Software Development... For All?

Democratisation and Automation in Software Development

Juan de Lara[0000−0001−9425−6362]

Computer Science Department
Universidad Autónoma de Madrid (Spain)
Juan.deLara@uam.es

Abstract. Our world runs on software. It governs all major aspects of our life. It is an enabler for research and innovation, and it is critical for business competitivity. Traditional software engineering techniques have achieved high effectiveness, but still may fall short on delivering software at the accelerated pace and with the increasing quality that future scenarios will require.
To attack this issue, some software paradigms raise the automation of software development via higher levels of abstraction through domain-specific languages (e.g., in model-driven engineering) and empowering non-professional developers with the possibility to build their own software (e.g., in low-code development approaches). In a software-demanding world, this is an attractive possibility. However, to make this possible, methods are required to tweak languages to their context of use (crucial given the diversity of backgrounds and purposes), and to assist all types of developers (professionals or not) throughout the development process.
This paper presents an overview of enabling techniques for this vision, supporting the creation of families of domain-specific languages (to consider a wide range of users and application scenarios); their automated adaptation to the usage context; and the augmentation of low-code environments with assistants to guide developers (professionals or not) in the development process.

Keywords: Software Development · Domain-Specific Languages · Model-driven Engineering · Product Lines · Conversational assistants.

1 Introduction

Software is essential in today's world. It is an enabler for most aspects of our daily lives, governing critical infrastructures like energy systems, transportation or communications, and playing a central role in many jobs and leisure activities. Quoting Bjarne Stroustroup (the creator of C++) *"our civilization runs on software"* and *"our civilization is as reliant on software as it is on water"*.

The central role that software plays today is raising its demand. Hence, professional programmers and software firms need to produce software for all sorts of platforms (web, mobile, desktop) and devices (including IoT devices, smartphones and tablets, robotic systems). Therefore, this trend requires software to be produced in greater quantities, faster, with higher quality. To achieve this goal, several strategies can be followed,

like aiming for more automation, devising more powerful languages and frameworks (to work at a higher level of abstraction) and proposing better tools.

However, professional developers are only one side of the coin, and we observe the need for non-professional programmers to develop software as well. On the one hand, this includes *amateur* programmers and hobbyists, who may use, e.g., spreadsheets [45] and other end-user programming techniques [44] to serve their needs. On the other, professionals in disciplines different from computer science that need to perform programming and development tasks as part of their work. The latter types of programmers may use low-code platforms [62,10]. However, the development languages offered by those tools are fixed and rigid, with no adaptation to the user profile and needs. Hence, I argue that there is still the need to serve low-code users better, lowering the entry barrier to software development, and making software development environments adaptable to the user profile. I argue that, only by supporting non-professional programmers, we may achieve a truly democratisation of programming.

This paper is based on the keynote talk given at the ICSOFT 2024 conference at Dijon [38]. In the rest of it, I discuss automation techniques for both, increasing productivity (to develop software faster), and democratising software development (to enable more people develop software). These techniques are founded on principles of model-driven engineering (MDE) [6], low-code development [65], software product lines [59], conversational assistants, generative artificial intelligence and large-language models (LLMs) [72,42].

2 Automating software development: models and DSLs

Fig. 1 shows a highly simplified schema of a traditional software development process [64]. For simplicity, the usual iterations, increments and the testing phases are excluded.

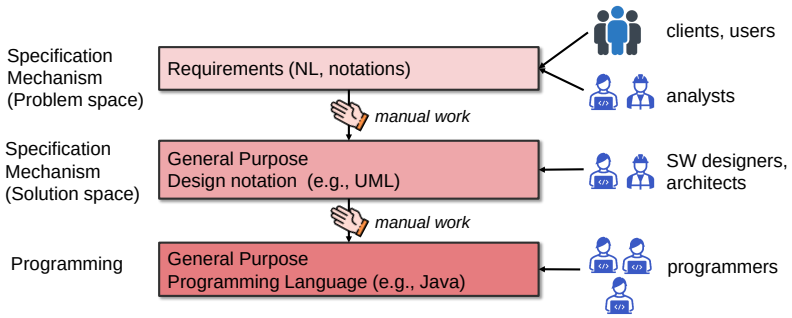

Fig. 1: A traditional development process (highly simplified).

The schema illustrates that the process starts by exploring the problem space, typically a work done by analysts together with clients and potential users. In this phase, notations for requirements engineering are used, like UML use cases [67], and also

natural language descriptions [37]. Then, the project moves to the problem space, constructing a design that satisfies the requirements. This is normally done by software designers and architects. In this stage, general-purpose design notations, like UML [67] can be used to express and reason about the design. Finally, an implementation is built based on the design. This is done by a team of programmers, using general-purpose programming languages like Java or Python. In this traditional approach, the transition from requirements to design, and from there to programming is done without automation.

Next, we will explore two ways to improve this process by exploiting the use of models and modelling, which are collectively termed as Model-driven Engineering (MDE) [6].

One possible way to increase the productivity of the previous process is to add automation in the transition from design to implementation (cf. Fig. 2). This is actually the schema proposed by OMG's Model-Driven Architecture (MDA) [47] in the early 2000's. Among other OMG standards, this approach proposes the use of UML to describe the software design, in enough level of detail to be able to produce the final code of the system (sometimes, through a series of transformations, from platform independent models, to platform specific ones, using platform description models). This entails using languages to express executable semantics, like the Action Language for Foundational UML (ALF) [2].

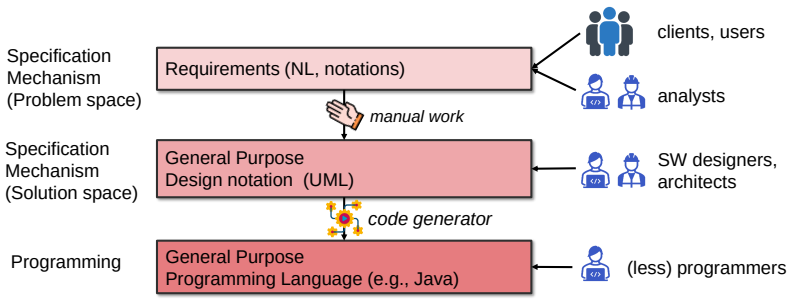

Fig. 2: Schema of the Model-driven architecture (MDA).

While partly successful[1], the MDA has some drawbacks. First, it uses UML as a programming language (and certainly UML was not invented for that[2]). Even though it is possible to define profiles [21], UML is a general-purpose modelling language, and hence may not be the best option for some specialized domains. Given the generality of UML, code generators need to be defined covering the language, and may become inflexible for certain needs. One can argue that, if all low-level details need to be specified at the UML level, the effort that it requires may yield low gain with code generation. Finally, the approach may not fit too well with agile development approaches.

[1] See `https://www.omg.org/mda/products_success.htm` for descriptions of success stories using MDA. [2] `https://www.informit.com/articles/article.aspx?p=1405569`

On reflection, it may be argued that the gap automated by MDA is much smaller than the gap between the problem and the solution spaces, as Fig. 3(a) shows. Therefore, the approach shown in Fig. 3(b) reduces the gap between the problem and the solution spaces by the use of domain-specific languages (DSLs) [35,71]. These notations are explicitly designed to solve problems in well-scoped domains. Hence, domain-specific modelling typically reduces the cognitive gap between the problem and the solution. This entails more powerful code generators able to bridge the gap between the DSL and the implementation, but it is also possible to use interpreters.

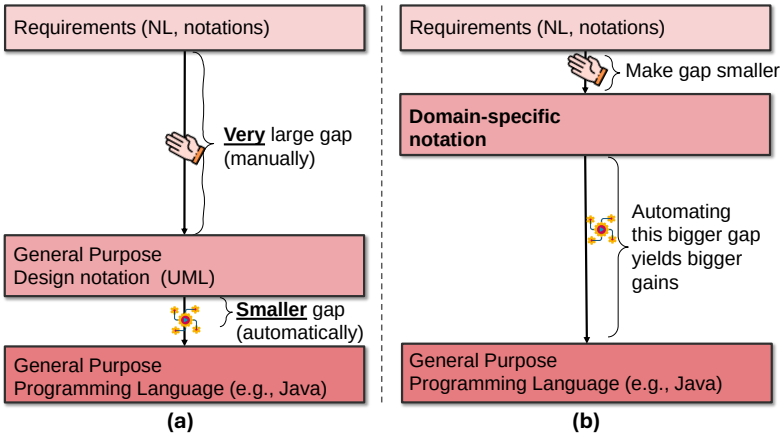

Fig. 3: MDA (a) vs Domain-specific Modelling (b).

One advantage of the domain-specific modelling approach is that the DSLs could be used by domain experts, who may not have a technical profile. Domain-specific modelling works well for specific, well-understood domains [35,52], with high gain by the automation provided. It allows modelling using the most appropriate language, at the right level of abstraction [51]. However, it also requires high investment for building the DSL-based infrastructure.

Building a DSL-based automation solution requires describing the DSL and its associated services, like code generators, interpreters, optimizers, analysis tools, debugging and testing tools, among others [68]. Normally, DSLs are built using an MDE approach via models and automation [6]. A DSL is made of an abstract syntax, a concrete syntax and semantics. The abstract syntax is typically described using a meta-model. This is a structural model (frequently, a class diagram) that describes the primitives of the domain, their properties and relations. It may also include integrity constraints described in languages like the Object Constraint Language (OCL) [55]. The concrete syntax – normally either graphical [35] or textual [68] – describes how the models of the DSL are represented. Finally, the semantics describe what the models mean [29], and may be described via code generators (which may produce executable code) and interpreters (which run the models). Additional services, e.g., for refactoring, optimising, analysing

and transforming the models may be described as well, usually via model transformations [63].

Even if many success stories about MDE and domain-specific modelling approaches have been published [14,52,35,31,9], they also have drawbacks. First, MDE solutions are often very technical, hard to install and use [8]. They are frequently oriented to developers, and many are deployed on Eclipse [20], with complex installation requiring managing many dependencies. Solutions tend to be rigid, since features outside the DSL may not be viable to incorporate into the system. Finally, evolving an MDE solution is hard, since it requires evolving the DSLs and the co-evolution of all the associated artefacts [61].

Next section analyses a modern evolution of MDE-based solutions, which aims at solving some of these issues.

3 Low-code: old wine in new bottles?

Low code development platforms (LCDPs) [62,10] enable the definition of applications with little or no need for coding. Being web platforms, they are cloud-first, with zero installation cost, since they are usable from a web browser. This effectively frees the users from complex software installation and hardware requirements. Moreover, they often host the developed application also on the cloud, hence greatly facilitating the deployment and operation of the defined applications.

LCDPs are oriented to so-called *citizen developers*, digital-savvy users with non-technical profile, and limited or no programming experience. This way, LCDPs typically offer graphical DSLs, forms, and drag & drop interaction to facilitate the specification of applications.

Many vendors offer LCDPs today, including Microsoft Power Apps [49], OutSystems [56], Appian [5], Mendix [48], or Google AppSheet [23], among many others. Regarding domains, numerous LCDPs for businesses have been proposed (e.g., Power Apps, OutSystems, Mendix), but there are also LCDPs for other specific domains, like IoT (e.g., Node-RED [54]), task-oriented chatbots (Google's Dialogflow [25], Amazon Lex [3], IBM's watsonx [32]), or machine learning applications (e.g., Google's AutoML [24], DataRobot [17], Akkio [1]), among many others.

Hence, LCDPs solve some of the issues with MDE solutions: zero-cost installation, easy deployment of the generated applications, and focus on the end users (who may not necessarily be professional programmers). However, many traditional vendors of model-based solutions have just rebranded their products as low-code. Therefore, how are low-code solutions different from MDE ones?. Inspired by [62], Fig. 4 aims at highlighting the commonalities and differences in a spectrum from 1 to 5.

First, the aim of low-code solutions is reducing the effort in creating software, which is very frequently the goal of MDE solutions, but not always. Then, we distinguish between low-code solutions that are platforms (LCDPs), from those that are not.

Hence, in point 1, we have MDE solutions that do not aim at reducing the effort needed to build an application, and therefore cannot be considered as low-code. These include solutions that use models e.g., for simulation, analysis or reverse engineering. In point 2, we find MDE solutions that aim at producing applications reducing the need to

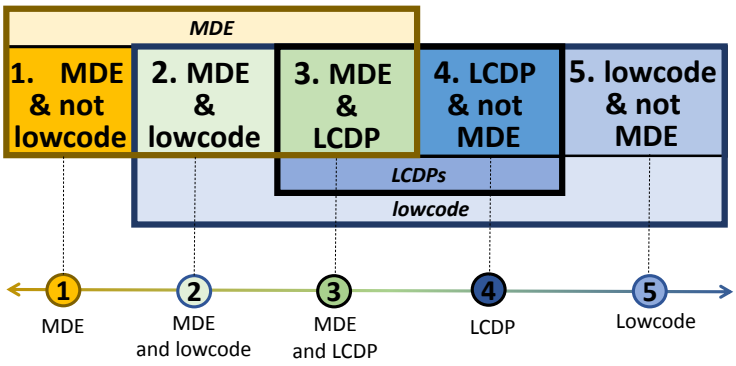

Fig. 4: The MDE – low-code spectrum.

code, and consequently are considered low-code. However, this kind of solutions are not platforms (i.e., are not accessible from a web browser, and require local installation). In step 3 we have LCDPs, built using MDE technologies. In steps 4 and 5 we have LCDPs that are not built using MDE technologies (i.e., they do not use artefacts that can be recognised as models, or DSLs), and low-code solutions that are not platforms and do not use MDE technologies. Thus, while we see a large overlap between MDE and low-code (points 2 and 3), there are also differences (points 1, 4 and 5). The reader is referred to [62] for a deeper discussion of commonalities and differences.

Generally, low-code development by means of platforms has the potential to broaden the range of their users, since these tools are accessible just via a web browser. It must be noted that not every user has the same background, the same knowledge and skills, the same interests and objectives, or uses the same device. However, s/he interacts with the platform in the same way, frequently as expected from the most expert user. Hence, low-code (and MDE) solutions suffer from *visual monolinguism* [50], where a DSL definition should fit all users and scenarios, which may not be reasonable. As a simile, learners of a foreign language do not talk as native speakers, and hence the other recipient of a conversation needs to adapt. Therefore, for a true democratisation of programming, I claim that DSLs – and their development environments – need to adapt to the user profile and context of use. To pursue this idea, next section introduces DSL families.

4 From languages to language families

This section introduces the need for language families (Section 4.1), and then reviews how to build them effectively (Section 4.2). The section continues providing an overview of the two main approaches to define language families: the annotative approach (Section 4.3) and the modular approach (Section 4.4). Finally, a discussion on how to define the concrete syntax and semantics of language families is provided in Section 4.5.

4.1 Why language families?

Language families are sets of language variants, where each variant may be suitable for different users, context of use, or modelling purposes. For example, instead of having just one notation for UML class diagrams – which should fit all users and purposes – we could have a language family, with variants to cater for:

- *Different skills and expertise levels*. Learners of UML could use a simple variant (e.g., with no composition, navigation, or access control inscriptions), and then move to more advanced variants as they learn. This idea is similar to the notion of *gradual language* proposed in [30] where different versions of Python – from simplified ones to the full language – are created to facilitate learning.
- *Project stages/level of precision*. Depending on the purpose of modelling, or the project phase, the user may prefer a more flexible notation to facilitate discussion with peers, or a more detailed and stricter notation for analysis or code generation [27].
- *Context of use*. Depending on the device the model is being accessed, the syntax of the language could change e.g., to present more or less model details, or even to show the model textually instead of graphically.
- *Project roles*. The language could present different information to the different stakeholders depending on their role.

The literature reports many examples of language families, like those for software architecture languages [46], Petri nets [53], access control languages [34] and symbolic automata [16], among many others. In many cases, the families include variants with different expressive power (e.g., black and white Petri nets vs. coloured Petri nets), which provide a trade-off between modelling capacity and analysis power.

4.2 How to build language families effectively?

Without proper support, a naive approach to define a family of languages would require the specification of each member of the family in isolation (i.e., a *clone and own* approach, using the terminology of product lines [19]). However, this may require high effort, does not scale, is error prone, hardly maintainable, and incurs in replications that one would like to avoid.

For example, in the case of the UML class diagrams example mentioned before, assume we would like to produce variants making methods, interfaces and association decorations (composition, aggregation, navigation and cardinalities) optional; to choose between multiple, single and no inheritance; and between references or associations. These 12 features lead to 288 language variants (some of them are shown in Fig. 5).

Hence, a more sensible approach – leading to a more compact language family specification – is to combine techniques from model-based software language engineering [71,6] and product lines [59] to define:

- The *variability* of the language family. This involves defining the features supported by the language family, and their dependencies.

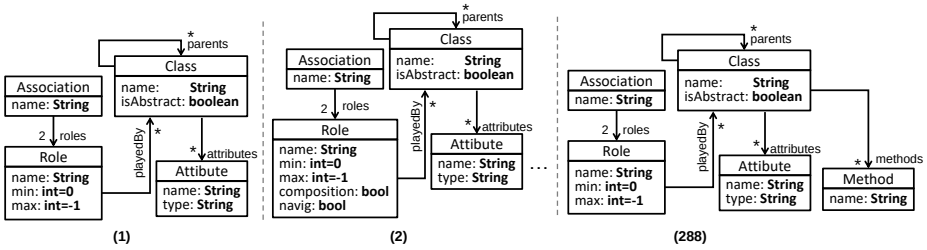

Fig. 5: Three meta-model variants of the class diagrams family (out of the 288 possible ones). (1) Variant with multiple inheritance and associations. (2) Variant with composition and navigation decorations in roles. (288) Variant with methods.

- The *mapping* of the language features to language elements. This way, whenever a certain feature is selected, some elements in the language will be activated.
- *Derivation* mechanisms to obtain a particular language of the family out of a configuration (a valid selection of features).

The variability is often defined via a feature model [33]. This is a diagrammatic notation to express hierarchies of features, and their selection constraints. The latter is achieved by indications of optionality in features, as well as the possibility to define *or-* and *alternative* feature groups. The former requires selecting one or more features in the group, the latter needs selecting exactly one feature among the set.

Fig. 6(a) shows a feature model for a family of class diagram languages (cf. [39]). The model permits selecting Methods for classes: the type of Inheritance (single, multiple, none); Interfaces; the style for associations (either full associations, or references); and decorations for association roles (composition, aggregation, navigability and cardinalities). Feature models may include boolean formulae that use features as variables, to express further constraints (cross-tree constraints). In the example, if Interfaces is selected, so should be Methods.

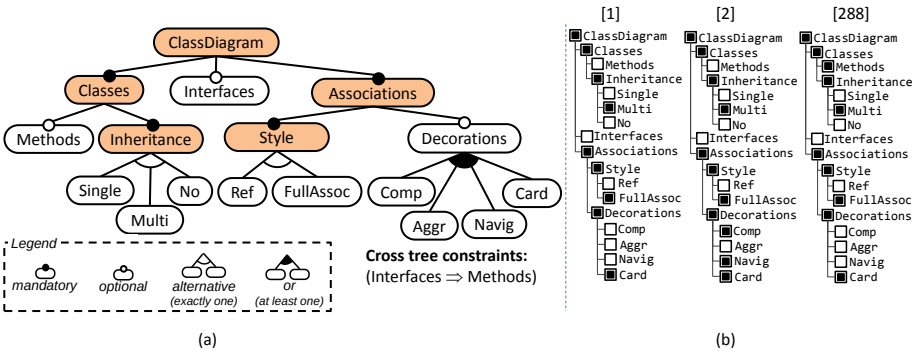

Fig. 6: (a) Feature model for a family of class diagram languages. (b) Three configurations corresponding to the three meta-model variants shown in Fig. 5.

A configuration is a selection of features that is valid according to the feature model. Fig. 6(b) depicts three configurations for the class diagram example, corresponding to the meta-models in Fig. 5. Configuration 1 selects multiple inheritance, full associations and cardinalities. The second one adds composition and navigation. The last configuration selects methods, but not composition or aggregation. Altogether, the example feature model accepts 288 configurations. Typically, the number of configurations of a feature model is exponential with the number of features.

4.3 Language families: the annotative approach

A first approach to map language features to language elements is by using annotations. Essentially, all meta-models of all language variants are superimposed [4], and then the elements are tagged with the features they belong to [15]. Such superimposed meta-model is commonly called a *150%* meta-model [28]. The annotations on its elements (classes, attributes, references) are called presence conditions (PCs), and are boolean formula whose variables are the features of a feature model.

Fig. 7 shows a 150% meta-model for the class diagrams example. The PCs are represented in boldface, between square brackets. For example, the PC of class Association is FullAssoc. This means that, when feature FullAssoc is selected, the class will be included in the derived product. If a class lacks an explicit PC, then it is assumed to be **[true]**. The figure makes the convention that the PC of the attributes is calculated by conjoining their displayed PC with the PC of the owning class. This ensures that, in the generated products, the attributes will always have an owner class (since, if the attribute is selected, so must be the owner). In [28], other well-formedness conditions regarding references and inheritance relations are discussed. Moreover, the work also proposed methods to analyse instantiability properties of 150% meta-models with OCL constraints.

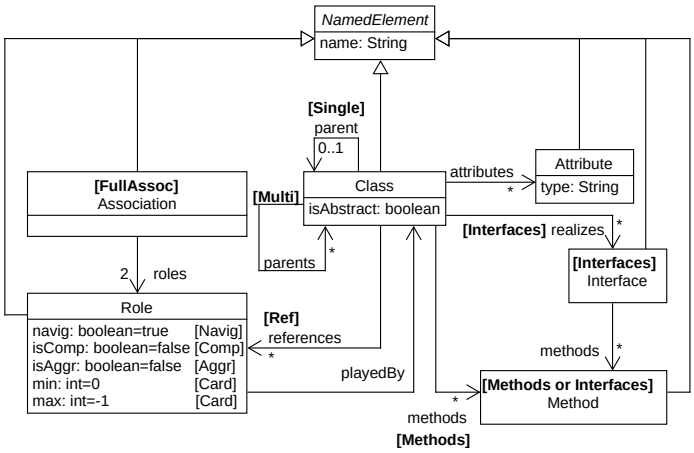

Fig. 7: 150% meta-model for a class diagrams language family.

The approach was realised on a tool called MERLIN, and available at: `https://miso.es/tools/merlin/`.

4.4 Language families: the modular approach

An alternative approach to the annotative approach is to split the family definition into modules, and then use a composition mechanism to produce the members of the family out of the selected features [40].

Fig. 8 shows an excerpt of the definition of the class diagrams family, using the modular approach proposed in [40]. In that approach, a language product line is made of a set of modules, each containing a meta-model fragment. For simplicity, the figure displays only five modules of the family (Classes, Methods, Multi, Single and No).

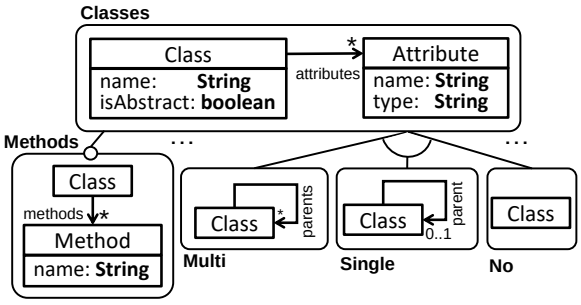

Fig. 8: Excerpt of a modular language product line for the class diagrams family.

A module may establish dependencies of different kinds (mandatory, optional, alternative, or), to a parent module. For example, the Methods module has a dependency to module Classes. The white circle at the top of module Methods indicates that it is optional (whenever Classes is selected, Methods can be optionally selected). If a module M_1 has M as its dependency, then M_1 is said to be its child. Modules can also include cross-tree constraints (boolean formulae that use modules as variables), expressing module selection conditions for valid configurations. A language product line is a set of modules, with their dependencies, and with a single top module with no dependencies.

Each element of a meta-model fragment within a module may define a mapping to an element of the meta-model fragment of the module's dependency. The figure depicts this mapping by equality of names. For example, class Class in module Methods is mapped to the class with same name in module Classes.

A member of the language family is chosen by selecting a valid set of modules, where the top always needs to be selected. If a module is selected, then its optional children can be selected, all its mandatory children should be selected, one or more of its or-children have to be selected, and exactly one of its alternative children need to be selected. The meta-model of the member is derived by composing all meta-model fragments of the selected modules, and merging all mapped elements. In [40] additional well-formedness conditions, and mechanisms to analyse them, are discussed.

The approach was realised on a tool called CAPONE, and available at: `https://capone-pl.github.io/`.

4.5 Language families: semantics and concrete syntax

The previous two subsections have dealt with the abstract syntax of the language family, but a language needs to define its concrete syntax and semantics as well.

For the concrete syntax, in [22], language modules were extended with a model fragment that specifies the graphical concrete syntax of the abstract syntax elements introduced by the module. More specifically, the work uses Sirius' *odesign* model fragments, and defines composition mechanisms for both meta-model fragments, and Sirius odesign model fragments. In practice, selecting a member of the language family generates a Sirius project with a customised graphical editor, synthesized according to the selected language features.

For the semantics, in [40], language modules were extended with graph transformation rules and extension rules. The latter increase rules of the dependency module with further elements (e.g., new elements to match, delete, create or forbid). A composition mechanism was proposed to compose rules and extension rules of the modules in a configuration. Hence, given a configuration, the end result is a meta-model that aggregates the meta-model fragments of each selected module, and a transformation system with rules resulting from composing the rules and extension rules of the selected modules. Interestingly, given certain conditions – which can be checked statically at the product line level –, it is possible to ensure behavioural consistency of all family members. This means checking whether the behaviour of each language variant is consistent with the behaviour of any "smaller" language variant. This implies that for any possible application of a rule r' of a language variant, which extends a base rule r in the smaller language variant, there is a corresponding application of r in the smaller language variant (cf. [40] for more details).

Finally, please note that other approaches exist to define transformational semantics for product lines of languages, like [58] and [41] for 150% meta-models.

5 Beyond language families: Adaptive languages

Language product lines as described in the previous section enable the definition of the abstract syntax, concrete syntax and semantics of a family of languages. Each member of the family may be suitable for a given context of use, or modelling scenario. However, those language product lines do not consider *how* or *when* to change from one language variant to another one.

Adaptive languages [39] consider exactly that possibility. An adaptive language is made of a language product line, plus trigger definitions (to account for *when* language reconfigurations should occur), and migration adapters (to describe *how* models need to be migrated from one language variant to another when a reconfiguration occurs).

Fig. 9 shows the working scheme of adaptive languages. The upper part of the figure displays the definition of an adaptive language. On the one hand, it contains a language product line (label 1), as described in the previous section (more specifically, [39]

uses an annotative approach, as in Section 4.3). On the other hand, it contains a set of *adapters* to describe model migrations between language variants (label 2), and trigger definitions (label 3) to describe conditions for reconfiguration.

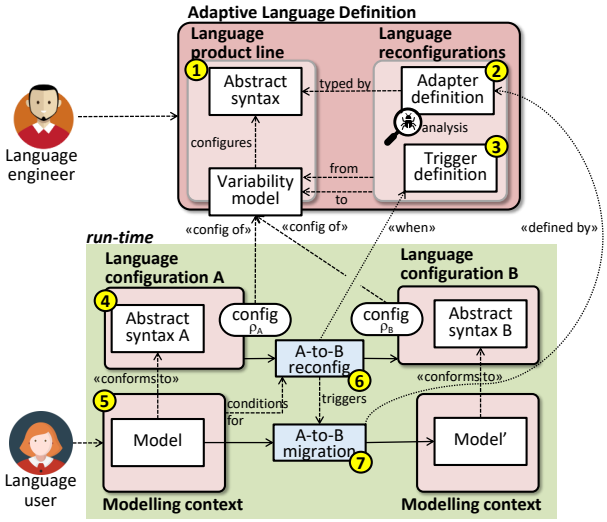

Fig. 9: Schema of adaptive modelling languages (adapted from [39]).

Please note that, given the potential high number of language variants within a family (e.g., 288 in the case of the class diagrams example), it is not feasible to explicitly define model migration transformations between each pair of variants (which would yield 8256 transformations). Adapters [39] are a mechanism to make this definition tractable. An adapter is a set of transformation rules, which work on the 150% metamodel, and perform small migration tasks, associated to the change of just one, or a few features in the language configuration. For example, we may define an adapter that handles the change between multiple and single inheritance; and another adapter to cater for the change between full associations and references. This way, whenever a language reconfiguration is to be performed, the feature changes between the source and target language configurations are collected. Then, the system selects the adapters whose specified feature changes are compatible (i.e., included) with the features changes between language configurations. The selected adapters form the migration transformation, and can be applied to the model of the source configuration to yield a model in the target configuration (cf. [39] for more details).

Following with the scheme in Fig. 9, the language user builds models using the current language configuration (labels 4 and 5). The trigger definition of the adaptive language monitors the model and its context to check if a reconfiguration is needed. Hence, a trigger definition may inspect the current model state, but also contextual conditions, like the modelling history – including errors committed by the user –, the (declared or inferred) level of expertise of the user, the language configuration used by

similar models in a repository, the device being used for modelling, or even additional information to indicate the modelling phase or desired level of flexibility. Once a trigger holds (label 6), it produces a language reconfiguration. This means, that the current language configuration needs to change, and the current model needs to be migrated. To create a migration (label 7), adapters are selected as described in the previous paragraph.

This approach was realised in the MERLIN-A tool, available at `https://miso.es/tools/merlin-adaptive/`.

6 AI assistants: the two sides of the coin

The recent advances in generative artificial intelligence and LLMs [72] have triggered the appearance of intelligent assistants for programming tasks, especially for coding. Hence, both general-purpose LLMs – like those of the GPT family [7] (by OpenAI), Llama [66] (by Meta) or Gemini [26] (by Google) – and LLMs optimised for coding tasks – like Codex [12] (by OpenAI), Code Llama [60] (by Meta), or StarCoder [43] – are used to assist in tasks like code generation from natural language, code autocompletion, or error finding.

Next, Section 6.1 overviews the use of such technologies for professional programmers, and Section 6.2 presents perspectives on the use of assistants to support citizen developers.

6.1 AI assistants for professional developers

Fig. 10 depicts the overall scheme of introducing AI code assistants in the development process. The figure shows the use of assistants to produce code in a general-purpose programming language, out of natural language. Please note that assistants can be used for other code-related tasks (like writing test cases [69]). However, their usefulness for design activities (like domain modelling) has currently proven to be very limited [11].

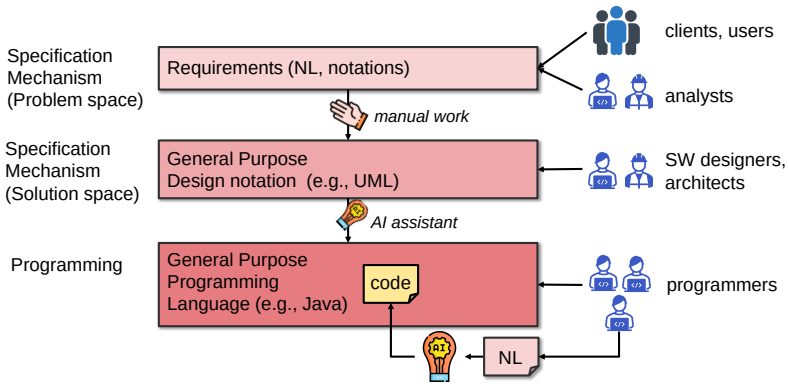

Fig. 10: AI assistants for professional developers.

Using a code assistant can be done by prompting an LLM-based chatbot, like ChatGPT, or by using an LLM integrated within the IDE, like copilot [12] or CARET [13]. Typically, the latter option is to be preferred for a streamline coding experience. However, AI assistants have only been started to be integrated within IDEs, and many factors require attention (see [13] for a feature-based analysis). For example, the IDE provides a context that should be transferred into the underlying LLM, and LLM responses should be translated into code, but also on IDE actions and commands. For professional development, traceability of AI-generated code into the project would be essential, to track the decisions made, storing them under standard version control systems, and supporting queries about which parts of the code generated the assistant, why, when, and who invoked the assistant.

To be useful in practice, IDEs need to enable extensibility of the assistants with new tasks (e.g., generate Javadoc comments for all code in a project), or tailor existing tasks to the programmer and company needs (e.g., generate code following in-house coding standards). Following research in [13], I argue that IDEs should offer extensibility mechanisms (like Eclipse extension points) to ease the integration of new AI assistants and of AI-supported tasks. We term such extensibility mechanisms AI extension points [13].

To be successful, AI assistants need to be trustworthy [70]. This means that the code suggested should have high quality, including both syntactic correctness (the fragment suggested by the assistant compiles when integrated into its context of use), and semantic correctness (the fragment works as expected by the developer), and follows the coding standards in use.

6.2 AI assistants for citizen developers

Current AI coding assistants may become powerful tools to improve the productivity of developers. These assistants rely on LLMs that have been trained with vast amounts of code, in mainstream programming languages like C, Java or Python. This means that, currently, they are mostly useful for users with a training on programming.

Therefore, how can citizen developers benefit from AI assistants? A first possibility is to combine AI assistants with DSLs. Fig. 11(a) shows the working scheme of such proposal, where the assistants would need to consume and produce DSL code. However, this is problematic, since – most probably – the underlying LLMs have not been trained with examples of use of the DSL.

Adapting AI assistants to DSLs is still an open problem, with several proposals. Early ones include SOCIO [57], where conversational syntaxes for DSLs are defined, and a Dialogflow modelling chatbot is automatically generated. More recent proposals include ModelMate [18], a system to create assistants for DSLs by fine-tuning language models with DSL data; and DSL-Xpert [36], a simpler approach that relies only on prompt engineering.

In the previous approach, the DSL is the medium to express what the software is to perform. Going one step further, one could envision the natural language acting as the programming medium, as shown in Fig. 11(b). This way, the AI assistant would directly synthesize an application (e.g., in Python) out of the natural language descriptions of the citizen developers. Such application would be hosted by the low-code platform (and

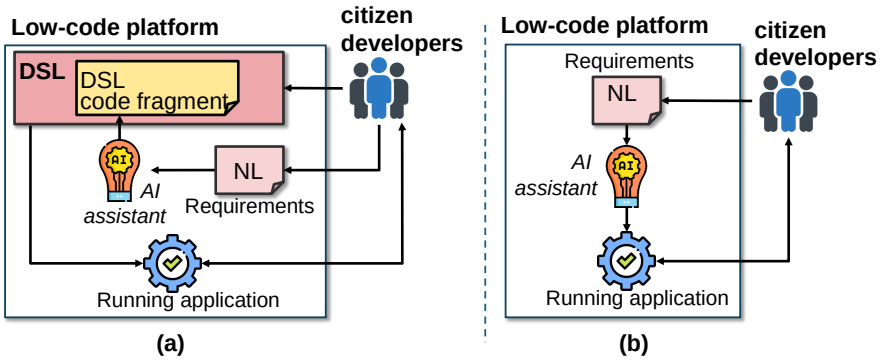

Fig. 11: AI assistants for citizen developers. (a) The DSL is the programming medium. (b) Natural language is the programming medium.

run in e.g., a sandbox), in such a way that the citizen developer would not need to check or even understand the synthesized code. While this approach might be the ultimate low-code platform – natural language becomes a programming language – it has shortcomings and limitations, e.g., related to the security and correctness of the generated applications. In addition, it may only be suitable for small applications since there is no medium to express a design, which is an enabler of scalability. However, small, simple, non-critical applications are often all that citizen developers aim at building. This approach would make programming an immediate activity, and its ease of use may change the status of such applications: they no longer need to be stored or maintained, but they can be disposable[3] – even for single-use – and discarded once they are not needed.

7 Conclusions and perspectives

In this paper, I have discussed techniques to respond to the increasing demand for software, answering the question *How to develop software faster, by more people?*. This way, the paper has reviewed the working scheme of professional vs. citizen developers, based on using code (in traditional IDEs) vs. DSLs (in low-code platforms) to build software. In the latter case, I have claimed that one language does not fit all users and contexts of use, and argued for the need of languages to become adaptive. Finally, the paper ended with a brief account of the increasingly important role of AI assistants in software development. Today, such role is mostly oriented to professional developers (working on mainstream, general-purpose programming languages), and I have discussed for the need to make them available to citizen developers as well. This can be done via DSLs as a medium for programming, but one could envision environments where natural language becomes the programming medium.

There are many challenges in this area, for example to make languages and IDEs more aware of their usage context (i.e., a more prominent role of pragmatics in language design). Regarding AI assistants for professionals, there is the need of a more

[3] Credits to Jesús Sánchez-Cuadrado for suggesting this notion.

streamlined integration with IDEs, languages and development processes. The impact of AI assistants on IDEs, languages and processes should be carefully analysed too, to be able create the next generation of development tools. Finally, we should not forget about the "other side of the coin" of the software development: the non-professionals. Hence, techniques to build AI assistants for citizen developers, working on (textual or graphical) DSLs within low-code platforms, or using only natural language, would help democratising software development.

Acknowledgments. Thanks to Esther Guerra for fruitful discussions on the paper's topics. This work has been funded by the Spanish MICINN with projects SATORI-UAM (TED2021-129381B-C21), FINESSE (PID2021-122270OB-I00), and RED2022-134647-T.

References

1. Akkio: https://www.akkio.com/ (2024)
2. Action Language for Foundational UML OMG specification. https://www.omg.org/spec/ALF/1.1/About-ALF (2017)
3. Amazon: Lex. https://cloud.google.com/dialogflow (2024)
4. Apel, S., Lengauer, C.: Superimposition: A language-independent approach to software composition. In: ICSC. LNCS, vol. 4954, pp. 20–35. Springer (2008)
5. Appian: https://appian.com/ (2024)
6. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice, Second edition. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, San Rafael, California (USA) (2017)
7. Brown, T.B., et al.: Language models are few-shot learners. CoRR **abs/2005.14165** (2020), https://arxiv.org/abs/2005.14165, see also https://platform.openai.com/docs/models/o1
8. Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Softw. Syst. Model. **19**(1), 5–13 (2020)
9. Buezas, N., Guerra, E., de Lara, J., Martín, J., Monforte, M., Mori, F., Ogallar, E., Pérez, O., Cuadrado, J.S.: Umbra designer: Graphical modelling for telephony services. In: Modelling Foundations and Applications - 9th European Conference, ECMFA. Lecture Notes in Computer Science, vol. 7949, pp. 179–191. Springer (2013)
10. Cabot, J.: The low-code handbook (2024)
11. Cámara, J., Troya, J., Burgueño, L., Vallecillo, A.: On the assessment of generative AI in modeling tasks: an experience report with chatgpt and UML. Softw. Syst. Model. **22**(3), 781–793 (2023)
12. Chen, M., et al.: Evaluating large language models trained on code. CoRR **abs/2107.03374** (2021), https://arxiv.org/abs/2107.03374
13. Contreras, A., Guerra, E., de Lara, J.: Conversational assistants for software development: Integration, traceability and coordination. In: Kaindl, H., Mannion, M., Maciaszek, L.A. (eds.) Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE. pp. 27–38. SCITEPRESS (2024)
14. Cuadrado, J.S., Izquierdo, J.L.C., Molina, J.G.: Applying model-driven engineering in small software enterprises. Sci. Comput. Program. **89**, 176–198 (2014)
15. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M.R. (eds.) Generative Programming and Component Engineering, 4th International Conference, GPCE. Lecture Notes in Computer Science, vol. 3676, pp. 422–437. Springer (2005)

16. D'Antoni, L., Veanes, M.: Automata modulo theories. Commun. ACM **64**(5), 86–95 (2021)
17. DataRobot: https://www.datarobot.com/ (2024)
18. Durá, C., López, J.A.H., Cuadrado, J.S.: ModelMate: A recommender for textual modeling languages based on pre-trained language models. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS. pp. 183–194. ACM (2024)
19. Echeverría, J., Pérez, F., Panach, J.I., Cetina, C.: An empirical study of performance using clone & own and software product lines in an industrial context. Information and Software Technology **130**, 106444 (2021)
20. Eclipse: http://www.eclipse.org (2024)
21. Fuentes-Fernández, L., Vallecillo, A.: An introduction to UML profiles. Upgrade **5**(2), 6–13 (2004)
22. Garmendia, A., Guerra, E., de Lara, J.: Product lines of graphical modelling languages. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS. pp. 69–79. ACM (2024)
23. Google: AppSheet. https://about.appsheet.com/home/ (2024)
24. Google: AutoML. https://cloud.google.com/automl (2024)
25. Google: Dialogflow. https://cloud.google.com/dialogflow (2024)
26. Google: Gemini. https://gemini.google.com/ (last access in 2024)
27. Guerra, E., de Lara, J.: On the quest for flexible modelling. In: MoDELS. pp. 23–33. ACM (2018)
28. Guerra, E., de Lara, J., Chechik, M., Salay, R.: Property satisfiability analysis for product lines of modelling languages. IEEE Trans. Software Eng. **48**(2), 397–416 (2022)
29. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? Computer **37**(10), 64–72 (2004)
30. Hermans, F.: Hedy: A gradual language for programming education. In: ICER. pp. 259–270. ACM (2020)
31. Hutchinson, J.E., Whittle, J., Rouncefield, M.: Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. Sci. Comput. Program. **89**, 144–161 (2014)
32. IBM: Watsonx. https://www.ibm.com/watson (2024)
33. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
34. Kashmar, N., Adda, M., Atieh, M.: From access control models to access control metamodels: A survey. In: FICC. LNNS, vol. 70, pp. 892–911. Springer (2020)
35. Kelly, S., Tolvanen, J.: Domain-specific modeling - Enabling full code generation. Wiley (2008)
36. Lamas, V., R. Luaces, M., Garcia-Gonzalez, D.: DSL-Xpert: LLM-driven generic DSL code generation. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems. p. 16–20. MODELS Companion '24, Association for Computing Machinery, New York, NY, USA (2024)
37. Laplante, P., Kassab, M.: Requirements Engineering for Software and Systems (4th ed.). Auerbach Publications (2022)
38. de Lara, J.: Software development... for all? In: Fill, H., Mayo, F.J.D., van Sinderen, M., Maciaszek, L.A. (eds.) Proceedings of the 19th International Conference on Software Technologies, ICSOFT. p. 5. SCITEPRESS (2024)
39. de Lara, J., Guerra, E.: Adaptive modelling languages: Abstract syntax and model migration. ACM Transactions on Software Engineering and Methodology (Nov 2024). https://doi.org/10.1145/3702975, just Accepted

40. de Lara, J., Guerra, E., Bottoni, P.: Modular language product lines: Concept, tool and analysis. Software and Systems Modeling **in press**, 1–30 (2024)
41. de Lara, J., Guerra, E., Chechik, M., Salay, R.: Model transformation product lines. In: MoDELS. pp. 67–77. ACM (2018)
42. Leung, M., Murphy, G.C.: On automated assistants for software development: The role of llms. In: 38th IEEE/ACM International Conference on Automated Software Engineering, ASE. pp. 1737–1741. IEEE (2023)
43. Li, R., et al.: Starcoder: may the source be with you! (2023), https://arxiv.org/abs/2305.06161
44. Lieberman, H., Paternò, F., Wulf, V. (eds.): End User Development. Human-Computer Interaction Series, Springer (2006). https://doi.org/10.1007/1-4020-5386-X, https://doi.org/10.1007/1-4020-5386-X
45. Luckey, M., Erwig, M., Engels, G.: Systematic evolution of model-based spreadsheet applications. J. Vis. Lang. Comput. **23**(5), 267–286 (2012)
46. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: A survey. IEEE Trans. Software Eng. **39**(6), 869–891 (2013)
47. MDA OMG specification. https://www.omg.org/mda/executive_overview.htm (2024)
48. Mendix: https://www.mendix.com/ (2024)
49. Microsoft: Microsoft Power Apps. https://www.microsoft.com/en-us/power-platform/products/power-apps (2024)
50. Moody, D.L.: The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Software Eng. **35**(6), 756–779 (2009)
51. Mosterman, P.J., Vangheluwe, H.: Computer automated multi-paradigm modeling: An introduction. Simul. **80**(9), 433–450 (2004)
52. Muñoz, P., Zschaler, S., Paige, R.F.: Preface to the special issue on success stories in model driven engineering. Sci. Comput. Program. **233**, 103072 (2024)
53. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
54. Node-RED: https://nodered.org/ (2024)
55. OCL: http://www.omg.org/spec/OCL/ (2014)
56. OutSystems: https://www.outsystems.com/ (2024)
57. Pérez-Soler, S., González-Jiménez, M., Guerra, E., de Lara, J.: Towards conversational syntax for domain-specific languages using chatbots. J. Object Technol. **18**(2), 5:1–21 (2019)
58. Perrouin, G., Amrani, M., Acher, M., Combemale, B., Legay, A., Schobbens, P.: Featured model types: Towards systematic reuse in modelling language engineering. In: MiSE@ICSE. pp. 1–7. ACM (2016)
59. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software product line engineering: Foundations, principles and techniques. Springer-Verlag, Berlin, Heidelberg (2005)
60. Rozière, B., et al.: Code llama: Open foundation models for code (2024), https://arxiv.org/abs/2308.12950
61. Ruscio, D.D., Iovino, L., Pierantonio, A.: Coupled evolution in model-driven engineering. IEEE Softw. **29**(6), 78–84 (2012)
62. Ruscio, D.D., Kolovos, D.S., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: Two sides of the same coin? Softw. Syst. Model. **21**(2), 437–446 (2022)
63. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
64. Sommerville, I.: Software engineering, 10th Edition. Pearson (2015)

65. Tisi, M., Mottu, J., Kolovos, D.S., de Lara, J., Guerra, E., Ruscio, D.D., Pierantonio, A., Wimmer, M.: Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. In: STAF 2019 Co-Located Events Joint Proceedings. CEUR Workshop Proceedings, vol. 2405, pp. 73–78. CEUR-WS.org (2019)
66. Touvron, H., et al.: Llama 2: Open foundation and fine-tuned chat models (2023), https://arxiv.org/abs/2307.09288, see also https://www.llama.com/
67. UML 2.5.1 OMG specification. http://www.omg.org/spec/UML/2.5.1/ (2017)
68. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013), http://www.dslbook.org
69. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, Q.: Software testing with large language models: Survey, landscape, and vision. IEEE Trans. Software Eng. **50**(4), 911–936 (2024)
70. Wang, R., Cheng, R., Ford, D., Zimmermann, T.: Investigating and designing for trust in ai-powered code generation tools. In: The 2024 ACM Conference on Fairness, Accountability, and Transparency, FAccT. pp. 1475–1493. ACM (2024)
71. Wasowski, A., Berger, T.: Domain-Specific Languages - Effective Modeling, Automation, and Reuse. Springer (2023)
72. Zhao, W.X., et al.: A survey of large language models. https://arxiv.org/abs/2303.18223 (2023)