Software and Systems Modeling (2020) 19:935–958
https://doi.org/10.1007/s10270-020-00788-4

SPECIAL SECTION PAPER

Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels

Perdita Stevens[1]

Received: 19 July 2019 / Revised: 12 December 2019 / Accepted: 17 February 2020 / Published online: 19 March 2020

Abstract
Software build systems tackle the problem of building software from sources in a way which is sound (when a build completes successfully, the relations between the generated and source files are as specified) and optimal (only genuinely required rebuilding steps are done). In this paper, we explain and exploit the connection between software build and the megamodel consistency problem. The model-driven development of systems involves multiple models, metamodels and transformations. Transformations—which may be bidirectional—specify, and provide means to enforce, desired "consistency" relationships between models. We can describe the whole configuration using a megamodel. As development proceeds, and various models are modified, we need to be able to restore consistency in the megamodel, so that the consequences of decisions first recorded in one model are appropriately reflected in the others. At the same time, we need to minimise the amount of recomputation needed; in particular, we would like to avoid reapplying a transformation when no relevant changes have occurred in the models it relates. The megamodel consistency problem requires flexibility beyond what is found in conventional software build, because different results are obtained depending on which models are allowed to be modified and on the order and direction of transformation application. In this paper, we propose using an orientation model to make important choices explicit. We show how to extend the formalised build system *pluto* to provide a means of restoring consistency in a megamodel, that is, in appropriate senses, flexible, sound and optimal.

Keywords Megamodel · Build system · Model transformation · Bidirectionality · Orientation model

1 Introduction

Model-driven development (MDD) is now well established in a number of niches such as automotive software [1]. It has the potential to fundamentally transform software development by enabling genuine separation of concerns so that decisions about software behaviour can be taken by those best placed to make them, where appropriate without the intervention of software specialists. However, it has been slow to emerge from its niches and become the dominant mode of software development. There are many reasons for this, some technical, some organisational.

Among those reasons—with both technical and organisational aspects—is that we so far lack a good understanding of how *collections* of models can be robustly and efficiently managed. The time taken to apply model transformation tool chains is already a problem [2], motivating our attention to optimality, but flexibility is an even greater concern.

The Object Management Group (OMG)'s original ideal of MDA [3] was basically unidirectional and tree-like: a highly abstract, platform-independent model would be transformed into a platform-specific model from which code would be generated. Megamodelling [4] recognises that real large-scale software development will typically require more flexibility than was envisaged originally, e.g. models will be related in graphs, not trees, and there are more relationships than "generates". A *bidirectional transformation* (bx) between adjacent models in the graph captures the appropriate notion of *consistency* between them. The notion of

Communicated by Richard Paige, Andrzej Wasowski, and Oystein Haugen.

✉ Perdita Stevens
perdita@inf.ed.ac.uk

[1] Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, UK

consistency maintenance is extremely general. The consistency relation to be maintained might be project-specific, or it might be a standard one, e.g. conformance between a model and metamodel. The bx also specifies how to restore consistency when it is lost. Unidirectional transformation is then a special case; for example, in compilation, the object code is considered consistent with the source precisely when it is the result of compiling the source; restoring consistency means recompiling. Note that throughout this paper, we take an "everything's a model" perspective: metamodels, code, etc. included. Thus, even though in some work on MDD the "conformance" relation is special, it does not need to be for all purposes: a metamodel is a model, and the relation "conforms to" between models and metamodels is a fine example of a consistency relation. Tools that check and perhaps even restore conformance fit into our consistency maintenance approach. This generality is an important aim of the work: we provide a disciplined approach to combining uni-, bi- and multi-directional transformations, which may be heterogeneous, in a network of models. We do not assume that models, or transformations, share a common technological base. This holds promise to enable the combination of best-of-breed technologies.

Specifically, we are concerned with settings in which there may be:

- Multiple models, maybe used by different people, recording different concerns,
- Several of which are simultaneously "live", that is, in which decisions may be recorded,
- And which are not completely orthogonal, so that a decision recorded in one live model may necessitate a change to another live model. (Sometimes a distinction is made between models being related "horizontally" or "vertically", on the basis of whether they are considered to be at the same "level of abstraction". That distinction does not matter for this paper).

These three factors are identified as the "essence of bidirectionality" in [5].

The collection of models that are relevant to a system, and the relationships between them, can itself be seen as a model, which may require and repay explicit attention as a designed artefact: this is what we mean by a *megamodel*.

In [6,7], we discussed networks of models connected by model transformations (which might be bidirectional) and pointed out, for example, that the result of consistency restoration will normally be different, depending on the order (and direction, for bidirectional transformations) in which the individual model transformations' consistency restoration processes are used. This presents management problems for consistency maintenance in megamodels: it is these problems that we address in this paper.

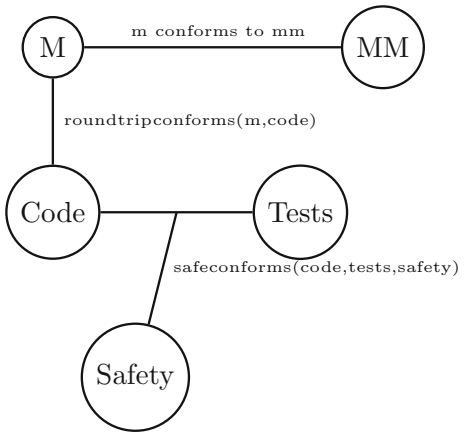

Fig. 1 Megamodel derived from [6]. (Notation: lower-case `model` is instance of upper-case `Model`)

Here is an example which we will consider in more detail in Sect. 5. Figure 1 informally illustrates a small megamodel derived from [6]. The circles represent model spaces within which different teams work, and the lines represent relationships that are supposed to hold between the models. So, at some point in development, there may be a design model (`m` in `M`) which is supposed to conform to a metamodel (`mm` in `MM`); there may be some `code` (in `Code`) which is supposed to satisfy some round-tripping relationship with the design model `m`, such as providing an implementation for all and only the classes mentioned in its class diagram; there may also be a test suite (`tests` in `Tests`) and a safety model (`safety` in `Safety`), with a more complex ternary relationship between them which we will return to. At a certain point, a modification has been made to the design model, such that it no longer conforms with the metamodel, nor satisfies the round-trip relationship. Perhaps a change has simultaneously been made to the test suite. What should be done? There is no straightforward answer, because the right thing to do depends on the circumstances. For example, if the metamodel to which the model is supposed to conform is the standard UML metamodel, then it is not sensible to try to restore that conformance relationship by modifying the metamodel, which should rather be considered *authoritative*; however, if the model is in an evolving domain-specific modelling language, it may be. For another example, even if the individual transformations `roundtripconforms` and `safeconforms` each provide a means of updating the code to bring it into consistency with the design model, respectively, with the tests, the result of applying these transformations will in general depend on the order in which they are applied. Worse, quite likely neither order will produce a desirable result, and some reconciliation between their actions will be required. Nevertheless, we would like to do better than giving up and assuming totally manual control of

the application of the transformations and the reconciliation of their results.

As this example illustrates, any approach for maintaining consistency in a megamodel faces several challenges. Unlike the situation in conventional software build, in a megamodel setting the same artefacts that human experts are working on must sometimes be changed by the automated system: this is a consequence of separating concerns into models that nevertheless cannot be perfectly independent. But when? If the model someone is working on is changed too often, in a way that is not under their control, this will be frustrating and confusing and may lead to errors. On the other hand, if development proceeds too far without reconciliation, effort may be wasted. Moreover, if the amount of work to reconcile models and generate software is considerable, the time taken to do it may be unacceptably long, which is already a problem in MDE [2].

It turns out that the concerns that arise when managing multiple models in an MDD process are related to, yet not subsumed by, those that arise when managing multiple program units in a conventional development process. In this paper, we bring recent advances in formalisation and optimisation of build processes to bear on megamodelling, to address these concerns. Our contributions are as follows.

1. We clarify the relationship between building software and maintaining consistency in a megamodel which may include bidirectional relationships, not just unidirectional generation relationships.
2. In particular, we discuss the role, in consistency maintenance, of the properties that build systems may or may not have, discussed in [8].
3. We propose the use of an *orientation model* to manage key decisions about how to restore consistency.
4. We show how to adapt the formalism of the sound and optimal incremental build system *pluto*[1] [9] to this setting, appropriately combining use of the orientation model with encapsulated decisions about how to update each model.
5. We demonstrate that a soundness result and an incrementality result can then (with care) be derived using those proved in [9], and we discuss the relevance of these results in an MDD setting.

Paper organisation The rest of the paper is structured as follows. First, in Sects. 2 and 3, we discuss related work and introduce some background ideas. Section 4 discusses how build system work can be applied in MDD; it discusses some extra difficulties that arise in the MDD setting, and introduces terminology from [8] and its application in our setting. In Sect. 5, we describe examples and scenarios, used throughout. Section 6 summarises the formalisation and relevant results from [9], and Sect. 7 shows how it is formally adapted to the megamodel setting. Section 8 gives the soundness and optimality results. Section 9 relates the custom stamper idea, key to the *pluto* work, with some notions known in MDD. Section 10 adds some further discussion, and Sect. 11 concludes and discusses future work and open problems.

This paper is an extended version of [10], which was presented at MODELS'18. It has been restructured in order to allow for better explanation of how it uses the underlying *pluto* theory and framework: examples, explanation and discussion have been added throughout. We expand on the role of custom stampers and how they might be automatically generated. We newly discuss the relation with Mokhov et al.'s paper [8], which appeared after [10] but seems to have high potential in this area; in Sect. 11, we discuss future work that will build further on this.

2 Related work from the MDD community

The topic of consistency maintenance between more than two models has been gaining attention in the MDD community [11], especially from theoretically minded MDD researchers including Trollmann and Albayrak from a graph transformation perspective [12] and Diskin and collaborators from a more purely categorical one [13]. It is conceptually challenging, and increasingly acknowledged as an essential part of what must be mastered for MDD to achieve its full potential. The *orientation models* we propose here bear a superficial resemblance to Trollmann and Albayrak's *graph diagrams* [12]. However, all of this previous work has in common that it makes strong assumptions about the possibility of representing all the models, and the changes to them, within a common framework. For example, Diskin's work, as is natural in a category–theory framework, is predicated on the availability of *updates* describing what has changed in a model, Trollmann and Albayrak's requires maintaining correspondences between parts of models. Real-world scenarios involving multiple models, however, may be arbitrarily heterogeneous. We cannot assume that all models are expressed in languages based on the same meta-meta-modelling language, or that we have any control over what tools will be used to edit the models, or that it even makes sense to describe changes to different models using the same language. Unlike the bidirectional transformation literature, the software build literature has always been forced to confront this lack of control, and has handled it by a strict separation of concerns; that is the approach we will take here.

Practically oriented work on the build process within MDD—that is, on the way in which the application of model transformations and other modelling related technologies is

[1] http://pluto-build.github.io/: not to be confused with Apache Pluto.

orchestrated—has generally been modelled closely on conventional software build, and has used only unidirectional model transformations. Representative examples are [14–16]; note that [16], though it shares an author with [9], does not build on *pluto* and has concerns largely orthogonal to ours.

Turning to the special needs of building in megamodels that might include automatically interrelated sources, two recent papers illustrate, in different ways, how far there is to go. In [6,7], I discussed what is lost by limiting bidirectional model transformations to relate just two models. I suggested that in many cases this is tolerable, so that MDD projects could work with networks of binary bidirectional transformations, instead of requiring explicitly multi-directional syntax. I pointed out that in such networks, many problematic issues arise. These include the (non-)existence of a globally consistent state, its (un-)reachability by means of the consistency restoration functions of the bidirectional transformations, and the fact that different sequences of applications of these may yield different results. That paper did not attempt to solve these problems, beyond pointing out some special cases in which consistency restoration is possible, and it did not address incrementality.

More positively, Di Rocco et al. [17] described an attempt at a concrete solution to this problem implemented in the web-based modelling platform MDEForge [18]. However, this solution was very limited in scope, because it disallows the cases identified as problematic by [6]: it requires that a reachable globally consistent state exists and not only that it be unique, but more, that it be reachable in only one way. A related journal paper, Di Rocco et al. [19] has very recently appeared; however, its focus is orthogonal to ours and it does not seem to lift the relevant restrictions.

3 Background from the software build community

In conventional software build, we start from a collection of human-authored source artefacts (hereinafter we will say "files": see Sect. 10.7) and combine these via a number of intermediate stages into runnable software. Intermediate stages may involve generating files from a subset of the sources and/or other generated files. Such a generation step is a function, which takes some sources and produces one or more generated files.

We need to be slightly more precise: it is a partial function. This is because it may happen that a given set of sources is inconsistent in the sense that it does not correspond to any set of generated files: the build step gives an error. However, the usual expectation is that this arises in relatively rare and easy-to-fix cases, thanks partly to technical mechanisms such as explicit interfaces, and partly to organisational ones such as the same team being responsible for all the files that are sources for a given build step. A key insight, explored in [8], is that partiality is but an example: a build step is an *effectful* function. Besides partiality, other relevant effects include non-determinism and dependence on the state of other artefacts. It is unusual in the software build setting for one of the effects of the function to be interactivity, but this may assume greater importance in an MDD setting.

We may see the build process as being a process of *restoring consistency* to the whole collection of files: source, intermediate and target. Each generated file is considered consistent with its sources if it has been built from them in the intended way, and the whole collection is consistent if this is true of the running software and everything it depends on (directly or indirectly).[2] Problems arise if a source is changed and something that depends on it is not rebuilt, or if the intended relationship between sources and generated file is changed without changing the generated file (and then everything that depends on it) accordingly. Typically, a *clean* build, in which all generated files are deleted and everything is regenerated from sources, is straightforward to get right, but expensive. The difficulty is typically to ensure correct *incremental* building: when some sources change, we prefer to save time by rebuilding only the generated files that are no longer consistent with their sources, iterating this process appropriately so that the final software is correctly built. What we mean by correctly built is, typically, that it is *identical* with what would be achieved by a clean build. Because there is a clean separation between sources (never automatically modified) and generated files (never manually modified), and because the generation steps are (partial) functions, so that at each stage there is at most one automatic way to restore consistency, this is (informally) equivalent to saying that the whole collection of files is consistent.

Our use of the weasel word "typically" in the previous paragraph alludes to a complicating factor that we, like other authors on related topics, shall mostly avoid: cyclicity, where a file depends indirectly on itself, i.e. there is a cycle in the dependency graph of files. Achieving a correct build in such circumstances needs special measures, such as repeating build steps until (hopefully) a fixed point is reached. Although familiar to users of LaTeX, this is generally seen as undesirable. As we will mention in Sect. 11, *pluto* does incorporate an attempt to permit cyclic builds [9]: but we, like Mokhov et al. in [8], will avoid them (by imposing an appropriate well-formedness condition on the orientation model).

[2] We phrase it this way because of a subtlety: if, in the current configuration, a generated artefact is not used, it may not be required to satisfy a consistency relation with its sources that would be needed if it were used. That is, the set of consistency relations that are relevant may, in general, change.

4 Towards applying build system work in MDD

MDD separates concerns into different models, which may be worked on by different people.

As mentioned, the Object Management Group (OMG)'s original view of Model-Driven Architecture (MDA) [3] was basically unidirectional: a highly abstract, platform-independent model would be transformed into a platform-specific model from which code would be generated. It was acknowledged that there would sometimes be a need for bidirectional transformations, such as model-code round-tripping; but language and tool support for such transformations have been slow to develop.

Modern MDD, especially what is called megamodelling [4], recognises that real large-scale software development will typically be much less regimented than in OMG's original vision. The collection of models that are relevant to a system, and the relationships between them, can itself be seen as a model which may require and repay explicit attention as a designed artefact. (Note that the term *megamodel* is used in several different senses, which can sometimes lead to confusion: e.g. sometimes for the model, whose elements are models and relationships between them, and sometimes for the actual collection of all the models and relationships that are represented in that model. In Sect. 7, we will formally distinguish megamodel-skeletons, megamodels and megamodel instances.) As real-world examples demonstrate [20], the models in a megamodel often involve overlapping information. This may show up as an "overlaps" relationship between models at the same ("horizontal") level of abstraction, or conformance, refinement or other "vertical" relationships: for our purposes, the technical concerns are the same. In either case, the models have a non-trivial consistency relationship which needs to be maintained as they are changed by developers. Maintaining such a relationship is the job of a model transformation; the simple case in which one model is generated from another can be seen as a special case, in which the consistency relationship that must be maintained between source and target is just that the target is equal to what is generated from the source. This simple special case can only pertain, of course, when the information contained in the target is a subset of that contained in the sources (and the transformation itself). We are more generally concerned with situations where each model contains information that is recorded nowhere else: e.g. information needed only by the humans who work with this particular model. The model transformation itself may be fixed, as when a commercial code generator is used, or may change over time and need to be managed like any other software artefact.

To get full benefit, we must allow more than one model to be simultaneously "live", that is, able to have decisions recorded in it. Otherwise, the humans who are working with those models, and recording their decisions in them, cannot work simultaneously. However, typically, these models are not perfectly independent: a change in one may necessitate a change in another. These factors are identified as the "essence of bidirectionality" in [5]. Today, restoring such models to a consistent state is often done manually. However, this is sometimes inconvenient or impossible. The models may be under the control of different humans, none of whom have sufficient familiarity with them all to be able to reconcile them manually easily and safely (e.g. the PIM and PSM in classic MDA [3]). And/or the notion of consistency between the models may make the reconciliation required very burdensome (e.g. round-tripping between a UML model and code). In either case, having to restore consistency manually may negate the benefit of separating the concerns in the first place.

A bidirectional transformation is, as mentioned, a means of maintaining consistency between two or more such models. Many approaches to defining bx exist, and, as discussed in Sect. 10.2, this paper places few restrictions: we focus on how bx, expressed in whatever formalism, can be used in a disciplined way as ingredients in a broader mechanism to restore consistency within a megamodel. We will assume that the bx, at least, specifies a consistency relation between the models, so that we know when nothing needs to be done. Note that this relation will not usually be bijective (if it were, the models would just be recording the same information in different forms). The bx's other job is to restore consistency when it is lost. Depending on the specific formalism chosen for the bx, it may do this deterministically (probably using the current state of more than one model) or non-deterministically, using search, or even with user interaction; it may or may not be allowed to fail. One assumption we will make about our bx: a bx provides a means to restore consistency by modifying just one model (as do bx in all major bx languages, and, of course, unidirectional transformations). This is less of a restriction than might at first appear; we will discuss this point in Sect. 10.1. Of course, the choice of *which* model to modify may be made differently at different times: we say that to make this choice is to choose the *direction* in which to *apply* the bx.

When, and how often, consistency must be restored is itself an interesting question (see Sect. 10), but typically a set of models will have to be consistent by the time code is generated from it, and indeed, depending on which artefacts are represented in the megamodel, the generation of code can be seen as part of the process of restoring consistency. As "everything's a model" (including code, compiled code and executable software), the problem of restoring consistency in a megamodel *subsumes* that of conventional software build.

4.1 Extra difficulties in consistency maintenance

Thus, consistency maintenance in a megamodel is a problem related to, but not identical with, the problem of software build. What further issues do we have to address in megamodel consistency restoration?

The chief assumption that is usual in software build systems but that does not hold for consistency maintenance in megamodels is that any source—file or model—is either a source or a generated file, never both. The distinction is that sources are modified by means that the build system must just take as given (typically, human choices) and must never be modified by the build system. The build system only modifies generated files; indeed, these may (if convenient) be assumed to be under the complete control of the build system. By contrast, a bidirectional transformation reads several models, and, using information about the state of all of them, modifies one of them. The modified file is therefore both source and generated.

This fact, that some models are both read and written by both humans and the consistency maintenance process, also presents new questions about how the process should be administered, so as not to interfere over-much with the work of humans using the models. Some of these questions were raised in [21] and subsequently discussed in the Dagstuhl meeting on multi-directional transformations reported in [11]. If two models are both being actively worked on, when, and under what (or whose) control, should consistency between them be restored? Moreover, if there is a choice of which of them to modify in order to restore consistency, how should that choice be resolved? Since restoring consistency may involve unavoidable inconvenience to the users of some model, we may argue that making these decisions is part of over-arching project management: certainly, we cannot allow people to cause changes to other people's models in an undisciplined way.

Notice, however, that the gulf between this situation and that of a collection of files managed using a software build system is not quite as wide as one might think at first sight. Even though the build system does not model, check or restore consistency between its sources, there *are*, conceptually, consistency relations between the sources. For example, a `.h` file and a `.c` file may both be sources to a build system, but the build will fail unless a certain consistency relation between the interface described in the `.h` file and the implementation in the `.c` file holds. The build fails, in such a case, because there is no way to restore this consistency relation automatically, and hence no way to produce a `.o` file consistent with both the `.h` and the `.c`. Similarly, when we maintain consistency between models in a megamodel, we do not necessarily record and formalise every consistency relation that must hold. It is normal that some aspects of consistency are maintained manually. Indeed, this is a strength of the approach, enabling gradual adoption (see Sect. 10.3).

Models which are analogous to sources in software build, in that they are modified only by humans, never by the build (rsp. consistency maintenance) system, will be termed *always-authoritative*. Models which are not to be modified in some particular application of the consistency maintenance system will be termed *authoritative*.

4.2 Problems and progress in build systems

Unfortunately, the engineering of conventional build systems is itself not a solved problem. It is recognised that build scripts are often hard to read and maintain (prominent estimates of the proportion of development effort devoted to the development of build scripts are 12% [22] and 27% [23]!) and error-prone. Developers using complex build scripts often end up feeling compelled, in an attempt to avoid being affected by subtle errors, to do clean rather than incremental builds (e.g. defaulting to `make clean; make all`). Correspondingly, maintainers of build scripts often shy away from incrementality for fear of introducing subtle problems. The result is often that builds are unacceptably slow. Heroic efforts (e.g. [24]) have been made to force `make` into doing the right thing as well as to replace it with better systems; yet problems persist.

Fortunately, in recent years this has been recognised as a major problem in software build. Hence, it is an area of active research, and progress is being made. In [9], Erdweg, Lichter and Weiel succeeded in proving soundness and optimal incrementality in *pluto*, which is a formalised build system: that is, *pluto* comprises both a formalism and a corresponding (open-source) software framework. In *pluto*, each generated file is the responsibility of just one *builder*, which is capable of rebuilding the file from its dependencies when the *pluto* algorithm detects that it is necessary to do so. The optimality result is that (subject to certain assumptions) as few builders (hence, e.g. compilations) will be run as possible, and within that, as few checks will be carried out as possible. A key contribution of that work is that they formalise the idea of custom stamps. Improving on the traditionally used timestamps, these give a more general, customisable notion of what it means for one file to be up-to-date with respect to others. Although such custom stamps have not been very widely used in practice, and (as far as I know) never in the full generality envisaged in the *pluto* framework, one related example is the way the Avaloq DSL Developer toolkit[3] uses object fingerprints for fine-grained dependency analysis, within the specific context of the building required in xtext-based language engineering.

[3] https://ddk.tools.avaloq.com/overview.html.

Most recently, Mokhov et al. [8] have begun to investigate the commonalities and differences between a collection of advanced build systems in order to build a common formalisation, enabling them to be compared and their advantages combined. In the process, they make precise and explicit several properties that build systems may, or may not, have. Since they also have relevance for the consistency maintenance problem, we briefly review them here. The build system on which we shall build, *pluto*, has each of these properties.

Dynamic dependencies. Traditionally, most dependencies—cases where a change to one file necessitates a change to another—are known statically; for example, they are written explicitly into each rule of a Makefile. If, by contrast, a dependency may be discovered only when a build step is executed, it is *dynamic*. A build system that is able to handle dynamic dependencies can be more efficient, because there is no need to specify dependencies so pessimistically; we avoid rerunning a build step just in case it is necessary to do so, when in fact the change that prompted the rerunning is currently not relevant.

In the present work, for comprehension and ease of presentation, we assume that the megamodel itself is fixed: thus, for each model, we know statically the (maximal) set of models on which it may depend. However, we make use of *pluto*'s dynamic dependency support to support restoring consistency differently in different circumstances. As we shall see, the *orientation model* will record key project management decisions about, for example, in which direction a bx shall be applied, and which models may be modified. Thus, it will be the *current content* of the orientation model which determines the build work to be done, including the dependencies.

Self-tracking. A build system may or may not be able to detect the logical implications of a change to the build task itself (e.g. the addition of a dependency or a change to a build rule) and do the necessary rebuilding, even if no source has changed. Crudely, this can be achieved in almost any build system by adding the build file itself as a dependency, but greater granularity is desirable. Three aspects of self-tracking are relevant to us. Our orientation model, which records dependencies, is itself a model and, as we shall see, custom stamping enables the consistency restoration process to react to changes in it in an efficient, fine-grained way. Second, *pluto* automatically adds dependencies on the Java classes comprising the *builders* of each model, which means that a change to how we want consistency restoration to be done will cause (only) the necessary rebuilding (up to the hot-swapping capabilities of the underlying JVM). Third, transformations can themselves be taken as (always-authoritative) models in the megamodel, and stamped like any other model, so that a relevant change to an individual transformation can be acted on appropriately.

Early cut-off. It may happen that a file is rebuilt because it appeared to be out-of-date, but, in fact, the result of rebuilding it is that it does not change. If the build system supports early cut-off, then this lack of change means that tasks that depend on this task are not caused to rerun. Building on *pluto* means that we get this benefit: in fact, the use of custom stamps actually gives us a stronger version of early cut-off, in which a change to a file that is not *relevant* to something that depends on the file is also prevented from causing later rebuilding steps to run. We will see an example of this in Sect. 8.1.

In future work, we hope to complete fitting the consistency maintenance work into Mokhov et al.'s ongoing work (see Sect. 11). However, the key idea of custom stamps is not yet incorporated in Mokhov et al.'s formalism: although they make allowance for different means of detecting whether a source is outdated, they do not allow for the same source to be used in different ways, with *different* notions of outdatedness. Because in our setting it is common that different aspects of the same model will be relevant to different consistency relations, this is important to us.

To summarise, build systems are an active and promising area for consistency maintenance to draw on. In the present work, we choose to build on *pluto* because, as well as being both a formalism and an open-source software framework, it incorporates two unusual capabilities that are useful in an MDD setting: the aforementioned custom stamps, and dynamic dependencies, such as the possibility that the content of one model determines whether or not a change to a second model necessitates a change to a third.

5 Examples

In this section, we introduce two examples. The first, drawn from [17], uses only unidirectional transformations, but enables us to motivate different scenarios of consistency restoration in which models may be authoritative or always-authoritative. The second, drawn from [6], involves bidirectional transformations and enables us to discuss matters including the use of consistency restoration procedures that do more than simply apply bidirectional transformations.

5.1 Unidirectional example

Figure 2 illustrates a megamodel, derived from [17], with a metamodel (`mm`), two models (`m1` and `m2`), and a delta (`delta`). The collection is consistent if: the models conform to the metamodel, the delta is the result of applying the `compare` operation to the models, and `m2` is the result of applying `patch` to `m1` and `delta`. Now, as a specification,

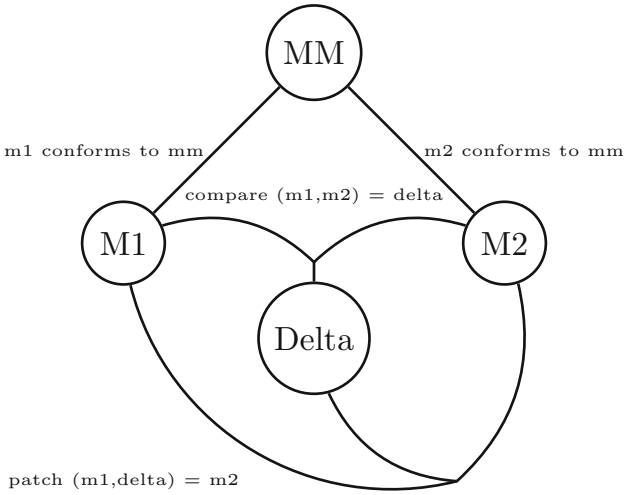

Fig. 2 Megamodel derived from [17]. (Notation: lower-case `model` is instance of upper-case `Model`)

this is redundant: as explained in [17], `compare` and `patch` have the usual joint specification, where

```
compare(m1,m2) = delta
```

iff

```
patch(m1,delta) = m2.
```

The main purpose of the `compare` and `patch` functions is that they provide means of restoring consistency when it is lost.

In [17], the idea is that consistency is restored after every change, and so the scenarios considered are those that start from a consistent set of models, just one of which is then changed. Even then, we must note that there may be a choice of how to restore consistency. If `m1` is changed, we may either apply the `compare` function to the new `m1` and the old `m2` to get a new `delta`, leaving `m2` unchanged, or we may apply the `patch` function to the new `m1` and the old `delta` to get a new `m2`, leaving `delta` unchanged. There is no a priori reason to prefer one of these solutions over the other: it depends on which of `m2` and `delta` should be taken to be authoritative.

Figure 3 represents those two situations using what we will shortly formalise as an *orientation model*. Solid blobs represent authoritative models; e.g. we suppose that the metamodel will always be authoritative (though as noted in Sect. 1, this is a fact about this example: not every metamodel will be always-authoritative). Di Rocco et al.'s assumption [17] is that the changed model, in this case `m1`, should always be authoritative; this makes sense in this setting, because we have no consistency restoration functions available that can take an old version of a model into account and produce a new version of that same model, so the only alternative would be to overwrite the changes just made entirely, which is presumably undesirable.

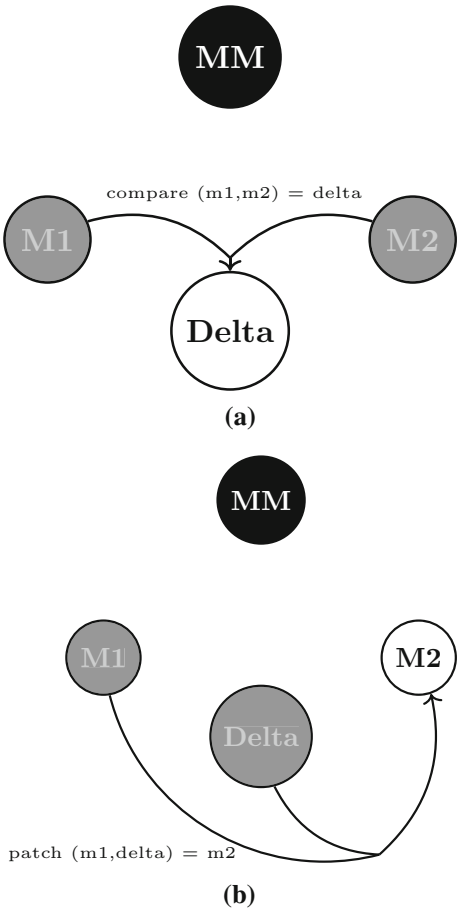

Fig. 3 Orientation models (grey = authoritative, black = always-authoritative)

The megamodel in [17] specifies that the models should conform to the metamodel, but it provides no operations to ensure this. The fact that there are no "conforms to" edges between `M1`, `M2` and `MM` in the orientation models in Fig. 3 corresponds to this fact: it shows that this aspect of consistency will have to be ensured and checked externally.

5.2 Bidirectional example

Recall from Sect. 1 that Fig. 1 illustrates a megamodel adapted from [6]. Here, we see a design model `m` that needs to conform to a metamodel `mm`; some `code` that must be consistent with the model `m` via a standard round-tripping relation; and a more interesting ternary relation between the `code`, a test suite `tests` and a `safety` model. The idea is that, at least, the `code` should pass the `tests` (otherwise no triple involving that `code` and those `tests` will be considered consistent) but also that the `safety` model records (among much other information not relevant here) whether or not the system is considered safety-critical. If it is, then the `tests` are also required to satisfy a coverage criterion.

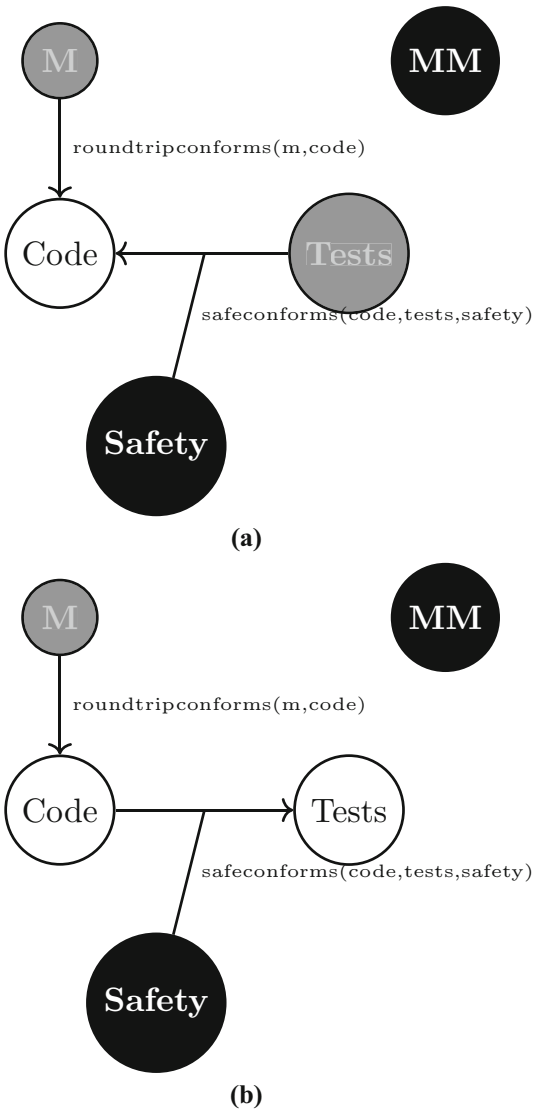

Fig. 4 Orientation models (grey = authoritative, black = always-authoritative)

Soundness. Even if we are provided with a bidirectional transformation that can restore each individual relation in the megamodel, we still need a disciplined way to roll changes through the network. For example, in Fig. 4b, if we want up-to-date `tests`, we must restore `roundtripconforms` first, then `safeconforms`.

Figure 4a represents the situation discussed in Sect. 1. Our framework allows us to encapsulate the reconciliation of different consistency relations impacting the same model in the builder of each model (here `code`). The orientation model records the *contracts* of the builders. To restore consistency respecting the orientation model in Fig. 4a is to modify only non-authoritative models (here, only `code`) in such a way as to make all the consistency relations on the edges hold. In this simple case, the arrows are redundant; but as Fig. 4b shows, an arrow might connect two non-authoritative models, in which case it indicates the priority of changes in the models, in a way which we will make precise by describing how consistency restoration is done. Note that such builders must in general be allowed to fail, as there may be no way to satisfy all the required relations.

Incrementality. We may suppose that checking the relationship between `code` and `tests` is expensive (it involves running tests and computing a coverage metric): we do not want to redo it more often than necessary. In particular, since the only change to the `safety` model that is relevant to this relationship is the one bit record of whether the system is safety critical or not, we do not want to recheck the relationship between `code` and `m` every time `safety` changes in any respect: changes to anything in `safety` that leave that one bit alone do not require us to do any rechecking. Similarly, if the use case diagram part of `m` changes, but this is not relevant to the consistency relation between `m` and `code`, we would like our system to be able to detect that there is no need to recheck that consistency. We can achieve this using a custom stamp: see Sect. 9.

Flexibility. Conventionally, e.g. in [6], we think about restoring consistency to the whole network. In practice, however, that may not be the right thing to do. For example, in the case that an operation changes in the model `m`, thereby breaking consistency with `code` (and `tests`), it may not be sensible to update `code` and `tests` immediately (especially if, say, three more changes will follow in quick succession). What we should be able to ensure is that someone who is relying on `tests` is able to ensure, when they wish to do so, that they are indeed using an up-to-date version of `tests`. We will therefore use a *demand-driven* approach. Rather than pushing the changes from model `m` to `tests`, as the approach in [17] does, we will say: the person who wants to use `tests` will submit a *build request* for `tests`. This will in turn submit a build request for any model on which `tests` depends, before using those updated models to recheck the consistency relation on `tests`. The *pluto* algorithm determines which builders actually need to be run in order to satisfy the build request. In this case, requests that `tests` be brought up-to-date will require that `code` is first brought up-to-date, and this will automatically be done when the build request is processed by the *pluto* algorithm.

6 About pluto

We need to introduce some background on *pluto*, but, of course, will avoid reproducing the technicalities [9] in full detail. We explain the parts we need, omitting features we do not make use of: for example, the reader familiar with *pluto*

$$
\begin{array}{lcll}
P & ::= & \langle \mathsf{path} \rangle & \text{path to a file} \\
\Omega & ::= & P \to \langle \mathsf{file} \rangle_\perp & \text{file system} \\
 & & & \\
FS & ::= & P \times \Omega \to S & \text{file stamper} \\
S & ::= & \mathsf{stamp}\ \langle \mathsf{value} \rangle\ FS & \text{file stamp} \\
 & & & \\
R & ::= & \mathsf{freq}\ P\ S \mid \mathsf{breq}\ B & \text{file or build requirement} \\
G & ::= & \mathsf{gen}\ P\ S & \text{provided file with stamp} \\
B & ::= & \{build : \Omega \twoheadrightarrow U \times \Omega,\ path : P\} & \text{builder} \\
U & ::= & \{builder : B,\ reqs : \overline{R},\ gens : \overline{G}\} & \text{build unit}
\end{array}
$$

Fig. 5 *pluto* concepts and syntax of build units, adapted (to remove variable input) from [9]

should consider that we take the input type to be unit. Figure 5 adapts Fig. 5 of [9] accordingly, and introduces standard concepts and notation which we will refer to throughout. We refer the reader to [9][4] for more detail and, of course, for proofs of the correctness and optimality theorems on which we rely.

We assume given notions of path (the type of operating system paths to files), file (the type of contents of files), and value (the type of whatever stamps we like to compute). Then, a filesystem (an element of type Ω) assigns to each path its current file contents, or $\perp$ if there is currently no file at that path.

pluto incorporates a build *algorithm* that accepts a *build request* (a request to (re)build a particular generated file) and determines when to invoke the *build method* (the *build* field) of a *builder* (of type B). These builders, including their *build* methods, are provided by a user of the framework. They must satisfy certain requirements, which we shall shortly list. Provided that they do so, the algorithm guarantees that the behaviour of the framework overall provides certain soundness and optimality properties.

Each generated file is a responsibility of just one *builder*, whose *build* method describes how its file(s) shall be generated, including what other builders must be up-to-date to do this properly. As part of the requirements imposed on it by the framework, it uses framework-provided methods to record what files it reads and writes. Before it reads a file which is itself a generated file—that is, which has its own builder—this builder must use a framework-provided method to ask that file's builder to check that it is up-to-date; this prevents stale versions of files from being used in the build. Formally, a *build* method operates on a file system, and (unless it fails) produces a record called a *build unit* (an element of type U) which it saves for later inspection. The build unit records: which builder built it (its *builder* field); a list (its *reqs* field) of its dependencies, i.e. what other builders were required (e.g. breq b indicating that this builder requested that builder b be rerun if necessary) and what files were read (e.g. freq f indicating that the file at path f was read); and a list (its *gens* field, which in our application will always have length 1) of the files written (e.g. gen g indicating that this builder wrote the file at path g). As well as the path to the file concerned, each freq and gen entry records a *stamp* to enable relevant later changes in the files to be detected (see below). All this information is later used by the *pluto* algorithm to decide whether it is necessary to invoke the builder's *build* method again, or whether it is unnecessary, because nothing it used has changed in a way that the build unit declares that this builder cares about.

[4] And/or to a video of the corresponding talk, at https://youtu.be/QsgLSDMLLTo.

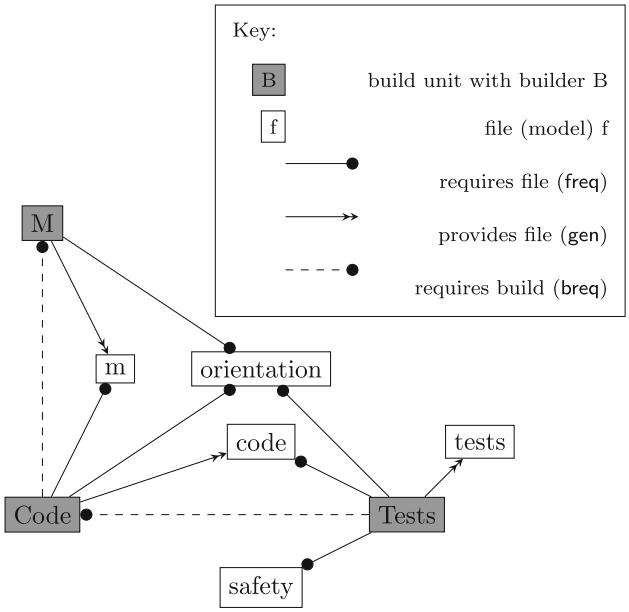

Fig. 6 Dependencies of a build in the context of decisions recorded in Fig. 4b (cf Fig. 3 of [9])

Figure 6 illustrates the gen, freq and breq dependencies between build units and the files (models) they refer to, in the context of an orientation model corresponding to Fig. 4b. Note that this abstracted figure does not show the order in which the dependencies are recorded in a build unit, which is important for the operation of the algorithm, nor the stamps that have been used. We will walk through this in Sect. 8.1.

It is important to understand that requiring (breqing) a builder does not invoke the *build* method of that builder directly: rather, it sends another build request to the *pluto* build algorithm, so that it can check whether or not a rebuild is required. That is done using stamps.

Stamps Each file said to have been read ("freqed") in the build unit is identified by giving a path (from P), and, crucially, a *stamp* (from S). This is a value determined by the builder *for this use of this file*.

Each stamp is associated with a *stamper* (from FS); the idea is that the builder chooses a stamper, which produces

the stamp (there are framework-provided stampers that produce, for example, the last modification time of a file, a hash of its contents, or a Boolean for whether it exists). Formally what the stamper has to be able to do is to take a path (in P) and a file system (in Ω) and compute a stamp for the file (if any) which is *currently* found at that path. A key part of the algorithm's checking whether a build unit is up-to-date, i.e. whether its builder needs to be rerun, is: look at the path and stamp of each file that it records having read; get the stamper from that stamp; ask the stamper to compute the stamp associated with the path *in the current file system*; compare this stamp with the one recorded. The file is considered up-to-date iff the recorded stamp is the same as the current stamp (e.g. the last modification time has not changed). The generality of the stamper set-up means, however, that a stamp could be anything convenient.

Thus, the choice of stamp(er) is made by the developer of the builder, and it is the key element in defining what it *means* for the system to be "built correctly": the choice must ensure that if two versions of the file at a given path have the same stamp then they are interchangeable to this builder, in the sense that a change from one to the other does not necessitate rerunning it. So the easiest, safest choice for the developer of a builder is to use the finest possible stamper, in which any change at all to a file will change the stamp; last-modified time, supported by the operating system, is traditional. As in conventional build, this can already avoid a lot of unnecessary work. In practice, though, it can happen that a file changes in a way which definitely does not cause a rebuild to be necessary. For example, if only comments in a source file change, it is (barring strange compiler bugs!) unnecessary to recompile it.[5] Therefore, we might be able to save rebuild effort by deciding that a stamp on a file should be computed ignoring the comments, so that changes to comments alone do not change the stamp. In our megamodel setting, this kind of thing happens *more* than in conventional system build, because it is normal that only part—perhaps only a small part—of the information contained in a model is relevant to a particular bidirectional transformation involving it. For example, if the `roundtripconforms` relationship in Fig. 1 only depends on a class diagram part of the model `m`, but `m` also includes other diagrams, the developer of the builder that builds `code` might choose to stamp its use of `m` with a stamp computed from the class diagram part alone.

We have been talking about the builder's choice of stamps to place on freq-ed files, i.e. files read. In fact, in *pluto* the builder also chooses a stamp for each generated (gen-ed) file. However, in our setting we always stamp gen entries with a stamp fine enough to detect any consistency violation (in practice, last-modified time will do). We discuss this in Sect. 10.4.

Correctness, soundness and optimality When a builder completes, a file it generates (gens) is considered correctly built, provided every file the builder read (freqed) was stamped with the same stamp as is computed from the current version of the file. The gened file is up-to-date for as long as this remains true.

A build unit is *internally consistent*, at a given moment when the builder returns, if all required and generated files are up-to-date (i.e. their recorded stamps are indeed equal to what their stampers produce on the files at this moment), and build units exist for all required builders.

We elide the details of what it means for a build system to be *sound*, but informally, it means that a non-failing build produces an internally consistent build unit for each build request, and for any build requests generated in the process of carrying these out, and that they are all properly linked with no stomping on one another's files. Crucially, only one build unit is allowed to have generated the file at any given path. In our setting, this means that each model is the responsibility of at most one builder.

Requirements that builders must satisfy Conditions that the developer of a builder must arrange to satisfy are formally given as requirements on the build unit that the builder produces; these are then assumed in the proofs of soundness and optimality. In practice, the software framework provides considerable support for meeting these requirements. Our megamodel extension will help even more: we will give a skeleton form of a *build* method which ensures all these conditions are met.

- **C1** breq before freq: If any file is required that is a generated file of another builder, then that builder must be required earlier in the build unit's list of requirements (*reqs*) than the file. This ensures that an out-of-date generated file is not used.
- **C2** The builder must either fail, or produce a build unit which is internally consistent. This is Assumption 4.1 in [9], and enables the soundness result.
- **C3** Enabling the optimality result, Erdweg et al. [9] has a further assumption (4.2), essentially that the list of requirements captures enough information to describe differences in the dynamic behaviour of the builder. It states that if a build unit's *reqs* list contains a build requirement breq b, and the only requirements coming *before* this in *reqs* are either build requirements, or file requirements that are up-to-date in the current file system, then rebuilding this builder in the current file system will give a build unit whose *reqs* list still includes breq b.

[5] Note that "only comments change" rules out (un)commenting code since that adds or removes code too!

The effect of this requirement is that any information that a build method relies on to determine whether or not to invoke a builder must itself be recorded as a dependency earlier in the list: so, if nothing recorded has relevantly changed, then neither has the need for the builder changed.

7 Adapting pluto for MDD

This section is the heart of the paper (although the reader should not be tempted to start here: it relies crucially on material presented earlier). In it, we will

- Give a simple formalisation of a megamodel, suitable for interpreting in the *pluto* framework (Sect. 7.1);
- Formalise the concept of *orientation model* for capturing (and varying) decisions about which models may be modified, and in which direction bx will be applied (Sect. 7.2);
- Formalise the concept of a *Megamodelbuild system* based on a megamodel and an orientation model, including
- Specifying how the *build* method of each builder in such a system must be constrained, in order that the system as a whole will be able to do its job of restoring consistency in the megamodel instance soundly and optimally. We first give a definition in terms of the *pluto* framework, and then show how such *build* methods are implemented using a template method (Sect. 7.3).
- We illustrate this in the context of the example from Fig. 4a (Sect. 7.4).
- Finally, we add some remarks (Sect. 7.5).

This sets us up to go on in Sect. 8 to prove soundness and optimality of a consistency restoration system built up in this way, satisfying these constraints.

7.1 Megamodels

Recall that a megamodel is a way of *specifying* a collection of modelling artefacts and relationships between them. These relationships may include that one model conforms to another, that one is generated from another, etc. Formally, let us give a very general description, in which we do not, for example, assume there are consistency restoration functions, nor make any distinction between models and metamodels, nor between different kinds of relationships between models.

Definition 1 A megamodel-skeleton $\mathscr{H}$ comprises:

- A finite set Node, called *nodes*
- A possibly empty $\mathsf{AAuth} \subseteq \mathsf{Node}$, the nodes that are *always-authoritative*
- A finite set Edge of (directed) *(hyper)edges*, together with a function

  $$\mathsf{nodes} : \mathsf{Edge} \to \mathsf{Node}^*$$

  such that for every $E \in \mathsf{Edge}$,
  - $|\mathsf{nodes}(E)| \geq 2$, and
  - The elements of $\mathsf{nodes}(E)$ are all distinct.

  This function need not be injective, i.e. we do not forbid multiple edges between the same list of nodes.

Definition 2 A megamodel $\mathscr{M}$ over a megamodel-skeleton $\mathscr{H} = (\mathsf{Node}, \mathsf{AAuth}, \mathsf{Edge})$ comprises:

- A valuation v_{Node} of nodes, giving for each $N \in \mathsf{Node}$ a set $v_{\mathsf{Node}}(N)$ of *models*.
- A valuation v_{Edge} of edges, giving for each $E \in \mathsf{Edge}$ with $\mathsf{nodes}(E) = (N_1, \ldots N_k)$, a relation on the model sets the edge connects, i.e. a subset $v_{\mathsf{Edge}}(E) \subseteq v_{\mathsf{Node}}(N_1) \times \cdots \times v_{\mathsf{Node}}(N_k)$.

Remark 1 In [10] I combined the concepts of megamodel-skeleton and megamodel into one (there only ever is one megamodel per megamodel-skeleton in that paper), and this led to a little confusion. Here, let us be more pedantic.

Definition 3 An *instance* of a megamodel $\mathscr{M} = (v_{\mathsf{Node}}, v_{\mathsf{Edge}})$ over a megamodel-skeleton $\mathscr{H} = (\mathsf{Node}, \mathsf{AAuth}, \mathsf{Edge})$ is a collection of one model n in each $v_{\mathsf{Node}}(N)$.

The instance is *consistent* if all the relations are satisfied, i.e. whenever $n_{i_1} \in v_{\mathsf{Node}}(N_{i_1}), \ldots, n_{i_k} \in v_{\mathsf{Node}}(N_{i_k})$ are models in this instance and $v_{\mathsf{Edge}}(E) \subseteq v_{\mathsf{Node}}(N_{i_1}) \times \cdots \times v_{\mathsf{Node}}(N_{i_k})$ is an edge in the megamodel, we have $(n_{i_1}, \ldots n_{i_k}) \in v_{\mathsf{Edge}}(E)$.

Remark 2 The reader may be wondering about models which may be absent from an instance, either

- Because they need to be generated from other models in order to restore consistency to the instance; or
- Because they are genuinely optional according to the megamodel.

Given the generality of the set-up, this can easily be modelled using a special "no information" model, say $\Omega_N \in v_{\mathsf{Node}}(N)$, to represent the absence of the model. The consistency relations, recording whether Ω_N is consistent or not with the possible values of other models, can capture both of the above situations.

This is nothing but a slight generalisation of the way the early bx literature [25] uses a special model $\Omega_N \in N$ in order to allow a consistency restoration function $\overrightarrow{R} : M \times N \to N$

to model a situation in which a model in N must be generated afresh: given a model $m \in M$, the newly generated model in N is $\overrightarrow{R}(m, \Omega_N)$. (The alternative approach, used in the early lens literature [26], requires a lens to provide a special create function, taking just a view to a source, in addition to the put function which takes a view and a source to a new source. The "no information model" approach we use is more notationally convenient, given that we wish to talk explicitly about consistency relations.)

Remark 3 Notice that this set-up encompasses two MDD situations:

1. All the nodes are models, and the transformations between them are encoded as edges;
2. Some of the nodes themselves represent transformations. ("Transformations are models!")

For example, if bidirectional transformation R (the currently programmed transformation from a set $\mathscr{R}$ of transformations) between model sets M and N specifies a consistency relation, we may choose whether or not to encode the transformation itself as a node. If we do not, we will simply have an edge R between nodes M and N, specifying that m and n are consistent precisely when $R(m, n)$ holds. If we do, we will have nodes M, N, and $\mathscr{R}$, with a hyperedge between them specifying that m, n and R are consistent precisely when $R(m, n)$ holds. The latter gives us the flexibility to react automatically (without needing to modify builders) to changes in the definition of the transformation. Our framework permits both variants without further ado.

7.2 Orientation model

Typically, there will be some nodes in a megamodel which it is helpful to include, but never appropriate to change automatically. (e.g. we never want our automated consistency restoration procedure to modify the UML metamodel). These are the *always-authoritative* nodes. For others, whether we permit them to change, or take them as authoritative, depends on the situation. We use a special model to capture such variations in the situation.

Definition 4 An *orientation model* O over a megamodel-skeleton $\mathscr{H} = (\mathsf{Node}, \mathsf{AAuth}, \mathsf{Edge})$ comprises:

- All the nodes of the megamodel-skeleton Node
- A possibly empty set of nodes Auth designated *authoritative*, satisfying

  $$\mathsf{AAuth} \subseteq \mathsf{Auth} \subseteq \mathsf{Node}$$

- A possibly empty set of edges $\mathsf{OEdge} \subseteq \mathsf{Edge}$
- An orientation for each edge, i.e. a function

  $$\mathsf{target} : \mathsf{OEdge} \to \mathsf{Node}$$

  satisfying $\mathsf{target}(E) \in \mathsf{nodes}(E)$ for each $E \in \mathsf{OEdge}$.

It is *well-formed* if

- It is acyclic, that is, there is no sequence $E_1 \ldots E_n$ ($n > 1$, each $E_i \in \mathsf{OEdge}$) such that $\mathsf{target}(E_i) \in \mathsf{nodes}(E_{i+1})$ for each i and $\mathsf{target}(E_n) \in \mathsf{nodes}(E_1)$, and
- No target node is authoritative, that is,

  $$\mathsf{target}(\mathsf{OEdge}) \cap \mathsf{Auth} = \emptyset.$$

7.3 Specifying a megamodelbuild system

Given this setting, we can equip our megamodel with *pluto* builders (constrained as we shall see) and use the *pluto* algorithm (unmodified) to restore consistency.

We are about to define builders for our models. As explained above, a build unit is a record of a successful run of a builder, which the *pluto* algorithm consults to see whether a builder needs to be rerun. In more detail, we have to define:

- A *build* (partial) function (implemented as a method of the builder: the implementation might fail, raising an exception), taking a file system to a build unit and a new (i.e. possibly modified) file system;
- A *path* which, in a file system, will either lead to a file where the latest build unit is stored, or will be undefined ($\perp$) in case the last build of this builder failed. That is, the builder only produces a path to an actual file where a build unit is stored, if the build succeeded. Hence the build unit, if it exists at all, may be relied upon.

The path is only interesting in that *pluto*'s soundness depends, naturally, on builders not overwriting one another's build units ("stomping on one another's files"); we will say no more of it, as no new considerations arise in the MDD context.

The interesting part of defining a builder B is the *build* function, i.e. determining, based on the initial state of the file system, what the build unit and the final file system must be. To recapitulate:

- The first element, labelled *builder*, of the build unit is always the builder that wrote this build unit: there is no choice for the builder-writer to make.
- The second element, *reqs*, is a list of build and file requirements: this is the interesting part. Definition 6 explains how this is determined, and the restrictions on the stamps that must be used on file requirements.

Table 1 Correspondence between formalisation and framework

Formalisation	Framework
breq	`requireBuild`
freq	`require`
gen	`provide`

– The third element, *gens* is a list of stamped provided files. In our setting, however, there is no choice for the builder-writer to make here. Each builder only generates one file, viz. the model it is responsible for, and we will always stamp this unique gen entry with a stamp fine enough that it changes when the file is modified in any way that might violate any consistency relation (in practice, last-modified time will do fine). See Sect. 10.4 for discussion of this choice.

The modified file system that is returned is just the result of carrying out the *build* method and saving each generated file at the appropriate path. Since in the implementation the file system may, of course, be modified in ways beyond the simple saving of a generated file (e.g. running a model transformation engine may produce a log file, which plays no role in the formalisation), some latitude is needed to match up the formalisation with reality so that the soundness and optimality results pertain. Here, the "no stomping on one another's files" assumption is again used to match up the actual implementation behaviour to its simplified formalisation. That is, it does not matter if the implementation makes unformalised modifications to the file system, provided that these do not do any harm to things that *are* relevant to the formalisation.

In the implementation, the build unit is built up by the *pluto* framework, based on the builder's calls to framework-provided methods. We shall look at an extract in a moment. For example, if the builder calls the framework-provided `requireBuild` method, this (as well as guiding the *pluto* algorithm) writes a breq entry to the build unit. Table 1 summarises the correspondence between build unit entries and framework methods.

Definition 5 A *Megamodelbuild system* for a megamodel $\mathscr{M}$ over skeleton $\mathscr{H}$ comprises a well-formed orientation model O for $\mathscr{H}$, together with, for each node M in Node \ AAuth, a *pluto* builder whose *build* method behaves according to the constraints in Definition 6. Such a builder is called an M-builder.

Recall that a *build* method formally reads and writes a file system and (unless it fails) returns a build unit in which the *builder* field simply specifies that this is the builder which wrote it. Specifically, in a Megamodelbuild system for megamodel $\mathscr{M}$, the *build* method of the M-builder will read the current instance of megamodel $\mathscr{M}$, and the orientation model, from the file system. It may write a modified version of the model it is responsible for, M, into the instance. Via the operation of the *pluto* algorithm, its build requests may indirectly invoke other builders, causing other models to be modified too. On successful completion, it returns a build unit (and saves it to the filesystem).

The contentful work, besides calculating the new model, is to specify the entries in the *reqs* and *gens* field.

Definition 6 Given a file system containing a megamodel instance and an orientation model O, the *build* method of the M-builder must construct a build unit by carrying out the following steps in this order.

1. freq the orientation model O. (That is, enter an freq entry for the path of the orientation model and the appropriate stamp (see below) as the first entry on the *reqs* list of the build unit being constructed.)
2. Determine, from O, the set $\mathscr{E}$ of directed (hyper)edges having M as target; let $\mathscr{N}$ be the set of nodes that are sources of these edges.
3. breq the N-builder for each $N \in \mathscr{N}$ (such that N is not always-authoritative). (That is, append these breq entries to the *reqs* list.)
4. freq all the models which are the values of nodes in $\mathscr{N}$ in the current instance. (That is, append these freq entries, for the paths of these models, with appropriate stamps (see below), to the *reqs* list.)
5. Calculate a new version of model $m \in M$ *that makes all the relationships in* $\mathscr{E}$ *hold* (and save it).
6. gen m, i.e. record that m has been (re)generated. (That is, make this the sole entry of the *gens* list in the build unit being constructed—recall that we always use a fine stamp on gen entries.)
7. Return (and save) a build unit recording this sequence of requirements and generation.

The builder must use stamps fine enough to ensure that if a file changes without changing the stamp, consistency will not be lost. It may stamp O just with a record of its own authority status and which edges target it (i.e. $\mathscr{E}$).

Implementation in Megamodelbuild In the prototype Megamodelbuild system,[6] a base class `MegaBuilder` for builders, providing a template method for the *build* method, is provided. Recall that each builder is responsible for a single model. The *build* method will be invoked if, and only if, the *pluto* algorithm has determined that this model may need to be changed. It must then meet the requirements laid out in Definition 6 above. Figure 7 summarises the template method's code.

[6] https://github.com/PerditaStevens/megamodelbuild.

Fig. 7 Abstract of template build method

```
@Override // of the Pluto framework build method
  protected Out<File> build(Input input) throws IOException, MegaException {
    // set file to be the model this builder is responsible for
    // set om to be the orientation model
    require(om, getOrientationStamper()); // step 1
    // step 2:
    String orientationInfo = orientationModel.getInfoFor(getName(), getOrientationStamper());
    boolean isAuthoritative = OrientationModel.authoritative(orientationInfo);
    if (isAuthoritative) {
      // do nothing - steps 3-5 become trivial
      report(getName()+" is authoritative, so no resolution to be done");
    } else {
      // do the actual work - steps 3-5:
      restoreConsistency(input, file, orientationInfo);
    }
    provide(file); // step 6
    return OutputPersisted.of(file); // step 7
  }
```

Here, `require` is the framework-provided method corresponding to freq-ing a file. Thus, the template method begins by requiring the file containing the orientation model, `om`, as required by Step 1 of Definition 6. It must specify which stamper to use against this file requirement, so that later the *pluto* algorithm can check whether a change to the file does or does not require the *build* method to be rerun. In this case, the appropriate stamper is the one returned by Megamodelbuild's `getOrientationStamper` method, i.e. a stamper written specifically for stamping orientation models: it captures, as specified in Definition 6, its own authority status and which edges target it. From the orientation model, the build method extracts the information that is relevant to the model for which this builder is responsible: this is Step 2 of Definition 6.

- If the model is marked as authoritative in the orientation model, then, since the orientation model is assumed to be well-formed, the sets $\mathcal{N}$ and $\mathcal{E}$ are empty: this model must not be modified, so there is nothing to be done. Steps 3–5 of Definition 6 become vacuous.
- Otherwise, some specific action must be taken, and this is encoded in the `restoreConsistency` method which is invoked by the template *build* method, and will be implemented in a model-specific subclass of `MegaBuilder`. This method must carry out steps 3–5 of Definition 6. It is passed the `orientationInfo` which tells it with which models it must restore consistency (i.e. sets $\mathcal{N}$ and $\mathcal{E}$). It must use framework methods `requireBuilder` and `require` to record those dependencies in the eventual build unit, and then restore consistency by modifying this model accordingly. How this is done is encapsulated: it is up to the implementer of the specific builder. If there is a unique edge incident on this model, and that edge is associated with an external transformation, then the builder may simply run that transformation, by invoking an external transformation tool. However, the situation may be more complicated, such as in Fig. 4a, where there are several separate consistency relations that must simultaneously be restored. The `restoreConsistency` method might invoke several consistency restoration procedures, perhaps from different external tools, and might also do arbitrary "fixing up" before determining that the model has had consistency fully restored, or else failing. We will see an example in a moment.

After the model has been brought into consistency with its neighbours, the template method uses the framework-provided `provide` method (with a fine, default stamp) to record that this model has been gen-ed (step 6 of Definition 6). It saves and returns a build unit (step 7).

7.4 Example in the context of Fig. 4a

Let us fill in a few more details in the specific context of the orientation model shown in Fig. 4a, as follows. Consider the task faced by the writer of the `Code`-builder, who must implement a `restoreConsistency` method. Sometimes, a builder-writer's job will be very easy: the builder will have to do nothing more than invoke a pre-existing model transformation. On other occasions, restoring consistency may require more intelligence: perhaps there is an available model transformation but it does not completely do the job, or there are several that must be combined in the right order, perhaps with some pre- or post-processing. Or perhaps there is no model transformation available to help, and the builder-writer must implement the `restoreConsistency` method from first principles, manipulating the model explicitly. What is fixed is the notions of consistency specified by the edges of the megamodel: in accordance with Step 5 of Definition 6, the `restoreConsistency` method of the `Code`-builder must produce a version of the `Code` that is consistent with the models identified from the orientation model, in this case `m`, `tests` and `safety`. If it cannot do so, it fails, raising an exception: the whole build fails.

In this case, we suppose that the ingredients available to the writer of the `Code`-builder's `restoreConsistency` method—whom we will call Sam from now on, for brevity—include pre-existing bx `roundtripconforms` and `safeconforms`. These incorporate the required consistency relations on the corresponding edges of the megamodel, and also provide consistency restoration behaviour, which Sam may use in combination to restore consistency to `code` on demand. We next give some more details about these hypothetical bx, in order to discuss how Sam might decide to use them.

- 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.
- The `safeconforms` edge requires (among other things, as previously discussed) 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.

Now, the two consistency restoration procedures are not *non-interfering* [7]: that is, applying them in different orders can yield different results. Sam's code gets to make the choice of order; the choice can be made statically, or it might even depend on the current states of the models.

Note, for example, 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, applying the two bx in either order may[7] restore consistency: the first one will generate Java code for the missing class, after which the second one will find a corresponding class and succeed. However, it may be that one of the bx is better at generating useful Java code than the other. Sam might choose to invoke this bx first, so that the most useful code gets generated in such a case; then the other bx, invoked second, will find a corresponding class in `code` and its less good code generation capabilities will not be 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, the flexibility to invoke the bx from a *build* method makes it easy to solve this problem. Most obviously, `restoreConsistency` might be written to:

1. Apply the `roundtripconforms` consistency restoration first, possibly creating new classes in `code`, then
2. Automatically 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.

7.5 Remarks

1. Generally the M-builder's newly calculated m will depend on the old value of m, as well as on any linked models. This is unusual in conventional build systems, but essential for bidirectional transformations. A careful read of [9]'s proofs shows that it is unproblematic and does not require breqing this builder (which would result in a build cycle) nor freqing this model (`m` itself).
2. Because an authoritative model is never the target of a (hyper)edge, the builder of a model that is authoritative in the current orientation model will, as expected, neither freq any model nor breq any builder, but just restamp this model (e.g. Test-Builder according to Fig. 4a). We are using *pluto*'s dynamic dependency capabilities here: the builder's requirements depend on the current contents of the orientation model.
3. As we saw in the example, the real work is done in Step 5. If there is a single incoming edge, and if the megamodel is associated with a way to restore consistency along this edge—e.g. the `compare` or `patch` function in Example 5.1, or the consistency restoration function of an individual bidirectional transformation—then all the builder has to do is apply it. In practice, this may be done by invoking a separate transformation engine. In this way, this approach supports the principled heterogeneous combination of different bx technologies.
4. If there is more than one incoming edge (e.g. CodeBuilder according to Fig. 4a), or if the megamodel is not associated with the means to restore consistency along

[7] Elided details prevent us saying definitely "will".

its edges, then, as we saw in the example, more interesting work is required. This might involve adjustment of the result of applying transformations, search, or even user interaction. The choice is encapsulated inside this builder: the requirement is just somehow to deliver a consistent m. The attempt must be allowed to fail, however, because as discussed in [6] there might simply be no solution. Soundness in this setting, as in conventional software build, does not mean that consistency will always be restored, but rather that *if* the algorithm succeeds *then* the result really is consistent.

8 Soundness and optimality

The builders in a Megamodelbuild system, defined following Definition 5, will automatically obey requirements **C1**, **C2** and **C3**; in particular, the sequence of requirements changes only if the orientation model changes, ensuring **C3**. These builders are now used with the standard *pluto* build algorithm, and we get:

Theorem 1 (Soundness) *Invoking the* pluto `build` *algorithm with a build request for the builder of any model M in the megamodel will either fail, or produce a new megamodel instance which is correct, in the sense that consistency holds in the subgraph of the orientation model from which M is reachable.*

Proof (Sketch) Theorem 5.3 of [9] tells us that the build request will, unless it fails, result in a totally consistent build unit for the M-builder: that is, one in which all the stamps are up-to-date, and the same is true of all the build units to which this one is transitively linked via breq entries. (Total consistency also guarantees appropriate book-keeping properties such as that the build units exist and have the expected attributes and links. We refer to [9] for full details.)

What we have to show is that this ensures the correctness we want in the megamodel. We proceed by induction on the length of the longest directed path in the orientation model, which is finite since the megamodel-skeleton has finitely many nodes and the well-formed orientation model is acyclic. The induction hypothesis is that, if there is a totally consistent build unit for a model N to which the longest directed path in the orientation model has length at most $k \in \mathbb{N}$, then consistency holds in the subgraph of the orientation model from which N is reachable. For $k = 0$, this is vacuously true. Suppose it is true for k, and that the longest directed path in the orientation model to M has length $k + 1$. Consider any edge in the orientation model targeting M. Total consistency of the build unit tells that the gen and freq stamps on the involved models are up-to-date. By Definition 5, together with the fact that the gen stamp we always use is fine enough to detect any consistency violation, this ensures that consistency holds along this edge. Total consistency tells us that all linked build units, i.e. the build units for any source models of edges targeting M, are also totally consistent; since the orientation model is acyclic, each directed path in the orientation model to one of these build units has length at most k, so the induction hypothesis applies and we are done.

Note that this is a stronger result than Theorem 5.3 of [9] because of the additional requirement we put on the megamodel builders, that they restore consistency along certain relationships (or fail). We cannot get a guarantee that *all* relationships in the megamodel hold, because this may be impossible.

Optimality Theorem 5.7 of [9], which says:

Theorem 2 (Optimality) *The number of builders executed by the build algorithm (in response to any build request) is minimal.*

transfers directly. Informally, this holds because the algorithm caches previous build results, repeating builds only when they are invalidated because a file changes in a way that the stamps indicate is significant, and then only when the file is genuinely required to build the requested artefact. In our setting, we see in particular that the only builders invoked in response to a build request for a model M are those of models from which there is a path to M in the orientation model; each of these is invoked at most once (by acyclicity), and only if required. For example, with the orientation model of Fig. 4b, if from a consistent state just Safety is altered, and then Test-builder is invoked, the Code-builder will not be invoked.

Note, however, that minimality means a builder is never rerun if it should have been apparent *from the stamps* that this was unnecessary: in *pluto*, the stamps specify what it means for the build to be correct. Of course, we cannot exclude that the model was, as it happened, still consistent with its neighbours—manual changes could have "by chance" maintained consistency in a way that is invisible to the build system until the builder is run.

8.1 Example in the context of Fig. 4b

Fully explaining and reproving the soundness and optimality result here would involve importing essentially all of the content of [9], especially fully reproducing the *pluto* algorithm. Instead, we will explain how the algorithm operates in an example: the interested reader is invited to read this explanation alongside Fig. 6 of [9], but the explanation should stand alone. Figure 6 may be helpful again.

We suppose we are in the context of Fig. 4b, and that a build request for the `Tests`-builder has been submitted: that is, we are restoring consistency to the `tests`. Suppose that,

since this builder was last successfully run, only model m has changed. Potentially, this might have caused code to be out-of-date, and tests in turn may need modification. What will the algorithm do?

Starting with a build request to build tests, the *pluto* algorithm will consult the build unit recording the last operation of the Tests-builder. We suppose that the build unit exists, and its *reqs* field records an freq on the orientation model, then a breq on the Code-builder, then an freq on the safety model (recall that, because Safety is always-authoritative, it does not have a builder, so there is no corresponding breq) and finally an freq on the code file. The algorithm checks the status of the *gens* field of the build unit; in our system, this list always contains simply the one file gened by the Tests-builder, viz. tests, the model for which this builder is responsible. Since we always use a fine stamp on generated files, any change to tests that could invalidate any of the consistency relations incident on it will cause the consistency restoration procedure to be run. In our scenario, there is no change to tests, so next, each of the requirements in the Test-builder's build unit's *reqs* list is checked in turn. In our scenario, the orientation model has not changed so its freqing does not necessitate any work.

The next requirement to be checked is the breq of the Code-builder. The algorithm now looks for its build unit, which it finds. The *gens* field just records that the builder is responsible for the code model, which has not changed. The *reqs* field records an freq on the orientation model, then a breq on the M-builder and finally an freq on the m file. Once again the check of the orientation model succeeds as it has not changed. The algorithm next checks the breq requirement on the M-builder which means finding its build unit. This time, when the status of the single generated file m in the build unit's *gen* field is examined, we find that indeed m has changed. Therefore, the M-builder's consistency restoration procedure is run. Because m is authoritative, this does not make any change to m: the result is a new build unit which records an freq on the orientation model, no other entries in the *reqs* list, and a new stamp on the single entry for m in the gens list. (The algorithm records that this build unit is known to be consistent, so that it would not have to be rechecked if something else breqed the M-builder later, but that will not happen in our scenario anyway.) Returning to the checking of the *reqs* in the Code build unit, we meet the freq on the m file. At this point, the stamp recorded in this freq is compared with the stamp produced by the same stamper on the current version of the m file, to see whether any *relevant* change to it has occurred. Let us suppose that, even though we posited that m had changed, it has not done so in such a way as to change the stamp—perhaps the changes to m were only in the use case diagram, and the stamp that the Code-builder placed on it was computed only from the class diagram. Then, even though we just reran the M-builder, this use of the stamp lets the algorithm know that the check on the freq of m succeeds: no relevant change has occurred. That concludes the checking of the *reqs* list in the Code build unit, so there is *no* need to rerun the consistency restoration procedure of the Code-builder: its build unit is recorded as known consistent.

Returning to the checking of the *reqs* list in the Tests build unit: the checking of the freq on the safety model succeeds because we are supposing that had not changed and, more interestingly, so does the check on the freq of the code. (Note that the same would have been true if the Code-builder *had* needed to run its consistency restoration procedure, but the result had not produced a change to the code that caused it to have a different stamp from the one recorded in the freq of code in the Tests build unit's *reqs*.) Therefore, there is no need to run the consistency restoration procedure of the Tests builder either. Here, we see how the use of custom stamps enables *early cut-off*: even though there was the potential for the change to m to necessitate running of both consistency restoration procedures of Code and Tests, in this case neither had to be done.

(Note that in this example, we have elided mention of some book-keeping aspects of the *pluto* algorithm (shown in Fig. 6 of [9]), especially the validate procedure (Fig. 7 of [9]), which do not give rise to any new considerations in our setting).

9 Custom stampers and bidirectionality

In this section, we discuss the relationship between custom stampers as used in *pluto* and related ideas in MDD. In build system work, a traditional rule is "if the target is already up-to-date with respect to the sources, do not run the builder". As explored by [9], the naive version of this, using time stamps, can lead to inefficiencies: a target may be out-of-date only because a source has changed *in a respect that is irrelevant to the relationship between source and target*. In effect, *pluto*'s stamps impose a builder-specific equivalence relation on the set of possible instances of a file depended on by the builder: instances are equivalent iff they have the same stamp, and this indicates that the instances are interchangeable as far as this particular builder is concerned.

A related idea in MDD is hippocraticness: "if the target model is already consistent with the source(s), do not apply a consistency restorer". This helps avoid disruptive and unnecessary changes to models, but it does not necessarily save computational effort, because checking consistency may itself be arbitrarily expensive. For example, checking whether a given triple of code, tests and safety model are consistent will involve rerunning the tests and computing a coverage metric. On the other hand, if we know that the only aspect of the safety model that is relevant to this con-

sistency is the one bit that says the system is safety-critical, we may safely say that a change to the `safety` model that does not flip that bit does not necessitate rechecking the consistency relation, because the two versions of the `safety` model are equivalent as far as this consistency relation is concerned.

In the safety case, there will be just two equivalence classes of the `safety` model, determined by the safety-critical bit. Or if m is a Java source file and R is maintaining consistency between the Java source file and an HTML documentation page, we may identify an equivalence class of Java sources files with the file comprising a particular set of extracted docstrings, discarding all the code.[8]

The idea of models being equivalent if they differ only in ways that never affect their consistency with another model via a given bidirectional transformation has been explored in [27], in the setting of simple relational state-base bx. Such a bx, relating model sets M and N, is formally defined by a triple:

1. The consistency relation that the bx checks and enforces, $R \subseteq M \times N$
2. A forward consistency restoration function $\overrightarrow{R} : M \times N \to N$
3. A backward consistency restoration function $\overrightarrow{R} : M \times N \to M$.

The bx is termed *correct* if the consistency restorers do restore consistency, i.e. $R(m, \overrightarrow{R}(m, n))$ always holds (and dually for $\overleftarrow{R}$). The formalisation of hippocraticness is that if $R(m, n)$ holds then $\overrightarrow{R}(m, n) = n$ and $\overleftarrow{R}(m, n) = m$. In that setting, as explained by [27], we may define equivalence relations $\cong_F^R$ and $\cong_B^R$ on M (and mutatis mutandis on N) by:

- $m \cong_F^R m'$ iff $\forall n \in N . \overrightarrow{R}(m, n) = \overrightarrow{R}(m', n)$—that is, for *every* model n, the result of using $\overrightarrow{R}$ to modify n so as to be consistent with m is the same as the result of using R to modifying n to be consistent with m'
- $m \cong_B^R m'$ iff $\forall n \in N . \overleftarrow{R}(m, n) = \overleftarrow{R}(m', n)$—that is, any differences between m and m' are such as to be obliterated by synchronisation with *any* element of N.

It turns out that any model $m \in M$ is *determined* by its equivalence classes under these two equivalence relations [27]. In many (but not all) natural cases, the equivalence class of m modulo $\cong_F^R$ is easily reified as the information from m that $\overrightarrow{R}$ looks at. If this equivalence class could easily be computed from m, it would serve as a suitable stamp for the N-builder to use on model m, because a change to m that does not change its equivalence class does not necessitate a change to $n \in N$. An interesting challenge in the context of a particular transformation language (related to slicing) would be: given a transformation, automatically generate a stamper that generates stamps corresponding to these equivalence classes. Indeed, the discussion in Sec. 7.1 of [7] suggests a way to do this, by statically analysing the text of a model transformation to determine which model elements—instances of which metaclasses—may be relevant: the automatic generation of an abstraction, or *part*, of a model that is potentially affected by a bx would give a (pessimistic) candidate for a section of the model to inspect for changes in order to decide whether any change has taken place that might be relevant to the bx. I am indebted to a reviewer for the suggestion that perhaps the intents of transformations, in the sense of [28], might also provide useful information from which stampers could be derived.

There is, of course, a pragmatic question about the trade-off between the expense of computing the stamp on a file, and the expense of rerunning a transformation. We might expect that in the case where a stamp is derived by looking in a safety model for a single bit, and seeing that it has not changed may save substantial effort, this is worthwhile; however, using a custom stamper in the Java/HTML case is less likely to be useful, because computing the stamp may be almost as expensive as regenerating the documentation.

10 Discussion

10.1 Bx modifying just one model

A major aim of this work is to support consistency restoration in networks of models that involve heterogeneous technologies, e.g. various models and transformations whose languages need not even share a common meta-meta-modelling language. Yet we have worked on the assumption that restoring consistency along any (hyper)edge in the megamodel instance involves modifying just one of the models involved. Is there a contradiction here?

The reason for the restriction is that it is essential to our adaptation of *pluto* that any model is the responsibility of just one builder in the system. We cannot, without completely reworking the framework, allow the same model to be modified by two different builders under different circumstances. So if two models must be modified by the same transformation, they must be the responsibility of the same builder. In such a case we will then, for practical purposes, regard them as one combined model.[9] Thanks to custom stamps, it

[8] For a concrete example, see the orientation stampers at https://github.com/PerditaStevens/megamodelbuild.

[9] A draft of this paper used the word "merged", but this caused anxiety in a reader familiar with the problems involved in "model merge". Here, there is no need for any actual merging—no need for the models

is still possible to react efficiently to changes. Say, for example, that we wish to make use of an existing transformation which modifies both a state diagram and a class diagram; and suppose we also wish to use a different transformation which reads only the class diagram. In our framework, there will be one builder which is responsible for both the class diagram and the state diagram, but the transformation that only reads the class diagram may freq the combined model with a custom stamper which ignores changes to the state diagram, so that it will not need to be reapplied if only the state diagram changes.

10.2 State-based versus delta-based approach

Another aspect of this work that might be perceived as a limitation, but which is in fact a deliberate choice, is that there is no requirement within the Megamodelbuild framework for information to be provided about deltas ("what has changed in a model since last synchronisation?"), edits ("what command(s) did the person editing this model issue?") or traceability links (a.k.a corrs, or correspondence graphs: "which part(s) of that model are relevant to this part of this one?"). When such information is available, it is often possible to use it to make more intelligent decisions about *how* to restore consistency than are possible without it [13,29,30]. However, as mentioned in Sect. 2, real-world scenarios that involve multiple models may be arbitrarily heterogeneous. Some of the files in our megamodel instance might be UML models, some Python programs, some models in a previously unknown and ad hoc DSL, some text files. We often cannot assume that all models are expressed as graphs, or in languages based on the same (meta)*-modelling language; or that we have any control over what tools will be used to edit the models; or that it even makes sense to describe changes to different models using the same language. In Megamodelbuild, the decisions about what needs to be done to bring a model into consistency with its neighbours are entirely encapsulated inside that model's builder. If information about deltas, edits or correspondences is available (stored in the same file as the model, or in a different one) it is open to the builder to use it, whether directly or by passing it to a transformation engine that needs it. But this is not—and does not need to be—treated specially in the framework: a file containing such information, if present, would just be another file used by the builder, to be stamped appropriately. As far as the framework is concerned, it does not matter whether a builder achieves its task of bringing its model into consistency with its neighbours using one or more external transformation engines, using its own internal programming, or even with the help of user interaction.

10.3 Humans in the loop and gradual adoption

Given the separation of concerns between the builders, which restore consistency by any means, and the Megamodelbuild framework which invokes builders as necessary, it is even possible for a builder's consistency restoration to involve human interaction (or search, or any other approach). Provided that before the `restoreConsistency` method returns successfully, it verifies that the relevant consistency relations have indeed been restored, the guarantees provided by the framework still hold. In practice, however, it is more likely that situations where human action is required will be captured as they are in conventional software build: that is, by the build failing, so that humans have to take action before rerunning the build. The approach of this paper allows for widely varying degrees of automation. At one extreme, it is possible to use it with no automated consistency restoration at all! That is, every builder's `restoreConsistency` method could merely check whether consistency along the relevant edges holds already; if so, it returns successfully, while if not, it fails, i.e. raises an exception that terminates the build. This already brings benefits: the automated consistency checking is proved sound, and when it fails, it provides an indication of where in the megamodel instance there is an inconsistency (which builder failed and what error message did it emit?). One could envisage gradual adoption of the Megamodelbuild framework proceeding on this basis: initially, builders would be given that minimal check-only behaviour, and then over time, as a cost-benefit calculation permitted, the `restoreConsistency` methods could be improved to automate more of the consistency restoration and fail less often.

10.4 Fine stamps on generated files

A design decision in the Megamodelbuild framework is that the template build method (Fig. 7) does the `provide` framework call, and hence, chooses the stamp that is applied to the gen entry in the build unit. In fact, as can be seen from the absence of a stamp argument to this call, the current framework implementation always uses the default last-modified stamp on gen entries. This is more pessimistic than strictly necessary: the interesting content of the decision is that the stamp should be fine enough to ensure that, if the stamp on the file does not change, then neither does the file's consistency with any other adjacent model in the megamodel. What we have to ensure is this: if the model is modified by something other than its builder—e.g. by a human working on it—and then a build request for it is submitted, the *pluto*

Footnote 9 continued
to be given a common metamodel, for example—so this anxiety is unnecessary. It is just that the framework, agnostic about how models are constructed, is *a fortiori* agnostic about whether someone might consider this model to be the combination of several smaller models.

algorithm needs to be able to detect that the builder's *build* method does need to be called, in order to check that the modifications have not broken consistency.

To see why this is necessary, consider again the scenario presented in Sect. 8.1, but in a modified version where the `tests` had been changed by a human in such a way as to invalidate the `safeconforms` consistency relation. If the gen stamp used by `Test`-builder on `tests` were so coarse that it could not detect these human modifications, then the scenario would play out exactly as presented: in particular, the `restoreConsistency` method of the `Test`-builder would never be called. Hence, the consistency restoration would not be correct, because the build would complete "successfully" but leave a megamodel instance in which `safeconforms` would not hold. (Note that this would not be a problem for soundness in *pluto*'s sense: for *pluto* the stamps, determine what it means for the build to be correct. The issue is that in our setting, we want to *use* the stamps to ensure the kind of correctness we actually care about, viz. restoration of consistency relations in the megamodel instance.)

Our choice to use a pessimistically fine gen stamp, within the template method, frees the builder-writer from the (perhaps error-prone) obligation to choose a stamp which is fine enough to catch all potentially consistency-violating external changes to the model that this builder controls. It could result in a *build* method being called when, in fact, no work needs to be done (note that this does not violate the Optimality Theorem, which only rules out work which is unnecessary *according to the stamps*). An avoidable call to a `restoreConsistency` method will only result, however, when a builder-controlled, non-authoritative model suffers an external modification which does not in fact impact consistency. Since such modifications normally will require a consistency check to be done, our choice seems pragmatically reasonable; nevertheless, it might be interesting to experiment with more flexibility in future.

Notice that this issue only applies to gen stamps, not to freq ones: a stamp used on an freq entry is specific to one use of the file, i.e. to one consistency relation.

10.5 Management of the orientation model

Because the orientation model is just a model (though always-authoritative, i.e. only manually changed!), it will be managed in a configuration management system as usual, and edited, probably by a project manager, to reflect current circumstances of the project, such as which models should be permitted to be modified by the build system. A typical project might have several versions of an orientation model over its lifetime; for example, a model may become authoritative after it is signed off by a customer. We may even have several variants that are interchanged as appropriate, e.g. one that labels a model as authoritative, for use while its own developers are working on it (so that their work is not interrupted), another that does not. As we have seen, the system automatically maintains soundness even if the orientation model changes.

10.6 Changes to the megamodel itself

For simplicity, we have assumed here that the megamodel does not change, although the orientation model may. That is, we use *pluto*'s dynamic dependency capability only to react to changes in the orientation model. It would be possible, however, to use it more; it is unclear whether this would be useful, or rather would diminish the value of the megamodel.

10.7 Files

In order to make use of the existing *pluto* software, which is based around the notion of file, we have adopted here the assumption that models are realised in files, and we have not considered serialisation and deserialisation explicitly. In [14], the authors make the point that for practical purposes it is highly advantageous for a model management workflow to avoid parsing the same model more than once, and they discuss how to use features of Ant to make this work. The concerns are orthogonal to those discussed here, however, and the use of the file system is not essential to anything we have proposed.

10.8 Demand-driven versus global consistency restoration

Following *pluto*, we have adopted here a demand-driven approach to consistency restoration: we provide a mechanism that will not necessarily restore all of the consistency relations in the megamodel, but only those that must be restored in order to produce an up-to-date version of the requested model. This approach is a contrast to earlier work on megamodel consistency, e.g. [6,17]. We think that, for MDD, it is an advance, but note that it is still possible that a rebuild of one model forces an update to another (e.g. `Test`-Builder in Fig. 4b may cause the `code` to be rebuilt, if it is currently inconsistent with the model `m`). This relates to.

10.9 Always-consistent versus stable

In modern software engineering, there is an interesting tension between (a) the desire to avoid duplicating information, and (b) the perceived need to tolerate inconsistency to permit creative flow [31] that may lead to step improvements. Prioritising (a) leads to a preference for having a "golden copy" of any piece of data; in an MDD context, it suggests that any inconsistency should be repaired immediately [17,32].

Nuseibeh et al. [33], argue for (b); in an MDD context, Kuhn et al. [2] make the point that engineers want to work independently on copies of the same model and then need good tool support for reintegration of their modified copies. At issue is the *length of time* for which it is appropriate for some expert (group) to proceed with changing an artefact independently, before bringing it into consistency with all other artefacts. Too short a time, and nobody achieves flow: everyone is constantly interrupted by their artefacts changing underneath them to take account of other people's decisions. Too long a time, and development returns to the bad old days of months-long integration phases. This work does not offer a silver bullet, but it does help to ease the management of such decisions, by embodying them in a concrete artefact (the orientation model), giving explicit constraints on how builders must be written, and providing, in return for the builder-writer abiding by these constraints, guarantees of sound and optimal behaviour. Making the pragmatically best choice in a given setting will inevitably, though, require skill and experience.

11 Conclusions and future work

We have proposed an approach to sound, optimal and flexible megamodel-based building, extending the work of Erdweg et al. [9] to tackle the problem of Di Rocco et al. [17], and to address some of the challenges raised by Stevens [6]. The soundness and optimality are precise results, based on [9]'s work. Flexibility is, naturally, more subjective. Here, we have externalised the decisions about which models are authoritative, etc. into the orientation model, which, being a model like any other, can be changed, such that affected models can be automatically rebuilt in response to the change while unaffected ones need not be. We have shown how decisions about consistency restoration can be encapsulated inside relevant builders. We think this will be more dependable than using a complex build script, especially where developers need to automatically reconcile the effects of several transformations, or use transformations provided by vendors or others and then systematically "tweak" their results.

Since the publication of [10], the appearance of the important work by Mokhov et al. [8] has changed the present author's future plans. By clarifying key concepts relating to software build and how they are related in various advanced build systems, it helps to situate the *pluto* work and clarifies what is important about it. A new project (of the author and James McKinna) aims to mechanise the formalisation of [8], extending it with a mechanised formalisation of custom stamps, and produce machine-verified versions of the soundness and optimality results we have used here. This is a very ambitious goal, made feasible only by the foundational work of [8], and it will take some years. In the short term, this will support an investigation of whether each restriction placed on the framework is required for soundness or optimality, or might usefully be relaxed. In the long term, this could then form the basis for an exploration of even more flexible, yet still sound, means of restoring consistency in megamodels, as follows.

We have shown how certain decisions about the restoration of consistency can be encapsulated in individual builders. In our early experimentation with *pluto* and its formal model, we explored a more radical proposal than is presented here. Here, we have presented the use of an orientation model that specifies, for example, in which direction each relevant bidirectional transformation should be applied, so these decisions are centralised. We could imagine a megamodel build system in which these decisions, too, were encapsulated in individual builders, leading to a radically different and even more flexible approach to the provably correct building of software. In future work, we would like to investigate the extent to which this is (a) technically feasible and (b) pragmatically helpful (vs confusing): both are at present uncertain.

For example, an M-builder would have the responsibility to restore consistency between m and its neighbours in the megamodel, and would be able to choose to do so by *whatever* means were appropriate, e.g. at one time to do so by modifying m, at another to do so by leaving m alone and instead asking the builder of a neighbouring model to modify itself. Such decisions might be based on anything the builder programmer chose, even, for example, local time of day, if we judge that a change to an expert's model in the middle of the night is likely to be less disruptive than one that takes place during the working day. However, pursuing this idea turned out to involve heavy use of *pluto*'s cycle-handling capabilities, which (unlike the cycle-free case) are described only informally in [9]. Thus, since the correctness of *pluto*'s cycle-handling is delicate and would be crucial, we would prefer to have a detailed and preferably mechanised proof of correctness before relying on it to this extent. Assuming that appropriate soundness results could be proved, understanding whether there are settings and assumptions under which this would be useful and usable is an intriguing direction for future work.

More prosaically, a specialised open-source framework for building from megamodels, on top of *pluto*, is available,[10] although currently primitive. (Given the retargeting of future work described above, it is not yet clear whether future development should be undertaken on the current *pluto* basis, however, or whether formal and theoretical work based on [8] will show a better way to proceed.) Manually implementing appropriate builders, as described, is routine, but we would further like to incorporate: wrappers to let builders invoke existing model transformation engines; auto-

[10] https://github.com/PerditaStevens/megamodelbuild.

matic generation of builders from a megamodel expressed in an appropriate language such as MegaL/Forge [17]; connections with further megamodelling work such as [34,35]; generation of custom stamps from transformations; validation of orientation models; exploration of scalability, etc. By permitting, for low effort, trustworthy and fully incremental build of model-driven systems, this is a step towards *continuous model-driven engineering*, as requested, for example, in [32]. Completing the path towards that goal will, of course, require further work, both practical and theoretical.

Acknowledgements I thank the reviewers of this paper and of [10] for their constructive suggestions. I also thank the Bx'18 audience (for a related talk https://youtu.be/Pp1BsQyHoMs with no accompanying paper), especially Sebastian Erdweg, Jeremy Gibbons, and James McKinna, for helpful discussion. Further, discussions at Dagstuhl no. 18491 on *Multidirectional Transformations and Synchronisations* [11] were also very helpful. I am grateful for support from the UK NCSC/RIVeTSS grant RFA20601-4214171, *Mechanising the theory of build systems*.

References

1. Whittle, J., Hutchinson, J.E., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2014)
2. Kuhn, A., Murphy, G.C., Thompson, C.A.: An exploratory study of forces and frictions affecting large-scale model-driven development. In: MoDELS, Volume 7590 of Lecture Notes in Computer Science, pp. 352–367. Springer (2012)
3. Object Management Group. Model driven architecture (MDA) MDA guide rev. 2.0 (2014)
4. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of the OOPLSA/GPCE Workshop: Best Practices for Model-Driven Software Development (2004)
5. Stevens, P.: Is bidirectionality important? In: Pierantonio, A., Trujillo, S. (eds) Modelling Foundations and Applications—14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Toulouse, France, June 26-28, 2018, Proceedings, volume 10890 of Lecture Notes in Computer Science, pp. 1–11. Springer (2018)
6. Stevens, P.: Bidirectional transformations in the large. In: MODELS, pp. 1–17. IEEE (2017)
7. Stevens, P.: Maintaining consistency in networks of models: Bidirectional transformations in the large. Softw. Syst. Model. (2019). Online first, May 2019
8. Mokhov, A., Mitchell, N., Peyton Jones, S.: Build systems à la carte. PACMPL **2**, 79:1–79:29 (2018)
9. Erdweg, S. Lichter, M., Weiel, M.: A sound and optimal incremental build system with dynamic dependencies. In: OOPSLA, pp. 89–106. ACM (2015)
10. Stevens, P.: Towards sound, optimal, and flexible building from megamodels. In: ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS'18). ACM (2018)
11. Cleve, A., Kindler, E., Stevens, P., Zaytsev, V.: Report from Dagstuhl Seminar 18491. Multidirectional Transformations and Synchronisations. Dagstuhl Rep. **8**(12), 1–48 (2019)
12. Trollmann, F., Albayrak, S.: Extending model synchronization results from triple graph grammars to multiple models. In: Van Gorp, P., Engels, G. (eds) Theory and Practice of Model Transformations—9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings, Volume 9765 of Lecture Notes in Computer Science, pp. 91–106. Springer (2016)
13. Diskin, Z., König, H., Lawford, M.: Multiple model synchronization with multiary delta lenses with amendment and k-putput. Formal Asp. Comput. **31**(5), 611–640 (2019)
14. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: A framework for composing modular and interoperable model management tasks. In: In Model-Driven Tool and Process Integration Workshop, pp. 79–90 (2008)
15. Lämmel, R.: Relationship maintenance in software language repositories. Art Sci. Eng. Program. J. **1**, 4 (2017)
16. Erdweg, S., Ostermann, K.: A module-system discipline for model-driven software development. Program. J. **1**(2), 9 (2017)
17. Di Rocco, J., Di Ruscio, D., Heinz, M., Iovino, L., Lämmel, R., Pierantonio, A.: Consistency recovery in interactive modeling. In: EXE at MODELS (2017)
18. Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: Mdeforge: an extensible web-based modeling platform. In: CloudMDE, Volume 1242 of CEUR Workshop Proceedings (2014)
19. Di Rocco, J., Di Ruscio, D., Härtel, J., Iovino, L., Lämmel, R., Pierantonio, A.: Understanding MDE projects: megamodels to the rescue for architecture recovery. Softw. Syst. Model. (2019). First Online: 18 July 2019
20. Hebig, R., Giese, H., Batoulis, K., Langer, P., Farahani, A.Z., Yao, G., Wolowyk, M.: Development of AUTOSAR standard documents at Carmeq GmbH: a case study. Universitätsverlag Potsdam (2015)
21. McKinna, J., Stevens, P.: How to regain equilibrium without losing your balance? scenarios for bx deployment (discussion paper). In: Anjorin, A., Gibbons, J. (eds) Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016, volume 1571 of CEUR Workshop Proceedings, pp. 32–34. CEUR-WS.org (2016)
22. Epperly, T., Kumfert, G.: Software in the DOE: the hidden overhead of "the build". Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, CA, USA (2002)
23. McIntosh, S., Adams, B. Nguyen, T.H.D., Kamei, Y., Hassan, A.E.: An empirical study of build maintenance effort. In: ICSE, pp. 141–150. ACM (2011)
24. Mokhov, A., Mitchell, N., Jones, S.P., Marlow, S.: Non-recursive make considered harmful: build systems at scale. In: Haskell, pp. 170–181. ACM (2016)
25. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. J. Softw. Syst. Model. (SoSyM) **9**(1), 7–20 (2010)
26. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), 17 (2007)

27. Stevens, P.: Observations relating to the equivalences induced on model sets by bidirectional transformations. In: *EC-EASST*, 049 (2012)
28. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M.K., Syriani, E., Wimmer, M.: Model transformation intents and their properties. Softw. Syst. Model. **15**(3), 647–684 (2016)
29. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds). Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012, pp. 495–508. ACM (2012)
30. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. J. Object Technol. **10**(6), 1–25 (2011)
31. Gray, J., Rumpe, B.: The importance of flow in software development. Softw. Syst. Model. **16**(4), 927–928 (2017)
32. Garcia, J.: Continuous model-driven engineering. https://modeling-languages.com/continuous-model-driven-engineering/ (2018). Accessed 16 Mar 2020
33. Nuseibeh, B., Easterbrook, S.M., Russo, A.: Making inconsistency respectable in software development. J. Syst. Softw. **58**(2), 171–180 (2001)
34. Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., Cabot, J.: Moscript: a DSL for querying and manipulating model repositories. In: SLE, Volume 6940 of Lecture Notes in Computer Science, pp. 180–200. Springer (2011)
35. Diskin, Z., Kokaly, S., Maibaum, T.: Mapping-aware megamodeling: Design patterns and laws. In: SLE, Volume 8225 of Lecture Notes in Computer Science, pp. 322–343. Springer (2013)

Perdita Stevens is Professor of Mathematics of Software Engineering in the Laboratory for Foundations of Computer Science at the University of Edinburgh, in Scotland.