The Future of Programming and Modelling: A Vision

The Future of Programming and Modelling: a Vision

Perdita Stevens[1][0000−0002−3975−7612]

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh, UK
Perdita.Stevens@ed.ac.uk
http://homepages.inf.ed.ac.uk/perdita

Abstract. What is the future of programming, and what does it have to do with modelling? In this paper we will first argue that, despite impressive achievements, software development now suffers from a *capacity crisis* which cannot be alleviated by programming as currently conceived. Rather, it is necessary to democratise the development of software: stakeholders who are not software specialists must, somehow, be empowered to take more of the decisions about how the software they use shall behave. We will suggest that a potential way to achieve this is that software should be delivered in the form of a collection of models, each expressed in a (domain-specific) language appropriate to its intended users, and all connected by bidirectional transformations. We emphasise the pragmatic need to accommodate a heterogeneous collection of formalisms so that solutions can incorporate pre-existing transformations, with automatic "fixing up" of their results as necessary. We discuss the advances that are needed to make this a reality, and some early progress in this direction.

Keywords: programming · modelling · bidirectional transformation · consistency maintenance

1 The software capacity crisis

In the early days of the telephone, subscribers called one another, not by entering a number into their handset, but by lifting their receiver and talking to a telephone operator, who plugged the caller's wire into the callee's socket. Soon, it became clear that the number of telephone operators could not scale to match the growth in subscriber numbers. People began to point out, rhetorically, that in the not-too-distant future everyone in the demographic from which telephone operators were drawn would have to be employed in that way. For example, around 1886 the Chief Engineer of the British Post Office estimated that by the year 2000 every woman in Britain would have to be a telephone operator (reported in [48], p52). Fortunately for the author, such dreadful predictions did not come to pass. Instead, the automated telephone exchange was invented, enabling subscribers to control their own connections. In a certain sense, every

one of us is now a telephone operator: but we do not find the occupation too irksome.

Software development today is in a state analogous to that of telephony at the end of the 19th century. We struggle to find enough people – there are already hundreds of thousands of unfilled ICT positions in the European Union [11]. Looking further ahead, we expect the demand for software to continue to increase; one estimate [12] is that 1.6 *million* ICT professional jobs will need to be filled in the European Union between 2018 and 2030. Big data and the rise of AI make new frontiers of potential software visible, while they also intensify concern about the properties of the software, including correctness and privacy. Businesses desire to update their software ever faster. Despite their resulting popularity, agile and DevOps techniques such as continuous integration/continuous delivery (CI/CD) are far from a panacaea [31]. For example, even in a survey of businesses using agile development [47], just 4% of respondents – 71% of which were planning DevOps adoption within the next year – agreed that agile practices were enabling greater adaptability to market conditions! Capacity and hiring regularly appear at the top of lists of software companies' concerns (e.g. [46]).

What makes the software situation even worse than the telephony one is the sheer **difficulty of modern software development**. It pushes the limits of human cognition: in order to make productive use of the technology we have today, people typically have to devote their full-time efforts to learning, retaining, practising and updating their software-related skills. Universities are asked to turn out increasing numbers of students with an extraordinarily wide and fast-changing skill-set; the difficulty of doing so underlies apparent paradoxes such as that, in the UK, despite the unfilled positions, the unemployment rate for computer science graduates is above that of other STEM subjects [39]. While the fundamentals of our discipline do not change, the devil is in the detail – and there is a lot of detail, as anyone can attest who has had to develop (say) a web application with mainstream tools, after a few years of not doing so. Consequently not every person is able to succeed in modern software development, so even if, as a society, we were willing to devote the efforts of even more of us to software development, this would not solve the problem.

Impressive advances in software development have, however, been made since the "thirty year crisis" [40] of approximately 1960–1990. We now have better languages, tools, and frameworks; we have consensus on the importance of testing being integrated into development; we are beginning to understand how to combine the safety of a high-ceremony process with the responsiveness to change that agile methods can bring.

Each of these advances can be seen as (partially successful) attempts to employ **separation of concerns** in Dijkstra's terminology [8] to help manage **information overload**, which is the main root cause of the difficulty of software development, because the amount of information in even a medium-sized software project vastly exceeds what humans can easily hold in mind. For example, high level programming languages allow the programmer to remove focus from

low level implementation details; the practice of unit testing enables someone investigating a bug to ignore certain parts of the code base with confidence; in a sprint, the developer focuses attention on a subset of the requirements and how to realise them.

Ideally decisions taken for the sake of one concern have no effect on any other; then different people may work with different concerns, independently. But usually this is not so (as indeed Dijkstra explained). Separating concerns is worthwhile because of the way it helps focus, but dependencies between them still have to be managed, usually manually and informally, relying on developers' knowledge. Unfortunately, we have (so far) not made advances, in this task of **reintegrating concerns**, that are comparable to those that we have made in managing each individual concern. Thus it turns out that the improvements in software engineering to date do not actually make it *easier* to develop software: they just enable the same software developers – tight-knit teams of people with a rare blend of up-to-the-minute technical, interpersonal and business skills – to achieve more, faster, than used to be possible.

In summary, the practice of software development has advanced enormously since the earliest days, so that we are now able to build large, complex software systems reliably. We have banished the original software crisis. However, current approaches to software development push the limits of human cognition. Acquiring and retaining the skills necessary to be effective in modern software development takes so much time and effort, even from the most talented people, that it is not possible to find enough skilled people to build all the software we would like built, given the way that software development is currently organised. Hence we have a software capacity crisis.

2 Modelling and its limitations

As we have seen, from the invention of the subroutine on, encapsulation, and, more generally, separation of concerns [8], have been understood to be important for managing the information overload that is characteristic of software development. This idea, together with the idea that, in particular, the concerns of different **stakeholders** should be separated, drove the rise of object orientation and the development of modelling.

In the 1990s many different modelling languages flourished, each typically promoted by a single guru and supported by a single tool. Eliding some political history: the Unified Modeling Language, UML [17], was developed in the late 1990s, with the aim of solving this Babel and permitting networking effects that would energise the tools market, permit easier transfer of people between projects, and generally increase the efficiency of software development by enabling decisions to be taken and recorded at higher levels of abstraction. The Object Management Group (OMG), many of whose members are tool development companies, standardised UML. (I wrote the first student textbook on it [49].) UML garnered a remarkable degree of buy-in and effectively wiped out most of the earlier modelling languages. It has since suffered a backlash, because

1. the need to get buy-in from all the key influencers (and to standardise using OMG's consensus-based process, which is itself designed to maximise buy-in) led to UML being huge and imprecise;
2. diagrams are slow to develop compared to code. It took the community a remarkably long time to appreciate the sense in which concrete syntax is superficial. We can have several concrete syntaxes for 'the same' language, e.g. graphical and textual presentations of the same information. For example, the metamodelling language Ecore has a textual syntax, Emfatic, in addition to the original diagrammatic syntax [14]. [1]

Unsurprisingly, we have seen a succession of papers about how little UML is used (e.g. [35]), although in fact, developers' use of diagrams to help focus their design thinking is ubiquitous [26].

One way to analyse the problem is to say that UML has a cost-benefit ratio problem: point 1 above causes the benefit to be too low, and point 2 means that the cost is too high.

2.1 Increasing the benefit that derives from modelling

Attempts to increase the benefit that is derived from the effort of developing models have led to **model-driven development** (MDD) and the related concepts of **language engineering** [23] and **low-code platforms** (estimated market size 27.23 billion US dollars by 2022 [29]). These can be seen as reactions to the backlash against UML: they make more use of tools, and hence, perforce, of languages as formal artefacts, in an attempt to increase the benefit derived from models and hence improve the cost-benefit ratio. A progenitor of this family of approaches was OMG's **model-driven architecture** [16]. This emphasised forward generation of platform-specific models from platform-independent models, and of code from platform-specific models: its underlying assumption was that important decisions about functionality could all be made at a high level of abstraction, so that human involvement in modifying code – programming – would be all but abolished. Modern, more flexible, successors of this approach include XMDD based on the "One Thing Approach" [27, 28]. With an insistence on replacing, rather than integrating, the old-fashioned approach of humans editing code, they provide an conceptually efficient methodology for greenfield development, in which all modelling can take place under the same aegis and there can be a single point of control for generating code once the models are ready. If testing of the code reveals that early decisions, embodied in highly abstract models, must be revised, then the necessary changes are recorded in the models and the automatic process of synthesising code is re-run. When this is a practical way to proceed, it is undoubtedly the right thing to do: it avoids repeating information in more than one place, and recording it in inefficient ways, and uses automation to best advantage. Successful examples include the development of single-page web applications using DIME [4].

[1] Indeed, this is why, in this paper, we do not make a hard distinction between "model" (often assumed graphical) and "code" (always assumed textual).

A key difference between OMG's original MDA conception and these later approaches – with wider implications which we shall shortly come to – is that while MDA envisaged models would be expressed in general purpose languages like UML, later approaches make use of multiple **domain-specific (modelling) languages (DS(M)Ls)**, each made just expressive enough for the concern it targets and endowed with syntax suitable for its users. (Traditionally a program in a DSL is expressed in text, while a model in a DSML is expressed diagrammatically; but we have already observed that this distinction is superficial, and in the context of DS(M)Ls the deliberately limited expressivity makes it easier, than with general purpose languages, to provide both textual and graphical syntax for the same language, and hence makes it even more difficult and unproductive to draw a distinction between program and model. We shall not do so, and shall use the shorter term DSL from now on without intending to limit its scope to textual languages.) DSLs can benefit their users by providing uncluttered, straightforward means to access *all and only* the information required for a particular task; they can be provided with tooling which is efficiently usable; and they are amenable to programmatic manipulation for synthesis, model-checking, etc.

There is, of course, no such thing as a free lunch. The DSLs themselves and their tooling have to be developed and maintained and, even with the best of language engineering support, this carries a cost. The DSL's users have to learn them, and great care is needed to ensure that the initial effort of doing so really is repaid by greater efficiency coming from the suitability of the language for the task. Different users have different backgrounds and skills, hence they may need widely different languages and tools. A poorly designed DSL can give the worst of all worlds. Nevertheless, these problems and their solutions are becoming well-studied and mainstream (see e.g. [15]).

Overall, DSLs are an important step forwards towards better separation of concerns. However, concerns must still be related, so that eventually software can be produced that is correctly modelled by all the models in the various DSLs. In an ideal world, any decision is recorded in only one model, so that all the human-modified models are orthogonal, with no dependencies between them. Then models that combine information from several of them, including ultimately the delivered software system itself, can be generated, unidirectionally, from them. Most DSL engineering still works on this premise, whether the generation is done by **transformations** as usual in MDD, by global constraint solving, or by another kind of search. From now on we focus on model transformations as the mechanism by which models are related. The term refers to any program, however expressed, that has models among its inputs and/or outputs.

2.2 Bidirectionality

From the beginning, potential users of transformations recognised that the world would not generally be ideal in the sense just referred to: the interesting Object Management Group document [50], produced in the run-up to its call for proposals for model transformation languages, records that the ability to resolve

bidirectional dependencies between models was important to potential users of such languages. Bidirectional dependencies, in which a change to either of two models may necessitate a change in the other, arise because information cannot, in fact, usually be partitioned between models. There is generally an overlap between the information that must be included in one model, and that which must be included in another, in order to allow the users of each to do their work. Thus it is not generally enough to accept arbitrary current states of all the models expressed in their DSLs, and synthesise code from them. It can, and does, happen that models get "out of sync": they record inconsistent information, and one or both must be modified before development can proceed. If only one of the models is user-facing, the other being generated from it, then of course there is no problem: we simply regenerate the generated model. However, if both models are under the control of human developers then these modifications have to be effected in a way which is acceptable to the humans. They may, for example, have to sit down together, identify the root causes of the inconsistencies between their models, and agree how to fix them. This can be an expensive, time-consuming and error-prone process, because it inherently requires the humans to understand information from outside their own model – precisely what DSL use was intended to avoid.

We use the term **bidirectional transformation** (abbreviated **bx**) for an *automatic* means of checking and restoring consistency[2] between two (or more) models, allowing for the possibility that a change in either might necessitate a change in the other. In an earlier paper [43] I listed the following three criteria as "the essence of bidirectionality":

1. There is separation of concerns into explicit parts such that
2. more than one part is "live", that is, liable to have decisions deliberately encoded in it in the future; and
3. the parts are not orthogonal. That is, a change in one part may necessitate a change in another.

Following the earlier observation about the superficiality of syntax, we call the parts "models", regardless of whether they are diagrams or text (including code), or recorded otherwise. ("Everything's a model".) Where bidirectional situations arise – and they do arise in any large cooperative software development – care must be taken to manage the relationship between the models. They sometimes need to be allowed to evolve separately – we say, to become "inconsistent" – for a while, especially when the owner of one model is making changes that may not prove to be durable [33]. At some points, though, it will be necessary to bring the models into consistency with one another. This can be done entirely manually, e.g. following discussion between the owners of the models. Restoring consistency automatically is the job of a bidirectional transformation.

[2] The now well-established use of the term "consistency" occasionally causes confusion. Consistency can be any desired relation between the models: models are consistent if the development they are part of is considered to be in a good state. The relationship between this and logical consistency is discussed in [43].

It is important to understand that, even though there may be many ways to restore consistency between two models, this does not imply that the bidirectional transformation must be non-deterministic, or must involve user interaction. It may do so, if desired: but the choice between different consistency restorations can be programmed in the bidirectional transformation. Indeed, this is the main job of the programmer of the transformation.

It has proved difficult to develop good languages and other technology to support bidirectional transformations, partly because the requirements for such a language cannot all be met simultaneously[3]. The various attempts have led to a **fragile tools problem**, in which solution approaches, each balancing the forces in different ways and making different compromises, are incompatible with one another and have idiosyncratic (and often incompletely documented) behaviour. Among other problems, we so far lack a principled way to allow **inter-operation of bidirectional transformations**. That is, it is difficult or impossible to manage a development that incorporates bidirectional transformations that have been developed in different ways at different times by different people and expressed in different languages.

Thus these model-based approaches have not (yet) solved the capacity problem. Whilst powerful walled-garden tools such as JetBrains MPS[4] can achieve amazing results in skilled hands, this leads to lock-in at personal and organisational level; it prevents the combination of advantages from different approaches, and makes network effects unavailable (although for commercial platform vendors, such lock-in gives a short-term advantage). And, despite the bullish projection of its market size, low-code platforms are reasonably seen as a "fad", the latest in a long sequence of candidate silver bullets, because "anybody coding really needs to understand what's going on behind it all" [36]: that is, today, the dependencies between concerns still have to be handled manually, which places a heavy burden on the developer.

3 A vision for the future

In summary, in order to address the software capacity crisis we need principled advances on several fronts.

1. Of course, we do need to continue to increase the productivity of today's best software specialists, that is, the speed at which teams of the most talented people – who can, for example, embrace techniques such as mechanised proof and functional programming with sophisticated type systems – produce dependable software. This is the aim of the vast majority of software engineering and programming language research today, but it is not the only important avenue to pursue.

[3] For example, it is extremely convenient if all one's bx have the property known as strong undoability, while not requiring auxiliary data beyond the models themselves, but insisting on this limits expressiveness too much.

[4] https://www.jetbrains.com/mps/

2. We need to reorganise software development, so that the effort of the most skilled software specialists can be applied where it is most needed (e.g. writing the bidirectional transformations that support the integration of concerns) allowing technically easier tasks (e.g. updating a model of a single concern) to be done by developers with less experience. Taking this to the extreme:
3. We need to distribute more of the decision-making about software's behaviour to people who are not software specialists, but are stakeholders in the software, perhaps experts in some completely different domain.

Let us go into a little more detail about how such a reorganisation of software development might look.

In future, rather than delivering a software system with fixed behaviour, and standing ready to change it whenever the required behaviour changes, software specialists should deliver something more like a **cloud of potential software systems**:

- a collection of distinct **model spaces**, within which each stakeholder can safely and easily change their decisions about how the software should behave, using whatever tooling they find appropriate;
- within each model space, a starting model, which incorporates the specialists' current understanding of what the stakeholders want;
- a mechanism, involving a collection of bidirectional transformations, for bringing together the separate collections of decisions made by different stakeholders and melding them into *well-behaving* software that meets all of its requirements.

Here is a very simple example. Suppose that a system involves: a form-based user interface, controlled by a UI designer; a database, controlled by a database designer; and a report production engine, the format of the report being controlled by an accountant. Let us suppose that data that needs to appear in the report must be collected from the users and stored in the database: that is, the consistency condition between the three models, that must be maintained, includes this constraint. (It may, or may not, also say a lot more, such as that data should not be collected from the user *unless* it is needed in the report.) If the accountant modifies the report format to include some extra data, then the three models will be considered inconsistent. It might be that the bx, delivered with the model spaces, are capable of automatically restoring consistency, by adding a field to the UI in a default, programmed way, and by adding another column to the database schema (and creating any necessary migration scripts etc.). Of course, not every change that a stakeholder wishes to make to their model will break consistency with other stakeholders' models. For example, following the change just discussed, the UI designer might decide on a better way to collect the new data than the default one chosen by the bx, but this would be entirely within the UI designer's concern and would not affect consistency with the other models. We would still like to have a mechanism for checking that consistency holds.

More generally, the vision is that whenever it turns out that – because their requirements were misunderstood, or because they have changed – a stakeholder's needs are not served by the current software system, they can change their own model within the provided model space. They can then use the bx to update the whole software system accordingly, including automatically making any necessary modifications to other stakeholders' models. Only when something is needed that is outwith the delivered cloud of software systems do software specialists need to get involved again. Of course this will sometimes happen: a stakeholder may need something that is not expressible within the provided model space, or the provided bx may be unable to synthesise well-behaving software from all the current needs of the stakeholders. The more expressive the model spaces, and the more powerful the bx, the less often this will happen. As usual we may expect to see a trade-off between effort invested up front and effort likely to be required later; but we may hope there is potential to eliminate a lot of routine maintenance work and a vast amount of stakeholder frustration, by **making easy changes easy**. The reader who doubts whether this is possible at all should observe that we already have a degenerate case of it: we expect to have settings screens which enable us to modify the behaviour of software in certain small ways which have no effect on other stakeholders. What is proposed here is that we harness the power of bx to broaden the scope of changes that can be automatically effected: an open question is to what extent this can be done. Another, equally important, is "how do we get there from here?".

Summarising the argument so far: we have understood the importance of **separation of concerns** since Dijkstra gave us the phrase. We separate out a concern by capturing all, and only, the information relevant to that concern in one artefact – today we call this artefact a model. The language of this model functions as a high-level, abstract language for expressing the part of the solution relevant to the concern. This helps the developer by allowing them to focus attention on the most relevant information, and by giving them the ability to express their decisions concisely.

However, attempts to use such an approach to democratise the task of telling a computer what to do have had very limited success, despite attempts going back at least to the development of COBOL. Fundamentally this is because of inadequate separation of concerns, which in turn results from a lack of support for **putting concerns back together again**. It looked as though someone could write a COBOL program without understanding full details of what the computer would do as a result, but this was an illusion. If (as still generally happens today) the developer is permitted to *write* only a comparatively small model, but still has to *understand* how their model fits into the rest of the development, what will be generated from it, etc., we may have saved them typing, but we have not really relieved them of information overload; we have simply handed them yet another power tool with which to manage it. To get more benefit, it needs to be possible for a developer to understand in detail *just* the model of this one concern. By taking that seriously as an aim, we can get:

– benefits for software specialists, who are free to not spend brain space on knowing a lot of detail about how their model fits with the rest of the system; but even more
– the possibility of opening up the use of the model to people who are not software specialists.

One might think that it is natural to concentrate on the first of these benefits, taking software specialists as intended users, and only later expanding focus to include non-software-specialists. However, aiming at the second possibility has a crucial advantage for the technology developer. If the people using a model are software specialists anyway, and especially if they *already know* a low-abstraction way to solve the problem, then it may not be possible to overcome the startup cost of learning to work with high-abstraction modelling languages and separate bx. In the early stages at least, we are vulnerable to "I can code this directly, faster". Non-software-specialists are not vulnerable to this: they genuinely need the abstractions, because the rest of their cognitive attention is on things other than software. They therefore automatically get more benefit than software specialists do from using the new approach. This is the sense in which focusing on the harder aim is sensible: it may actually make us more likely to succeed.

4 Bidirectional transformations

We have argued that it is desirable that different stakeholders should be able to work on different models, with the relationships between them maintained automatically, and we pointed out that this has long been recognised. However, even something as generic as UML-Java round-tripping has not been taken up as widely as one might expect, because in practice the accidental complexity [22] imposed by today's tools is too high; so maintaining consistency between models and code is perceived as an important barrier to the use of modelling [35, 32, 20], despite the long-standing availability of tools that target exactly this problem. If, rather than using general purpose modelling and programming languages like UML and Java, we want to use custom-designed DSLs, better adapted to the people using them, then we must also custom-design the means of maintaining the desired consistency relationships between the models. That is, we must have good ways to develop dependable bx. In this section we briefly consider the state of the art.

Consistency checking and restoration can be done by programs written in conventional – unidirectional – languages, and in practice, today's bidirectional situations are often handled that way. In the simplest formulation, we can write three distinct, but related, programs that each operate on two models m and n whose consistency is supposed to be maintained: one consistency-checker, which returns true iff m and n are consistent, and two consistency-restorers, one that returns an m' which is a version of m modified to be consistent with n, and dually one that returns a version n' of n, modified to be consistent with m. However, since the functionality for the consistency checking, and for the restoration in

either direction, must then be written largely separately, it is tedious and error-prone. For example, the structure of the models tends to get encoded in all three programs, all of which must be updated if the structure changes. A bx language is a language in which one artefact can represent all of these tasks. A good example of what can be achieved today with bx languages is BiYacc [51]: this domain-specific language allows a single bx program to represent both a parser and a printer for the same grammar. Moreover, since BiYacc is based on a body of bx theory, it offers "reflective" printing with guaranteed round-trip properties, allowing it, for example, to avoid losing comments in program text which is parsed, optimised, and printed again.

The design space of general purpose, unidirectional programming languages has been extensively explored; although advances continue, much is understood about the options for structuring, typing and supporting such languages. Despite important advances in recent years, bx languages are nothing like so well understood. A handful of languages have been developed [37, 38, 3, 19, 7] and a few have had some success in applications [18, 25]. However, they are very different from each other, difficult to learn and practically impossible to combine. Classification has been attempted [9] but is not yet mature. The Object Management Group developed a standard for a bidirectional language, QVT-R [34], but the standard has so many problems [41, 42, 5], including not only "accidental" problems but also "essential" problems with the structure of the language, that with hindsight this standardisation effort was premature.

The active Bx community, especially through its annual workshop and its collaborative events, brings together diverse constituencies – chiefly software engineering, programming languages, databases and graph transformations – and is making great progress in understanding the commonalities and differences between approaches to bx (e.g. [21]). It has also built up a useful catalogue of examples and benchmarks [6, 1]. Nevertheless, the area is still desperately immature compared with that of unidirectional programming. We need experimentation with different languages to continue, but even more, we need investigation into the foundations of such languages, to improve our understanding of the design space of bx languages.

One axis on which approaches differ is which bx task should be uppermost in the bx programmer's mind. Let us explain in the special case of an asymmetric bx, where one of the models being reconciled is a **view** which is a strict abstraction of the other, its **source**. (The term **lens** is often used for such asymmetric bx, following seminal early work [13].) A programmer following a **bidirectionalization** approach [30] thinks principally of the `get` direction, from source to view, and in practice the same is true of the programmer in lens languages such as Boomerang [3]. There is a field explicitly called **put-back based programming** [19] in which the `put` direction, which takes an updated view and a source and updates the source, is primary. (One advantage of this approach is that the `get` function is then determined by the `put` behaviour, given mild well-behavedness assumptions.) Relation-based languages such as QVT-R, like constraint-based approaches [24], put the consistency relation itself first in the programmer's

mind. Because this approach does not privilege one restoration direction, it is suitable for writing **symmetric** bx, where each model contains information that is not present in the other. These are ubiquitous in MDD: for example, source code typically omits the use-case information from a UML model, but includes detailed code which the UML model does not. In MDD understanding *what it means* for models to be consistent is both easier and more important than understanding *how* consistency can be restored after it has been lost. It is more likely to be specified correctly, and less susceptible to being generated automatically.

In summary, there is currently, for good reasons, a wide and growing range of bx languages and the field is still in its infancy. Unfortunately, it is not straightforward to compose transformations written in today's bx languages and this situation is not likely to improve any time soon. So, in order to "get there from here", we need to tackle this problem. As so often in software engineering, we may proceed by adding an extra level of indirection[5]. But first let us consider the broader implications of having more than two models in play.

5 Specifying networks of models: megamodelling

As we have seen, there is a bewildering variety of approaches to the problem of maintaining consistency even between just two models. Until recently, most work on bidirectional transformations was focused on this binary case. This is unfortunate for our vision, since any non-trivial software system has more than two concerns! Elsewhere in MDD, however, more complex configurations of models were getting more attention. The term *megamodel* was coined by Jean Bézivin [2] in recognition of the fact that the collection of models and their relationships can itself be seen as a model (but that the term *metamodel* is already in use for something quite different!)

Concretely, consider Figure 1 as a small but not trivial example of how models work together to separate concerns in software development. The diagram represents: a model `M` (say, a diagram showing the structure of the software to be built, together with a use-case diagram giving an overview of its requirements); a metamodel `MM` to which the model should conform; some `Code`; some `Tests`; and a `Safety` model. The model `M` and the `Code` are supposed to be related by a standard **round-tripping** relationship. For example, we might expect that the same classnames will appear in the structure diagram as in the code, while the detailed code has no equivalent in the model, and the use-case diagram has no equivalent in the code. There are several different possible relationships that might be desired between the `Code` and the `Tests`, for example a coverage criterion might or might not be included; the diagram represents that a `Safety` model, recording among other things whether the system is considered safety-critical, may have an influence on what relationship is desired.

In principle, the requirements for a way of restoring consistency between separated concerns do not imply presenting the concerns in the form of a megamodel

[5] See https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering.

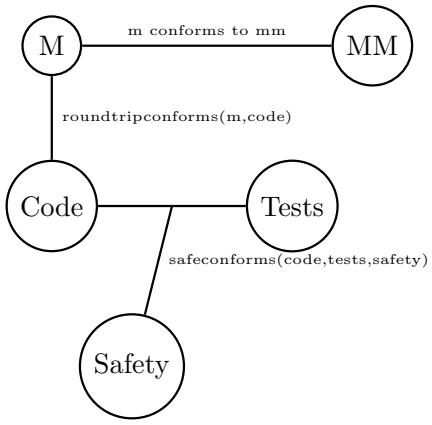

Fig. 1. A small megamodel: models connected by desired relationships (from [45]). Notation: lower-case `model` is instance of upper-case `Model`.

like this: we could in principle specify and restore a single, five-place consistency relation expressing precisely which collections of models `m` ∈ `M`, `code` ∈ `Code` etc. are to be considered consistent. This is impractical, however, for many reasons. An important reason is that such a five-place consistency relation would be entirely bespoke, and would be prohibitively costly to specify. Since it most likely incorporates, conceptually, standard notions such as conformance between a model and its metamodel, and roundtrip consistency between a model and some source code, we would like to be able to reuse those bx, perhaps even buying them off the shelf. We expect, therefore, that some edges in a megamodel will represent such off-the-shelf standard bx, while others may represent bx written for a specific software system. Remember that, given the lack of a single best bx technology, these bx may well be written in different languages and executed by different bx engines. We should not assume *any* homogeneity or compatibility between their formalisms.

Particularly when considering what future, better bx languages should be like, one early question among many is: does it suffice to have languages in which to express consistency, and its restoration, between just two models – we say, languages to express binary bx – or do we need multiary transformations (multx for short), to express and restore consistency relations between more than two models? Figure 1 illustrates a ternary bx between `Code`, `Safety` and `Tests`, although the other edges in this megamodel are all binary. This question is addressed in detail in [44]: here it suffices to say that in many situations it is reasonable to proceed by putting together binary bx in a network of models. Then each edge in the network represents a binary bx: a restorable consistency relation between two models.

Even if, as in Figure 1, edges are not restricted to being binary, it is useful to express the consistency of a whole collection of models forming a megamodel

by means of the edges in a network. The entire network is considered consistent when every edge in it is consistent. To restore consistency in the network, we apply the consistency restoration capabilities of each edge, in some sequence, until, hopefully, the entire network is consistent. For example, we might apply the `roundtripconforms` bx on the `M-Code` edge to update the `Code` with respect to some changes in the model `M`, and then we might apply the `safeconforms` bx to update the tests.

This approach has the advantage that it gives us a way to talk about the overall consistency of the network, and even about how we restore consistency in the network, even though the network is heterogeneous in both the expression of consistency and the consistency restoration mechanisms.

Unfortunately, it is easy to see that we cannot hope for an arbitrary collection of bx (even if they all happen all to be in the same language) to comprise a complete solution to the problem of maintaining consistency in the network. As explored in [44], several problems can arise. Most significant among them is that when a model $m \in M$ is connected by bx to several other models in the network, restoring consistency in the whole network requires that an $m' \in M$ be found which is consistent with *all* its neighbours. Even if each individual bx can restore consistency with *one* of m's neighbours, such an m' may not exist. Even if it does, reaching m' from m may not be possible using any sequence of applications of the consistency restoration procedures of the individual bx. And even if there is a simultaneous solution and it is achievable, we may not have confluence: that is, the eventual result achieved may depend on the order (and, in general, direction) in which bx are applied.

6 Restoring consistency in megamodels

The problem of how to reason about the restoration of consistency in a network of models can seem overwhelming. Even if we start with a collection of bidirectional transformations that are, in principle, adequate, how on Earth do we manage the process, avoiding confusion caused by the problems just discussed, viz., that solutions may not exist or may not be unique? There may be no practical alternative to doing some "fixing up" in order to make bx incident on the same model "play nicely together", e.g. preventing the second bx applied from undoing some necessary change made by the first; but requiring even a trivial amount of manual work to be done after applying the bx negates some of the value of using the bx. It is especially damaging to our vision of consistency restoration being done without reference to software specialists.

An example (from [45], referring again to Figure 1) illustrates. Consider the bx incident on the `Code` (a particular instance will be referred to as `code` following our standard convention), and think about the problem of using these bx to change `code` so as to bring it into consistency with its neighbours. (For the sake of giving a simple example, we suppose that in this situation only `code` must be altered – we say, its neighbours are for the present *authoritative*, that is, must not be altered by the automated consistency restoration process.) For concreteness,

- Suppose the `roundtripconforms` edge requires that every class in `m`'s class diagram should have a corresponding (in some sense we need not go into) Java class in `code`. When the bx's consistency restoration is invoked in the direction of `code`, then if there is a class in `m` with no corresponding class in `code`, one will be generated. No comments will ever be inserted in the Java.
- Suppose the `safeconforms` edge requires (among other things) that every Java class in `code` corresponds to a test class in `tests`, *unless* the Java class is marked with a special comment (`// Not Yet To Be Tested` or similar). When this bx is invoked in the direction of `code`, any Java class that has neither that special comment nor a corresponding test class will be deleted. If there is a test class that lacks a corresponding Java class, then a Java class will be generated.

First, observe that the order in which these bx are applied matters (we say that they are not *non-interfering* [44]). One reason why this is so is that each of the two bx will generate a missing Java class if necessary. Consider the case that the "same" class exists in `m` and in `tests`, but there is currently no corresponding class in `code`. Then the first bx to be applied will generate Java code for the missing class, after which the second one will find the Java code already present and not need to generate it. However, it may be that one of the bx is better at generating useful Java code than the other. We would like human intelligence, not an automatic framework that proceeds in ignorance of the specific setting, to be making the choice of order of application of the bx, so that the better code generator is used.

More interestingly, consider a case where a class is present in `m`, but not in either `code` or `tests`. Here neither order of application of the available bx, without adjustment, will succeed in restoring both the consistency relations. For if `roundtripconforms` is applied first, it will create a Java class – but because it does not insert the special comment, application of `safeconforms` will then delete it again, breaking consistency according to `roundtripconforms`. On the other hand, if `safeconforms` is applied first, and then `roundtripconforms`, the result will be that a Java class is present in `code`, without the special comment, but is not present in `tests`, so the `safeconforms` consistency relation does not hold. However, some intelligent "fixing up" can easily solve this problem. What we want to do is:

1. apply the `roundtripconforms` consistency restoration first, possibly creating new classes in `code`, then
2. add the special comment to any such new classes, before
3. invoking the `safeconforms` consistency restoration.

In this way, a fully consistent state may be reached even though this would not be possible with any combination of the bx unaided. Of course, a human could carry out this procedure, manually invoking the bx and doing the "fixing up" as necessary. But in order to realise our vision of most software maintenance taking place without the involvement of software specialists, we need to automate the whole process.

Alongside tackling these semantic issues and ensuring that consistency can be restored in a sensible way, we also note that, in practice, model transformations can be computationally expensive and it will be important not to do unnecessary work. We will want to avoid applying model transformations in situations where we "should know" that they are not required.

It turns out we can make progress on all of these problems via the observation that the problem of restoring consistency in a network of models is closely related to the problem of *software build*, where both correctness (ensuring that software is built correctly from its sources, according to the build rules, incorporating the latest changes to every source) and optimality (ensuring that no unnecessary compilation etc. is done) have been the subject of extensive study. Work by Erdweg et al. on the *pluto* build system framework [10] is especially helpful: it gives us the means to handle the problem of wanting human intelligence to control the application of the bx and any necessary "fixing up", as follows.

For each model that might need to be modified in the process of restoring consistency overall, there is a **builder** which owns the responsibility of doing that modification. That is, this builder controls the invocation of any bx that will modify this model, and does any necessary "fixing up". The builder is a program: it might be a very simple one, which simply invokes one or more bx in a fixed order, or it might be arbitrarily intelligent. The effect is to allow the inter-operation of heterogeneous technologies; eliding some details, the builder's key post-condition is simply that, on successful completion, this model should, somehow, have been brought into consistency with its (relevant) neighbours. The builder provides the extra level of indirection advertised earlier.

Space forbids telling the full story of how the builders cooperate to restore consistency in the megamodel as a whole. To cut a long story short, it turns out that the *pluto* framework [10] can, with care, be adapted to our needs: provided that we write builders obeying some natural constraints, *pluto* can manage the invocation of the right builders in the right order, so that (if the build completes without error) consistency is restored in the relevant part of the whole network, without any unnecessary work having been done.

Key ingredients of the adaptation are:

- the decision to adopt a "pull" rather than a "push" model: rather than rolling changes in one model out through the network, a build request produces a version of a specified model, which has been brought into consistency with its dependencies (transitively, but without modifying any model on which the specified model does not depend);
- the use of an *orientation model* to capture project-level decisions about which models may be automatically modified (and which are authoritative, i.e. may not be modified right now) and in which direction bx should be applied (hence, which model takes priority, right now, in the case of conflicts).

For more details, see [45].

7 Further work needed to realise this vision

Programming This vision has not, by any means, eliminated the need for programming. What it has done is to concentrate it. Someone has to program the bx, and the builders. There is something to be said for having the builders all in Java, or another general purpose language, but, as mentioned in Section 4, it is advantageous to write the individual bx in a specialist bx language. We have already remarked that the development of bx languages is in its infancy, and "bx programmer" is not yet a career. Perhaps it will be in future.

Modelling Achieving separation of concerns which is effective enough to make it genuinely practical for non-software specialists to change the behaviour of software by using only their own model, without needing to understand the rest of the system into which their model fits, requires excellent support for both developers and users of DSLs. It remains to be seen how far the idea can be pushed, but it is a field which is already active [15].

Explainability, verification, validation More challenging may be the need to achieve overall dependability of the framework into which the DSLs fit. When the consistency restoration process produces results that surprise someone, how can they tell whether there is a bug that should be reported? And, if the consistency restoration process fails – e.g. because different stakeholders have made decisions for which no simultaneous solution exists – what then? We will need *explainability* beyond anything achieved so far.

The correctness and optimality of any framework realising the vision suggested here is both crucial and subtle. Megamodelbuild [45], building on *pluto* [10], is supported by hand-written proofs, but, especially in order to explore more flexible variants, mechanisation is desirable. This is work in progress.

Enabling gradual adoption Something which is both a challenge and an opportunity is the flexible range of possible ambition inherent in this approach. At the least ambitious end, we could have a set-up in which all we can do is *check* consistency: every builder checks consistency of its model with relevant neighbours and fails if any inconsistency is found. This might already be very useful, even if the actual restoration of consistency has to be done manually following meetings between stakeholders (and presumably, in this case, involving software specialists). For example, it would permit any stakeholder to make any change that does not break consistency. Over time, the builders, and the bx that they apply, could be replaced by more sophisticated versions that can more often succeed in restoring consistency automatically. We could even envisage a *learning* framework, in which the consistency restoration processes become automatically more powerful over time, as they incorporate knowledge of what humans do to restore consistency so that the next time a similar change is required it can be made automatically. There is intriguing crossover with artificial intelligence (principally good old-fashioned AI rather than machine learning, though that too might have its place).

8 Conclusions

In this paper I have argued that we need a radical change in how software is conceived, developed and delivered. Without it, we have little hope of solving the software development capacity crisis. I have suggested that a reorganisation of software decision-making that empowers stakeholders to take more of the decisions pertaining to their own concern has potential. To make this a reality we will still need all the old programming language skills, but they will be directed to where they are most needed: programming the consistency checking and restoration processes. If achieved, this vision might deliver more flexible software for us all; but many challenges need to be met to make it a reality.

Acknowledgements

I am grateful to NCSC/RIVeTSS project RFA20601-4214171 *Mechanising the Theory of Build Systems* for funding, and to Steffen Zschaler, Julian Bradfield, Robin Bradfield and all the participants of Dagstuhl no. 18491 on *Multidirectional Transformations and Synchronisations* [40] for helpful comments and discussion. I thank the anonymous reviewers for insightful comments, questions and pointers to relevant literature.

References

1. Anthony Anjorin, Zinovy Diskin, Frédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. Benchmarx reloaded: A practical benchmark framework for bidirectional transformations. In *BX@ETAPS*, volume 1827 of *CEUR Workshop Proceedings*, pages 15–30. CEUR-WS.org, 2017.
2. Jean Bézivin, Frédéric Jouault, and Pierre Valduriez. On the need for megamodels. In *Proc. OOPLSA/GPCE workshop: Best Practices for Model-Driven Software Development*, 2004.
3. Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *PoPL*, 2008.
4. Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. DIME: A programming-less modeling environment for web applications. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, volume 9953 of *Lecture Notes in Computer Science*, pages 809–832, 2016.
5. Julian C. Bradfield and Perdita Stevens. Enforcing QVT-R with mu-calculus and games. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013*, volume 7793 of *Lecture Notes in Computer Science*, pages 282–296. Springer, 2013.
6. James Cheney, James McKinna, Perdita Stevens, and Jeremy Gibbons. Towards a repository of bx examples. In K. Selçuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference*

(EDBT/ICDT 2014), Athens, Greece, March 28, 2014, volume 1133 of *CEUR Workshop Proceedings*, pages 87–91. CEUR-WS.org, 2014. See also http://bx-community.wikidot.com/examples:home.
7. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A bidirectional and change propagating transformation language. In *SLE 2010*, volume 6563 of *LNCS*, pages 183–202. Springer, 2010.
8. Edsger W Dijkstra. *Selected writings on Computing: A Personal Perspective*, chapter On the role of scientific thought, pages 60–66. Springer-Verlag, 1982.
9. Zinovy Diskin, Hamid Gholizadeh, Arif Wider, and Krzysztof Czarnecki. A three-dimensional taxonomy for bidirectional model synchronization. *Journal of Systems and Software*, 111:298–322, 2016.
10. Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In *OOPSLA*, pages 89–106. ACM, 2015.
11. Digital Economy European Commission and Skills (Unit F.4). Digital skills and jobs. https://ec.europa.eu/digital-single-market/en/policies/digital-skills, November 2019.
12. Cedefop: European Centre for the Development of Vocational Training. ICT professionals: skills opportunities and challenges (2019 update). https://skillspanorama.cedefop.europa.eu/en/analytical_highlights/ict-professionals-skills-opportunities-and-challenges-2019-update, November 2019.
13. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.
14. Eclipse foundation. Emfatic: A textual syntax for emf ecore (meta-)models. https://www.eclipse.org/emfatic/.
15. Martin Fowler and Rebecca Parsons. *Domain-specific Languages*. Addison-Wesley, 2010.
16. Object Management Group. Model driven architecture (MDA) MDA guide rev. 2.0, 2014.
17. Object Management Group. Unified modeling language v2.5.1. OMG document formal/17-12-05, available from https://www.omg.org/spec/UML/2.5.1, 2017.
18. Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, and Claudia Ermel. Triple graph grammars in the large for translating satellite procedures. In *Proceedings of the 7th International Conference on Model Transformation (ICMT)*, volume 8568 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2014.
19. Zhenjiang Hu and Hsiang-Shang Ko. Principles and practice of bidirectional programming in BiGUL. In Jeremy Gibbons and Perdita Stevens, editors, *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*, volume 9715 of *Lecture Notes in Computer Science*, pages 100–150. Springer, 2016.
20. John Edward Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *ICSE*, pages 471–480. ACM, 2011.
21. Michael Johnson and Robert D. Rosebrugh. Symmetric delta lenses and spans of asymmetric delta lenses. *Journal of Object Technology*, 16(1):2:1–32, 2017.
22. Frederick P. Brooks Jr. No silver bullet – essence and accident in software engineering. In *Proceedings of the IFIP Tenth World Computing Conference*, pages 1069–1076, 1986.

23. Ralf Lämmel. *Software Languages*. Springer, 2018.
24. Kevin Lano. Constraint-driven development. *Information & Software Technology*, 50(5):406–423, 2008.
25. D. Lutterkort. Augeas: A linux configuration API, version 0.10.0, December 2011. Available from http://augeas.net/.
26. Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. How software designers interact with sketches at the whiteboard. *IEEE Trans. Software Eng.*, 41(2):135–156, 2015.
27. Tiziana Margaria and Bernhard Steffen. Business process modeling in the jabc. In Jorge S. Cardoso and Wil M. P. van der Aalst, editors, *Handbook of Research on Business Process Modeling*, pages 1–26. IGI Global, 2009.
28. Tiziana Margaria and Bernhard Steffen. Service-orientation: Conquering complexity with XMDD. In Mike Hinchey and Lorcan Coyle, editors, *Conquering Complexity*, pages 217–236. Springer, 2012.
29. marketsandmarkets.com. Low-code development platform market by component (solution and services (professional and managed)), deployment mode, organization size, vertical (telecom and it, bfsi, government), and region - global forecast to 2022. https://www.marketsandmarkets.com/Market-Reports/low-code-development-platforms-market-103455110.html, January 2018.
30. Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP*, pages 47–58, 2007.
31. Bertrand Meyer. *Agile! The Good, the Hype and the Ugly*. Springer, 2014.
32. Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoît Combemale, Robert B. France, Rogardt Heldal, James H. Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave R. Stikkolorum, and Jon Whittle. The relevance of model-driven engineering thirty years from now. In *MoDELS*, volume 8767 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2014.
33. Bashar Nuseibeh, Steve M. Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, 2001.
34. OMG. MOF2.0 query/view/transformation (QVT) version 1.3. OMG document formal/2016-06-03, 2016. available from www.omg.org.
35. Marian Petre. UML in practice. In *ICSE*, pages 722–731. IEEE Computer Society, 2013.
36. Bob Reselman. Why the promise of low-code software platforms is deceiving. https://devopsagenda.techtarget.com/opinion/Why-the-promise-of-low-code-software-platforms-is-deceiving, January 2018.
37. Andy Schürr. Specification of graph translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science (WG94)*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
38. Andy Schürr and Felix Klar. 15 years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT*, volume 5214 of *LNCS*, pages 411–425. Springer, 2008.
39. Sir Nigel Shadbolt. Shadbolt review of computer sciences degree accreditation and graduate employability. https://www.gov.uk/government/publications/computer-science-degree-accreditation-and-graduate-employability-shadbolt-review, April 2016.

40. Stuart Shapiro. Research abstract. In William Aspray, Reinhard Keil-Slawik, and David L. Parnas, editors, *History of Software Engineering*, pages 45–46. Dagstuhl Seminar, August 1996.
41. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Journal of Software and Systems Modeling (SoSyM)*, 9(1):7–20, 2010.
42. Perdita Stevens. A simple game-theoretic approach to checkonly QVT Relations. *Journal of Software and Systems Modeling (SoSyM)*, 12(1):175–199, 2013. Published online, 16 March 2011.
43. Perdita Stevens. Is bidirectionality important? In Alfonso Pierantonio and Trujillo Salvador, editors, *Modelling Foundations and Applications - 14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Toulouse, France, July 25-29, 2018, Proceedings*, volume 10890 of *LNCS*, pages 1–11. Springer, 2018. Keynote paper.
44. Perdita Stevens. Maintaining consistency in networks of models: Bidirectional transformations in the large. *Software and System Modeling*, 19(1):39–65, 2019. In print Jan 2020. Online first, May 2019.
45. Perdita Stevens. Connecting software build with maintaining consistency between models: Towards sound, optimal, and flexible building from megamodels. *Software and System Modeling*, 2020. In press. Online first, March 2020.
46. Tamás Török. Software development trends 2018: Latest research and data. https://codingsans.com/blog/software-development-trends-2018, April 2018.
47. VersionOne. 12th annual state of agile report. https://explore.versionone.com/state-of-agile/versionone-12th-annual-state-of-agile-report.
48. Charles W. Wessner, editor. *New Vistas in Transatlantic Science and Technology Cooperation*. National Academies Press, 1999.
49. Perdita Stevens with Rob Pooley. *Using UML: software engineering with objects and components*. Addison-Wesley Longman, 1999. Current edition updated for UML2: first published 1998 (as Pooley and Stevens).
50. Steven Witkop. MDA users' requirements for QVT transformations. OMG document 05-02-04, 2005. Available from www.omg.org.
51. Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. Parsing and reflective printing, bidirectionally. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 2–14. ACM, 2016. See also http://biyacc.yozora.moe/.