The Art of Bootstrapping

Andreas Prinz[1](✉) and Gergely Mezei[2]

[1] Department of ICT, University of Agder, Grimstad, Norway
andreas.prinz@uia.no
[2] Department of Automation and Applied Informatics,
Budapest University of Technology and Economics, Budapest, Hungary
gmezei@aut.bme.hu

Abstract. Language workbenches are used to define languages using appropriate meta-languages. Meta-languages are also just languages and can, therefore, be defined using themselves. The process is called bootstrapping and is often difficult to achieve.

This paper compares four different bootstrapping solutions. The EMF environment and the Meta-Programming System (MPS) use a compiled bootstrapping for their own definition. The platforms LanguageLab and DMLA are using interpreted bootstrapping. This paper compares these kinds of bootstrapping and relates them to the definition of instantiation. Besides the structural aspects of the bootstraps, the dynamism is also elaborated. It is shown how the bootstrap is related to the execution environment. Finally, the level of changeability is also discussed. It is shown that all approaches are quite similar and provide very flexible environments.

Keywords: Language workbench · Bootstrapping · Metamodelling

1 Introduction

Metamodelling [12] is an approach to define languages using metalanguages. In turn, these languages can be used to define specifications (programs) of new languages. Typically, one would use a metamodelling environment, also called language workbench[1] [7,9,25,28], to define languages and metalanguages. The metalanguages are also languages and thus they can be defined using the same or different metalanguages. This process is called bootstrapping, and it is mainly an advantage for the workbench developers. In general, the term 'bootstrapping' is used in several contexts, e. g. to refer to the process to load the initial constructs of an engine to start it. In this paper, bootstrapping refers to a self-descriptive process that acts as the base point of modelling. Bootstrapping is typically not visible to the users. This paper discusses the similarities

[1] There are also grammar-based language workbenches in addition to the metamodelling environments.

and the differences between four language workbenches, namely Eclipse Modeling Framework (EMF) [24], JetBrains Meta Programming System (MPS) [3,22], LanguageLab [11] and Dynamic Multi-Layer Algebra (DMLA) [5].

EMF is tightly connected to the Eclipse [4] infrastructure. In particular, it is used for the structure definitions of many Eclipse projects. EMF provides high-level descriptions of classes and their relationships similar to MOF [17]. EMF descriptions are simple class diagrams that are translated into Java classes that integrate with the Eclipse infrastructure. Another output from the EMF specifications is an exchange storage format XMI [18] the production of which is auto-generated.

The open-source industrial-strength *Meta-Programming System (MPS)* is provided by the company Jetbrains. MPS has several meta-languages covering a wide range of language-design elements in order to support comprehensive language design. All the meta-languages within MPS are defined using MPS itself making the platform bootstrapped. MPS features a language called Base Language resembling Java, which is then available for language extension and development inside MPS. For example, it is possible to define new constructs for Java (Base Language) within MPS. In the project mbeddr [26], MPS provides an almost complete definition of C++, allowing C++ to be used and extended within MPS. This is exactly what mbeddr does: extending C++ in MPS with state machines and units for use in embedded device programming.

LanguageLab is an academic project at the university of Agder, Norway. It is not concerned with an industrial strength environment, but rather with the concepts that are needed to make metamodelling user-friendly and feasible. The main goal of LanguageLab is to show the essential concepts of metamodelling in a clear and understandable tool, such that it can be used in university teaching [10]. In the same way as MPS, also LanguageLab is bootstrapped such that the few meta-languages of LanguageLab are defined within LanguageLab.

Dynamic Multi-Layer Algebra (DMLA) is also an academic project. It is developed at the Budapest University of Technology and Economics, Hungary. DMLA provides an environment, where stepwise refinement of concepts is possible from the highly abstract initial ideas to the fully concretized final specifications. The approach ensures rigorous validation along the whole process based on constraints specified during the refinement. The goals of the approach are achieved by a multi-layer modelling structure with a built-in, fully modelled operation language. The initial modelling structure, i.e. the bootstrap of DMLA is self-describing and self-validating.

These four language workbenches have different focus. EMF is mostly related to defining structure. The general idea of MPS is to provide the user with as much as possible help in defining languages and using known notation when possible. LanguageLab is focused on a clean environment, and in this context, it tries to avoid exposing the underlying language of the platform to the user. DMLA has focus on validated multi-layer modelling and refinement.

All four platforms are implemented in Java, and this is very visible in EMF and MPS, but not much visible in LanguageLab and DMLA. All four platforms are used to define themselves, and this paper compares these four self-definitions.

The paper starts with an introduction of the essential terms and concepts of metamodelling in Sect. 2. In Sect. 3, more details of the four platforms are provided, before we describe their bootstrap in Sect. 4. Runtime aspects are discussed in Sect. 5. Finally, we conclude in Sect. 6.

2 Terminology

This section introduces the concepts and terms used in metamodelling in general, and in EMF, MPS, LanguageLab and DMLA in particular. We use basically the terms of MPS in this paper, and indicate the different terms used in EMF, LanguageLab, and DMLA.

Metamodelling is an application of model-driven development [1] to language engineering and compiler construction. Instead of implementing language tools and compilers by hand for each domain-specific language, a model of the language is created. From the model, tools are automatically generated [16]. This way, languages can be designed quickly for the programming tasks at hand [29].

A *language description* has several different aspects, which together give a complete description of all important properties of the language. There are the aspect groups e.g. of structure or semantics, as well as a group of tool-related aspects. In this paper, we focus mainly on the most relevant aspects for bootstrapping, namely structure, constraints, and semantics. Concrete syntax is not important in this context, since we can always use a predefined concrete syntax based on the abstract syntax.

The structure (abstract syntax, metamodel) aspect defines the *concepts* that are used in the language and their relationships with each other. In EMF a 'concept' is represented by a 'class'. LanguageLab uses the term 'type', while DMLA uses the term 'entity' instead.

Concepts can own *properties* (of basic types) and *children* (enclosed concepts). Properties are called 'attributes' in LanguageLab and EMF. Children are referred to as 'aggregates' in LanguageLab, while in EMF, they are represented by containment associations. In DMLA, the term 'slot' is used for both of the terms mentioned above. A slot is a placeholder that can have constraints on its type, or cardinality and can have a value. The value can be a primitive or complex value (child).

In modelling, relations between concepts are essential. Typical examples are inheritance, containment, aggregation and UML-like association relations. The containment relation is already discussed in the previous paragraph. Inheritance is referred to as *parent*, while associations are called *references* in EMF, MPS and LanguageLab. In DMLA, containment and association are not distinguished, thus, the term 'slot' is also used for references, while inheritance is currently not supported natively.

The abstract syntax introduces a large range of possible specifications. However, not all of them are meaningful, and thus, constraints are often used to define additional conditions for valid specifications. For example, there can be size constraints (not more than one parent), or type constraints (the actual parameter type must match the formal parameter type), or reference constraints (attribute types have to be defined in the enclosing scope). Constraints are boolean conditions that all have to be true for a specification to be valid.

The meaning of programs in a language is defined by its semantics. There are two main ways to achieve this: transformations (compiler) and executions (interpreter). In the first case, the model is mapped to another model or to source code, and then this other representation is executed. A typical example is a code generator that produces Java code from the model. In the second case, we have a virtual machine that can directly interpret the language models. This is discussed in more detail in Sect. 5.

3 Language Workbenches

In this section, we present the four language workbenches EMF, MPS, LanguageLab, and DMLA. They are selected because they provide very different views onto languages and bootstrapping, and they have different focus.

3.1 EMF

The Eclipse Modeling Framework (EMF) [24] is possibly the most complete and without doubt the most well-known modelling platform nowadays. EMF and Ecore (the underlying modelling foundation) are often referred to as the defactor standard of modelling. There exist many tools and applications using and extending the Eclipse-based framework of EMF in many fields, including visual modelling, metamodeling and model processing. Models are specified in XML Metadata Interchange (XMI) from which EMF provides tools and runtime support to produce a set of Java classes for the model, and also a set of adapter classes that enable viewing and editing of the model. EMF has runtime support for manipulating the models including change notification, persistence support and a reflective API for manipulating modelled objects generically. The platform focuses not only on models themselves but also on editing the models. EMF provides a basic editor and several ways to extend the editor in a textual or visual way.

From the modelling point of view, EMF is based on Ecore which itself is a reference implementation of the EMOF standard [19]. Although MOF has four modelling layers, in practice, EMF is restricted to two modelling layers (metamodel and model) from the users point of view, while the core metamodel of Ecore is self-defining.

Although the potential extension points of an EMF domain are countless, the usual modelling scenario is often code-based, not modelled. Typically, the user creates the structural definition of her domain model based on the Ecore metamodel. Then the definition is refined by additional constraints (OCL constraints or custom, Java validation methods) if required. If execution semantics is needed, the user may implement the corresponding methods in Java and execute them from the modelling environment, or use approaches such as Kermeta3 [13] to overcome the limitations of Ecore. To sum up, in case of EMF, only the structural aspect of the model definitions is modelled by its bootstrap.

3.2 MPS

JetBrains MPS is a metaprogramming system which is being developed by JetBrains. MPS is a tool to design domain-specific languages (DSL). It uses projectional editing which allows users to overcome the limits of language parsers, and build DSL editors using text, tables and diagrams. It implements language-oriented programming [29]. MPS is an environment for language definition, a language workbench, and an integrated development environment (IDE) for such languages.

Developers from different domains can benefit from domain specific language extensions in general purpose programming languages. For example, Java developers working with financial applications might benefit from built-in support for monetary values. Unfortunately, traditional text based languages are subject to text ambiguity problems which makes such extensions problematic.

MPS supports composable language definitions. This means that languages can be extended, and embedded, and these extensions can be used, and will work, in the same program in MPS. For example, if Java is extended with a better syntax for collections and then again extended with a better syntax for dates, these extensions will work well together.

MPS solves grammar ambiguity issues by working with the abstract syntax tree directly. In order to edit such a tree, a text-like projectional editor is used.

MPS provides a reusable language infrastructure which is configured with language definition languages. MPS also provides many IDE services automatically: editor, code completion, find usages, etc.

The boostrap of MPS includes a definition of almost complete Java called Base Language. It also provides languages for collections, dates, closures and regular expressions. All meta-languages (language definition languages) of MPS are defined in MPS, with languages for structure, editor, constraints, type system, and generator.

For better applicability of MPS, there is also a connection to C and C++ given by the project mbeddr [26]. Its main purpose is to provide support for the development of embedded system using MPS. It has languages tailored to embedded development and formal methods: core C language, components, physical units, and state machines.

3.3 LanguageLab

In metamodel-based language design, a major challenge is to be able to operate on an adequate level of abstraction when designing a complete computer language. There are several different technologies, meta-languages and tools in use for defining different aspects of a language, that may or may not satisfy the needs of a DSL developer when it comes to abstraction level. Before starting the design and development of the LanguageLab workbench, we set out to examine what concepts are needed for defining the different aspects of a computer language, and discuss how to apply them on a suitable level of abstraction. If the abstraction level is too high, the definition of behaviour may be a challenge, while if the abstraction level is too low, the language developer will spend too much time on unnecessary details.

The LanguageLab platform facilitates operation on a suitable abstraction level. It provides user-friendliness and a low threshold to getting started, in order to make it useful for teaching of metamodelling. The platform is open for third party language modules and it is intended to facilitate reuse of language modules, modular language development and experiments with multiple concrete syntaxes. Another goal is to supply some basic guidelines for developing LanguageLab modules that can further add to the features and capabilities of the LanguageLab platform.

Based on experiences from teaching, the core LanguageLab is a very simple metamodel-based language definition platform, attempting to remove some of the complexity of the more popular existing tools. This simplicity allows students to grasp the basic principles of metamodelling by working on small language examples on a suitable level of abstraction for each relevant language aspect.

In LanguageLab, a language definition consists of one or more modules with structured elements that can be run as an IDE for that language. Program (code) specifications work in the same way. Specifications can also be run as programs and create the intended execution as described by the language used.

LanguageLab exists as a simple prototype based on Eclipse/EMF. There is also a more advanced version built in MPS. LanguageLab is simplistic by design.

3.4 DMLA

The Dynamic Multi-Layer Algebra (DMLA) [5] is a multi-layer modelling framework based on Abstract State Machines (ASM) [2]. DMLA consists of two parts: (i) the Core containing the formal definition of modelling structures and their management functions; and (ii) the Bootstrap having a set of essential reusable entities for all modelled domains in DMLA. We have intentionally separated the two parts to be able to use the same structure with different bootstraps, like a computer, which can be used with different operating systems.

According to the Core, each concept is defined by a 4-tuple (unique ID, meta-reference, attributes and concrete values). Besides these tuples, the Core also defines basic functions to manipulate the model graph, for example, to create new model entities or query existing ones. This solution makes it possible to create a virtual machine handling the concepts and simulating the ASM functions and thus act as an implementation-independent interpreter for the models.

Although the tuple-based structure is defined by the Core, it is useless without a proper bootstrap. The role of the Bootstrap is to define the basic, essential built-in building blocks for practical modelling. For example, concepts such as

'Entity', 'Slot' and 'Constraint' are introduced here. Based on the complete setup of these initial concepts, the definition of domain models is possible. Instantiation in DMLA means gradual constraining and thus has several peculiarities. Whenever a model concept claims another concept as its meta-concept, the framework automatically validates if there is indeed a valid instantiation between the two concepts. Similarly to concepts, slots are also automatically validated against their meta-slots (each slot has a reference to its meta-slot). The rules of valid instantiation are not encoded in an external programming language (e.g. Java), but modelled by the bootstrap. The operations needed for encoding the concrete validation logic of instantiation are modelled by their abstract syntax tree (AST) representation as 4-tuples within the bootstrap.

In DMLA, multi-level behaviour is supported by 'fluid metamodeling', that is, instead of following a rigid hierarchy between the modelling levels, each concept can refer to any other concept along the complete meta-hierarchy, unless the reference is found to be contradictory to the validation rules. This way, instantiation in DMLA acts as a refinement relation, where the concepts to be refined can be referenced anywhere within the modelling space. DMLA also supports horizontal refinement, when previously existing slots are cloned, not concretized, but new slots may be added and thus the original concept may be extended. Note that this behaviour is very similar to inheritance, therefore we use it to simulate inheritance relationships.

From the practical applications point of view, there are two major versions of DMLA workbenches. The old version has an XText-based editing environment, where users can edit their models using our scripting language, DMLAScript [27]. DMLAScipt is only syntactic sugar to hide the tuple-based structure from users. The models defined by the scripts are translated to tuples and the tuples to Java. The generated code acts as a set of instructions for the underlying abstract state machine developed in Java. The main limitation of this approach is that the model is validated as a whole and dynamic modification is not supported. In other words, one can define a hierarchy of refinements and validate it, but cannot apply operations on the models interactively. In order to solve this, we are working on a new workbench based on GraalVM [20] and Truffle [21]. In this new environment, we provide a complete virtual machine for the users, which can be used interactively, e.g. run operations altering the modelling concepts. In this paper, we will refer to only this new version of DMLA.

4 Bootstrap

Bootstrapping is connected to a circularity in definition. In the context of language workbenches, this means that one language is defined using itself or a set of languages is defined using that same set. It is a good idea to distinguish between the bootstrap situation and the bootstrap process.

Bootstrap Situation. The bootstrap situation is a static situation describing how languages and concepts depend on each other. For normal use in a language workbench, new languages are defined using existing meta-languages. Here, the new languages depend on the meta-languages, as they are defined using them. The meta-languages are just used and have no dependency on the new languages.

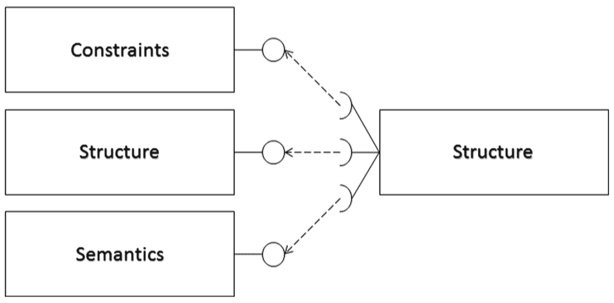


Fig. 1. Circularity between languages.

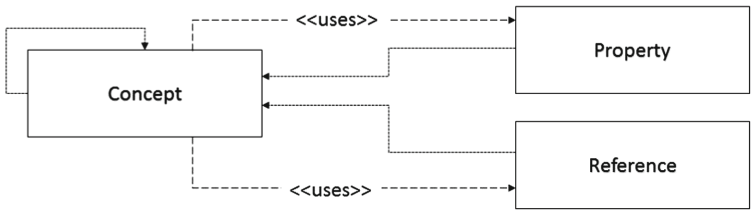


Fig. 2. Circularity between concepts.

Looking at the meta-languages themselves, they are typically also defined using the same meta-languages, thereby establishing a circular definition-use dependency. For example, the structure meta-language is defined using three meta-languages: the structure meta-language, the constraint meta-language and the semantics meta-language (Fig. 1).

Just by looking at the language level, it is hard to open up the challenge of circularity. Instead, it is valuable to look at the dependencies on the concept level. In Fig. 2, we show the details of the circularity displayed in Fig. 1 from the concept level's point of view. The three concepts 'Concept', 'Property', and 'Reference' are defined in the structure language, but all of them are in fact instances of the concept 'Concept'. They are concepts and therefore they may have properties (like a name), children (like methods) and references (like a type), which are instances of Property, Child and Reference, respectively. The dependency relationship that leads to the circularity is the instantiation relation.

Language workbenches may offer different solutions, but at the very core, they have to answer the same question: how are the concepts defined using each other?

Bootstrap Process. The bootstrap process is the technical process that leads from a situation without circularity to the situation with circularity. Often, this is a technically challenging step and is difficult to understand, see [14,23].

In the process of bootstrap, the definition of Concept needs reference to properties, children and references, but in order to achieve this, we need to have the concept of property, containment and reference, respectively. However, the definition of property should be based on the concept 'Concept' that we are just trying to create. Often, it is possible to start with empty definitions that can be filled later, leading to several concepts to be defined at once and referencing each other. This leads to a tightly coupled set of concepts together forming the essence of the bootstrap definition.

The tricky part is typically the self-reference, which would imply that a concept is defined before it is defined, which is impossible. Often, this situation has to be established using means from outside the language workbench. In this paper, we focus more on the bootstrap situation as the process is only needed once, and then the bootstrap situation can be used again and again.

4.1 The Bootstrap of EMF

The main motivation behind creating EMF was to have a universal modelling and model processing platform that can be used in practical scenarios and application development. The infrastructure is therefore made to be highly customizable and easy to extend using a flexible modelling solution behind. This modelling solution is Ecore [24], which has a strong resemblance with the EMOF variant of MOF [17], but has more focus on practical functionality.

Ecore is the metamodel for EMF. It is self-defining because it is used to explain its own classes. The elements and the structure of Ecore are translated into native Java classes, which again are used to represent the metamodel of Ecore. This way, Ecore concepts act more as interface specifications rather than implementation classes, since their logic is not modelled. For example, the root concept EObject has a method eContainer() that returns its container concept (or null, if it does not have one), but it is not specified in Ecore how the container is retrieved. More precisely the specification is available but only as an informal textual description written in English.

The reason for this is that EMF is only a structural tool with focus on navigating and manipulating the concepts and their properties. ECore does not take into account aspects of constraints or semantics. The structural parts described by Ecore can be extended by constraints as EAnnotations. From the practical points of view, several methods exist to handle the constraints e.g. adding annotated Java methods, or using projects such as Eclipse OCL [6]. However, from the theoretical points of view, unfortunately, this also means that Ecore itself does not support adding and validating constraints.

There is no support for modelling the operation logic (i.e. dynamic semantics) in Ecore either. Similar to constraints, we can easily attach methods to the model written in Java, or use Eclipse-based projects to model the operations, but Ecore is capable only to describe the interface of operations.

Ecore, as an interface specification, is not only able to describe itself, also the complete infrastructure of EMF is built upon the current version of Ecore. So changes to the Ecore, although possible, might invalidate many Eclipse projects being build using EMF, since the Java implementation relies on existing concepts and relations between the concepts.

4.2 The Bootstrap of MPS

MPS uses all its meta-languages for the bootstrap, i.e. structure, editor, constraints, behaviour, typesystem, intentions, accessories, generator, runtime, actions, dataflow, refactorings, textgen and version control. Based on the focus of this paper, we only consider the aspects structure, constraints, and typesystem as well as generator (semantics). All the other aspects are mainly used to improve the user experience, and the bootstrap works also without them.

This leads to the language dependencies shown in Fig. 3, which are explained in more detail in the next subsection. At the concept level, the name of the self-referential concept is *ConceptDeclaration*, not *Concept*. Properties and references are instances of *ConceptDeclaration*, and they are attached to the concept declarations as already shown in Fig. 2. This way they are not involved in the direct circularity. A similar argument applies to *EditorDeclaration* and *ConstraintDeclaration*, which are instances of *ConceptDeclaration*. They are attached to concept declarations using references.

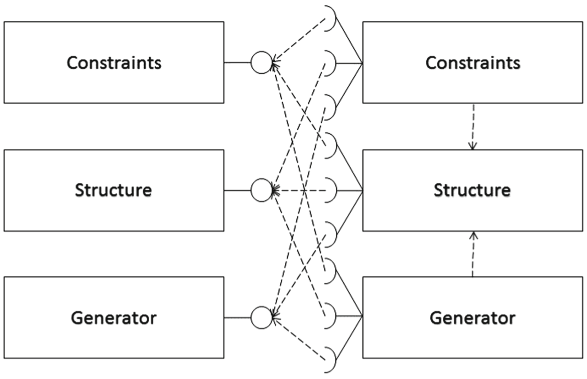


Fig. 3. Bootstrap in MPS and LanguageLab.

The bootstrapping process in MPS is enabled by the generated Java code. In a first attempt, the code is created semi-automatically, and then the loop is established and the new code generation will recreate the correct code. After the loop is existing, also changes to the bootstrap situation are possible.

4.3 The Bootstrap of LanguageLab

The bootstrap of LanguageLab is shown in Fig. 3. This figure depicts three languages in their meta-language role on the left, and in their language role on the right. Still, the languages are the same as indicated by the same name.

There are two kinds of dependencies in Fig. 3: use dependencies and references. A use dependency is a connection between a meta-language and a language, i.e. an arrow from the right to the left. Essentially it means that a description is dependent on the language it is written in. A reference is given between languages on the same meta-level. In our case, both the constraints language and the generator language depend on the structure language, as the constraint and the generator descriptions always refer to their concept description.

At the concept level, the situation is very similar as before. Here, the name of the self-referential concept is ‘Type’, not ‘Concept’. Property and Reference are instances of Type, and they are attached to Types. In the same way, Constraint and Semantics are instances of Type and they are attached to Types using references, see again Fig. 2.

This leads to the the structure language being central to the bootstrap, as indicated in Fig. 3. There are the following reasons for the structure language being special, all of them being related to the Type concept.

– The structure language has five incoming dependencies, three with instantiation to structure, constraints, and semantics, and two as references from constraints and semantics to Type. In fact, these two references are the only incoming cross-language references in the bootstrap.
– The structure language is needed already for the bootstrap situation. This means it is needed even without running a specification.
– Type is the only concept with a self-definition loop (Type being defined by a Type).

LanguageLab has two features to allow creating the bootstrapping situation using an appropriate bootstrapping process (see also [23]). The first feature is that LanguageLab is interpreted, that is the language description (file) is used as it is. This can be exploited by simply changing the dependency in the language description file (i.e. outside the LanguageLab platform) in order to establish the bootstrapping situation. A second helpful feature of LanguageLab is that each language use is connected to its language definition via an interface, where the actual language can be exchanged when the instance is loaded. This means that there is no direct connection between language use and language definition. With these interfaces, bootstrapping is even possible within the platform itself.

4.4 The Bootstrap of DMLA

In this paper we give a brief summary of the bootstrap elements to illustrate the mechanisms. For more details, please refer to [5].

The bootstrap of DMLA is not structured using meta-languages: the definition of constraints and operations are merged with the structural parts. They are not separated, because the structure, the validation (constraints) and the dynamic behaviour (operations) are all essential when creating model definitions, none of them would work without the other two. The main reason behind is that even the definition of type and cardinality checking are modelled. For example, by changing the implementation of the built-in TypeConstraint entity, one can alter what type conformance means. Note that the term 'implementation' refers here to an operation fully modelled by tuples of DMLA, not to a method written in code. This is why DMLA is said to be self-validating, as the validation methods are essential parts of the bootstrap model. Although in DMLA the bootstrap is not structured into separate languages, we have the same challenges to solve: the definition of the basics of structure, constraints and execution.

The topmost concept is referred to as 'Base', all other concepts are direct or indirect instances of it. Base grants three features for its instances: (i) the ability to have slots (properties, children and references), (ii) the ability to have constraints and (iii) it also defines the basic validation formulae.

The first feature grants composability. Note that even the definition of 'slot' is modelled (by the concept SlotDef), not hard coded.

The second feature (constraint containment) is used to fine-tune instantiation rules. Constraints (originated from the concept 'Constraint') are re-usable validation logic definitions and they are intensively used to specify type-, and cardinality restrictions as mentioned earlier, since no a priori instantiation rules are implemented. The same mechanism can also be used to add custom, user-defined constraints to further restrict instantiation.

The third feature of 'Base' (validation formulae) is responsible to define how to collect attached constraints of the given concept and evaluate them. The formulae are defined as operations and they are stored in slots. Instances may refine these formulae, but each concept is validated against all formulae of its meta-concepts (along the whole hierarchy). This means that the formulae may be extended making them more rigorous, but they cannot be relaxed.

All other concepts defined in the bootstrap inherit the behaviour described above since they are originated from Base. It is worth mentioning that besides the structural, and constraint-related concepts, the elements of the operation language (the expressions and statements composing an operation) are also modelled. In order to achieve this, every language element has a corresponding entity and when an operation is defined, its abstract syntax tree is built from the instances of these entities. For example, all conditional statements refer to the concept 'If' as their meta-concept.

Note that unlike in the case of the other three approaches, in DMLA, the bootstrap definition is not available as a pre-compiled binary code, when domain languages and models are built. Instead, the bootstrap is interpreted every time it is used. This way, the bootstrap situation is not handled differently than the usual models. The same rules, the same validation logic applies to the concepts of the bootstrap as for modelling concepts of other domains. Handling all concepts uniformly simplifies the underlying framework and helps in eliminating errors.

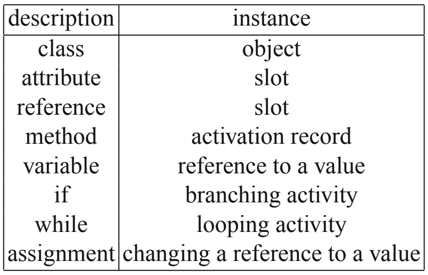
5 Dynamic Semantics

The discussion so far was related to the structure and the static aspects. Even though this is the central aspect of a language, the semantics is needed to make the language work. The structure allows us to talk about descriptions (specifications), i.e. drawings and texts. In contrast, the semantics talks about running systems and executions of the descriptions. Moreover, the semantics is the place to solve the bootstrapping circularity, as the circularity is between the description of a language and its later use, i.e. an execution.

5.1 Executing Specifications

The description provided by the model elements is passive in nature. It does not allow any activities and it will not change by itself. It has to be placed in a proper environment to be activated. We will refer to this environment as the *execution machine*. The execution machine is always bound to a language, which is the language of the descriptions it can run. The execution machine can be concrete, like a concrete computer for machine code, or abstract like the Java virtual machine for Java.

An execution machine is in some sense physical, as it can be used and executed. It has an innate ability to create executions for a description that is placed into it. In order to describe these machines, we will consider an abstract view of executions as sequences of runtime states. These runtime states are structured, and their structure is coming from the specification under execution and from additional structures used by the execution machine. As an example, if we specify a class Person in Java, then the structure of Person objects will be available as a template in Java runtime states. Figure 4 shows several examples how specification elements (descriptions) can appear at runtime (as instances).



description	instance
class	object
attribute	slot
reference	slot
method	activation record
variable	reference to a value
if	branching activity
while	looping activity
assignment	changing a reference to a value

Fig. 4. Different possible execution instantiations.

In addition to structures described by the models, the runtime state may also include elements like a program counter and exception storage. These additional structure parts are not derived from the specification but created by the execution machine mainly to help in managing the execution of the model. The structure of these additional parts is universal in the sense that it is not affected by the current specification but only by the execution machine. This also means that if we run the same specification more than once in an executuion machine, then the runtime states may differ. Often, the possible runtime states are called the runtime environment (RTE). Note that by observing the run of a program using a debugger, the current runtime state is observed.

It is the task of the execution machine to provide a way to embody the current runtime state and also to define the rules of when and how to advance the state based on the specification and the language semantics. This way, it is an execution engine which turns descriptions into living things, see also the left side of Fig. 5. Here, the specification is written in the Spec language. The execution machine for the specification (EM 4 Spec) is bound to the specification language. The running specification is placed inside the execution machine. It will have runtime states as given by the language semantics.

Introducing the execution machine breaks the self-reference loop, as the machine is working based on the laws of nature (it is physical), and the loop is embedded in the working of the execution machine.

From the abstraction levels point of view, the use of an execution machine is a vertical process in the sense that the specification – a high-level description (text) – is turned into low-level runtime states and state changes of the running specification, see again Fig. 4. For the sake of clarity, we refer to this vertical process as *execution instantiation*. It is depicted as the arrow between the running specification and the specification itself in Fig. 5. Execution instantiation relates to structural constructs like classes and methods, but also to dynamic constructs like expressions and statements.

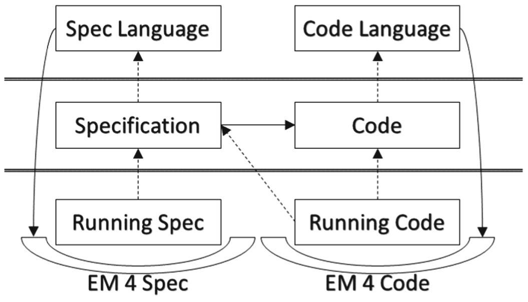


Fig. 5. Semantic approaches.

5.2 Compiling Specifications

Execution can also be achieved by translating the specification into another specification in a different language that has already an execution semantics, i.e. an execution machine. We will refer to this translation as compilation. In contrast to the aforementioned execution instantiation, compilation is a horizontal process, as the result is not an execution, but still a description. We can consider several compilation steps, but in the end, there has to be a final step to execute the description.

Figure 5 shows the same execution situation for the code (to the right) as the situation for the specification (to the left). The code is run in its own execution machine. For our discussion, we consider the connection between the original description and the final execution as a similar runtime connection, see the arrow from the running code to the specification in Fig. 5. It is irrelevant how many compilation steps are between the description and the execution - it is still one vertical step.

5.3 Semantics in the Case Languages

In EMF, there are three stages of language definitions: (i) the language description is created as the instance of the meta-languages which are later saved in the form of an XMI file; (ii) from the description Java source code is generated (compilation step); (iii) the Java source files are compiled to bytecode (compilation step), which is then loaded and thereby activated in EMF runtime (execution step). EMF runtime is basically the same as Eclipse runtime. EMF is not a complete language definition framework - the constraints and semantics parts are missing. Typically, they are specified using Java, so stage (ii) is not needed for the semantics part. From stage (iii), the JVM execution machine enriched with Eclipse functionality takes over and handles the compiled specification, thus allowing to close the bootstrapping cycle.

In MPS, the process is very similar, but MPS has its own file format and does not use XMI. In addition, MPS has meta-languages for all aspects, such that the semantics and constraints can be specified in MPS itself. Otherwise, the JVM execution machine is again the final stop. In the case of MPS, the JVM is running in the context of the IntelliJ IDEA [8], giving a richer execution machine.

The situation in LanguageLab is slightly different, given by its interpretive nature. The language description is loaded directly into the LanguageLab execution machine (LLEM), without an intermediate compilation step. The LLEM itself is defined in Java (using EMF). The semantics of LanguageLab ensures that the defining language is loaded read-only and before its instances. Changes to this language are not done until the next load of the language. This way it is possible to close the bootstrap loop.

DMLA is also interpretive and does not need a compiler step. At runtime, all specifications are given as tuples translated from DMLAScript. This way, the Core component of DMLA can be considered the execution machine for DMLA, implemented based on GraalVM and Truffle. DMLA allows changes of languages and specifications at any time at any level (even in the bootstrap). The effect of changing a concept is immediately reflected by the runtime state of the execution machine.

Both LanguageLab and DMLA are interpretive language environments allowing a complete definition of a language including semantics. The semantics is given by the underlying execution machine. Both environments can specify languages and specifications in the same framework in their languages. We can refer to these connections as in-level instantiation, which is a different instantiation than the execution instantiation introduced earlier. Execution instantiation crosses the machine boundary, while in-level instantiation is on the same level, i.e. instances are still descriptions.

LanguageLab uses its languages to extend its (generic) execution machine with the language-specific parts of the language loaded, thereby creating a specialized execution machine for the language loaded. In this situation, the language and the execution machine cannot be changed. In contrast, DMLA uses its generic execution machine for both the language and the specification. This way, the level of specification is left only at the very end of the process by the execution machine, when the machine evaluates the final form of models by interpreting them. This solution allows a very flexible change handling as explained in the next section.

5.4 Changeability

Many modelling environments are not modelled, but hard-coded and available only as a binary. In contrast to these approaches, having a self-defining, modelled bootstrap allows to modify, fine-tune or refactor the basic modelling principles and thus the modelling mechanisms much easier. Changing the bootstrap means to change all its aspects, namely structure, constraints, and semantics.

Here it is essential to remember that changes to a meta-language might make all its own definitions invalid, such that the whole platform does not work any longer. It is important to make sure that the changes maintain validity. This is discussed in more detail in [15].

Therefore, we consider the gradual change of the bootstrap situation rather than a complete replacement of it. This way, the previous self-references stay intact and the result of the bootstrap is comparable with the situation before. We assume that the changes to the bootstrap still fit with the underlying execution machine. In some sense, this sound like there is not much difference between hard-coded and bootstrapped environments. However, a bootstrap has only a few (typically far less than ten) self-referential concepts that are not allowed to change, in contrast to the hard-coded situation, where we have dozens (sometimes hundreds) of such concepts.

Changing the bootstrap of EMF is not easy. The platform, the modelling environment and the extensions (thus the whole Eclipse modelling ecosystem) rely heavily on the binaries generated from the EMF bootstrap, thus any modification to the bootstrap may cause problems. The meta-languages for EMF are Java and EMF. The bootstrap modification would work in two steps: first the new language is defined, and then it is compiled and used. As long as the generated code still adheres to the Eclipse platform requirements, such changes are possible and have been done before.

MPS is open to changes as long as the generated code fits into its execution machine (a variant of the IntelliJ [8] platform). This implies that the structure, constraint, and generator languages can be partly or completely replaced, but the core of the bootstrap, the Concept being an instance of itself, cannot be changed. The modification of the bootstrap is a two-step process similarly to EMF: first, the new languages are defined, and then the languages are compiled in order to be available for use. It is used frequently to create new versions of the meta-languages of MPS.

The situation in LanguageLab is similar, although the platform is interpreted. All meta-languages can be changed or replaced, as long as they still fit into the LanguageLab execution machine.

In the case of DMLA, the bootstrap can be easily modified. There are only a few core concepts (Base, Slot, Constraint) and basic operational statements (e.g. conditional statement), whose structure and semantics must not be modified, but otherwise, all other elements of the bootstrap and even the validation logic can be easily modified without needing to change anything in the execution machine. This is also true for operations including their inner implementation. They are executable not only directly by the execution machine, but also callable from operations. Since operation definitions are composed of model elements, they can also be changed by operations, thus it is possible to create self-changing, self-refactoring models. It is even possible to create an operation that changes its own definition.

6 Conclusion

Self-defining language workbenches tend to be more flexible and more consistent than environments defined by an external language. The methods of bootstrapping of a compiler have been discussed and used for decades in the field of the classical programming languages, but nowadays, similar solutions are needed in the field of model-based language workbenches as well. Having a precise, adaptable, self-describing bootstrap definition is not an easy task to achieve and even then, we are only at halfway, since the challenge of evaluating the definition is also to be solved.

In this paper, we have compared the bootstrapping strategies and mechanisms of four modelling platforms: EMF, MPS, LanguageLab, and DMLA. Although both the main goals and the modelling paradigms are different in the approaches, it is clearly visible that the basic constructs involved in the bootstrap situation are very similar for them.

On the language level, bootstrapping gives a very tight connection between all the involved meta-languages, which might seem to make bootstrapping impossible. However, the situation becomes solvable if we look at the concept level, where there is normally just a few concepts that are self-referential or mutually referencing each other. All four platforms have similar solutions to define the very core concepts of the bootstraps, although there are differences: EMF focuses merely on the structural aspects, MPS and LanguageLab have different languages to handle structure, constraints and semantics, while DMLA has only one language that covers all of these features.

The actual bootstrapping is given by the execution of the languages in an execution machine, which can be accessed directly (interpretive) or via transformations (compiled). In the compiled case (as in EMF and MPS), the language description is translated to another language, which is then run in its own execution machine (an extended Java VM). In the case of an interpretive approach, there are two options: (i) the execution machine can be specialized to each language and be able to handle specifications of this language as in LanguageLab; or (ii) the execution machine can be generic and both language descriptions and specifications are evaluated by the execution machine directly as in DMLA.

As the paper shows, bootstrapping is not an easy task to solve. The art of bootstrapping is multicoloured. Existing approaches are very similar in some sense, but at the same time, they are also very different. The main goal of having a self-describing bootstrap can be achieved in different ways depending on the main requirements of the users. Practical approaches, such as EMF and MPS, focus on usability and expressivity, while academic approaches, such as LanguageLab and DMLA, focus on creating a theoretically pure solution. By elaborating the main challenges of creating a bootstrap, we believe that this paper can act as a guideline to choose the right approach for the specified needs and to create new solutions having their own bootstrap.

References

1. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**(5), 36–41 (2003). https://doi.org/10.1109/MS.2003.1231149
2. Boerger, E., Stark, R.: Abstract State Machines: A Method for High-Level System Design. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
3. Campagne, F.: The MPS Language Workbench, Vol. I. Fabien Campagne (2014)
4. D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley, Boston (2005)
5. DMLA Developers: Dynamic multi-layer algebra (DMLA) official webpage. https://www.aut.bme.hu/Pages/Research/VMTS/DMLA. Accessed 12 June 2019
6. Eclipse OCL. http://projects.eclipse.org/projects/modeling.mdt.ocl. Accessed 12 June 2019
7. Erdweg, S., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 197–217. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_11
8. Fields, D., Saunders, S.: IntelliJ Idea In Action. Dreamtech Press, New Delhi (2006)
9. Fowler, M.: Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/languageWorkbench.html (2005)
10. Gjøsæter, T., Prinz, A.: Teaching computer language handling - from compiler theory to meta-modelling. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 446–460. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_14
11. Gjøsæter, T., Prinz, A.: Languagelab 1.1 user manual. Technical report, University of Agder (2013), http://brage.bibsys.no/xmlui/handle/11250/134943

12. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley Publishing, Hoboken (2008)
13. Kermeta3 developers: Kermeta3 homepage: K3 - breathe life into you metamodel. http://diverse-project.github.io/k3/index.html. Accessed 12 June 2019
14. Konat, G., Erdweg, S., Visser, E.: Bootstrapping domain-specific meta-languages in language workbenches. In: SIGPLAN Not, vol. 52, no. 3, pp. 47–58 (2016). https://doi.org/10.1145/3093335.2993242
15. Meijler, T.D., Nytun, J.P., Prinz, A., Wortmann, H.: Supporting fine-grained generative model-driven evolution. Softw. Syst. Model. **9**(3), 403–424 (2010). https://doi.org/10.1007/s10270-009-0144-1
16. Nytun, J.P., Prinz, A., Tveit, M.S.: Automatic generation of modelling tools. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 268–283. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_21
17. Object Management Group: Meta Object Facility (MOF) Core Specification. Object Management Group (2006). http://www.omg.org/cgi-bin/doc?formal/2006-01-01.pdf
18. Object Management Group: XML Metadata Interchange. Object Management Group (2015). https://www.omg.org/spec/XMI/About-XMI/
19. OMG: MOF 2.5.1. specification. https://www.omg.org/spec/MOF/2.5.1/. Accessed 12 June 2019
20. Oracle: Graalvm. https://www.graalvm.org/. Accessed 12 June 2019
21. Oracle: Truffle GitHub. http://github.com/oracle/graal/tree/master/truffle. Accessed 12 June 2019
22. Pech, V., Shatalin, A., Völter, M.: JetBrains MPS as a tool for extending java. In: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ 2013, pp. 165–168. ACM (2013). https://doi.org/10.1145/2500828.2500846
23. Prinz, A., Shatalin, A.: How to bootstrap a language workbench. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, 20–22 February 2019, pp. 345–352 (2019). https://doi.org/10.5220/0007398203450352
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley, Boston (2009). Professional
25. Stoffel, R.: Comparing language workbenches. In: MSE-seminar: Program Analysis and Transformation, pp. 18–24 (2010). http://wiki.ifs.hsr.ch/SemProgAnTr/files/ComparingLanguageWorkbenches-Roman-Stoffel-2010-12-23.pdf
26. Szabó, T., Voelter, M., Kolb, B., Ratiu, D., Schaetz, B.: Mbeddr: extensible languages for embedded software development. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT 2014, pp. 13–16. ACM, New York (2014). https://doi.org/10.1145/2663171.2663186
27. Urbán, D., Theisz, Z., Mezei, G.: Self-describing operations for multi-level metamodeling. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, 22–24 January 2018, pp. 519–527 (2018). https://doi.org/10.5220/0006656105190527
28. Völter, M.: Generic tools, specific languages. Ph.D. thesis, TU Delft, Delft University of Technology (2014). http://resolver.tudelft.nl/uuid:53c8e1e0-7a4c-43ed-9426-934c0a5a6522
29. Ward, M.P.: Language oriented programming. In: Software-Concepts and Tools, vol. 15, no. 4, pp. 147–161 (1994) http://www.tech.dmu.ac.uk/~mward/martin/papers/middle-out-t.pdf