Supplemental note to "Towards Sound, Optimal and Flexible Building from Megamodels"

July 13, 2018

Perdita Stevens

ACM Reference Format:
Perdita Stevens. 2018. Supplemental note to "Towards Sound, Optimal and Flexible Building from Megamodels": July 13, 2018. In *Proceedings of NONE (NO NOT AN ACM CONFERENCE).* ACM, New York, NY, USA, 4 pages.

This note gives more detail than would fit into the MODELS page limit of how the approach described in the MODELS paper works with the pluto algorithm described in [1], especially as regards formalisation – that is, it expands Section 5 of the MODELS paper. It is still not completely self-contained (that would require reproducing most of [1]) but I hope it will satisfy people wanting a bit more detail and explanation. Eventually, it should form the foundation of a journal version of the paper. For now, however, it is written hastily, and in chatty style. It may exist in several version: see date above. Comments are welcome.

Important remark: although I have only expanded Section 5 of the MODELS paper here, **the rest of the MODELS paper is relevant!** Especially, it is important to read Section 2.5, Summary of pluto, and Section 4, Custom stampers and bidirectionality, of the MODELS paper, before reading further here.

1 FORMALISATION

In this section we show 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 1.1. A megamodel-skeleton $\mathcal{H}$ comprises:

- a set Node, called *nodes*
- a possibly empty AAuth $\subseteq$ Node, the nodes that are *always-authoritative*
- a set Edge of (directed) *(hyper)edges*, together with a function
$$\mathsf{nodes} : \mathsf{Edge} \to \mathsf{Node}^*$$
such that for every $E \in$ 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 1.2. A megamodel $\mathcal{M}$ over a megamodel-skeleton $\mathcal{H} = (\mathsf{Node}, \mathsf{AAuth}, \mathsf{Edge})$ comprises:

- a valuation v_{Node} of nodes, giving for each $N \in$ Node a set $v_{\mathsf{Node}}(N)$ of *models*.
- a valuation v_{Edge} of edges, giving for each $E \in$ 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. In the MODELS paper 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. I tried to add a few words in the CRC to head off such confusion, but here let us be more pendantic.

Definition 1.3. An *instance* of a megamodel $\mathcal{M} = (v_{\mathsf{Node}}, v_{\mathsf{Edge}})$ over a megamodel-skeleton $\mathcal{H} = (\mathsf{Node}, \mathsf{AAuth}, \mathsf{Edge})$ 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 $(n_{i_1}, \ldots n_{i_k}) \in v_{\mathsf{Edge}}(E)$.

Notice that this encompasses two MDD situations:

(1) all the nodes are models, and the transformations between them are encoded as edges;

NO NOT AN ACM CONFERENCE, NEVER,

(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 1.4. An *orientation model* $\mathcal{O}$ over a megamodel-skeleton $\mathcal{H} = (\mathsf{Node}, \mathsf{AAuth}, \mathsf{Edge})$ 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 OEdge $\subseteq$ 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,

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

Given this setting we can equip our megamodel with *pluto* builders and use its algorithm to restore consistency.

To explain this in more detail than fitted into the MODELS paper, let us introduce a bit more detail about *pluto*. Figure 1 is a screenshot of Figure 5 of [1]). (In the eventual journal version this will obviously have to be latexed, but I'm writing this in limited time between notification and CRC date, in order to respond to reviews!)

P	::=<path>	path to a file
Ω	::=$P \to$ <file>$_\perp$	file system
FS	::=$P \times \Omega \to S$	file stamper
S	::=stamp <value> FS	file stamp
R	::=freq $P\ S$ | breq $B\ I$	file or build requirement
G	::=gen $P\ S$	provided file with stamp
I	::=<value>	builder input
B	::=$\{build\colon I \times \Omega \to U \times \Omega,\ path\colon I \to P\}$	builder
U	::=$\{builder\colon B,\ input\colon I,\ reqs\colon \overline{R},\ gens\colon \overline{G}\}$	build unit

Figure 5. Syntax of build units and requirements

Figure 1: Screenshot from [1]

We are about to define builders for our models. As explained in Section 2.5 of the MODELS paper, 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:

Recall that in our application, the input type is unit. So to define a builder, we have to define

- a build function (implemented as a method of the builder), taking a file system to a build unit and a new (i.e. possibly modified) file system;
- a path function whose domain is unit, i.e. a path to a file (or undefined) – this is the path where the build unit produced is stored. Importantly, Undefined corresponds to the build having failed; that is, the builder only produces a path to an actual file where a build unit is stored, if the build succeeded nd hence the build unit may be relied upon.

The path is only interesting in that pluto's results depend on builders not overwriting one another's build units ("stomping on one another's files" as I wrote in the MODELS paper); we will say no more of it.

About builders. 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.

The first element, labelled *builder*, of the build unit is always the builder that wrote this build unit.

The second element, *input*, is always unit in our case: boring.

The third element, *reqs* is a list of build and file requirements: this is the interesting part. Definition 1.5 explains how this is determined, and the restrictions on the stamps that must be used on file requirements.

The fourth element, *gens* is a list of stamped provided files, but in our setting each builder only generates one file, viz. the

model it is responsible for. As remarked in paragraph Stamps of Section 2.5 of the MODELS paper, we will assume (for technical reasons) that these 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 will do fine).

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 doesn't 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. See https://github.com/PerditaStevens/megamodelbuild for examples. E.g. if the builder calls the framework-provided requuireBuild method, this (as well as guiding the pluto algorithm) writes a breq entry to the build unit.

Definition 1.5. 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. (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 $\mathcal{E}$ 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). (That is, append these breq entries to the *reqs* list.)
(4) freq all the models which are the values of nodes in $\mathcal{N}$ 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* $\mathcal{E}$ *hold.*
(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 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.

Remark [a new one, not in the MODELS paper]. What we just did was to explain how to get a set of pluto builders from a megamodel and an orientation model over its skeleton. We did this by explaining what the builder would do in the presence of a file system recording an instance of the megamodel and recording the orientation model. The key observation is that provided the builder of each model is indeed designed following the above instructions, then these builders will satisfy the assumptions on builders that are used in [1]'s proofs, so these will transfer. By examining what the builders do, specifically, to restore consistency along certain edges, we get a slightly stronger result in fact.

Remarks [that were already in the MODELS paper].

(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 [1]'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 **??**(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 **??**, 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. CodeBuilder according to Figure **??**(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 [2] 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 [1] holds). These builders are now used with the standard *pluto* build algorithm, and we get:

Theorem 1.6 (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 [1]. □

Note that this is a stronger result than Theorem 5.3 of [1] 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 [1]:

Theorem 1.7 (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 **??**(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.

REFERENCES

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

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