Towards sound, optimal, and flexible building from megamodels

Perdita Stevens
School of Informatics
University of Edinburgh
Perdita.Stevens@ed.ac.uk

ABSTRACT

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. In general, however, 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 explain the relationship between software build systems and the megamodel consistency problem. 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

ACM Reference Format:
Perdita Stevens. 2018. Towards sound, optimal, and flexible building, from megamodels. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239378

1 INTRODUCTION

Model-driven development (MDD) is now well-established in a number of niches such as automotive software [22]. It has 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 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 [11], motivating our attention to optimality, but flexibility is an even greater concern. The Object Management Group (OMG)'s original ideal of MDA [8] 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. Megamodeling [2] 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* between adjacent models in the graph captures the appropriate notion of *consistency* between them (which might be standard, e.g. conformance between a model and metamodel, or project-specific), and 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.)

In [19] 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. Here is an example which we will consider in more detail in Section 3. Figure 1 informally illustrates a small megamodel derived from [19]. 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 (in M) which is supposed to conform to a metamodel (in MM); there may be some code (in Code) which is supposed to satisfy some round-tripping relationship with the design model, such as providing an implementation for all and only the classes mentioned there; there may also be a test suite (in Tests) and a safety model (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.

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 megamodeling, 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.

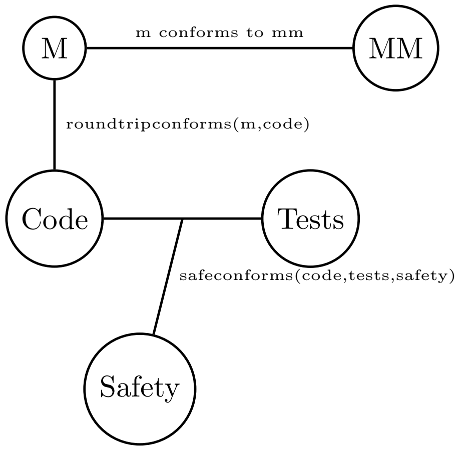


Figure 1: Megamodel derived from [19]. (Notation: lower-case `model` is instance of upper-case `Model`)

(2) We propose the use of an *orientation model* to manage key decisions about how to restore consistency.
(3) We show how to adapt the formalism of the sound and optimal incremental build system *pluto*[1] [4] to this setting, appropriately combining use of the orientation model with encapsulated decisions about how to update each model.
(4) We demonstrate that a soundness result and an incrementality result can then (with care) be derived using those proved in [4], and we discuss the relevance of these results in an MDD setting.

The rest of the paper is structured as follows. First, in Section 2, we discuss related work and take the opportunity to introduce elements of it on which we shall build. In Section 3 we describe examples and scenarios, and in Section 4 explain how custom stampers can help. We go on to provide a formalisation and soundness result in Section 5. Section 6 adds some further discussion, and Section 7 concludes.

2 RELATED WORK AND BACKGROUND

2.1 Build in MDD

Work on the build process in MDD has generally been modelled closely on conventional software build, and has used only unidirectional model transformations. Representative examples are [5, 10, 13]; note that [5], though it shares an author with [4], 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 [19] I discussed what is lost by limiting bidirectional model

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

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, and 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 modeling platform MDEForge [1]. However, this solution was very limited in scope, because it disallows the cases identified as problematic by [19]: 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.

2.2 Building software

In conventional software build, we start from a collection of human-authored source artefacts (hereinafter we will say "files": see Section 6) 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, 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.

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 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 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.

[2]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. I.e. the set of consistency relations that are relevant may, in general, change.

2.3 Model-driven development

MDD separates concerns into different models, which may be worked on by different people. To get full benefit, we must allow more than one model to be simultaneously "live", that is, able to have decisions recorded in it. 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 [20]. 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 [8]). 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* (bx) is a means of maintaining consistency between two or more such models. Many approaches to defining bx exist, and this paper places few restrictions. 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. 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 is allowed to fail. However, we will define a separate builder for each of the models that must be automatically updated, so (for now) we expect the bx to provide a means to restore consistency by modifying just one model (as do bx in all major bx languages, and, of course, unidirectional transformations).

When, and how often, consistency must be restored is itself an interesting question (see Section 6) but typically a set of models will have to be consistent before code is generated from it, and indeed, the generation of code can be seen as a process of restoring consistency. As "everything's a model" the megamodel consistency restoration problem *subsumes* software build.

2.4 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% [12] and 27% [14]!) 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. [15]) have been made to force `make` into doing the right thing as well as to replace it with better systems; yet problems persist.

Nevertheless, in [4] Erdweg, Lichter and Weiel succeeded in proving soundness and optimal incrementality in a formalised build system called *pluto*, in the sense that (subject to certain assumptions) as few builders (units of the build system, e.g. compilations) will be run as possible, and within that, as few checks will be carried out as possible. A key contribution 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.

There is a large literature on build systems, which we do not have space to survey, and correctness and incrementality are topics of increasing interest. 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.

2.5 Summary of *pluto*

We need to introduce some background on *pluto*, but space limits forbid reproducing the technicalities in detail. We explain the parts we need[3] tersely, and refer the reader to [4][4] for more detail.

Pluto is a formalism and software framework incorporating 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* of a *builder*. 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: the developer of the builder must satisfy certain requirements on which the soundness and optimality of building rely. Especially, it uses framework-provided methods to record what files it reads and writes. Formally a build method operates on a file system, and produces a record called a *build unit* which it saves for later inspection. The build unit records, in a list, what other builders were required (e.g. breq b indicating that this builder requested that builder b be rerun if necessary), what files were read (e.g. freq f indicating that file f was read) and written (e.g. gen g indicating that this builder wrote file g). This information is later used by the algorithm to decide whether it is necessary to invoke the builder's build method again.

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, and, crucially, a *stamp*. This is a value determined by the builder *for this use of this file*.[5] Each stamp is associated with a *stamper*; the idea is that the builder chooses a stamper, which produces the stamp (there are 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 and a file system 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 re-run, 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.

[3]e.g. the reader familiar with *pluto* should consider that we take the input type to be unit

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

[5]In fact the same is true of each generated file: but we will assume (for technical reasons) that gen entries are always stamped with the finest possible stamp, which changes when the file is modified in any way detectable by any other stamper used on that file (in practice, last modified time).

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. [6] 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 Figure 1 only depends on a class diagram part of the Model, but the Model also includes other diagrams, the developer of the builder that builds the Code might choose to stamp its use of the Model with a stamp computed from the class diagram part alone. We return to this in Section 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 (that is, 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.

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

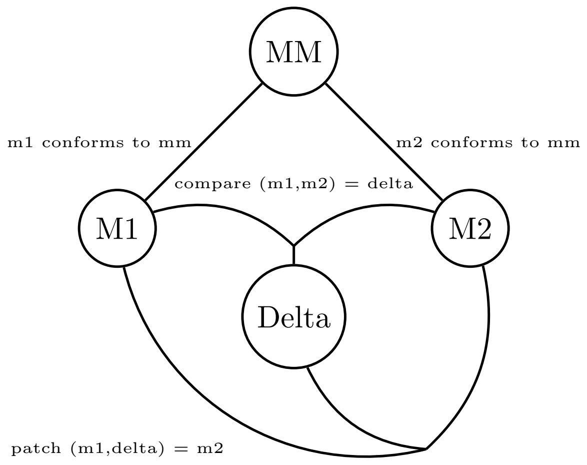


Figure 2: Megamodel derived from [17]. (Notation: lowercase `model` is instance of upper-case `Model`)

Requirements that builders must satisfy. Conditions that the developer of a builder must 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.

- 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 than the file. This ensures that an out-of-date generated file is not used.
- The builder must either fail, or produce a build unit which is internally consistent. This is Assumption 4.1 in [4], and enables the soundness result.
- Enabling the optimality result, [4] 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. We omit details for space reasons.

3 EXAMPLES

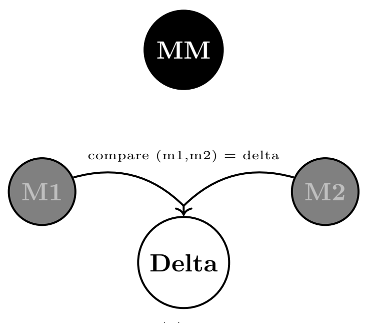
3.1 Unidirectional example

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, 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.

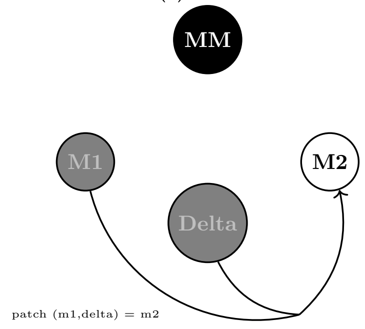
In [17] the idea is that consistency is restored after every change, and so the scenarios considered are those that start from a consistent set of models, just one of which is then changed. Even then, we must note that there may be a choice of how to restore consistency. If `m1` is changed, we may either 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 Section 1, this is a fact about this example: not every metamodel will be always-authoritative). Di Rocco et al.'s assumption 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.

The [17] megamodel specifies that the models should conform to the metamodel, but it provides no operations to ensure this. The orientation models in Figure 3 reflect a choice that, therefore, this consistency will be ensured and checked externally.



(a)



(b)

Figure 3: Orientation models (grey=authoritative, black=always-authoritative)

3.2 Bidirectional example

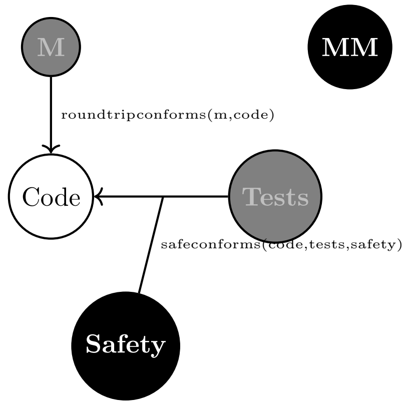
Recall from Section 1 that Figure 1 illustrates a megamodel adapted from [19]. Here we see a design model that needs to conform to a metamodel; some code that must be consistent with the model via a standard round-tripping relation; and a more interesting ternary relation between the code, a test suite 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.

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 Figure 4(b), if we want up-to-date tests, we must restore `roundtripconforms` first, then `safeconforms`.

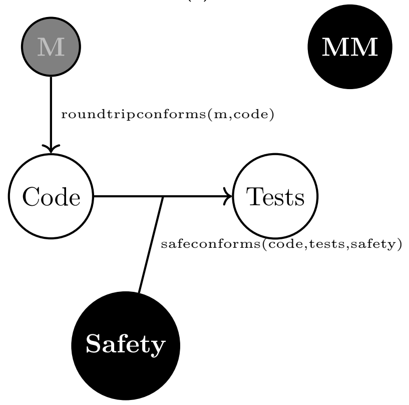
Figure 4(a) represents the situation discussed in Section 1. Our framework allows us to encapsulate the reconciliation of different consistency relations impacting the same model in the builder of each model (here Code). The orientation model records the *contracts* of the builders. Note that such builders 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: 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 model every time the safety model changes in any respect. We can achieve this using a custom stamp: see below.

Flexibility. Conventionally, e.g. in [19], 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, thereby breaking consistency with the code and the tests, it may not be sensible to update the tests immediately (especially if, say,



(a)



(b)

Figure 4: Orientation models (grey=authoritative, black=always-authoritative)

three more changes will follow in quick succession). What we should be able to ensure is that someone who is relying on the tests is able to ensure that they are indeed using an up-to-date version of the tests. We will therefore use a *demand-driven* approach. Rather than pushing the changes from model to tests, as the approach in [17] does, we will say: the person who wants to use the tests will submit a build request for the tests. This will in turn submit a build request for any model on which the tests depend, before using those updated models to recheck the consistency relation on the tests. The *pluto* algorithm determines which builders actually need to be run in order to satisfy the build request.

4 CUSTOM STAMPERS AND BIDIRECTIONALITY

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 [4], 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 needed by the builder: instances are equivalent iff they have the same stamp.

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 re-running 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 consistency 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.

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 [18]; e.g. $m \cong^R_F m'$ iff for *any* model n, the result of using 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'[7]. In many (but not all) natural cases, the equivalence class of m modulo $\cong^R_F$ is easily reified as the information from m that R looks at. 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]

An interesting challenge in the context of a particular transformation language (related to slicing) would be: given a transformation, automatically generate stampers. 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 re-running 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.

[7] F in $\cong^R_F$ is for Forward: notation of [18]

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

5 FORMALISATION

In this section we sketch[9] how a simple formalisation of a megamodel can be interpreted in the *pluto* formalism and augmented by a variable *orientation model*. We give a skeleton for builders of models, and hence derive soundness and optimality results for megamodel consistency restoration.

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 5.1. A megamodel $\mathcal{M}$ comprises:

- a set Node of *model sets*. That is, an $N \in \mathsf{Node}$ is itself a set of models, which may be interpreted as a type.
- a possibly empty AAuth $\subseteq$ Node, the nodes that are *always-authoritative.*
- a set Edge of *model relationships*. That is, a (hyper)edge $E \in \mathsf{Edge}$ connecting distinct nodes $N_1, \ldots N_k$ is a subset of $N_1 \times \cdots \times N_k$.

(More formally, we have a hypergraph, and a valuation of its nodes and edges into model sets and relations on them. However, since the valuation is fixed, we elide the distinction and trust no confusion results.)

An *instance* of a megamodel is a collection of one model n in each N in Node. The instance is *consistent* if all the relationships are satisfied, i.e. whenever $n_{i_1} \in N_{i_1}, \ldots, n_{i_k} \in N_{i_k}$ are models in this instance and $E \subseteq N_{i_1} \times \cdots \times N_{i_k} \in \mathsf{Edge}$ we have $(n_{i_1}, \ldots n_{i_k}) \in E$.

Notice that this 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!")

E.g. if bidirectional transformation R (the currently programmed transformation from a set $\mathcal{R}$ 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 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 $\mathcal{R}$, 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.

Typically, there will be some nodes in a megamodel which it is helpful to include, but never appropriate to change automatically. (For example, 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 5.2. An *orientation model* over a megamodel $\mathcal{M} = (\mathsf{Nodes}, \mathsf{AAuth}, \mathsf{Edges})$ comprises the nodes of the megamodel and a subset of the edges. It designates a subset of its nodes as authoritative, and it orients each of its edges, i.e. designates one node as target. It is *well-formed* if it (strictly, the underlying hypergraph) is acyclic and no target node is authoritative.

Given this setting we equip our megamodel with *pluto* builders, and use its algorithm to restore consistency. In the formalism, we are specifying the build units the builders produce, and their effect on the filesystem. In the implementation, if the builder (e.g.) calls the framework-provided `requireBuild` method, this (as well as guiding the *pluto* algorithm) writes a breq entry to the build unit.

Definition 5.3. A pluto *build system* for a megamodel $\mathcal{M}$ comprises a builder for each node M in the megamodel that is not always-authoritative. Given an instance of $\mathcal{M}$ and a well-formed orientation model O for $\mathcal{M}$, the build method of the M-builder must (unless it fails):

(1) freq the orientation model O (adding an freq entry for its path and the appropriate stamp (see below) to the build unit).
(2) Determine, from O, the set $\mathcal{E}$ of directed (hyper)edges having M as target; let $\mathcal{N}$ be the set of nodes that are sources of these edges.
(3) breq the N-builder for each $N \in \mathcal{N}$ (such that N is not always-authoritative) (adding breq entries).
(4) freq all the models which are the values of nodes in $\mathcal{N}$ in the current instance (adding freq entries).
(5) Calculate, and write, a new version of model $m \in M$ *that makes all the relationships in* $\mathcal{E}$ *hold.*
(6) gen m, i.e. record that m has been (re)generated.
(7) Return 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.

[9]more detail, omitted for space reasons, is in [21]

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 [4]'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.

(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 Figure 4(a).) We are using *pluto*'s dynamic dependency capabilities here: the builder's requirements depend on the current contents of the orientation model.

(3) 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 3.1, 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.

(4) If there is more than one incoming edge (e.g. Code-Builder according to Figure 4(a)), or if the megamodel is not associated with the means to restore consistency along its edges, then 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 [19] 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.

Soundness. The builders in such a build system will automatically obey the requirements we placed on builders; in particular, the sequence of requirements changes only if the orientation model changes (hence Assumption 4.2 of [4] holds). These builders are now used with the standard *pluto* build algorithm, and we get:

Theorem 5.4 (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.*

Sketch. Since the orientation model is always acyclic and each builder ensures that its model is consistent with the models that have arrows into it, induction on the length of the longest directed path in the orientation model, which is finite by assumption, together with Theorem 5.3 of [4]. □

Note that this is a stronger result than Theorem 5.3 of [4] 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 [4]:

Theorem 5.5 (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 of file changes that the stamps indicate are significant, and then only when they are 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 Figure 4(b), 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. 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.

Flexibility. 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.

6 DISCUSSION

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, another that does not. As we have seen the system automatically maintains soundness even if the orientation model changes.

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.

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 [10] 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.

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. [17, 19]. We think that, for MDD, it is an advance[10], but note that it is still possible that a rebuild of one model forces an update to another (e.g. Test-Builder in Figure 4(b) may cause Code to be rebuilt, if it is currently inconsistent with the Model). This relates to:

[10] even though S. Erdweg says pluto has recently been looking at the opposite strategy for conventional software build

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 [7] 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 [6, 17]. [16] argues for (b); in an MDD context, [11] makes the point that engineers want to work independently on copies of the same model and then need good tool support for reintegration. 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. Making the right choice in a given setting will inevitably require skill and experience.

7 CONCLUSIONS AND FUTURE WORK

We have proposed an approach to sound, optimal and flexible megamodel-based building, extending the work of Erdweg et al. [4] to tackle the problem of Di Rocco et al. [17], and to address some of the challenges raised by Stevens [19].

A specialised open-source framework for building from megamodels[11], on top of *pluto*, is work in progress: manually implementing appropriate builders, as described, is routine, but we would further like to incorporate: wrappers to let builders invoke existing model transformation engines; automatic generation of builders from a megamodel expressed in an appropriate language such as MegaL/Forge [17]; connections with further megamodelling work such as [3, 9]; 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 [6].

ACKNOWLEDGMENTS

I thank the reviewers, and the Bx'18 audience[12], especially Sebastian Erdweg and Jeremy Gibbons, for helpful discussion.

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

[12] for a related talk https://youtu.be/Pp1BsQyHoMs with no accompanying paper

REFERENCES

[1] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. 2014. MDEForge: an Extensible Web-Based Modeling Platform. In *CloudMDE*, Vol. 1242. CEUR Workshop Proceedings, 10. http://ceur-ws.org/Vol-1242/paper10.pdf

[2] Jean Bézivin, Frédéric Jouault, and Pierre Valduriez. 2004. On the need for megamodels. In *Proc. OOPLSA/GPCE workshop: Best Practices for Model-Driven Software Development.*

[3] Zinovy Diskin, Sahar Kokaly, and Tom Maibaum. 2013. Mapping-Aware Megamodeling: Design Patterns and Laws. In *SLE (Lecture Notes in Computer Science)*, Vol. 8225. Springer, 322–343.

[4] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. In *OOPSLA*. ACM, 89–106.

[5] Sebastian Erdweg and Klaus Ostermann. 2017. A Module-System Discipline for Model-Driven Software Development. *Programming Journal* 1, 2 (2017), 9. https://doi.org/10.22152/programming-journal.org/2017/1/9

[6] Jokin Garcia. 2018. Continuous Model-driven Engineering. https://modeling-languages.com/continuous-model-driven-engineering/.

[7] Jeff Gray and Bernhard Rumpe. 2017. The importance of flow in software development. *Software and System Modeling* 16, 4 (2017), 927–928.

[8] Object Management Group. 2014. Model Driven Architecture (MDA) MDA Guide rev. 2.0. http://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[9] Wolfgang Kling, Frédéric Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. 2011. MoScript: A DSL for Querying and Manipulating Model Repositories. In *SLE (Lecture Notes in Computer Science)*, Vol. 6940. Springer, 180–200.

[10] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2008. A framework for composing modular and interoperable model management tasks. In *In Model-Driven Tool and Process Integration Workshop.* 79–90.

[11] Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson. 2012. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In *MoDELS (Lecture Notes in Computer Science)*, Vol. 7590. Springer, 352–367.

[12] Epperly T Kumfert G. 2002. *Software in the DOE: The Hidden Overhead of "The Build"*. Technical Report UCRL-ID-147343. Lawrence Livermore National Laboratory, CA, USA.

[13] Ralf Lämmel. 2017. Relationship Maintenance in Software Language Repositories. *The Art, Science, and Engineering of Programming Journal* 1 (2017), 27. Issue 1.

[14] Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An empirical study of build maintenance effort. In *ICSE*. ACM, 141–150.

[15] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive make considered harmful: build systems at scale. In *Haskell*. ACM, 170–181.

[16] Bashar Nuseibeh, Steve M. Easterbrook, and Alessandra Russo. 2001. Making inconsistency respectable in software development. *Journal of Systems and Software* 58, 2 (2001), 171–180. https://doi.org/10.1016/S0164-1212(01)00036-X

[17] Juri Di Rocco, Davide Di Ruscio, Marcel Heinz, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. 2017. Consistency recovery in interactive modeling. In *EXE at MODELS*. 6. http://www.modelexecution.org/media/EXE2017/papers/EXE_2017_paper_6.pdf

[18] Perdita Stevens. 2012. Observations relating to the equivalences induced on model sets by bidirectional transformations. *EC-EASST* 049 (2012), 16.

[19] Perdita Stevens. 2017. Bidirectional transformations in the large. In *MODELS*. IEEE, 1–17.

[20] Perdita Stevens. 2018. Is Bidirectionality Important?. In *Modelling Foundations and Applications - 14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Toulouse, France, June 26-28, 2018, Proceedings (Lecture Notes in Computer Science)*, Alfonso Pierantonio and Salvador Trujillo (Eds.), Vol. 10890. Springer, 1–11. https://doi.org/10.1007/978-3-319-92997-2_1

[21] Perdita Stevens. 2018. Supplemental note to "Towards Sound, Optimal and Flexible Building from Megamodels". (July 2018). Available from http://homepages.inf.ed.ac.uk/perdita/MegamodelBuild.

[22] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. 2014. The State of Practice in Model-Driven Engineering. *IEEE Software* 31, 3 (2014), 79–85.