Fusing Modeling and Programming into Language-Oriented Programming

Our Experiences with MPS

Markus Voelter

independent/itemis
voelter@acm.org • http://voelter.de

Abstract. Modeling in general is of course different from programming (think: climate models). However, when we consider the role of models in the context of "model-driven", i.e., when they are used to automatically construct software, it is much less clear that modeling is different from programming. In this paper, I argue that the two are conceptually indistinguishable, even though in practice they traditionally emphasize different aspects of the (conceptually indistinguishable) common approach. The paper discusses and illustrates language-oriented programming, the approach to {modeling| programming} we have successfully used over the last 7 years to build a range of innovative systems in domains such as insurance, healthcare, tax, engineering and consumer electronics. It relies on domain-specific languages, modular language extension, mixed notations, and in particular, the Jetbrains MPS language workbench.

Keywords: Domain-specific Languages, Language Modularity, Function Programming, Language Engineering, Meta Programming

1 Introduction

1.1 What this Paper is (not)

This paper describes the author's perspective on the differences between modeling and programming. It is not a scientific (and hence, complete and objective) survey of the literature on how modeling and programming relate.

In particular, in the paper, *modeling* is interpreted as the automated construction of software (source code and other artifacts) from prescriptive models. Of course the term modeling can be interpreted differently, which would lead to a completely different discussion of the subject. However, because of its large scope, such a discussion is either very superficial or even longer than the paper as it is now, which is why we focus on this particular interpretation of the term. We elaborate on the terminology below.

1.2 Definition of Terms

Before discussing our particular view on programming and modeling, we first define what *we* understand the term "modeling" to mean.

The Essence of Models The generally agreed definition of a model is that a model abstracts from the reality of a system: certain properties of the system are ignored and others are approximated. Which properties are ignored and how others are approximated depends on the purpose of the model. A *good* model is one where the decisions about ignoring and approximating properties are well aligned with the model's purpose, i.e., the set of analyses, decisions or derivations performed on/with the model.

Descriptive vs. Prescriptive Models can be descriptive or prescriptive. A *descriptive* model describes an *existing* system for the purpose of understanding how it works, communicating this understanding to stakeholders and to make predictions about how the modeled system's behavior will evolve as time passes or as a reaction to stimuli. Most models in science are descriptive models, because science is about understanding the (existing) world. Examples include climate models, the standard model of particle physics, models of how populations evolve, or finite element models in structural engineering. In contrast, a *prescriptive* model represents a plan for a system that *does not yet exist*. Its primary purpose is to guide the construction of that system. Of course one still analyses prescriptive models in order to ensure certain properties of the future system, and exploits the abstraction and/or notation to make the system accessible to particular stakeholders. Examples include the CAD models for cars, electrical plans for buildings, or models used in biology to synthesize new compounds.

Modeling in Software Engineering In software engineering, we use both descriptive and prescriptive models. For example, in reverse engineering, when we represent the high-level structure of the code as components and visualize their dependencies, this is a descriptive model whose purpose it is to help estimate ("predict") how much effort it is to make particular changes to the software. However, prescriptive models are much more common:

- A PROMELA [9] model used to model check a state machine using Spin [1] before it is implemented.
- A state machine model that abstracts from the low-level details of implementing one in C using a `switch` statement.
- A decision table or decision tree used to analyze completeness and consistency of the decision criteria based on an SMT solver
- A model of the rules that govern when a truck driver is required to take breaks that is used to generate a C implementation for a tachograph.
- A model of a component architecture that is used to generate middleware infrastructure in an AUTOSAR system.

For a prescriptive model in software engineering to be useful, it is critical that the behaviors expressed in the model are transferred faithfully into the to-be-built system; otherwise the predictions or analysis results derived from the model are of no use. This leads us to the notion of model-*driven*.

[1] `http://spinroot.com`

Model-Driven[2] The "driven" expresses that the to-be-developed system is *automatically* constructed[3] from its prescriptive model. In other words, from the perspective of a user, only the model has to be adapted for the system to change. This can be achieved through (multi-step) transformation/generation/compilation or through interpretation.

Automatic derivation of the real system from the model is usually not the *only* (or even the primary) reason why one uses a model-driven approach – all other reasons mentioned above might apply. But the automatic derivation is a crucial[4] to ensure consistency between the model and the system; otherwise models usually get outdated as the system evolves, or vice versa. Automatically deriving the real system is thus the defining characteristic of model-driven.

Separation of Concerns and Domain Logic Most practical systems address many different concerns. In addition to the core functionality, concerns such as authentication and authorisation, encryption and privacy, transactions and persistence, or timing and throughput are common. Describing them all in one model or program is usually a mess.[5] Separation of concerns helps by allocating different concerns to different models, each of them expressed with a language that is suitable for the particular concern. To "prescribe" the complete system, all concern-specific models have to be created, aligned, and then fused during the construction of the final system. The more concerns have to be addressed, the more challenging is the construction process. In many systems there is one concern that is special, often called the domain logic:

- In a system that creates monthly salary and wage statements for employees [23], the domain logic comprises the rules that determine what counts as work time, as well as the laws that drive the deductions and benefits that apply to the employees' gross salary, plus their evolution over time.
- In a medical application that helps patients deal with the side effects of treatments [25], the domain logic is a set of data structures, algorithms, decisions procedures and correctness criteria that judge the criticality of side effects relative to the current phase of the treatment.
- In a tachograph, the device that monitors driving and break periods of truck drivers, the rules that govern when a driver has to take a break, and for how long, depending on the driving history over days and weeks, make up the core domain logic of the system.

[2] We intentionally omit the typically following word (Engineering, Development, Software Development or Architecture) because the supposed differences not relevant here.

[3] For the purpose of this paper, we ignore the question of how to ensure that this automatic construction process is correct. It is a problem where a theoretically satisfying answer is hard to give, but can be achieved relatively easily in practice.

[4] Alternatively, one could implement the system manually and then use tools to ensure consistency with ethe model. However, the author has never seen this approach used in practice.

[5] In fact, a major reason why it is so hard to understand and automatically migrate legacy software is exactly this mess.

– In an observation planning system for a radio telescope, the domain logic is the parameters needed to perform a successful observation of a particular spot in the sky, including positioning, focus, filtering and image processing.

The reason why it is useful to treat the domain logic specially in the context of software engineering is that, in many systems, the domain logic is not naturally contributed by software engineers but instead by experts in the domain [21]. It is thus especially useful to extract it into a separate model, ideally one that uses abstraction and notations from that domain. This way, the models become accessible to domain experts, and they are potentially analyzable by tools that find problems that are relevant on the level of the domain.

Platforms Whenever we construct the system from one or more concern-specific models, we expect that system to run on some kind of platform. The platform includes the operating system, middleware and application servers, but also libraries and frameworks available to us. Parts of most concerns are addressed by the platform, and the respective model only "instructs" the system how to use the platform – the models configure the platform.

~ ~ ~

This leads us to the following definition of model-driven:

Definition: Model-driven refers to a way of developing a software system S where users change a set of prescriptive models M_i representing concerns C_i at an appropriate abstraction level in order to affect the behavior of S. An automatic process constructs a full implementation of S that relies on a platform P.

In the rest of this paper, we understand the term "modeling" to mean "model-driven" in the sense of this definition.

~ ~ ~

Notably, the definition does not use the words *declarative* or *executable*, even though they are widely used in this context. This is because the author considers both of these terms not useful.

Declarativeness Using "declarative" [22] to characterise models is problematic because it implies that programs, in contrast, are not declarative. This is obviously not true, because there is a whole field called declarative programming.[6] More specifically, *declarative* is interpreted in two different ways. The strict interpretation says that a declarative program describes *the logic of a computation without describing its control flow*.[6] This is definitely not true for all kinds of models: state machines or workflow models (as in BPMN) model control flow explicitly. The less strict definition of *declarative* just means that something specifies the "what" but not the "how". Or in other words: avoid the details that are not necessary (for a particular purpose). This makes the word synonymous with "abstraction", and so it also does not add anything to our understanding of models.

[6] `https://en.wikipedia.org/wiki/Declarative_programming`

Executability This term is problematic for two reasons. First, for a model that describes only the structure of a system (and can be used to automatically construct that system) the term "executable" makes little sense unless one treats it as a synonym for "driven" (which then also makes it meaningless). Second, execution itself is not well-defined. For example, the domain logic in the above salary/tax example can be executed in the sense that unit tests can be written against it to verify the correctness in terms of compliance with the salary and tax law. But the "real" system that is later executed in the data center requires addressing scalability and security, as well as other non-functional concerns. These cannot be automatically derived from the domain logic alone. So, is the model that describes the domain logic concern "executable"?

2 Comparing Programming and Modeling

Reading the definition of *model-driven* above, it is apparent that it (also) describes programming, with the construction part being the compiler. We might argue that the difference between modeling and programming is

- that compilers produce machine code (not true for the Java compiler, it produces bytecode, another "model"),
- that the semantic gap between input and output is bigger in case of model-driven (not true, consider an optimizing compiler for a functional language),
- that the opposite is the case (also not quite true, consider a Modelica compiler that creates efficient C code from differential equations),
- or that only in model-driven we use separation of concerns (not true, consider SQL + Java + HTML + JavaScript + CSS).

It is impossible to draw a hard line between programming and modeling[7] (in the sense of model-driven). However: programming and model-driven clearly have distinct histories, traditions and communities. As a consequence, although the two are fundamentally the same, they emphasize different aspects of a common approach. Figure 1 highlights this emphasis. Of course Figure 1 paints with a broad brush, and for each of the criteria you will find a counterexample. However, we think the general trends are true; we elaborate below.

A Model-driven is often targeted at non-software domains, thus integrating more of this domain's concepts directly into the language (insurance contract, break period, image filter). Programming on the other hand emphasises orthogonal and flexibly composable concepts (function, class, module).

[7] It should also be mentioned that we are comparing programming and modeling specifically because this was the "task" set by the ISOLA track. There are other related fields one could compare, such as scripting (Perl, awk, what sysadmins do) and end-user programming (often using Excel, Access or low-code environments). In our opinion these are also basically the same, but also emphasize different aspects because of their unique context and tradition.

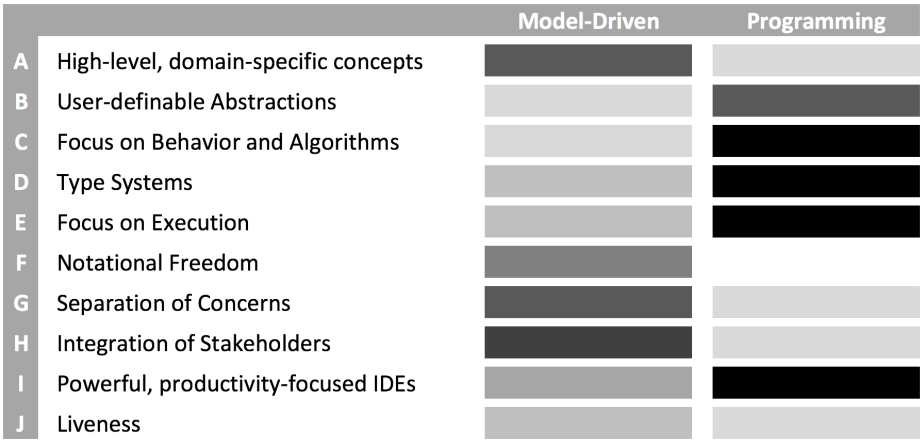

Fig. 1. Comparison between Programming and Modeling (in the sense of the definition given at the end of Section 1). The darker the color, the more emphasis.

B Consequently, programming (and its languages) optimises for letting users build their own, new abstractions. In modeling, the abstractions are often predefined and can be assembled into programs only in very particular ways. For example, in the tachometer example, users can define rules (the purpose of the language), but cannot create their own abstractions for defining rules; `break` or `driving period` are predefined.[8]

C Programming almost always includes the specification of behavior, the notion of algorithms is fundamental. In model-driven, there are many useful languages that only specify structure (UML Class Diagrams, SysML material flow, ADLs) or specify behavioral patterns instead of formulating the behavior algorithmically (consider an `async` flag on a remote communication specification).

D As a consequence of focussing on behavior, algorithms and user-defined abstractions, programming languages usually rely on sophisticated type systems. In contrast, model-driven can often make do with much simpler constraint checks.

E Programming has a clear focus on execution: if it doesn't run, it's not a program. In model-driven, while execution is always a factor as per its definition, other aspects such as analysis or communication with stakeholders might be the primary reason for using models.

F For all intents and purposes, programming relies on textual notations (the odd graphical programming language, such as ASCET[9] or Scratch [12], is the exception). On the other hand, model-driven is more flexible in its use of other

[8] Formal specification languages such as Z[30], VDM++[4] or OBJ[7] provide sophisticated means for defining abstractions for downstream analysis. However, these are outside the scope of this paper because they not used in the context of model-driven as defined in this paper.

[9] `https://www.etas.com/de/products/ascet-developer.php`

notations such as math, tables or diagrams. One could even argue that model-driven puts too much emphasis on graphical notations.

G Traditionally, a particular program has been written in *one* programming language, and concerns tended to be mixed. More recently, this has been changing for programming: SQL + Java + HTML + CSS + Javascript. Modeling has always had a tendency to describe the different concerns of the system separately, with different models and different languages.

H Partially because of the emphasis on separation of concerns, but also because of domain-specific abstractions and more diverse notations, model-driven is better at integrating non-programmers or domain experts into the software development process. The attempts at non-programmers reading source code (for example, to give feedback) are not very successful.

I Anybody who has ever used the "typical" tools for model-driven will probably agree that programming IDEs are more powerful and usable: from code navigation to refactorings to integrated test execution, model-driven tools cannot compete.

J Liveness broadly refers to removing the distinction between a program and its execution [17]. Since it is a relativerly new trend, both fields are relatively weak. We suggest that modeling is slightly better, because in some cases the models are created specifically for analysis and rapid feedback, and because liveness can be supported more easily for more narrow domains (i.e., it is harder for general-purpose languages).

3 A Hybrid Approach

In our[10] work on domain-specific languages (DSLs) in the context of business and engineering software, we have combined characteristics from the programming and model-driven column in Figure 1, further complicating an attempt at distinguishing the two. In this section we will illustrate, for each of the characteristics, what we have done, and why it is useful.

~ ~ ~

All our work is based on JetBrains MPS.[11] MPS is a language workbench [6], a tool for developing ecosystems of languages. MPS supports a wide range of modular language composition approaches, in particular, extension and restriction are supported directly [20]. This is possible because of two fundamental properties of MPS. First, it relies on a projectional editor [29]. Projectional editors do not use parsing. Instead, a user's edit operations *directly* change the abstract syntax tree, from which an updated visual representation is then projected. This way, ambiguities that result from grammars and parsing cannot arise when

[10] This refers to a team of engineers at itemis Stuttgart who specialize in language engineering with MPS. Between 2010 and 2018, the team has grown from 2 to 15 people, and we have been developing languages in a wide variety of domains such as healthcare, automotive, aerospace, robotics, finance, embedded software, science and government.

[11] `https://www.jetbrains.com/mps/`

independently developed languages are combined. This works for essentially any notation, including tables, diagrams, math symbols as well as structured ("code") and unstructed ("prose") text [27], a feature we exploit extensively in the construction of DSLs. Projectional editors have historically had a bad reputation regarding usability and editing efficiency. However, recent advances as implemented natively in MPS and in an extension called grammar cells [29] lead to good editor productiviy and user acceptance [1]. Second, MPS has been designed from the start to not just develop one language at a time, but ecosystems of collaborating languages. The formalisms for defining structure, type systems and scopes have all been designed with modularity and composition in mind. We analyze MPS' suitability for modular language composition in [26] (the paper also evaluates MPS more generally).

While other language workbenches exist [5], none of them provides the unique properties, scalability and robustness of MPS, which is why the approach – at least currently – cannot be fully implemented with another language workbench.

~ ~ ~

In this section, we pick up the differences in emphasis discussed in the previous section, talk briefly about how in our approach we blur the disctinction between programming and model-driven even further, and then provide lots of examples to make the case; in each section, examples follow after the separator (the three tildes). The examples center around three ecosystems:

mbeddr mbeddr is an implementation of the C programming language in MPS, plus a set of 30+ modular language extensions for embedded software development [28]. Extensions include state machines, physical units and interfaces/components. mbeddr has been in development since 2011 and is used in several commercial projects, including a smart meter [24].

KernelF KernelF is an extensible and embeddable functional language [23]. Its purpose is to serve as the core of DSLs. We have been using it in a variety of DSL projects, including in the finance, healthcare and smart contract domains.

IETS3 IETS3 is a set of languages for systems engineering, including a widely customizable language for component architectures as well as a feature modeling language. It is currently used in three different systems engineering projects, mostly in the automotive industry.

3.1 A High-level, Domain-Specific Concepts

In programming, new abstractions are provided through libraries,[12] developed with the language itself. In idiomatic use of MPS, new abstractions are provided through language extensions, defined outside the language, using MPS' language definition facilities. A language extension can be seen as a library plus syntax, plus type system and plus IDE support (and a semantics definition via an interpreter or generator). In other words, many more abstractions are first-class, with the resulting advantages in terms of analyzability and IDE support.

[12] In this context, we consider frameworks a form of library.

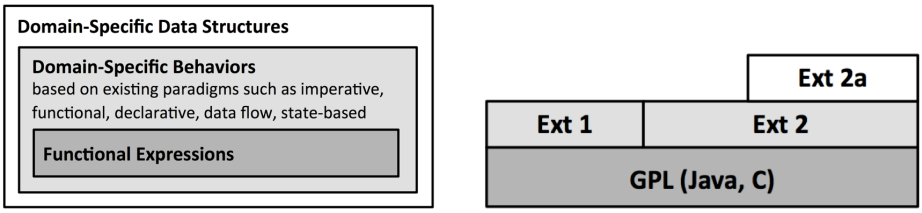

Fig. 2. Our two most important patterns for DSL design: a custom language with a (resricted) embedded functional language and base language that is incrementally grown towards a domain.

Libraries, in general, can be composed. For example, you can use the collections from the Java standard library together with the Joda Time[13] library for date and time handling and the Spring[14] framework for developing server-side applications. There is no need to explicitly compose the frameworks, the combination "just works". While this composability is not true for language modules in general (primarily because of syntactic ambiguities), it is true with MPS: for all intents and purposes, language extensions can be composed just like libraries. The composition also has similar limitations: one cannot statically prove that the composition will not have unintended side-effects, and the set of libraries/language extensions might not fit well in terms of their style. However, if language extensions are developed in a coordinated, but still modular way, as a stack of extensions, these limitations do not apply.

We rely on two primary patterns for language design, both shown in Figure 2. The first one uses a three layer approach where the domain contributes the core structural abstractions and the coarse-grained behaviors, but the low-level expressions and functional abstractions are reused from an existing language; we use KernelF [23] for this purpose. In the second approach, we take a full existing language (such as Java, C or again, KernelF) and "grow" it towards a domain by adding more and more domain-specific abstractions (implemented as modular language extensions). In the first case, only selected parts of the embedded functional language are available to the user, which makes analysis of the whole program simpler. In the second case, one typically allows the full base language in user programs; this nicely supports gradual abstraction in the sense that users can always fall back to the base language if no suitable high-level abstraction exists. However, analyzability potentially suffers because low-level constructs can "pollute" high-level, analyzable abstractions (think: C pointer arithmetics embedded inside a state machine described first-class).

~ ~ ~

Decision Processes In an extension of KernelF for Smart Contracts,[15] we have developed first-class support for collaborative decisions [19]; Figure 3 shows examples. A `MultiPartyDecision` is a top-level declaration (like a record or a

[13] `http://www.joda.org/joda-time/`
[14] `https://spring.io/`
[15] `https://en.wikipedia.org/wiki/Smart_contract`

function) that supports a declarative configuration of the decision: which parties are involved, and can that list of parties be changed dynamically (at runtime, as the decision process runs), what is the decision procedure (unanimous, majority, specific threshold or completely custom), is a minimum turnout required, is there a time limit for making the decision, and can a party revoke their votes. The complete, potentially non-trivial implementation of this process is handled internally to the decarative specification. In addition, the IDE only allows those commands on a decision object that are valid based on its configuration.

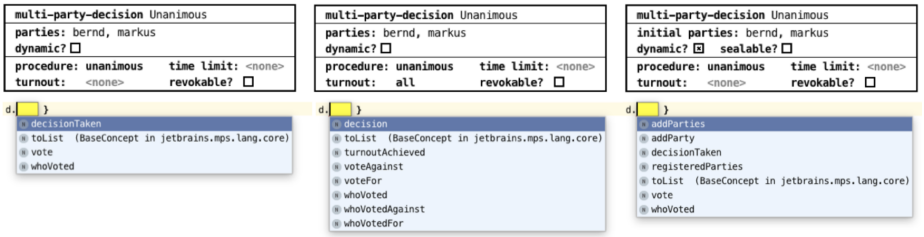

Fig. 3. Declarative decision procedures, each with a different configuration and resulting code completion proposals.

State Machines As part of mbeddr's extensions for embedded programming, we have developed first-class support for state machines. They support the textual notation shown in Figure 4, but also tabular and graphical notations; those can be switched by the user. As a consequence of state machines being first-class, mbeddr provides automatic model checking [2] support for properties such as reachability of states and activeness of transitions, but also allow users to define their own temporal properties which are then checked by the model checker.

```
statemachine HierarchicalFlightAnalyzer {
  macro stopped(next) = tp->speed == 0 mps
  macro onTheGround(next) = tp->alt == 0 m
  in event next(Trackpoint* tp) <no binding>
  in event reset() <no binding>
  out event crashNotification() => raiseAlarm
  readable var int16 points = 0
  initial state beforeFlight {
    entry { points = 0; }
    on next [tp->alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
  } state beforeFlight
  state crashed {
    entry { send crashNotification(); }
  } state crashed
```

```
composite state airborne initial = flying {
  on reset [ ] -> beforeFlight { points = 0; }
  on next [onTheGround && stopped] -> crashed
  state flying (airborne.flying) {
    on next [onTheGround && tp->speed > 0 mps] -> landing
    on next [tp->speed > 200 mps] -> flying { points += VERY_HIGH_SPEED; }
    on next [tp->speed > 100 mps] -> flying { points += HIGH_SPEED; }
  } state flying
  state landing (airborne.landing) {
    on next [stopped] -> landed
    on next [ ] -> landing { points--; }
  } state landing
  state landed (airborne.landed) {
    entry { points += LANDING; }
  } state landed
} state airborne
```

Fig. 4. The textual notation for state machines in mbeddr.

Timelines In a project in the context of tachographs, we built a DSL to describe the rules that determine when truck drivers have to make breaks. The language has first-class abstractions for timelines, driving periods, breaks and other abstractions core to that domain. The relationships between the various periods can also be described. The abstractions also come with a very particular notation shown in Figure 9 A and discussed in Section 3.6. The language builds

on top of mbeddr, and embeds C expressions in the timelines. From the rules, we generate executable C code that goes into the tachograph firmware.

3.2 B User-definable Abstractions

Technically there is no reason why one could not build languages that allow users to build their own abstractions, just like any programming language. In fact, MPS ships with Java, mbeddr implements all of C, and KernelF is a full functional programming language. By growing a language incrementally towards a domain, users can choose if they want to use predefined abstractions (such as the mbeddr state machines mentioned above), or whether they want to define their own abstractions using the facilities of the base language.

~ ~ ~

mbeddr Components mbeddr has an extension for component-oriented programming that includes interfaces with operations; hierarchical components with ports that provide and require those interfaces; interface polymorphism; and instantiation of components and wiring through connectors. The idea is similar to ADLs [13] and UML composite structures, but directly integrated into C. Similar to classes in C++, users can define their own abstractions using components. Section 5.1 and Figure 1 in [24] illustrates the architecture of a Smart Meter built with mbeddr's components. itemis France is currently extracting a reusable platform from this architecture, again illustrating the ability to create user-defined abstractions. *Inside* the components, users can use low-level C code (or other extensions) to implement a components's behavior; an example of an abstraction gradient [8].

Functional Abstractions in KernelF KernelF supports functions, function calls, currying, lambdas, higher-order functions and function references, as well as enums, tuples, collections and records. Similar to any other functional programming language, users can use these language concepts to create their own abstractions. However, KernelF does not support algebraic data types, a module system or function composition. This is because KernelF is intended to be embedded in (or extended towards) DSLs which typically supply their own first-class domain-specific abstractions instead for these.

3.3 C Focus on Behavior and Algorithms

Structure-only models are useful in the context of model-driven. For example, they can be used to generate distribution middleware, database schemas and access layers, or UIs. However, historically, the prevalence of structure-only models in model-driven is also a consequence of limitations in the tools: trying to graphically model the algorithmic parts of tax calculations really is a pain. With MPS, we can mix arbitrary notations, so algorithmic aspects are not problematic; in particular, we are not forced to represent them graphically. It turns out that in practice, lots of the business value in the domain logic of a system lies in the way *decisions* are made, and how values are *calculated*, the two main aspects of behaviors.

~ ~ ~

KernelF Utilities KernelF supports various utilities for making decisions, such as graphically-rendered decision trees and various forms of decision tables (both shown in Figure 5); all of these notations are fully editable. In addition, KernelF also has support for mathematical symbols to support concise notations for calculations (rightmost item in Figure 5). "Regular" programming with (higher-order) functions are of course also supported for decisions and calculations.

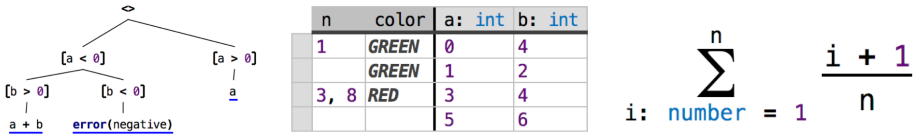

Fig. 5. Decision trees and tables, as well as math symbols available in the KernelF util package. All are expressions, so they can be used widely in KernelF programs.

Insurance Math Actuaries use many conventions when writing down their heavily numerical, recursive functions. For example, they disinguish between iterator variables and parameters, where parameters remain constant in (recursive) calls to functions that declare the same parameters (see Figure 6). Sameness is established by relating them to a definition in a data dictionary, which is why the parameters do not declare types when used in functions; those types are in the data dictionary. The language, an extension of KernelF, also relies heavily on various forms of lookup tables, built as another extension.

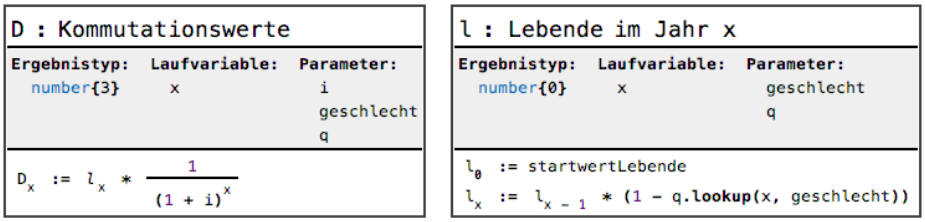

Fig. 6. A compact numerical functional notations for insurance math; parameter values do not have to be specified for calls to functions that take the same ones.

Temporal Data A KernelF extension developed for a salary/tax calculations system has temporal types. The system supports bitemporal [10] data, and the temporal types shown here represent one of the two dimensions. The notation `TT[U]` represents a temporal version of a base type `U`. Temporality means that a variable `ttu: TT[U]` does not represent a single value; instead, `ttu` is a sequence of `(date, U)`-pairs, expressing that on `date` the particular value of `ttu` changed to a particular `u: U`. The following example states that on Jan 1, 2017 the `salary` became 5.000 EUR, and on May 1 it changed to 6.000 EUR.

```
val salary : TT[currency] = TT | 2017 01 01  =>  5.000 EUR |
                               | 2017 05 01  =>  6.000 EUR |
```

The extension overloads the basic operators (`+`, `-`, `*` or `/`) for temporal types to "reslice" the temporal periods (Figure 7); the semantics of the operators regarding their basic types `U` remain unchanged. These overloaded operators let users write arithmetic code that works with temporal data as if it was regular, scalar data. To effectively work with temporal data, more support is required. An example is the `reduce` operator: `ttv.reduce(S, r)` (where `r` is a `daterange`, a type that represents time periods) reduces a temporal value back to a scalar. The operation takes into account the slices within `r` (for example, the month for which taxes are calculated) and a reduction strategy `S`. The strategy includes `LAST` (the value of the last slice in `r`), `SUM` (sum of all slice values), and `WEIGHTED_AVERAGE` where the sum is weighted with the relative duration of each slice.

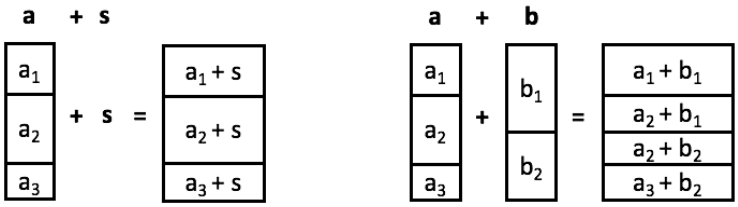

Fig. 7. Reslicing of temporal values. `a` and `b` are temporal values where the actual value (`a1, a2, a3` and `b1, b2`) is different in subsequent time periods (represented by the slices); `s` is a regular scalar. When a temporal value is "operatored" with a scalar, the slices remain the same, but their values change. In the case of two temporal values, the slices intersect, and the values are computed per intersection.

3.4 D Type Systems

Classical constraint checking is used in many of our languages: Are the names of states in state machines unique? A decision procedure that allows revocation of votes also requires a minimum turnout. Are there circular imports between mbeddr modules? However, we also use type checking. It is required once expressions, such as those from KernelF, are embedded into a DSL. Similary, if full programming languages are implemented for the purpose of extension (as in mbeddr with C, or as Jetbrains did with Java), the full type system of this language must be available in MPS. On the other hand, we do not require type systems that support meta programming (reflection, type bounds, implicit operations and other "magic" that can be found for example in Scala), because we can use MPS' meta programming facilities to build language extensions.

~ ~ ~

Number Types in KernelF The `number` type in KernelF is interesting because it has a range and a precision. This is very useful in many DSLs; for example, in a project in the healthcare domain, the ranges help validate domain-specific data such as blood pressure. The following patterns are supported:

```
number             => number[-inf|inf]{0}     // integer type, unlimited range
number[0|inf]      => number[0|inf]{0}        // positive integer
number[10|20]      => number[10|20]{0}        // integer type, range as specified
number{2}          => number[-inf|inf]{2}     // 2 decimal places, unlimited range
number[3.3|4.5]    => number[3.3|4.5]{1}      // range as specified, precision derived
```

KernelF operators perform basic arithmetic computations on the ranges of numeric types. However, we have not implemented a fully dependent type system [32] since this is too complicated for language implementors and for the developers of language extensions; ours is pragmatic in that gives up early on trying to compute the specific ranges and just assumes `[-inf|inf]`, i.e., unlimited range.

```
42 + 33                  ==> 75     <number[75|75]{0}>
42 + 2 * 3               ==> 48     <number[48|48]{0}>
aReal + anInt            ==> 75.33  <number[75.33|75.33]>
if aBool then 42 else 33 ==> 42     <number[33|42]{0}>

type tt: number[-10|10]
val n3, n4 : tt                = 0
val n34    : number[-100|100]  = n3 * n4
```

Units in mbeddr mbeddr supports physical units. Every numeric C type (`short int, int, double` and the like) and their literals can be annotated with a physical unit. The system ships with the SI base units (`m`, `s`, `kg`, `K`, `A`, `mol`, `cd`), and users can derive their own units from those base units. Figure 4 has a few examples in the state machine. Here are a few more:

```
derive N   as kg * m / s / s;         // defining a new unit for force
derive mps as m / s;                  // defining a new unit for speed
int/m/   length = 20 m;               // variable definition with unit type
int/mps/ speed  = length / 20 s;      // performs computations with units
int/mps/ kaputt = 20 * length;        // error caught by the type system
```

Temporal Types The two examples in this section are *only* type system extensions. During execution, they do not play a role: KernelF's number types are mapped to Java's `BigInteger` and `BigDecimal`, and mbeddr's physical units are completely elided during reduction to plain C. The temporal shown in the previous section also involves adaptations in the type system, but the execution semantics have to be adapted as well (to compute the sliced values).

3.5 E Focus on Execution

Because of the ability to add arbitrary abstractions as first-class concepts, it is easier to add meaningful (i.e., relevant to the user) analyses and verifications. We have exploited this in several projects and languages, as we show below. On the other hand, almost all of the DSLs we have built over the last years are executable; an exception is a set of languages for performance analysis and security assessments in embedded systems.

~ ~ ~

The KernelF Interpreter KernelF ships with an interpreter that runs directly on the AST of the program in MPS. Thus, the effect of a change to a program or

its input data can be seen immediately. The most used form of specifying inputs and observing outputs is as part of test cases. They can be executed automatically (technically, as part of MPS' type checking the program) or on demand. The actual value is rendered inline in the code, and the assertion/cases/suites are rendered in green/red, corresponding to test success or failure. Like the core language definition, the interpreter infrastructure is modular to allow language extensions to also supply an interpreter for that extension. For example, the decision processes (Section 3.1) and the temporal arithmetics (Section 3.3) come with modular interpreter extensions.

The mbeddr Build Infrastructure mbeddr does not ship with an interpreter. Instead, the extensions, such as components or state machines, are reduced to plain mbeddr C by a multi-step reduction pipeline (see [24] for the performance of this generated code). The last step is then the generation of textual C code, and its compilation with a configurable C compiler (gcc by default). The whole process is completely integrated with MPS' build infrastructure; `Ctrl-F9` runs everything. Programs can also be executed directly from the IDE, their output rendered in the MPS console.

Scenario Testing and Simulation In a medical project we developed a set of DSLs based on KernelF to model the algorithms in mobile phone apps that help patients with medicine dosage and the side effects of therapies [25]. One language supports given/when/then-scenario descriptions as well as their execution as tests using the interpreter. In addition, there is an interactive simulator, based on an in-IDE rendering of the final phone app, where healthcare professionals can interactively experiment with the algorithms they are currently developing.

Model Checking for mbeddr Components mbeddr can check components for their compliance with the semantics defined with pre/postconditions on the operations in interfaces using model checking; see Figure 8. The contract is translated to labels in the generated C code, and the integrated CBMC model checker [11] performs a reachability analysis for those labels. The results are lifted back to the level of the DSL in order to be meaningful to the developer. This approach combines the convenience of specifying the semantics via the DSL with verification of the final system, the generated C code [14,15].

SMT Solving for KernelF Decision Tables Several of KernelF's decision support abstractions support consistency checking based on an SMT solver. In particular, decision tables are checked for completeness (are all possible inputs handled) and overlap (for every particular set of inputs, will only one row/cell combination be valid). For multi decision tables, we also check coverage, and ensure that the condition in row `i` is either distinct from, or a subset of all lower rows `i + k`. This is important for the "shadowing" semantics of these tables. Results are reported as regular error markers on the code. The solver takes into account restrictions on the inputs from ranges in number types or other constraints; we are currently working on translating all of KernelF to the solver in order to take the complete context into account for these and other checks.

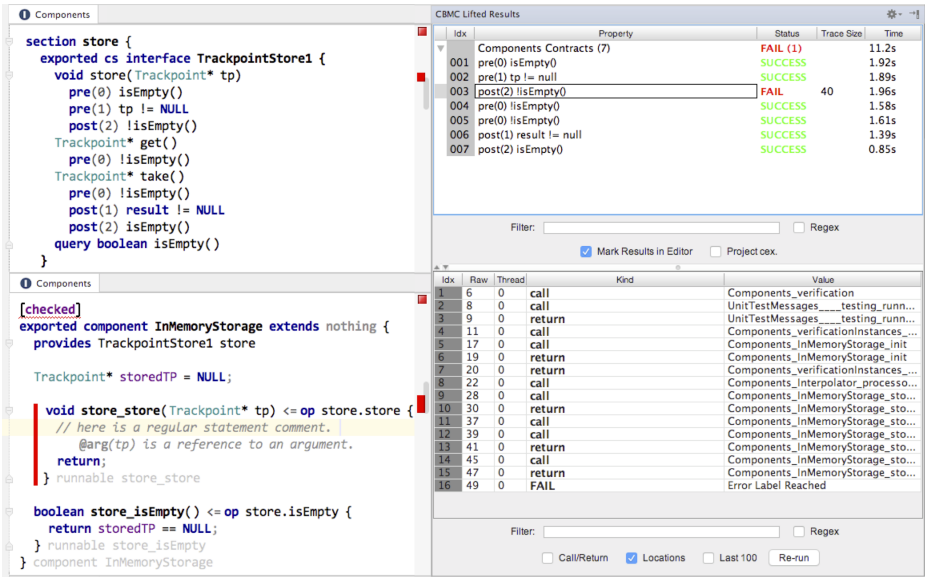

Fig. 8. Model checking component contracts. **Top Left:** an interface definition in mbeddr that expresses the semantics of the operations with pre/post conditions. **Bottom Left:** A component providing the interface and implementing the operations; implementation is arbitrary C code. **Top Right:** Verification results lifted from the raw CBMC result to the level of the DSL; a list of succeeded and failed pre/postcondition. Clicking on one selects the respective location in the code. **Bottom Right:** Execution trace for a failed check. Clicking selects corresponding location in code.

Performance Analysis using Simbench Simbench relies on the system engineering-level component language we developed for IETS3 to model automotive E/E architectures. A modular extension of the components language allows systems engineers to model resources (such as CPUs, memory or disk storage) as well as the components' use of these resources (CPU instructions, memory amount or disk access bandwidth) during the execution of particular scenarios (such as the boot/startup of an in-car navigation system). Simbench then performs a discrete event simulation to analyze timing properties and detect resource bottlenecks and the resulting critical paths. This helps engineers optimize the system architecture for performance through improved scheduling of tasks.

Security Analyst Also relying on IETS3 components to model E/E architectures, the tool is used to analyze the security properties of these architectures. In particular, users can model attack scenarios and risks regarding the component-based system architecture. The tool then computes risk/attack propagation vectors and lets users define mitigations. The result of the analysis is a set of reports that demonstrate how and why a particular set of security measures lead to a secure system relative to the previously defined risks and attacks. This tool is also example of purely structural models: no actual behaviors are described.

3.6 F Notational Freedom

The importance of the notation, or concrete syntax, cannot be overstated in the context of modeling languages: users initially always judge a language by the notation. Even if the concepts behind the language are great, users might not get to appreciate this if they cannot get past the "syntax they don't like". This becomes even more crucial for languages that address domain logic, and hence target people who may not be professional developers. Three ingredients allow us to use notations that are closely aligned with domains. First, the projectional editor can support essentially any notation; we have syntax libraries for structured and unstructured text, tables, math and diagrams. Second, because the modular language extensibility lets us easily define new first-class concepts, we have "hooks" to which we can attach specialized notations. Third, since MPS is bootstrapped, extending MPS with completely new notational primitives (that can then be used with multiple languages) is just a matter of modularly extending the notation meta language; we discuss the meta level in Section 4.

~ ~ ~

Many of the screenshots shown in this section are also examples of customized syntax, in particular Figures 3, 5 and 6. More examples follow.

Timelines The previously mentioned tachograph timelines are shown in Figure 9 A. Clearly, the notation it is very much inspired by Excel, which our customer had used to prototype the language. The timelines are an example of a project-specific notation: it is not built with our generic table notations framework, but has been built into MPS specifically for this project.

Excel-style The previously mentioned Security Analyst relies mainly on tables, some of them rather large. Figure 9 B shows a small one. While spreadsheets such as Excel are a good way for ad-hoc problem solving, they are rarely a good implementation technology for core domain logic, because of Excel's very limited support for managing complexity (the "Horror Stories" list at the European Spreadsheet Risk Group[16] provides ample evidence for this claim). However, they are often a good source from which to extract the core concepts, and sometimes also parts of the notation of DSLs.

Feature Models Feature models are a widely used formalism for describing variability in systems [3]; an example is shown in Figure 9 C. From a notational perspective, the decorations on and between the lines are interesting – such decorations are also useful in other tree-based notations. Our tree notation language (which is different from the language to describe box-and-line notations described next) generically supports decorations "on both ends".

Box-and-Line Diagrams Figure 9 D shows the graphical notation for IETS3 components. Note that the diagrams are fully editable (through a palette, in-diagram buttons, intentions and drag/drop). The diagrams can embed textual notations with full IDE support (for example, one can use KernelF expressions to define values for component parameters in the diagram) and the diagram can

[16] http://www.eusprig.org/horror-stories.htm

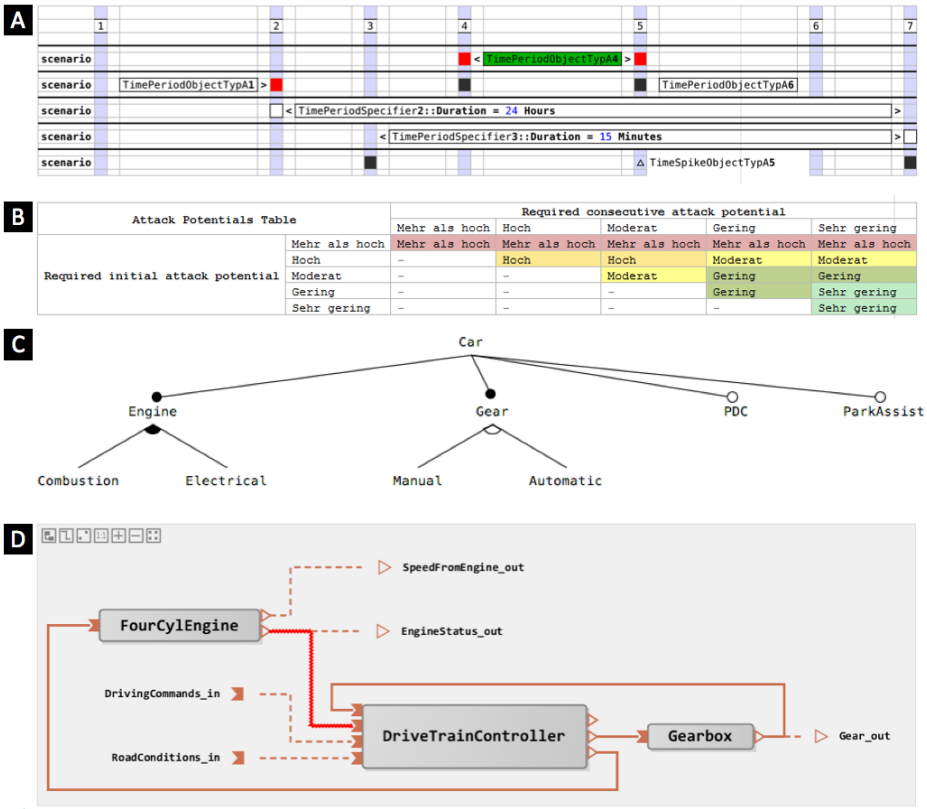

Fig. 9. Further examples of notational flexibility we use in MPS. **A:** a custom-developed notation for timelines; **B:** tabular notations inspired by Excel; **C:** decorated tree notation for feature models; **D:** box-and-line diagrams.

itself be embedded in a textual editor. All our graphical editors have a textual secondary notation that can be enabled by users; often, bulk editing is much faster using a textual syntax. The integration between text and diagrams is maybe the most convincing demonstration of the benefits of a unified (projectional) editor architecture. Integrating diagram editors with parser-based text editors is significant work otherwise [18]. In addition to the components, we have used the box-and-line notation also for state machines, entity relationship-style data models, and for dataflow programming (in the Siemens ESD[17] tool).

Unstructured Text Code that conforms to a well-defined language has a rigid structure, especially in a projectional editor. However, many parts of programs are best expressed as unstructured text: comments are the obvious example, but mixed content, such as markup (think: HTML) or requirements documents are also examples. To enable this, MPS supports rich text: rich text paragraphs

[17] `https://www.plm.automation.siemens.com/de/products/lms/imagine-lab/embedded-software-designer.shtml`

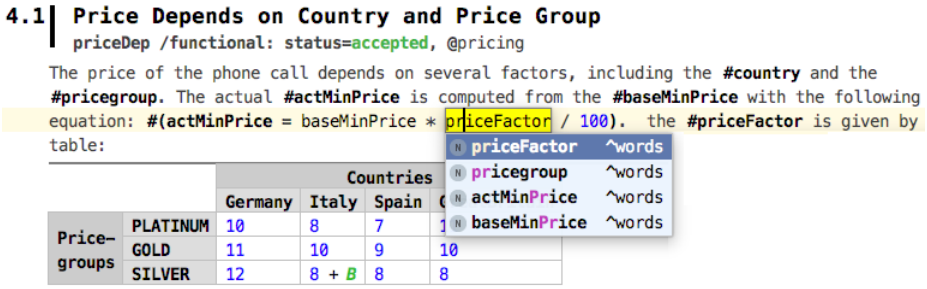

Fig. 10. Example from the mebddr requirements language. The number (`4.1`) is automatically computed from the nesting level and is not editable. The title is simple text. The `/functional` is one value from an enumerated type, same with the status. The `@pricing` is a predefined tag. The text paragraph is unstructured text with the `#` introducing embedded variable definitions. Those variables can contain an expression that can in turn reference other embedded variables, with full IDE and type system support. Finally, the table is a modular extension of the requirements language for data tables. The table supports full expressions as its data, as illustrated by the `8 + B`, where `B` is a reference to a constant.

are fundamentally unstructured, so users can use all the editing gestures known from regular text editors (which is unnatural to MPS' projectional editor, see [1]). However, the text is still stored in an AST (a list of `Word` nodes), and by implementing custom `Words`, one can embed arbitrary structured nodes into the text paragraph. This leads to full and extensible support for semi-structured content. The requirements document in Figure 10 is an example.

3.7 G Separation of Concerns

In software engineering, concerns are separated[18] for many reasons. Among the most prevalent ones are:

1. Different stakeholders contribute different ingredients to a system, and each of them should be able to concentrate on their part, with an optimized language.
2. Different ingredients of a system are contributed at different times during the development process, so separating the respective artifacts is useful.
3. There is a `1:n` relationship between instances of concern C_1 and concern C_2; separating the concerns allows having multiple instances of C_2 for a given single instance of C_1 (for example, having multiple UI definitions for a single data model definition).

A drawback of concern separation is that the concerns usually have to be recombined at some point, for example for an analysis or for constructing the full system; so some semantic connection is required between the models (and the underlying languages). It can also be hard for users to understand the interactions of the separated conderns; they have to recombine them in their head.

[18] `https://en.wikipedia.org/wiki/Separation_of_concerns`

The needs of use cases 1 and 2 (stakeholder/process-step specific abstractions and notations) often lead to the use of different tools for the different concerns, which, in turn, makes rejoining the separated concerns hard from a technical perspective (as opposed to the semantic perspective mentioned above). This problem does not exist in our approach: because of MPS' flexibility with abstractions and notations, we can credibly represent a very wide range of concerns within MPS[19], making reintegration technically trivial.

Many of our DSLs are explicitly built to factor out domain logic from technical aspects: the models/languages capture the domain logic, and the generators/interpreters/platforms contribute the technical aspects. In this case, the models often mix the various concerns *within* the domain logic, while the technical ones handled separately by the infrastructure.

MPS supports annotations. An annotation can be used to add additional data (AST nodes) to a model *without the underlying language definition knowing about it*. This means that, on model level, concerns are not separated, because the annotation is structurally and synantically inlined in the model. However, on language level, concerns remain separated. We use this regularly, for example, to parametrize advanced analyses on models, to attach documentation or to attach presence conditions for product line variability.

Finally, as mentioned above, MPS supports multiple projections for the same underlying AST. In particular, a particular projection can *not* show parts of the AST. This can be used to render a model in a concern-specific way while the underlying AST stored all concerns' data in one integrated resource.

~ ~ ~

mbeddr's Build mbeddr programs use a build configuration to specify how the executable is built from the source modules. It can be seen as a mix of makefile, compiler switches as well as instructions on how to reduce language extensions to plain C. For example, one can configure whether the components should support interface polymorphism (flexible) or not (faster, lower memory consumption). The main drivers for separating this concern are the above-memtioned reasons 2 (process) and 3 (`1:n` – many executables from the same sources).

Verification in mbeddr CBMC-based verifications of C code in mbeddr requires non-trivial parametrization of the verifier: which default analyses should be performed (NaN, div-by-zero, array-bounds), how far should loops be unwound, should slicing be turned on or not. Setting these parameters sensibly requires experimentation and/or experience with CBMC. This is why the configuration is kept in a separate concern model (use cases 1 and 2).

Healthcare DSL In the KernelF-based healthcare DSL mentioned earlier we separate four different concerns, mainly for reasons 1 and 2: complex decisions

[19] Of course, in some projects it is necessary for organizational reasons to keep some concerns in different tools; typical examples are requirements, architecture models or low-level implementation. There are various ways of integrating external data into MPS, from actual import to stub models to using URLs or other pointers to reference data stored in non-MPS files or other resources.

(defined and reviewed by healthcare professionals), overall execution of the algorithm (contributed by healthcare and software people), test scenarios (reviewed by healthcarers) and configuration of derived artifacts (contributed by software people, later in the process). There is also a use of multiple projections: states and transitions in the main algorithm state machine can be annotated as `good` and `exceptional` tags; a particular projection shows only the good case to allow reviewers to understand the essence of the algorithm.

Spacecraft Architecture based on mbeddr Wortmann [31] describes an approach where the abstractions of ESA's reference architecture for spacecraft software[20] were directly integrated into C through mbeddr language extensions. This is an example of the integration (as opposed to separation) of concerns: in the traditional approach based on a UML modeling tool and a classical C IDE, integrating the architecture description (UML model) and the implementation (C code) was *technically* hard, so they were separated. mbeddr, plus extensions, allowed OHB to create an integrated program/model of the two without losing the semantics of the abstractions defined in the reference architecture.

3.8 H Integration of Stakeholders

Integrating domain experts, who are typically not programmers, more directly into the development of software has been a long-standing goal of software engineering in order to get rid of inefficiencies in the development process. Domain-specific languages at the appropriate abstraction level that employ notations borrowed from the domain can make a significant contribution toward this goal. Bringing models "alive" through simulation and testing is another important ingredient.

~ ~ ~

Public Benefits Another DSL that extends KernelF is used for public benefits calculations. It is quite different in style from the salary/tax calculation language in that it relies much more on first-class domain abstractions and uses a notation that more closely resembles forms, while still supporting expressions (with full tool support) inside form fields (see Figure 11). Execution of the models for the purpose of testing is supported through extensions to KernelF's interpreter.

Healthcare In the healthcare DSL, healthcare professionals and software engineers create algorithms collaboratively, essentially by pair programming. While the asynchronous, state-based main algorithm is harder for the medical personnel to get used to, the tabular and tree-based decisions procedures are very accessible. The overall algorithm is then validated by doctors through generated flowcharts, and through interactive simulations based on interpreters.

Salary/Tax In the salary/tax DSL, one reason for acceptance by domain experts is that the language directly supports working with temporal data, which dramatically simplifies expressing idiomatic calculations. In addition, the language is designed in a way to support efficient, reactive execution; the domain experts, when writing calculation code, are isolated from this technical concern.

[20] `http://savoir.estec.esa.int/SAVOIRDocuments.htm`

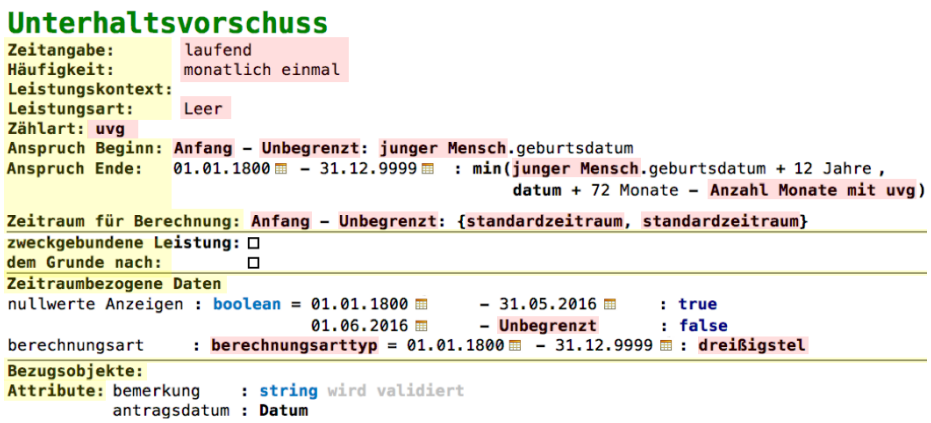

Fig. 11. Public benefits calculations using a form-style DSL. The DSL is quite form-oriented; the yellow shaded parts are rigid, and cannot be edited. The parts shaded red are domain-specific expressions and types that play an important role in the usability of the DSL for the end users. Both shadings are not present in the actual tool.

3.9 I Powerful, productivity-focused IDEs

MPS provides all the features known from programming-language IDEs: syntax highlighting, code completion, error markup, quick fixes (aka intentions), refactorings, goto definition, find references, search and replace, and version control integration including diff/merge *for all supported notations*. Most of these editor services work automatically once the language and its notation is defined, but all of them can also be customized. In addition, for non-textual notations, buttons, menus and palettes can be added to toolbars, views or inline in the notations themselves (such as little buttons to add new connections in diagrams, or buttons to add/delete rows/columns in tables). Below we discuss particular examples that go beyond these generic IDE features.

~ ~ ~

Validity of Rules In the salary/tax language, calculation rules have a validity period in order to represent the evolution of the underlying law. For example, the church tax might be calculated in one particular way until the end of 2017, but then in another way starting from 2018; each rules specifies its validity period. Once the system has been used for a while, there will be different incarnations of calculation rules for each particular data item, and developers will have a hard time understanding the set of rules valid for a particular point in time. To solve this issue, the IDE allows users to select a date, and it then renders the program code in a way where only the rules relevant for this date are shown.

Context Navigation MPS supports context actions, i.e., buttons in a palette that are specific to the currently selected program node. In the salary/tax language we use this to support context navigation (Figure 12). For a given data item, the actions allow quick navigation to all rules that calculate that particular item, and to all downstream calculation rules that depend on the current data item.

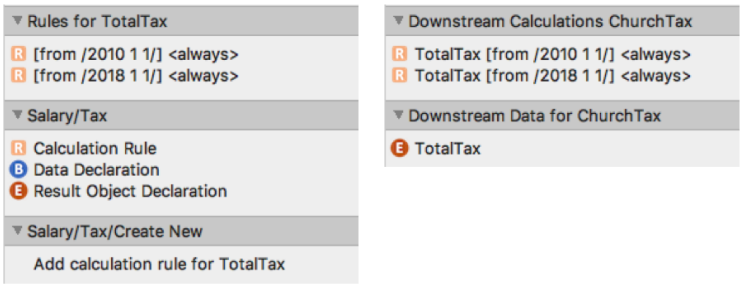

Fig. 12. Palette actions that support context-specific navigation for the currently selected program node in the salary/tax language.

Similarly, when a rule is selected, all alternative rules (different validity period or different precondition) are shown.

Variants in mbeddr In mbeddr, parts of programs can be annotated with presence conditions, i.e., Boolean expressions over features in a feature model (this can be seen as a structurally clean version of `#ifdef`s). Users can select a particular configuration (i.e., set of selected features), and the code can be projected to only show those parts that are present in this configuration, based on evaluating the Boolean expressions. In the full projection, parts that have the same presence condition are shaded with the same color to help users quickly identify parts that are in the same configuration

Tool as Language In traditional languages and their IDEs, many added-value services, for example, those for program understanding, testing, and debugging, rely on service-specific UI elements (such as tool windows, buttons, or menus). Because of MPS' flexibility in how editors can be defined, we use languages and language extensions instead of these UI elements. Examples include the REPL and its rendering of strucutured values (Figure 15), the overlay of variable values over the program code during debugging Figure 14, test coverage and other reports over the program, generated test vectors and their validity state (Figure 13) and the diffing of mutated programs vs. their original in the context of mutation testing. This leads to a less complex appearance of the IDE, which fits well with our domain expert users.[21]

3.10 J Liveness

Programs (or models that describe behavior) are essentially recipes of how something will behave once it is fed with particular inputs. In other words, the *program* is different from its *execution*: a program can "run". Special tools called debuggers help programmers understand the execution of their programs. Contrast this with a spreadsheet: a spreadsheet does not run (alternatively you can also say that it always runs). In any case there is no distinction between

[21] We have to make more progress in this direction to make MPS suitable for a wider range of domain experts.

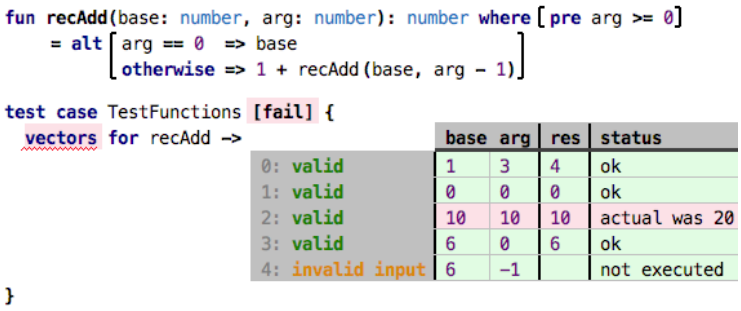

Fig. 13. Test vectors for testing a function `recAdd`; changing the function implementation or any of the inputs will immediately update the success status (incl. shade color) and the actual output value.

the recipe its execution for a particular set of inputs. This is generally helpful because during programming, users do not have to imagine how the program will run later.[22] Live programmig is about removing the separation between a program and its execution, or at least using tools to help users bridge this gap more easily than by staring at program code. For example, the Live workshop[23] defines liveness as follows:

> *Live programming systems abandon the traditional edit-compile-run cycle in favor of fluid user experiences that encourages powerful new ways of "thinking to code" and enables programmers to see and understand their program executions.*

According to this definition, we have made progress in this direction.

~ ~ ~

In-IDE Interpreter We mentioned the in-IDE interpreter before. It is typically executed as part of the type system.[24] This means that any change to the input data, or the program itself, will instantly update the output, very much "abandoning the ... edit-compile-run cycle". With the projectional editor it is easy to render computed data, such as a program's outputs, directly into an editor. One use of this appraoch is with (generated or manually written) test vectors, as shown in Figure 13. Since the programs (in this language) are functional, this setup is just like Excel – the computation state is "right there".

Debugger for KernelF We also support a debugger for KernelF. Again, since (at least in its basic version) KernelF does not support effects, a computation can be shown as a "state" – no interactive stepping, as a means of representing the passage of time, is necessary. However, it is possible that the same piece of code

[22] On the other hand, it is hard to see how a spreadsheet-like approach can be generalized, and made to scale. It is well known that there are serious problems with the quality of spreadsheets. There is probably a reason for the recipe/execution separation.

[23] `https://2017.splashcon.org/track/live-2017`

[24] If the computation is big, it can also be triggered explicitly, avoiding slowing down the type system.

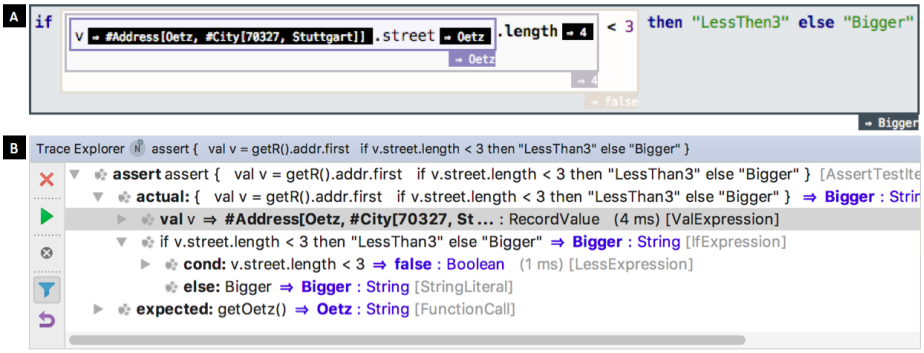

Fig. 14. KernelF's debugger. B: Tree view that represents *executions* of the underlying program nodes. A: An execution's value overlay over the corresponding source code.

is executed more than once because of loops (for example, through higher-order functions on collections) or through recursion. This is why the debugger has two components: a tree view (Figure 14 B) shows the computation tree, where each tree node represents an *execution* of a program node and shows its value *in that execution*. In addition, the values of the program nodes in a particular execution can be overlaid over the source by double-clicking the execution node in the tree. This is shown in Figure 14 A.

REPL and Interactors KernelF also comes with a REPL where users can enter expressions, see their values, and then use the value in downstream expressions (by referencing the `i`-th output by `$i`). Users can open a REPL for any expression via an intention. The REPL has special support for interactors, such as the decision procedures discussed earlier in the context of smart contracts. Interactors expose their (mutable) state through a generic API, so users can see the interactor evolve as they send commands via the REPL. In addition, because the state is represented in a homogeneous, structured way, the REPL can highlight the diff of the state between subsequent steps (Figure 15).

4 The Meta Level

Large parts of MPS are bootstrapped: the facilities for language definition are built with MPS itself. In other words, the discussion about programming and modeling can also be applied to MPS itself. We do this briefly in this section.

A High-level, Domain-Specific Concepts MPS has first-class abstractions for many of the ingredients of languages including structure, notation or type systems. DSLs for expressing IDE services are also available. These are defined as MPS languages, and they can be extended like any other MPS language. For example, the grammar cells [29] that make it easier to define usable text editors are an extension of MPS' language for defining textual editors. The languages for defining tables and diagrams have also been developed by our team, as a non-invasive extension to MPS itself. The same is true for the DSL used to define

Fig. 15. A REPL session that modifies an interactor through commands; the REPL shows the state internal state of the interactor, as well as its evolution based on a diff.

interpreters. IncA [16] is a DSL for incremental program analysis built with MPS, deeply integrated into MPS to support analysis of MPS-defined languages.

B User-definable Abstractions All language definition languages (except the structure) are extensions of BaseLanguage, MPS' version of Java. So while the DSLs for type systems, editors or constraints do not allow the definition of abstractions using these languages, users can always use BaseLanguage and factor behavior into classes.

C Focus on Behavior and Algorithms The only language that is purely structural is the structure definition language. All others express some kind of

behavior, be it typing rules, scope definitions or the interactive aspects of editors based on the various supported notations.

D Type Systems MPS has a type system that bascially resembles the Java type system, including generics. In addition, the type system also reflects MPS' meta meta model, so users can work with ASTs in a typesafe way (up to a point). For example, when users construct an AST of a `Function`, and they programmatically insert an instance of `Statement` in the `arguments` slot (which structurally expects `Arguments`), the type system reports an error.

E Focus on Execution The MPS language definition languages focus on execution in the sense of "running the language". Technically, all language definition models are generated into Java code which is then dynamically loaded and executed to "run" the new language in MPS.

F Notational Freedom The language definition languages rely largely on textual notations; only the notation definition languages uses a grid-style layout. The major reason for this focus on text is probably that, at the time the language definition languages were implemented, the non-textual notations were not yet reliably supported in MPS.

G Separation of Concerns Every language aspect has its own language, and the various aspects are also defined in separate models.

H Integration of Stakeholders The stakeholders of MPS are primarily the language developers. Integration of domain experts, i.e., the users of languages, is done by running the language so they can experience it. The meta languages are not suitable for review by non-software people.

I Powerful, productivity-focused IDEs Since MPS languages are developed in MPS, and since language definition languages are MPS languages, all MPS IDE features are available for the language developer.

J Liveness After a change to a language definition model, users press `Ctrl-F9` to build the language implementation (compile, dynamic reload). This is reasonably fast for realistically-sized languages. So while the edit-compile-run cycle still exists, it is fast enough for fluid work. Liveness in the stricter sense of the word is not supported.

5 Conclusions

Programming and modeling, in the sense of model-driven, where models are automatically transformed into the real system, cannot be categorically distinguished. However, the two have traditionally emphasized different aspects differently, making each suitable for different use cases.

Our approach to building (more or less) domain-specific modeling languages and environments combines ingredients from both fields in a novel way, relying on two pillars: projectional editing, which lets us use a wide range of notations, and language modularity, which allows us to reify library abstractions as language abstraction, which, in turn, leads to better analyzability and IDE support. Those

two, plus the practice of growing a general-purpose base language towards a domain, give us the necessary range in abstraction to cover "typical" modeling and programming tasks, as the many examples in this paper demonstrate.

~ ~ ~

MPS' spiritual father, Sergey Dmitriev, calls this approach language-oriented programming; language-oriented programming can be seen as the 21st century version of model-driven. Based on the observation that it is essentially impossible to nail down the difference between a (prescriptive) model and a (suitably abstract) program, we propose the following, very pragmatic definition of "model":

Definition: A model is an artifact expressed with a language defined in a language workbench that supports the features illustrated in this paper.

The reason for this definition is that, because of the language workbench's capabilities, one can incrementally add first-class abstractions, analyses, tool support and notations to "programs" to give them all the properties of a "model".

Acknowledgements While I *wrote* the paper, the material discussed in the paper would not have been possible without the team at itemis. So I want to acknowledge everybody who contributed ideas, code, or validation. In addition, I want to thank the MPS team at Jetbrains for building an amazing tool and for helping us use it productively over the years. Finally, Tamas Szabo provided usesful feedback on the paper, so I want to thank him specifically. The same thanks goes to my ISOLA reviewers.

References

1. T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing: A controlled experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 763–774. ACM, 2016.
2. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
3. K. Czarnecki, U. W. Eisenecker, and K. Czarnecki. *Generative programming: methods, tools, and applications*, volume 16. Addison Wesley Reading, 2000.
4. E. Durr and J. Van Katwijk. Vdm++, a formal specification language for object-oriented designs. In *1992 Proceedings Computer Systems and Software Engineering*, pages 214–219. IEEE, 1992.
5. S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.
6. M. Fowler. Language workbenches: The killer-app for domain specific languages. 2005.
7. J. Goguen and J. Meseguer. Rapid prototyping: in the obj executable specification language. *ACM SIGSOFT Software Engineering Notes*, 7(5):75–84, 1982.

8. T. R. Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.
9. G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.
10. C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
11. D. Kroening and M. Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.
12. J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.
13. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on software engineering*, 26(1):70–93, 2000.
14. Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific c verification with mbeddr. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 539–550. ACM, 2014.
15. D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, pages 9–15. IEEE, 2012.
16. T. Szabó, S. Erdweg, and M. Voelter. Inca: A dsl for the definition of incremental program analyses. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 320–331. IEEE, 2016.
17. S. L. Tanimoto. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*, pages 31–34. IEEE Press, 2013.
18. O. van Rest, G. Wachsmuth, J. R. Steel, J. G. Süß, and E. Visser. Robust real-time synchronization between textual and graphical editors. In *International Conference on Theory and Practice of Model Transformations*, pages 92–107. Springer, 2013.
19. M. Voelter. A smart contract development stack. [Online; posted 6 Dec, 2017].
20. M. Voelter. Language and ide development, modularization and composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.
21. M. Voelter. An argument for the isolation of 'fachlichkeit', 2017. https://languageengineering.io/an-argument-for-the-isolation-of-fachlichkeit-3a67a939d23b.
22. M. Voelter. Thoughts on 'declarativeness', 2017. https://languageengineering.io/thoughts-on-declarativeness-fc4cfd4f1832.
23. M. Voelter. The design, evolution, and use of kernelf. In *International Conference on Theory and Practice of Model Transformations*, pages 3–55. Springer, 2018.
24. M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: A case study. In *Proceedings of OOPSLA 2015*, pages 655–674. ACM, 2015.
25. M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann. Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling*, pages 1–24, 2018.
26. M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, Jan 2017.

27. M. Voelter and S. Lisson. Supporting Diverse Notations in MPS' Projectional Editor. *GEMOC Workshop*.
28. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):1–52, 2013.
29. M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 28–40. ACM, 2016.
30. J. Woodcock and J. Davies. *Using Z: Specification ,Refinement ,and Proof*. Prentice Hall International, 1996.
31. A. Wortmann and M. Beet. Domain specific languages for efficient satellite control software development. In *DASIA 2016*, volume 736, 2016.
32. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.