure 3 respectively in Figure 6.

the logical combinations of BDDs, and actually far beyond what could ever be achieved in a general purpose setting.

For a fixed order of the involved predicates, Boolean functions have a canonical BDD representation. If a number of BDDs share the same predicate ordering, the logical combinations of their individual canonical BDD representations can be efficiently evaluated to obtain a corresponding resulting canonical BDD, which is computationally optimal [4]. This evaluation uses standard powerful tools and frameworks [16, 1].

Figure 5 visualizes the overall decision structure resulting from the generation process for the five decision services of the email example: the four predicate-level evaluations of the profile in Figure 3 and the composition model of Figure 4. If an email's subject contains neither `urgent` nor `deadline`, the overall result is clear and none of the remaining predicates is evaluated. This illustrates the inherent optimality aspect of canonical BDDs: only the required predicates are considered, in this case two of four.

The optimized Binary Decision Diagram in Figure 5 is good for visualization, but not executable, so it must be translated into code. The current mIDE comes with a code generator for Java that provides the executable version with ease, without need to program. The mIDE also allows users to view the canonical diagram with one click, and to transform any of the models to a canonical version of a decision diagram model. These features support users not only to

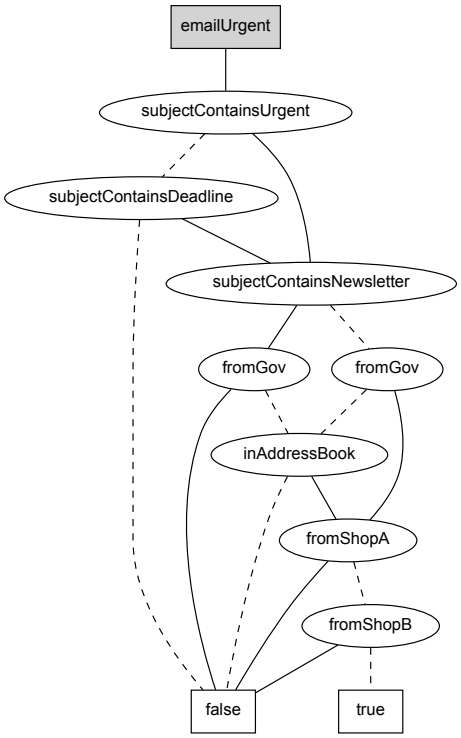


Fig. 5: Canonical decision diagram generated from the decision service models (cf. Fig. 3 and Fig. 4) to detect urgent emails.

model decision services and integrate them into their applications, but provide a quick and direct way to experiment with Binary Decision Diagrams, in a rapid prototyping fashion. For the moment, the mIDE supports Java, but it could be extended to support other target languages with relatively little effort.

3 Evolving the Modelling Language and its mIDE

Detecting urgent emails in a realistic stream of incoming emails is not an obvious task. A Boolean classifier may be too coarse: for some emails, the result may be ambiguous, others may not conform to the modeller's expectations, so that a too blunt decision model treats them differently than the modeller intended. A decision service that provides a *degree of certainty* about how urgent an email is could be more useful than a yes or no answer.

Generalizations capable to deal with degrees of certainty exist for both Boolean logic and Binary Decision Diagrams. Various fuzzy logics [9, 8] operate on the complete interval $[0, 1]$ rather than on just two values. The corresponding algebraic structures require a generalization of the BDDs to *Algebraic Decision*

Diagrams (ADDs) [2], which incorporate the underlying algebraic structure into the decision diagrams.

The new algebraic structure can be incorporated in our modelling languages and their corresponding mIDEs, too. Fuzzy logics have the same set of operations as standard Boolean logic, so previously modeled services can be easily adapted to the newly introduced concept of certainty: the only difference is that decision diagrams must now deliver a real value within the interval $[0, 1]$ instead of the previous Boolean values. In fact, the fuzzy logics used in the remainder of this paper generalize standard Boolean logic, in the sense that a Boolean model is an instance of fuzzy model where all the Boolean models' semantics remains untouched and 1 is used instead of *true* and 0 instead of *false*.

3.1 Language-Driven Engineering

Applying the fuzzy generalization to our domain-specific modelling languages requires adapting both the languages and the mIDEs. In this case, we incorporate the concept of degree of certainty into the modelling language for decision diagrams, and change the mIDEs accordingly. This is the key characteristic of Language-Driven Engineering [17]: developing domain-specific languages with mIDE support becomes part of the application development itself [3]. The change needed to accomplish this is small and close to the users' mindset, rather than forcing a new one on them. The powerful optimization of the language is maintained. As stated before, the development of domain-specific languages has become cheap, and this applies to their evolution as well.

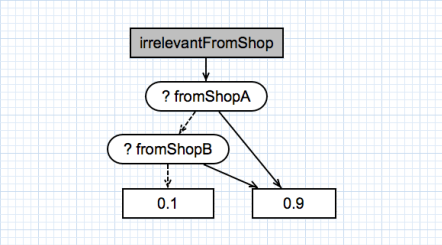
In the following, we present two variants of fuzzy logic, *fuzzy min-max logic* and *fuzzy probabilistic logic*. We realize them in different ways, in order to showcase two different ways of coping with evolution. In the fuzzy mindset, a value of 0 resp. 1 indicates maximum certainty that the truth value is *true* resp. *false*.

3.2 Fuzzy Min-max Logic

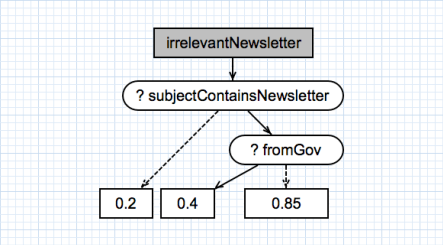
A particularly simple variant of fuzzy logics is min-max logic. Its carrier set is the interval $[0, 1]$ and it defines conjunction, disjunction, and negation as follows:

$$\begin{aligned} MinMaxLogic &:= (\ [0,1],\ \{\ \wedge_m,\ \vee_m\ \},\ \{\ \neg_m\ \}\ ) \text{ with} \\ a \wedge_m b &:= min(a, b) \\ a \vee_m b &:= max(a, b) \\ \neg_m a &:= 1 - a. \end{aligned}$$

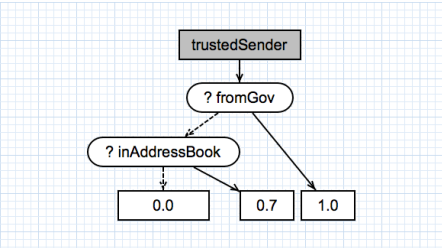
Adapting the modelling language: Because all operation symbols remain the same as in the Boolean case, the only change is the carrier set, namely the type of the values in the decision diagrams' terminal nodes. Adapting the modelling language means incorporating the new value type in the language's metamodels and the semantic change of the operations, which requires the implementation of the new operations' definitions, $\wedge_m, \vee_m$, and $\neg_m$. These are only minor changes to the code generator.



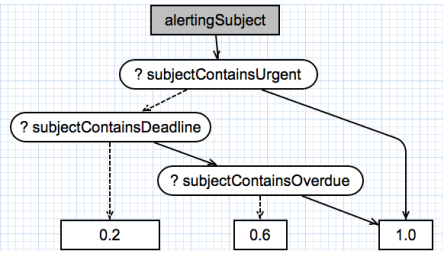
(a) Indicates degrees of unimportance of emails from one of two online shops on a scale from 0 to 1.



(b) Indicates degreed of irrelevance of newsletters on a scale from 0 to 1.



(c) Indicates degrees of trustworthiness of emails from known contacts and trusted senders on a scale from 0 to 1.



(d) Indicates the degree of conspicuity based on the occurrence of certain keywords in an emails subject on a scale from 0 to 1.

Fig. 6: Fuzzy variants of the BDDs displayed in Figure 3 [17].

Adapting the mIDE: Everything else, namely the entire tool support, can be regenerated by the metamodelling framework, making the evolution of our language easy and cheap.

Also, the already modelled decision diagrams can easily be adapted to conform to the new language. Figure 6 shows the fuzzy adaptations of the BDD models of Figure 3. Now these predicates take into account the different accuracy of the rules, improving the overall model accuracy. The composition model remains untouched and with the same operation symbols as before. However, their semantics has changed in the code generator, allowing for an adapted form of reuse, even in an evolved domain-specific language. The resulting canonical decision diagram in Figure 7 is obviously different from the previously seen case in Figure 5. As was to be expected, it distinguishes more than the Boolean decision service, and the new classifier has six different categories, with certainty degrees ranging from 0.0 to 0.8.

We showed how to manually evolve from BDDs to fuzzy min-max logic. However, in LDE, there is another, more elaborate, option, which we will discuss in the upcoming sections.

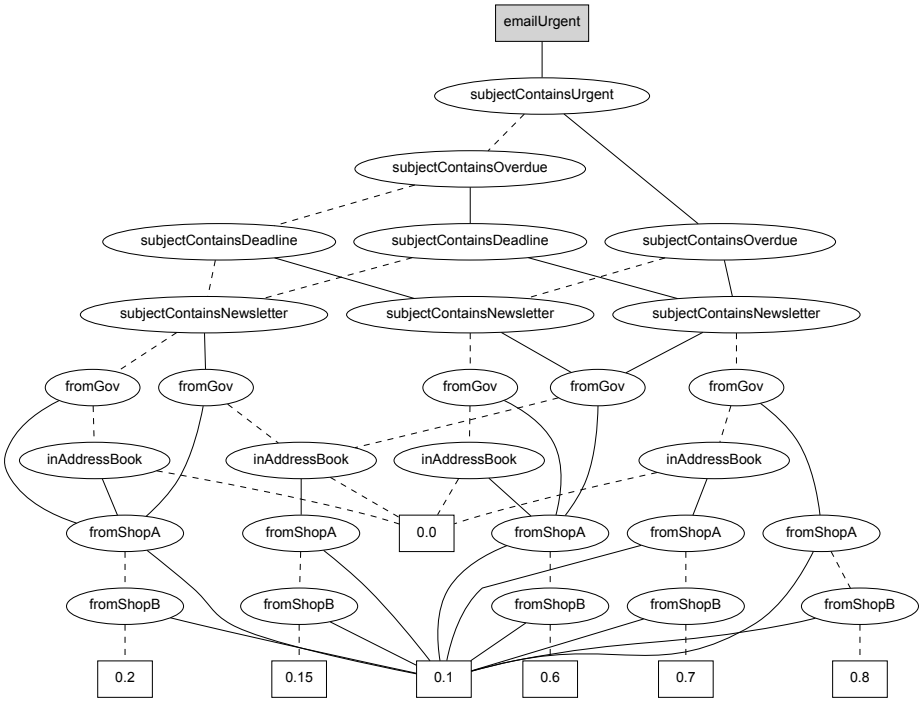


Fig. 7: Canonical fuzzy decision diagram generated from fuzzy decision service models (cf. Fig. 6 and Fig. 4) to detect urgent emails. In this case the underlying code generator implemented a fuzzy min-max logic.

3.3 Another Level of Language Refinement

Min-max logic is just one realization of fuzzy logics. Many alternatives to it have all their strengths and weaknesses, and correspond to different languages in Language-Driven Engineering. As the LDE paradigm is all about the evolution of domain-specific languages, our modelling languages can be adapted to other fuzzy logic variants within the paradigm. The wealth of domain-specific languages in this development paradigm introduces a new stakeholder type, responsible for the maintenance and evolution of the DSLs and their associated mIDE.

Instead of evolving the DSL through manual adaptation of its metamodels and code generators as just showed, we can embrace the LDE paradigm and add another refinement level: we introduce a new domain-specific language for the *evolution of the decision service modelling languages*.[3]

[3] See [3] for a discussion on completely loosening the meta-level classification of languages.

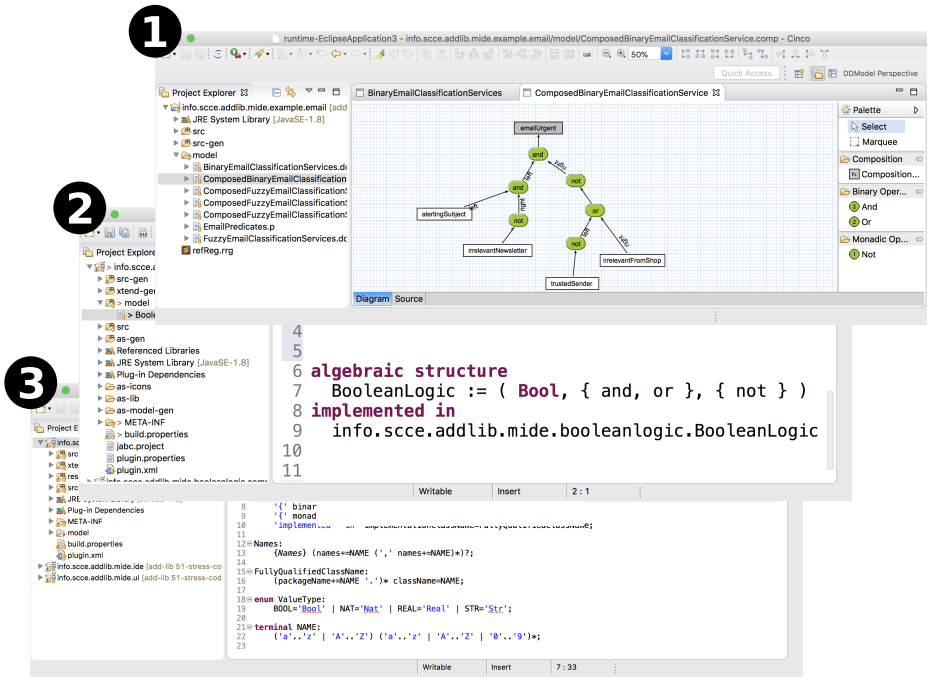


Fig. 8: Three meta levels of the domain-specific language for decision diagrams: 1) the concrete language for decision diagrams with an arbitrary algebraic structure, 2) the meta-level domain-specific language for the definition of algebraic structures, and 3) the implementation of the meta-level domain-specific language.

We demonstrate the power of this approach by introducing, now in the new way, probabilistic logic as a second variant of fuzzy logics[4], leading altogether to a third step of our language evolution. The language element subject to change is again the underlying algebraic structure: initially standard Boolean algebra, it then evolved to min-max fuzzy logic, and it changes again.

The three meta levels we use now are illustrated in Figure 8. The *modelling language designer* works with an own mIDE for language definition at the middle level (2): this designer defines the algebraic structure to be used by the end user, so that end users can use the concrete modelling tool shown in the top-most mIDE (1), which is the concrete modelling tool for the end user. To implement the defined language primitives, the modelling language designer works at the bottom level (3) where he or she now implements the just defined domain-specific language for algebraic structures. This idea was sketched

[4] We are not claiming this logic to be more or less appropriate to the email case study. Rather, we want to put the users in the focus and give them the choice of mindset among many.

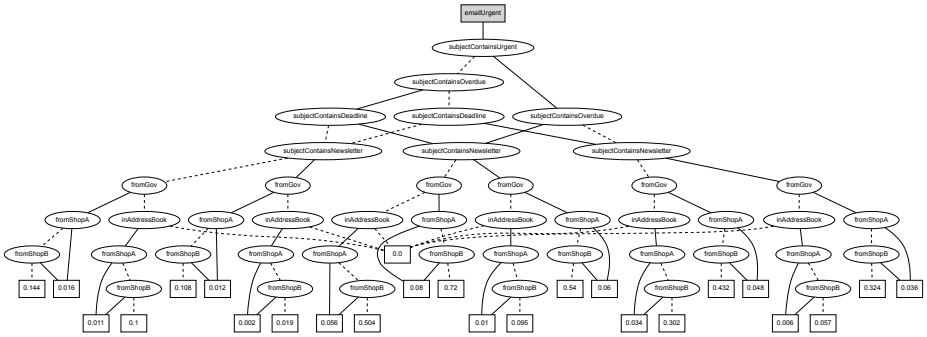


Fig. 9: Canonical fuzzy decision diagram generated from fuzzy decision service models (cf. Figs. 6 and 4) to detect urgent emails. In this case the underlying code generator implemented a fuzzy probabilistic logic.

as $meta_n modeling$ in [12]. Following that nomenclature, here the algebra definition is the meta$_3$model of the decision and composition models.

3.4 Fuzzy Probabilistic Logic

Probabilistic logic is a variant of fuzzy logics that resembles probabilities, with the implicit assumption that variables are independent. Its operations are defined as follows:

$$
\begin{aligned}
ProbabilisticLogic := (\ [0,1],\ \{\ \wedge_p,\ \vee_p\ \},\ \{\ \neg_p\ \}\ ) \text{ with} \\
a \wedge_p b &:= a * b \\
a \vee_p b &:= 1 - ((1-a) * (1-b)) \\
\neg_p a &:= 1 - a.
\end{aligned}
$$

The assumption of independent variables has remarkable consequences, e.g., the conjunction of a variable and its negation is no longer guaranteed to have probability 0. However, while probabilistic logic may not conform to all the properties one naturally expects from a logic, it is nevertheless a useful mindset for specific tasks. Here, we will use it primarily to show the adaptability of our modelling languages with the LDE approach of adding a level of meta languages.

From the point of view of adaptation needs, the operation symbols are the same as in fuzzy min-max logics. Consequently, the exact same models from our previous examples remain valid, and the newly introduced semantics only plays a role in the code generator. The resulting canonical decision diagram is shown in Figure 9.

In the following, we show how to achieve this within our Language-Driven Engineering approach by adding a new domain-specific language for the definition of algebraic structures.

3.5 Domain-Specific Language for Algebraic Structures

Applying the necessary changes appears a minor task at first glance, and in fact most of the implementation remains untouched and becomes an Archimedean point of the evolution step [18]. However, the few changes that are required spread across the entire project. In this section, we show how the service-oriented refinement of the BDD DSL and mIDE tackle this problem: instead of doing manual changes all the time, all the required changes can be elegantly captured using a specific DSL for defining algebraic structures equipped with a syntax close to the well-established mathematical notation. Once this is available, the use of probabilistic logic becomes very simple:

```
algebraic structure
  ProbabilisticLogic := (Real, {and, or}, {not})
implemented in
  info.scce.addlib.mide.ProbabilisticLogic
```

Essentially, the new DSL allows one to define the signature of the considered algebra and to link it to an implementation of the operators' semantics , here $\wedge_p, \vee_p$, and $\neg_p$, in a service-oriented fashion. Based on this description, the graphical language's metamodel, the code generator, as well as features to transform decision models to canonical decision diagrams are fully automatically generated.

Service-oriented language refinement makes the evolution of modelling languages fast, cheap and compliant with the LDE paradigm. In summary, the language refinement can happen at two levels, as sketched in Figure 8:

- directly at the language level, as in the evolution of the decision service modelling language from Boolean logic to fuzzy min-max logic, or
- at the meta-level, as in the just described DSL refinement, that eases the definition of algebraic structures in general.

The flexibility achieved by defining the algebraic structures this way is significant: models developed in the concrete mIDE can be reused, even when the language they were modeled in changes. As long as the carrier set type remains stable and operation symbols are only added, models remain completely valid, and only their semantics shift. Even when the type of the carrier set changes, models may still be adaptable, as seen in the first step of evolution from standard Boolean logics to min-max logics.

3.6 Colours as Algebraic Domain

The evolution of the language for decision services is by no means limited to logics. Graphical properties can be used to support the visualization, like the use of colours to highlight the email classification with colour codes. We consider now the algebraic treatment of colours according to the following signature:

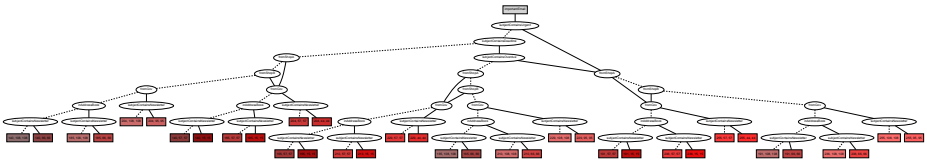


Fig. 10: Canonical decision diagram to colourize important emails. The decision structure was generated from colour decision service models structurally similar to those in Figure 6 and Figure 4).

$$
\begin{aligned}
\mathit{ColourAlg} &:= (\ [0..255]^3,\ \{\ +,\ \mathit{avg}\ \},\ \{\ \mathit{inv}\ \}\ ) \text{ with} \\
a + b &:= (\ \mathit{min}(a_0 + b_0, 255),\ \mathit{min}(a_1 + b_1, 255),\ \mathit{min}(a_2 + b_2, 255)\ ) \\
\mathit{avg}(a, b) &:= (\ \frac{a_0 + b_0}{2},\ \frac{a_1 + b_1}{2},\ \frac{a_2 + b_2}{2}\ ) \\
\mathit{inv}(a) &:= (\ 255 - a_0,\ 255 - a_1,\ 255 - a_2\ ).
\end{aligned}
$$

This algebraic structure interprets colour in the commonly used 'RGB' representation, i.e., as a combination of its red, green, and blue component. Binary operations are additive colour mixing ($+$) and mean colour mixing (avg), while inversion (inv) is the only unary operation. Like these, many more operations on colours can be lifted to decision diagrams as well as to their composition.

Exploiting LDE for algebraic structures, the following specification, together with the implementation of the operators' semantics, is sufficient for the required service-oriented language refinement:

```
algebraic structure
  ColourAlg := (Nat*Nat*Nat, {add, avg}, {inv})
implemented in
  info.scce.addlib.mide.ColourAlg
```

The corresponding decision diagram language and mIDE can now be generated in the exact same way as seen in previous examples, except that terminals are now labeled with RGB triples representing the resulting colours. The languages for the corresponding decision diagrams, for their composition, and also their optimization remain essentially unchanged. To evolve our ongoing email example to the colour domain, we used red as an indicator for the most urgent emails and cyan for unimportant emails. All emails were assigned a colour in this spectrum according to their importance. The generated decision structure is visualized in Figure 10.

The advantage of using the colour domain is that it can provide a comprehensive multi-dimensional overview. With a slightly elaborated set of models the following exemplary decision structure can be generated that highlights

– newsletters in red

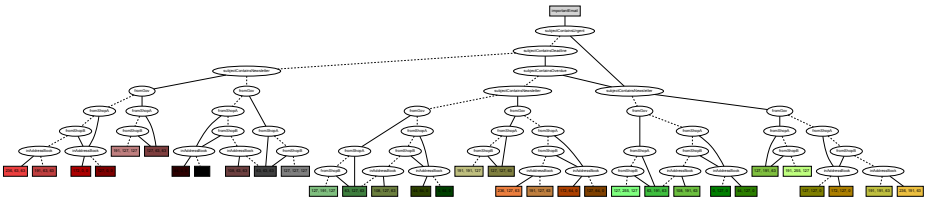


Fig. 11: Canonical decision diagram to colourize emails: Red for newsletters, Green for alerting email subjects, lighter colours for trusted senders, and darker colours for suspicious senders.

– alerting emails in green
– trusted senders with lighter colours, and
– suspicious senders with darker colours.

4 Conclusion

We showcased the flexibility of the Language-Driven Engineering paradigm on an increasingly smarter email selection/classification system. The power of the approach resides in the ability to define according languages and meta languages, and to evolve and refine over time both

– the specific Domain-Specific Languages used at the application level, in this case the decision structures that classify the mails and thus make the mailbox organisation system smart, but also
– the meta level in order to feed new, more powerful entities into the DSL level world and correspondingly also its mIDEs.

In the specific example, one can go from a binary classifier to a fuzzy min-max logic classifier by manually modifying the metamodel of the DSL for binary classifiers. Meta-level language refinement has then been illustrated by providing a dedicated mIDE for defining the algebra underlying the terminal nodes of the decision diagrams. With this refinement, changing from min-max logic to probabilistic logic is just a matter of a few lines: the algebra specification and a slight modification of the code generator. Considering a colour and application for mixing application illustrates the generality of this approach.

Evolution is typically done by substitution or refinement, and follows the changing needs of the user's mindset, in this case, the wish of increasing precision in the classification and of increased user support, e.g., with colour coding for provenance or urgency level. Contrary to the general perception, a language-driven approach can accomplish two things at the same time:

– capture its users' mindset and provide descriptive means for specification (at the WHAT-level), and
– provide powerful domain-specific optimization and code generation hidden from its users (at the HOW-level).

The interplay of DSLs growth over time – due to, e.g., the addition of predicates, and inherent sophistication of the analysis, due to, e.g., the addition of uncertainties and then of richer algebraic structures – supports an evolution-friendly and manageable style of application design for knowledge-intense domains.

This flexibility goes far beyond state of the art development scenarios, where application experts are often unable (or not allowed by the used IT systems) to change and evolve the design or configuration environment they use. With LDE, language primitives and the entire support mechanisms (editor, composition mechanisms, code generators) evolve along the needs and gracefully accompany the increasing sophistication of the entire environment.

References

1. ADD-Lib. http://add-lib.scce.info
2. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebric Decision Diagrams and their Applications. Formal Methods in System Design 10(2), 171–206 (1997), http://dx.doi.org/10.1023/A:1008699807402
3. Boßelmann, S., Naujokat, S., Steffen, B.: On the Difficulty of Drawing the Line. In: Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018) (2018), in this volume
4. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions Computers 35(8), 677–691 (1986)
5. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Software Tools for Technology Transfer (STTT) 3(2), 112–136 (may 2001)
6. Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley / ACM Press (2011), http://books.google.de/books?id=ri1muolw_YwC
7. Gossen, F., Margaria, T.: Generating Optimal Decision Functions from Rule Specifications. Electronic Communications of the EASST 74 (2017)
8. Klir, G.J., Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1995)
9. Kosko, B., Isaka, S.: Fuzzy Logic. Scientific American 269, 76–81 (1993)
10. Margaria, T., Steffen, B.: From the how to the what. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 448–459. Springer (2005), https://doi.org/10.1007/978-3-540-69149-5_48
11. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-specific Languages. ACM Computing Surveys 37(4), 316–344 (Dec 2005)
12. Naujokat, S.: Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools. Dissertation, TU Dortmund, Dortmund, Germany (Aug 2017), http://hdl.handle.net/2003/36060
13. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools. Software Tools for Technology Transfer 20(3), 327–354 (2017)
14. Naujokat, S., Neubauer, J., Margaria, T., Steffen, B.: Meta-Level Reuse for Mastering Domain Specialization. In: Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016). LNCS, vol. 9953, pp. 218–237. Springer (2016)

15. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. Scientific Affairs Division, NATO, Brussels 39 Belgium (1969)
16. Somenzi, F.: Efficient Manipulation of Decision Diagrams. International Journal on Software Tools for Technology Transfer 3(2), 171–181 (May 2001), https://doi.org/10.1007/s100090100042
17. 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
18. Steffen, B., Naujokat, S.: Archimedean Points: The Essence for Mastering Change. LNCS Transactions on Foundations for Mastering Change (FoMaC) 1(1), 22–46 (2016)