SHORT-PAPER

A Virtual Global Monorepo of Immutable Linked Data

JOERI EXELMANS, University of Antwerp, Antwerpen, VAN, Belgium

JAKOB PIETRON, Ulm University, Ulm, Baden-Wurttemberg, Germany

ALEXANDER RASCHKE, Ulm University, Ulm, Baden-Wurttemberg, Germany

HANS L VANGHELUWE, University of Antwerp, Antwerpen, VAN, Belgium

Open Access Support provided by:

Ulm University

University of Antwerp

Published: 22 September 2024

Citation in BibTeX format

MODELS Companion '24: ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems
September 22 - 27, 2024
Linz, Austria

Conference Sponsors:
SIGSOFT

https://doi.org/10.1145/3652620.3688222
ISBN: 9798400706226

A Virtual Global Monorepo of Immutable Linked Data

Joeri Exelmans
Hans Vangheluwe
<firstname.lastname>@uantwerpen.be
University of Antwerp
Flanders Make
Antwerp, Belgium

Jakob Pietron
Alexander Raschke
<firstname.lastname>@uni-ulm.de
Ulm University
Ulm, Germany

ABSTRACT

The data layer of today's model management solutions often is either centralized or Git-based. We point out a number of limitations of current approaches, such as poor replicability, manually configured access control, centralization, hard-coded 'meta-data', and inflexible encodings. We argue for a set of fundamental features / restrictions (most importantly immutability and capability-based security) for decentralized model management systems to adapt, to solve these problems at their root. We distinguish a fundamental core from non-fundamental applications (such as versioning), that can be built on top.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**; **Model-driven software engineering**; **Software configuration management and version control systems**; **Software libraries and repositories**.

KEYWORDS

model management, capability-based security, versioning

ACM Reference Format:
Joeri Exelmans, Hans Vangheluwe, Jakob Pietron, and Alexander Raschke. 2024. A Virtual Global Monorepo of Immutable Linked Data. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3652620.3688222

1 INTRODUCTION

In model-driven engineering, model management concerns handling the various heterogeneous models used throughout the development process, by different stakeholders. The development process may also be explicitly modeled. Models may refer to each other, and can be (in)consistent with one another, over time. Research topics include collaboration, inconsistency detection, automated synchronization, and industrial applications thereof.

In this paper, we take a step back, and look at model management as the challenge of dealing with inter-dependent, evolving and composable data, that needs to be access-controlled. These challenges are not unique to model management, and we will often draw parallels with solutions from the software community.

The rest of this paper is structured as follows: In Section 2, we describe problems and limitations of how code, data and models are typically stored, shared and reused today. In Section 3, we propose a solution to these problems. In Section 4, we point out some challenges of our solution. In Section 5 we discuss related work. In Section 6 we conclude this paper.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688222

2 PROBLEMS

2.1 Replicability - Dependency Management

Software and data appear to rot over time: a package that was developed five years ago, may fail to build or run on your system today. Or you can no longer open an old model, because you created it with an old, incompatible version of a modeling environment. These problems can be reduced entirely to poor management of *dependencies*, i.e., the context in which your program or data was supposed to run or used, no longer exists. Dependencies can take the form of software and hardware. An example of the latter: if you write x86 assembly, you have a dependency on that instruction set architecture. Hardware dependencies can be avoided by using high-level programming / modeling languages, and can be remedied by hardware emulation (e.g., QEMU [2]) or binary-to-binary translation (e.g, Apple Rosetta).

In this paper, we focus on software dependencies. A common problem is that software dependencies are implicit: importing a library that is globally installed on *your* system, won't necessarily work on another system. Nearly as bad, are imprecise dependencies: a Python program may be 'compatible with Python >= 3.8': regression bugs and the removal of deprecated features in newer versions may prove this statement wrong. Another source of problems is when dependencies don't *compose*. An example of non-composable (conflicting) dependencies, is when two different versions of the Python interpreter (e.g., `3.7` and `3.8`) try to install their binary in the same global location (e.g., `/usr/bin/python3` on Linux). The conflict could be avoided by using different names, and both versions could live side-by-side.

Deterministic package managers such as Nix [7] and Guix [5] force explicit dependencies by *sandboxing* builds and executions, and guarantee composability by prefixing all artifacts by hash-based IDs. This approach was demonstrated to work extremely well for the replicability of build environments [11]. Containers (e.g., Docker)

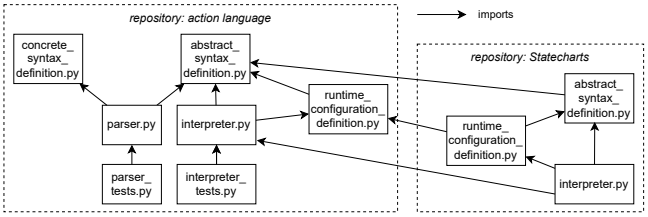

Figure 1: Dependencies within and between repositories

also perform sandboxing to eliminate missing dependencies, but they are not composable[1].

The sandboxing follows the "principle of least privilege". It is remarkable that this principle has become accepted as an important security practice in privacy-critical systems, but not to dependency management. Perhaps because the perceived benefit is smaller: replicability versus "not getting hacked".

We are convinced these insights should shape the future of model management. We advocate that dependencies must:

- point to *precise* versions of data and code / models, in a formal, machine-readable fashion.
- be *complete*: any reference (e.g., import statement) to any external artifact, implies a dependency.
- *compose*: depending simultaneously on two different versions of an artifact, should not cause a conflict.

2.2 Coarse-grained Dependencies

Software packages are typically contained in 'repositories' (e.g., Git). Package managers allow specification of dependencies *between* repositories, but dependencies typically exist *within* the repository as well (e.g., import statements between modules). A consequence is that if a repository depends on a single file of another repository, the entire repository needs to be depended upon. As an example, Figure 1 shows two repositories: an action language, and a Statechart language, and the import-dependencies between their modules. The Statechart language's abstract syntax imports the action language's abstract syntax, because Statechart models contain action language expressions and statements. Further, the Statecharts interpreter needs the action language interpreter in order to evaluate and execute these. However, the action language repository contains a number of modules that aren't needed by the Statecharts language, such as the tests. These are unnecessarily fetched. We state:

- The notion of a 'repository' is an unnecessary, artificial box around a set of artifacts. What matters, are the already existing dependencies between the artifacts themselves.

2.3 Coarse-grained, Inconsistent Access Control

Typically, read and/or write access permissions are (manually) granted to an *entire* repository. In our example (Figure 1), the Statecharts interpreter team must have access to the action language interpreter, but not necessarily to the parser. The maintainer of the action language repository needs to manually configure this, and possibly 'split off' the interpreter into a separate repository.

In other words, a trade-off needs to be made: many small repositories with finer-grained control, but more administration, or fewer larger repositories with less administration, but less control. Instead, we propose that:

- Access control should be fine-grained, and automatically consistent with dependencies: having access to some artifact, implies having access to its transitive closure of dependencies.

2.4 Centralization

Access control is almost always implemented via a central authority: e.g., GitHub / SVN / PLM systems / ...User accounts need to be created on each platform, and explicitly granted read/write-access. This involves manual administration, an opportunity for mistakes, and makes it more difficult to migrate repositories between hosting platforms, because access control rules are specific to the platform.

Centralized access control also forms a technical obstacle to cross-organization collaboration: when all data is tied to a single server deployment, collaboration with an external (untrusted) organization requires granting it access to an internal asset. Administrators are bothered with creating and maintaining user accounts for external users. Other typically centralized features, such as issue tracking and pull requests, further lead to vendor lock-in. It doesn't have to be this way: distributed issue tracking systems, such as "Bugs Everywhere" [4], store issues in the repository itself.

A system that has been designed from the start to be decentralized, can always be hosted on a single central server. The other way around, distributing a system that assumes centralized deployment, can be incredibly hard to do correctly. Therefore, we advocate:

- Access control should be fundamental and decentralized.
- All (meta-)data such as versioning information, issue tracking, and pull requests should be completely decentralized.

2.5 Fixed Data Model and Encoding

Versioning systems usually make an explicit and hard-coded distinction between meta-data (e.g., versioning information), and the data (e.g., what is versioned). For instance, in Git, every *commit* points to a snapshot of a file-system *tree*, and to a set of parent-commits. The user cannot define her own (meta-)data types (e.g., an 'issue'), or express dependencies between them. Meta-data and data have a fixed 'shape': e.g., the data that is versioned must be tree-shaped, not a DAG. Encodings are also fixed: e.g., commits are textually encoded.

We are convinced that the distinction between data and meta-data is unnecessary, and that custom, arbitrary encodings should be allowed. A model itself could be a data type (*not a file* containing a model). A model contains, and therefore depends on, a set of model elements, which also have their own data types. Models could be composed: a model becomes a model element in the context of another model (e.g., hierarchical formalisms such as Statecharts). The same model could be reused in different contexts, and thus, dependencies form a DAG in the general case (not a tree).

Even if conceptually, all data can be joined together in a single DAG of dependencies, encoding all of it in a uniform graph-format

[1]The fundamental difference is that containers flatten the closure of all dependencies into a single image, similar to a virtual machine. *Layering* of images can reduce data duplication to some extend, but only allows images to branch (by operational build scripts, mutating the image arbitrarily), not merge: two containers cannot form the basis of a third container.

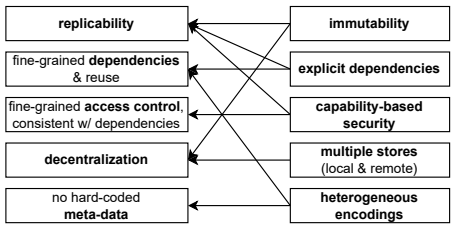

Figure 2: Relations between requirements and solution

(e.g., XMI) would be highly inefficient: for a project, we transformed a 10 MB CSV-file to OWL, which resulted in a 1 GB file size. The effect would be even more dramatic when starting from a highly-optimized encoding such as Apache Parquet [1]. Thus, data is best kept in its original (often already optimized) encoding.

In summary, we advocate:

- Fundamentally, no distinction between data and meta-data (e.g., versioning information) needs to be made.
- The system should support arbitrary encodings.

3 PROPOSED SOLUTION

Our proposed solution starts with a minimal foundation, which is mainly a set of restrictions on data stores. We then show how versioning and typing information can be added.

3.1 Foundation

3.1.1 Immutability of Data. As a bare minimum requirement to solve the 'Replicability' problem, in a distributed setting, it must be possible to refer to a piece of data by an ID, and *always* point to the same data. If we drop this requirement, all bets are off.

Any object (= piece of data), small or large, that can be shared, and thus become a dependency of another object, is assigned a globally unique content-based ID (i.e., hash of its contents and dependencies). Objects are immutable: making a change to an object, results in a new object, with a new ID (copy-on-write). Examples of objects are: a model, a model element, executable code, a version of something, a directory, a traceability link, …

Copy-on-write should not be confused with *versioning*. With copy-on-write, the new, updated object has no explicit relation with the old object. We'll see later how versioning information can be added.

3.1.2 Explicit Dependencies. Every object explicitly states its dependencies. When an object's dependency 'changes', the object itself must also change (and receive a new ID) in order to 'update' the dependency. Because of immutability, objects and their dependencies form a DAG, by construction. We'll discuss how to deal with cyclic references in Section 4.1.

3.1.3 Capability-Based Security. Since object IDs are hashes, they cannot be guessed / forged, and the ID of an object *is* the read-access right to the object, following the principle of Capability Based Security (CBS) [10]. Objects can be shared by sharing their IDs (similar to 'share-by-link'). Having read-access to an object, implies having read-access to its dependencies (but not the other way). Write-access does not need to be restricted: immutability ensures nothing can ever be overwritten. Every user or program can create objects, compute their IDs and share them.

To enforce this, a data store only needs to implement two functions:

```
get(hash) -> object
put(object) -> hash
```

Note that there is no operation to list all objects in the store (would violate CBS). Given an object, its dependencies can be retrieved:

```
getDependencies(object) -> set<hash>
```

3.1.4 Multiple Stores. Conceptually, our solution is a single large mono-repository. This makes Git's traditional usage model of keeping a local clone of everything, infeasible because of scalability and privacy reasons. One option would be to use a distributed, location-transparent content-addressed store like IPFS [3], which transparently looks up and replicates data over a dynamic network via a distributed hash table. However, IPFS is not private: network nodes publicly announce the IDs of the objects they serve. Lookup times can also be long. We'd rather sacrifice location transparency to gain privacy and improved latency.

We foresee that objects can reside in multiple stores at the same time. For instance, a local store, for all the projects the user is/has been working on, and a remote store, for all the projects of an organization. Synchronization between stores is trivial, because of immutability: conflicts cannot occur – a store either has an object or not.

3.1.5 Heterogeneous Encodings. In model management, a great deal of data heterogeneity needs to be dealt with: models of different types, spreadsheets, rich markup text documents, code (typically already in Git-repositories), traceability links, execution traces of experiments and workflows, …

We may want to expose the internal structure of data artifacts (e.g., the cells of a spreadsheet, the components of an electric circuit model) as individual objects, to allow the creation of fine-grained traceability links. (A traceability link would be a separate object depending on the elements it relates.) As mentioned in Section 2.5, re-encoding all data in a uniform graph format does not scale. The only workable solution is to keep the data in its original encoding, and to write an adapter that generates graph data *lazily*: only those parts of the graph / DAG that are navigated, are generated.

An adapter must implement the `getDependencies` function for all the objects it generates. Further, it may implement additional *transformations*: for instance, an adapter for unordered sets may implement `add(key)` and `remove(key)`-transformations, each of which return a new (updated) set.

Adapters integrate especially well with data that is already immutable / content-addressed: An adapter could be written that exposes a Git-repository, its commits, directories and files as a graph structure (which it *already* is…).

3.2 Applications

Given our proposed foundation of a store of immutable inter-dependent, arbitrarily-encoded, reference-counted objects, features such as versioning and typing, become *applications* that can be built on top.

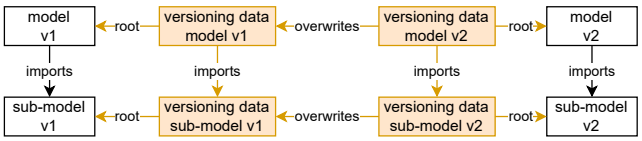

Figure 3: Versioning data is just data. All edges are dependency-links.

3.2.1 Versioning. We do not fundamentally enforce one particular approach to versioning, in fact, we do not enforce versioning at all: data can exist without having a history. Versioning information, too, is just data, and existing state-based (e.g., Git) or operation-based (e.g., our earlier work [8]) versioning approaches can be integrated with.

Obviously, for model versioning, we need to move beyond Git. Breaking through the boundaries of 'repositories', and making models and their history truly composable, without need for package managers. Versioning information should exist at multiple levels in the graph: suppose a model imports a sub-model. The sub-model can be versioned (and is imported by several models). The model itself is also versioned. Figure 3 shows multi-level versioning information. When merging two branches of the model, their sub-models are merged too, similar to Git 'submodules'.

We can take this to the extreme, and create versioning information for every model element and value in a model. The history of a single attribute (e.g., the price of a car) can then be isolated and shared (without sharing any other information of the containing model). For attributes that are computed (rather than manually edited), e.g., the mass of a car is the sum its parts' masses, we can even record the read-dependencies: what were the inputs of the computation? This way, if the mass of a part changes, the total mass needs to be re-computed. This can be a foundation for consistency checking. We are currently experimenting with this approach in a live modeling demo, where conflicts between edit operations and an ongoing execution by an interpreter are detected [9].

3.2.2 Typing. Typing information allows for answering the question "what can I do with my data?". This information can be implicit (derived from context), or explicit. As an example of implicit typing, consider the following fragment of C code:

```
struct Person { int age; };
int getAge(struct Person *p) {
    return p->age;
}
```

At run-time, the function `getAge` cannot verify whether it actually receives a pointer to a `Person`: it could be a null-pointer or a pointer to a different kind of data structure. However, the C programmer doesn't care: he is confident not to make mistakes.

Explicit typing information exists at design-time in statically typed languages, and at run-time in dynamically typed languages. Explicit typing information makes data self-describing, enabling features such as conformance checking and auto-completion.

We argue that the data itself is independent of its (optional) typing information: implicitly typed data can still be used (in the right context), as in the C example. Types are also just data. Typing information is a morphism (a 'mapping', a set of links) between instances and types. Figure 4 shows an example of an instance and

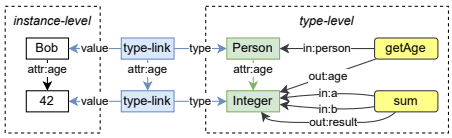

Figure 4: Typing information is just data. All edges are dependency-links.

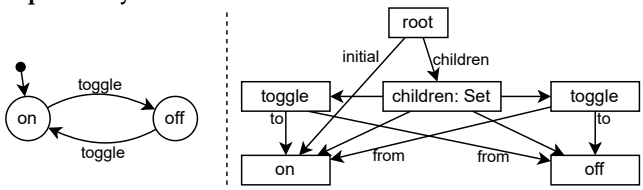

Figure 5: Left: A Statechart (with 'cycle'). Right: A DAG encoding the same information.

typing information. Transformations (yellow boxes) are declared at the type level: `getAge` takes as input a `Person`, and gives as output an `Integer`. Because of immutability, transformations must be *pure functions*. The distinction between instance-level and type-level is not fundamental: it should be possible for types to have types.

Relation between encoding and typing. Every object always has an encoding (by some adapter), and the adapter must implement all the transformations defined on the object's (implicit or explicit) type. For instance, the `sum`-transformation in Figure 4 must be implemented by the adapter that implements the `Integer` type.

Relation between typing and versioning. Typing and versioning are orthogonal features: We can version instances without having their type information. We can have typing information without having versioning information. And we can have versioned typed instances, and typed versioning information.

4 DISCUSSION

4.1 Dealing with Cyclic References

All data forms a DAG, so cyclic references within and between models cannot be represented, and need to be broken up.

WebGME [12] breaks cycles by turning every link into a node (pointing to their source and target) that becomes a child of the lowest common ancestor of the source and target. This technique comes at a price: suppose we want to execute the Statechart model in Figure 5. To select a transition, the interpreter needs to know the outgoing transitions of the current state. There is no efficient way of finding those transitions: dependency-links can only be navigated downstream, so we must search from the root for all transitions, and look for those whose source is equal to the current state. We could implement this in a function:

```
getOutgoingTransitions(root: State, state: State)
  -> set<Transition>
```

The good news is that, because of immutability, for the same inputs, this function will always produce the same output, so we can *memoize* the result – all subsequent calls return the cached value.

4.2 Scalability

Immutability implies copy-on-write, which leads to high disk space usage over time and slower writes, especially when modifying large

data structures (e.g., large dictionaries). We propose two complementary approaches to solve these issues: (1) Purely Functional Data Structures (PFDS) [13] implement copy-on-write efficiently, by sharing data between the original, and the updated object. This greatly reduces the time- and space-overhead (but still more costly than in-place update). PFDS exist for many data types, including lists, sets and dictionaries. (2) A garbage collection mechanism can clean up old objects, that have become unreachable from an explicit, mutable set of *entry point* IDs. An example of an entry point would be a directory-object containing a user's projects. A remote store could have multiple entry points (e.g., one for each user). *Reference counting* can be used, because of the acyclic nature of dependencies.

4.3 Inability to Revoke Read-Access

With centralized access control mechanisms, such as Access Control Lists, read-access can be revoked on the central server. With our approach, once a person (or program) receives read-access to an object, and consequently all of its dependencies, none of it can ever be revoked. Nevertheless, objects are immutable, so having access to an object, does not imply having access to future updates of the object. When an employee is fired, he only retains access to the data he already had access to. Finally, putting things in perspective, we argue that in the digital age, information leaks are inherently irreversible.

5 RELATED WORK

We now compare our work with existing model management and model repository solutions, by our own set of criteria, summarized in Table 1.

DesignSpace [6] is a centralized, graph-based model management solution. Versioning is done by recording CRUD-operations that mutate state. These operations are 'special' meta-data, not just data. Adapters exist to bring external data into DesignSpace, but all data is copied, and encoded graph-like.

In previous work [8], we created a decentralized, graph-based versioning system that records CRUD-operations that mutate state. Operations are 'special' (hard-coded), inter-dependent, and immutable. As with all other approaches (except ModelVerse, see below), versioning information was mandatory: no data could exist without having a 'history'.

WebGME [12] is a web-based, collaborative (meta-)modeling tool. State is an immutable DAG, operations are not recorded. Versioning information is 'special' (hard-coded). Access control is implemented centrally.

The ModelVerse [14] proposes a graph-based foundation for encoding models and their (inter-)relations (typing, transformations and processes). Everything is uniformly encoded as a graph: there is no 'special' meta-data. Versioning is ignored, and state is mutable. Access control is implemented centrally.

Finally, we mention OpenCAESAR [15], where models and their relations are translated to (OWL) ontologies, allowing powerful querying and consistency checking. Versioning is offloaded to Git, and therefore, versioning information is 'special' (and not part of the ontology).

	decentralized	no hard-coded meta-data	immutable state	immutable operations (versioning)	heterogeneous encodings
DesignSpace				~	
earlier work [8]	✓			✓	
WebGME	~		✓	N/A	
Git	✓		✓	N/A	
ModelVerse		✓		N/A	
OpenCaesar	~			N/A	
This paper	✓	✓	✓	✓	✓

Table 1: Classification of related work by our criteria.

6 CONCLUSION

In this paper, we have motivated a fundamental set of constraints that data stores for model management should enforce: immutability, explicit dependencies, access control consistent with dependencies, garbage collection of unreachable objects (via reference counting), and heterogeneous encodings. To our own surprise, we come to the conclusion that versioning is **not** fundamental. However, versioning usually implies immutability, which is fundamental.

There doesn't have to be a single implementation to dominate the world: As long as model management systems stick to these same principles, especially immutability, they can easily integrate with one another. The prime example here is Git, for which we can almost trivially create an adapter.

REFERENCES

[1] Apache parquet. https://parquet.apache.org/. Accessed: 2024-07-15.
[2] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (ATEC '05), FREENIX Track*, page 41. USA, 2005.
[3] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. *arXiv preprint arXiv:1407.3561*, 2014.
[4] Bugs everywhere. https://bugseverywhere.org/. Accessed: 2024-07-09.
[5] Ludovic Courtès. Functional Package Management with Guix. *arXiv preprint arXiv:1305.4584*, 2013.
[6] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. Designspace: an infrastructure for multi-user/multi-tool engineering. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1486–1491, 2015.
[7] Eelco Dolstra, Andres Löh, and Nicolas Pierron. NixOS: A purely functional Linux distribution. *J. Funct. Program.*, 20(5-6):577–615, 2010.
[8] Joeri Exelmans, Jakob Pietron, Alexander Raschke, Hans Vangheluwe, and Matthias Tichy. A new versioning approach for collaboration in blended modeling. *J. Comput. Lang.*, 76:101221, 2023.
[9] Joeri Exelmans, Ciprian Teodorov, Robert Heinrich, Alexander Egyed, and Hans Vangheluwe. Collaborative live modelling by language-agnostic versioning. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1-6, 2023*, pages 364–374. IEEE, 2023.
[10] Henry M. Levy. *Capability-Based Computer Systems.* Butterworth-Heinemann, USA, 1984.
[11] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. Reproducibility of Build Environments through Space and Time. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24. ACM, April 2024.
[12] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60, 2014.
[13] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.
[14] Yentl Van Tendeloo and Hans Vangheluwe. The Modelverse: A tool for Multi-Paradigm Modelling and simulation. In *Winter Simulation Conference, WSC 2017, Las Vegas, NV, USA*, pages 944–955. IEEE, 2017.
[15] David Wagner, So Young Kim-Castet, Alejandro Jimenez, Maged Elaasar, Nicolas Rouquette, and Steven Jenkins. CAESAR Model-Based Approach to Harness Design. In *2020 IEEE Aerospace Conference*, pages 1–13, 2020.