The Computational Notebook Paradigm for Multi-Paradigm Modeling

Bentley James Oakes*†, Romain Franceschini*‡, Simon Van Mierlo*†, Hans Vangheluwe*†
*University of Antwerp, Belgium
†Flanders Make vzw, Belgium
‡University of Corsica, France
{Bentley.Oakes, Simon.VanMierlo, Hans.Vangheluwe}@uantwerpen.be, R.Franceschini@univ-corse.fr

***Abstract*—*Computational notebooks* are gaining widespread acceptance as a paradigm for storage, dissemination, and reproduction of experimental results. In this paper, we define the *computational notebook paradigm* (CNP) consisting of entities and processes and discuss how the *reproducibility* of the experimental process and results is enhanced by each element. This paper also details the interactions of CNP and multi-paradigm modeling (MPM), with an aim of understanding how to support MPM within the CNP, and improve the reproducibility aspects of both the CNP and MPM.**

***Index Terms*—Computational notebooks, Multi-paradigm modeling, Paradigms, Reproducibility, Experimental Process.**

I. Introduction

To have scientific value, scientific experiments must be *repeatable* and their observations *reproducible*. This enables the wider community to explore and understand the underlying system under study. Without precisely defining the experiment's starting condition, set-up, process, and expected results, peers may be unable to reach the same conclusions.

Experimental notes can be recorded either on paper or increasingly in a digital form. The entries in these notebooks may cover a wide range of media, such as a combination of sketches, plots, text, code, or (in a digital system) audio and video. A well-structured notebook thus represents the owner's investigation and understanding of the system under study by recording the process of experimentation with the system. When these results are documented (such as in a scientific paper) and disseminated to others, the community gains the contained insight about the studied system.

Computational notebooks offer an executable way for the user to perform experimentation and dissemination activities. A key aspect is that users may write both text and code together, and results are stored directly alongside the code which produced it. When disseminated, others may also experiment further by changing parameters and (re-)executing the notebook. An example of this *documentation artefact* is a computational notebook produced for gravitational wave education [1]. This precise record of the experimental process therefore enhances the reproducibility and learning opportunities of the experiments. In fact, the visual and interactive properties of digital notebooks may be superior to typical paper-like representations when presenting scientific results [2], [3].

This paper examines how the Computational Notebook Paradigm (CNP) can be applied to Multi-Paradigm Modeling (MPM), which advocates the explicit modeling of every (relevant) aspect of a system, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s), and with explicit modeling of the development process(es) [4]. Our contributions are a discussion about the CNP enhancing reproducibility in experiments, definition of CNP entities and processes, and an examination how MPM and the CNP combine to increase reproducibility in CNP and MPM activities.

The notion of reproducibility in scientific experiments is introduced in Section II. Computational notebooks themselves are presented in Section III, along with a short example. The elements of the CNP are examined in Section IV, discussing the entities and processes of CNP and how they relate to reproducibility. The interactions between the CNP and MPM are examined in Section V, highlighting how MPM can be supported within the CNP, and how the *reproducibility* aspects of both the CNP and MPM can be improved. Finally, Section VI presents a brief conclusion and discussion of future work.

II. Reproducibility

Among the criteria allowing peers to assess the scientific quality of an experimental study is *reproducibility*, which guarantees that a particular phenomenon can be (re-)observed in a well-defined context [5]. Providing initial and experimental conditions thus allows others to corroborate or contradict the conclusions of an experiment. In experimental science, such criteria are generally ensured through data recording, as well as precise material and method documentation [3]. The primary record of a researcher is the laboratory notebook, which is then published as a scientific publication in a paper-like format (such as an electronic document).

However, as scientific fields increasingly use computational science to conduct numerical experiments, reproducibility concerns arise in the digital world [5]. For example, the 'reproducibility crisis' in biomedical research means that clinical results cannot be confirmed [3]. To avoid these issues, the results of an experiment must be reproducible to accurately document and disseminate results. This enables peers to verify, follow along with the explanations contained, and to validate the results.

Dalle proposes four levels of reproducibility for computational experiments [6], depending on the availability of the

source code, the non-deterministic aspects of the calculation, the similarity of the experimental setup, and of the results. Stodden *et al.* define five levels according to the availability of tools, results, data, and documentation [7].

Schnell proposes a terminology for classifying *repeatability*, *replicability* and *reproducibility*, as these terms can commonly be confused [3]. *Repeatability* is the ability for the *same* team with the *same* methods to obtain the same results (with some measure of precision). *Replicability* is when a *different* team can obtain the same results with the *same* methods. Finally, *reproducibility* is the ability of a *different* team with *different* methods to obtain the same results. Towards this goal, computational notebooks (CN) are envisioned to support all three of repeatability, replicability and reproducibility [8], [9]. They are regarded as an appropriate media to support *repeatability* and *replicability* [3], [10], while the documentation process of notebooks (see Section IV-B2) aims to disseminate knowledge about the system to encourage *reproducibility*.

Koop *et al.* discuss an architecture for maintaining *executable papers* [11], which are conceptually similar to computational notebooks. Their work defines seven components in executable papers: *text*, *workflows*, *data*, *results*, *source code*, *libraries*, and *visualizations*. To ensure that authors, readers, reviewers, and publishers can reproduce the paper, a *provenance*-based system is presented. Provenance is defined as "the documentation of exactly how data, experiments, and results were generated" [11]. For the above components, the full history of modifications is stored in a versioned manner with explicit dependencies, allowing the user to track exactly how each part of the experiment was conducted.

However, having the correct version of the components above, along with an exact copy of the experimental machine, may not be sufficient to reproduce an experiment [12]. This is due to the *state* of the computations, which may be partially "hidden" and not captured by the notebook or documentation. For example, a seed for a simulator's random number generator may not be initialized every time by the notebook, rendering experiments impossible to reproduce. This implies that the state of the computation, as well as all relevant initialization parameters must be made explicit in the computational notebook paradigm for reproducibility. In the software engineering domain, this capturing of initialization and state may be performed by virtualization or containerization technologies [13]. A model-based engineering approach is to formalize the *experimental frame* of models in the system [14]. Section V-A examines how MPM techniques (including frames) can enhance reproducibility by explicitly modeling the initialization, workflow, and execution of a system.

III. Computational Notebooks

This section presents computational notebooks and an example. Section IV then details the paradigm by referring to this example.

A. Computational Notebooks

The computational notebook (CN) concept first appeared in the late 1980s [9]. Similar to a read-eval-print loop (REPL) as in a shell, a CN allows code to be executed and the results to be shown directly beneath. However, in a CN, text, images, and other media can also be intertwined with the code. This can be used to add explanation to the code and results, offering a record of insights. Code and results are persisted in the notebook as well, offering a record of the activities.

Through their mirroring of the structure of a physical laboratory notebook, CNs provide a way to gather documentation, workflows, and data within the same document just as their physical counterpart does. However, CNs have added value due to their interactive execution and visualisation. Collaboration is also possible in CNs. If enabled by the underlying technology, multiple users may be experimenting and documenting in a CN at the same time, thus simultaneously disseminating the results [15], [16].

An important component of CNs is the ability to re-execute pieces of code (stored in *Cells*) at the user's direction with a *Kernel*, such that the corresponding *Output Cell* is updated. These *Kernels* are processes to execute types of code cells. Some notebooks only have one code kernel, such as Mathematica notebooks[1] and Matlab "live-scripts"[2]. Other notebooks such as the example below have a variety of heterogeneous kernels, including LaTeX, Markdown, and Python kernels.

Section III-B below demonstrates the popular Jupyter [8] implementation of a CN by reproducing (with minor edits) the text, code, and figures contained in an example notebook. Code cells are represented below as outlined boxes marked with `In:`. When executed, they produce the output which immediately follows. As an example, the first cell shows LaTeX code marked with the special tag `%%latex`, which produces the equations for the bouncing ball model in Equation 1.

B. Bouncing Ball Notebook

The aim of this notebook is to detail the experimentation results for modifying and optimizing the parameters of a bouncing ball model. This model is taken from the examples of the PyFMI library[3], which simulates models using the Functional Mock-up Interface (FMI) standard[4]. The intent of the FMI standard is to present system components as black boxes, where internal details are hidden. This is representative of industrial systems with intellectual property concerns.

The bouncing ball equations and initial parameters are:

```
In: %%latex
\begin{equation}
\frac{dh}{dt} = v; \frac{dv}{dt} = -g;\\
\mathit{when}~h < 0~\mathit{then}~v := -e * v;\\
\mathit{initial:}~e = 0.7, g = 9.81
\end{equation}
```

[1] http://www.wolfram.com/notebooks/

[2] https://www.mathworks.com/help/matlab/matlab_prog/what-is-a-live-script-or-function.html

[3] https://pypi.org/project/PyFMI/

[4] https://fmi-standard.org/

$$\begin{aligned} \frac{dh}{dt} = v; \frac{dv}{dt} = -g; \\ when\ h < 0\ then\ v := -e * v; \\ initial : e = 0.7, g = 9.81 \end{aligned} \quad (1)$$

Import PyFMI and load the Bouncing Ball FMU:

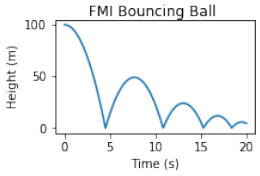
```
In: from pyfmi import load_fmu
model = load_fmu('bouncingBall.fmu')
```

Set the initial height to 100, and simulate for 20 seconds:

```
In: model.set('h',100)
result = model.simulate(final_time = 20.)
```

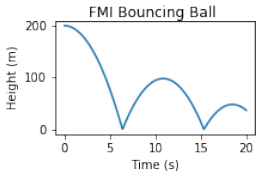
Define a plot function `plot(result)` and call it:

```
In: def plot(result):
--omitted for brevity--
plot(result)
```



1) Parameter Exploration: Reset the model, set the height to 200, and simulate. This experimentation could be repeated to explore the behaviour of the bouncing ball.

```
In: model.reset()
model.set('h',200)
result = model.simulate(final_time = 20.)
plot(result)
```



2) Parameter Optimization: The goal is to determine a height such that the ball bounces precisely halfway through the simulation. First, define a function to return the height at that time.

```
In: def halfway_height(init_h):
model.reset()
model.set('h', init_h)
result = model.simulate(final_time=20.)
halfway_index = len(result['h'])//2
return result['h'][halfway_index]
```

Then, use an optimization library to choose a height between 50 and 200 metres which minimizes that height.

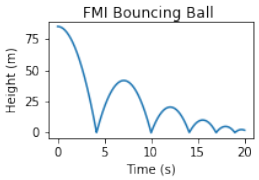
```
In [20]: import scipy
opt_result = scipy.optimize.brute(halfway_height,
 ranges=((50, 200),)
print(opt_result)
```

```
[84.91984558]
```

When the ball starts off at around 85 m high, it will bounce halfway. Plot these results to confirm.

```
In: h = 84.91984558
print("Height: " + str(halfway_height(h)))
model.reset()
model.set('h', h)
res = model.simulate(final_time=20.)
plot(res)
```

```
Height: -0.11444301199967911
```



IV. Computational Notebook Paradigm

This section investigates the computational notebook paradigm (CNP), using the bouncing ball example in Section III-B as a running example. This examination is divided into focusing on the *entities* and the *processes* involved in the paradigm. Section V then builds on this section to examine how the CNP can be used for multi-paradigm modeling to enhance reproducibility.

A. Entities

The entities involved in the notebook paradigm are represented by a class diagram in Figure 1, which has been created in the AToMPM modelling environment [17].

The core element of the computational notebook is the *Notebook* itself, which is structured as a linear sequence of *Cells*. As shown in the example notebook in Section III-B, these cells can contain a variety of typed elements, such as executable code, text, plots, or other media. The left-hand side of Figure 1 shows *User*(s) of the *Notebook*(s), where the user *accesses* (creates, reads, updates, deletes) multiple notebooks. A notebook itself may be *linkedTo* other notebooks, as in the laboratory notebook management system Prism [18]. A notebook can also produce *documentation artefacts* such as a PDF or LaTeX code. For example, the LaTeX code for Section III-B was exported (with edits for formatting and clarity) from the Jupyter notebook used for the experimentation.

Cells are divided into pairs of *InputCells* and *OutputCells*. *Cells* are typed by a particular *Language*, which captures the syntax and semantics for that *Cell*. When the *InputCell* is executed, the associated *Kernel* for that *Language* receives the code in the *InputCell* and executes it. Results are passed to a language-specific *Visualizer*, which produces an *OutputCell* to be paired with the *InputCell*.

For example, in the example notebook, most *InputCells* are *typedBy* the Python language. When code is executed, such as to plot the simulation results, the code is communicated to the Python *Kernel*. This kernel produces results, which are passed to an appropriate *Visualizer* (such as a plotter) to produce the *OutputCell*. Multiple kernels are present in that example, such as the Markdown kernel to produce text, and the LaTeX kernel to produce the equations.

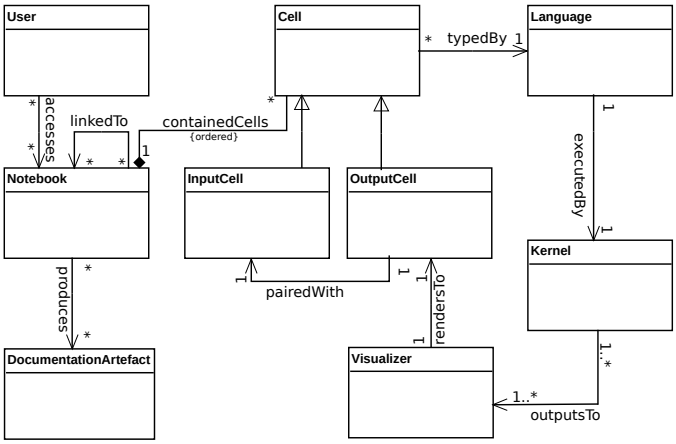


Fig. 1: Computational notebook paradigm entities.

B. Processes

As a computational notebook can represent the endless possibilities of a paper notebook, it is impossible to fully capture the involved processes at a fine-grained level or in a complete activity diagram. Instead, this section highlights the main activities in the paradigm: *experimentation* and *documentation*. Reproducibility is key for both processes. During the experimentation process, reproducibility enables users to accurately reproduce results. Additionally, when the notebook or other documentation artefact is disseminated, those who wish to reproduce the results or to experiment with the system further rely on the ability to obtain the same results, as described in Section II.

1) Experimentation: Experimentation is a crucial activity in computational notebooks, where the user applies the scientific method of hypothesizing and designing experiments, as well as the classic coding processes of writing and modifying code. In the notebook, the user repeatedly modifies and executes cells to produce desired results, based on their intentions and the properties of interest. This refinement loop continues until the produced results are as the user expects, or the user has achieved some understanding of the system. For example, Section III-B demonstrates a notebook for experimenting with the parameters of a system. The ability to modify small selections of code and immediately see the results reduces the iteration process in experimentation, allowing for fast insights on the system's properties [9]. The ability to execute multiple language kernels in one notebook also allows the user to avoid switching language tools or contexts.

2) Documentation: The documentation process of the notebook paradigm may be intertwined with the experimentation process or may be a separate process. For example, the computational notebook in Section III-B explains how to explore and optimize the height parameter for a bouncing ball simulation. Along with text and images, other media may also be present in this documentation, such as interactive diagrams [19]. This documentation may be created for different purposes, such as recording observations, thoughts and intentions about an experimental process, instructions, or to document understandings about a system.

Another portion of the documentation process is the dissemination of results as a *documentation artefact* such as a PDF or a copy of the notebook itself. This allows for different levels of reproducibility (see Section II), such as producing documentation about a system for *reproducibility*, or offering code and techniques (or the entire notebook) to replicate the results for *repeatability* and *replicability*. Dissemination also has a greater scientific goal, in that it allows others to better understand the behaviour of a system when they can modify the models involved [19]. Proper documentation and dissemination therefore allows others to re-start the *experimentation* process.

V. Computational Notebook Paradigm for Multi-Paradigm Modeling

This section examines the support needed for reproducible multi-paradigm modeling (MPM) in the computational notebook paradigm (CNP). First, Section V-A presents three improvement areas for enhancing the *reproducibility* of MPM activities in the CNP. Then, the following sections detail how the entities and processes of the CNP can be employed to support the three orthogonal components of MPM: multi-abstraction, multi-formalism, and explicit workflows [4].

A. Enhancing Reproducibility

The CNP has a high focus on reproducibility as this is required for the key processes of *experimentation* and *dissemination* (Section IV-B). As the next sections discuss, MPM also aids reproducibility with the explicit modeling of abstractions, formalisms, and workflows.

An immediate point of compatibility between MPM and the CNP is the notion of typing code cells in the CNP and executing them with a kernel. This maps directly onto conformance and executing a formalism with precise semantics in MPM. We have however identified three areas (*model frames*, *domain-specific languages*, and *explicit history and traceability*) where the reproducibility of MPM activities in the CNP could still be improved. Further development in these areas would help ensure that the properties, design, and workflow of a system can be explicitly modeled to a sufficient degree in a notebook, such that the result will be exactly reproducible when the notebook is shared or accessed at a later time.

1) Model Frames: The experimental conditions of a system must include the initial state and parameters for reproducibility. For example, the bouncing ball example in Section III-B is only reproducible with the same height and elasticity parameters for the ball. *Model frames* [14] capture this required information, A model frame consists of a) a *modeling activity* such as calibration or verification, with *inputs*, *outputs*, and a *process* description, b) a *context* with *objectives*, *assumptions*, and *constraints*, and c) zero or more sub-frames.

This frame information could be contained within a computational notebook at three levels: not-at-all, as informal code and text, or as (a) formalized decision process(es), which

ensures that the models and processes are valid at all times. As an example, consider a model frame which is (metaphorically) constructed around a code cell in the bouncing ball notebook. This model frame would only allow the simulation code inside to be executed if all necessary model parameters were set within valid ranges, such as a positive height of the ball. This model frame may also require the state of the cell's kernel to be in a pre-determined state, removing the possiblity of under-specified behaviour (see Section II). This technique of experimental condition modeling therefore raises the reproducibility of experimentation within the CNP.

2) Domain-Specific Languages (DSLs): DSLs are one approach to managing complexity in a system, such that the model's language encodes concepts from the system itself, rather than the solution domain [4]. DSLs thus aid reproducibility by focusing on the essential complexity of a system such as the parameters and processes.

Adding DSLs to the CNP requires the notion of *language engineering* where a meta-modeling language is developed in the notebook and used to type models. This dynamic creation of languages requires support in the CNP, as custom languages will need to be mapped (through a model transformation) to known semantics for simulation, debugging, or verification operations. The *conformance* relationship must also be present in the CNP whereby one model conforms to the syntax and semantics of its meta-model. This is required for *a-posteriori* typing, where the models can be re-typed by other meta-models [20]. With this support, models and meta-models can co-evolve during an experimentation process in the notebook, allowing the user to incrementally build the DSL and the models during experimentation or ideation.

3) History and Traceability: The iterative nature of cell execution in the CNP means that previous changes in code and results are lost when the cell is executed again. This is especially problematic in the case of a model in a cell which conforms to a meta-model in a different cell. It is not feasible to repeatedly 'copy-paste' these cells to retain old versions and avoid momentary conformance failure, as there may be rapid iterations of both model and meta-model.

The computational notebook paradigm may therefore need to be enhanced with history for each cell to support MPM activities such as domain-specific language creation and iteration. This history could enable different versions of a cell to be retained, allowing for versioned traceability links to retain the proper conformance information. This history also enhances the reproducibility of the notebook, as an interested peer could examine the precise evolution of the model and meta-model.

B. Notebook Entities for MPM

The entities in the notebook paradigm can be divided into two relevant parts to discuss their interaction with MPM: a) *users and notebooks*, and b) *blocks, language, and kernels*.

1) Users and Notebooks: As described in Section IV-A, the relationships between users and notebooks in the computational notebook paradigm include *Users accessing Notebooks*, which may be *linkedTo* other *Notebooks*.

There can be multiple levels of abstraction and differing formalisms within and between these notebooks. For example, two notebooks (or two different cells) may explain a system at different levels of abstraction. As notebooks may be linked together in a notebook management system (such as Prism [18]), this supports reasoning of a system at different levels of abstraction or different formalisms, or comparison of the results of differing approaches. For example, the notebook in Section III-B could be linked to another notebook representing the bouncing ball model in the Causal-Block Diagram formalism [21] with exact semantics instead of the black-box model presented.

The explicit modeling of workflows in MPM could also involve multiple users and notebooks. As an example, an industrial activity may be represented as an MPM process divided across different stakeholders who perform component design and verification activities. These activities may take place in multiple notebooks or in different cells within those notebooks. Workflows could also be modeled inside the notebook itself to define cell execution, although the linear nature of a notebook may make this less appropriate.

2) Cells, Languages, and Kernels: The key feature in the CNP is code *Cells* which can be (re-)evaluated at any time. As mentioned in Section IV-A, these cells are typed by a particular language with an associated kernel. To place these entries within an MPM context, cells are typed by a particular *formalism* (including domain-specific languages) with syntax and semantics, and (if executable) executed by (a) simulation kernel(s) for that formalism. Different cells within the same notebook may be typed by different formalisms and executed by heterogeneous kernels.

The multi-abstraction and multi-formalism aspects of MPM are thus naturally represented by the typing of notebook cells. However, explicit models of the state and semantics of the underlying kernels must also be available, such that experiments can be restarted from the same initial state. This requires techniques such as model frames (Section V-A1).

C. Notebook Processes for MPM

This section details the support required for combining MPM with the two core processes in the notebook paradigm: *experimentation* and *documentation*.

1) Experimentation for MPM: Experimentation within the MPM context is a broad umbrella encompassing many activities, such as ideation, model transformation, simulation, and verification. Each of these activities provides a view on the system, offering insight by representing the system in different abstraction levels and formalisms. As the user creates and iterates on multiple notebooks or cells focused on these different views, their understanding of the system and its behaviour increases.

To support this multi-abstraction and multi-formalism experimentation, heterogeneous kernels must be available in the computational notebook so that the user may use the "most appropriate" language. As the user may be creating their own domain-specific languages (see Section V-A2), techniques

may need to be developed such that semantics from different modeling languages can be (semi-)automatically combined for simulation, debugging, or other activities.

The combination of the computation notebook paradigm and multi-paradigm modeling offers the explicit modeling required for a reproducible global view of a complex system, including cyber-physical systems which involve multiple domains, with properties and workflows for each. During the experimentation process, the components of the full system evaluation would exist in the same document: a) the properties of interest, b) the design, and c) the processes for checking the properties against the design. Throughout the evolution of the system, the properties would then be checked against the design using the processes as needed. The explicit workflow component of MPM may also be beneficial, as an explicit version of an experimentation activity could be used for (semi-) automatic enactment [22]. The CNP also offers support for full documentation and dissemination such that full-system evaluation can be performed by other groups at other times, ensuring full reproducibility (see Section II).

2) Documentation for MPM: As described in Section IV-B2, computational notebooks can be used to document systems and improve the understanding of the system under study. In the MPM context, this can be employed for educational or training purposes. For example, a notebook could contain an explanation of how property satisfaction changes with the level of abstraction, along with the relevant models and verification code. This would allow the user to experiment with these representations and understand their effect.

Documentation of results is also aided by explicit modeling of the system and workflows. For example, the documentation of a system's workflow could be made actionable by including an explicit representation such as an Formalism Transformation Graph and Process Model (FTG-PM) [22]. This would enable other readers of the notebook to execute that workflow themselves, or to experiment with modifications.

VI. Conclusion

This paper has presented *reproducibility* in the context of the computational notebook paradigm (CNP). The entities and processes involved in the paradigm have been examined, as well as the combination with multi-paradigm modeling (MPM) to enhance reproducibility in both the experimentation and documentation processes.

Future work will implement the above ideas to verify their practicality, especially with regards to the complexities of language engineering and traceability (Section V-A). Additionally, we will examine the integration of the Modelverse [23] into the CNP. The Modelverse is a tool for language and process engineering, performing model operations, and acts as a repository of models, languages, and processes. The Modelverse could therefore be a comprehensive kernel to support MPM activities, such as providing model transformation enactment and full conformance checks on all artefacts.

As computational notebooks become more collaborative, it is also important to provide robust model management tools and techniques. If multiple users modify the same model cell or evolve a meta-model cell, our future work will make clear how inconsistencies are detected and resolved through such techniques as model difference detection and resolution, or co-evolution of models and meta-models.

References

[1] Gravitational Wave Open Science Center, "Binary black hole signals in LIGO open data," https://www.gw-openscience.org/GW150914data/LOSC_Event_tutorial_GW150914.html, 2017.

[2] J. Somers, "The scientific paper is obsolete," *The Atlantic*, 04 2018, https://www.theatlantic.com/science/archive/2018/04/the-scientific-paper-is-obsolete/556676/.

[3] S. Schnell, ""Reproducible" Research in Mathematical Sciences Requires Changes in our Peer Review Culture and Modernization of our Current Publication Approach," *Bulletin of Mathematical Biology*, vol. 80, no. 12, pp. 3095–3105, Sep. 2018.

[4] P. Mosterman and H. Vangheluwe, "Computer automated multi-paradigm modeling: An introduction," *Simulation*, vol. 80, no. 9, pp. 433–450, 2004.

[5] R. Franceschini, P.-A. Bisgambiglia, and D. R. Hill, "Reproducibility study of a PDEVS model application to fire spreading," in *Proceedings of the 50th Computer Simulation Conference*. Society for Computer Simulation International, 2018, p. 29.

[6] O. Dalle, "On reproducibility and traceability of simulations," in *2012 Winter Simulation Conference*. IEEE, Dec. 2012, pp. 1–12.

[7] V. Stodden, J. Borwein, and D. H. Bailey, "Setting the default to reproducible in computational science research," *SIAM News*, vol. 46, no. 5, pp. 4–6, 2013.

[8] T. Kluyver *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows," in *ELPUB*, 2016, pp. 87–90.

[9] S. Wolfram, "What is a computational essay?" https://blog.stephenwolfram.com/2017/11/what-is-a-computational-essay/, 2017.

[10] B. G. Fitzpatrick, "Issues in Reproducible Simulation Research," *Bulletin of Mathematical Biology*, vol. 81, no. 1, pp. 1–6, Sep. 2018.

[11] D. Koop *et al.*, "A provenance-based infrastructure to support the life cycle of executable papers," *Procedia Computer Science*, vol. 4, pp. 648–657, 2011.

[12] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 63, 2018.

[13] Jupyter *et al.*, "Binder 2.0-reproducible, interactive, sharable environments for science at scale," *Proceedings of the 17th Python in Science Conference*, 2018.

[14] S. Klikovits, J. Denil, A. Muzy, and R. Salay, "Modeling frames," in *CEUR workshop proceedings*, 2017, pp. 315–320.

[15] B. Ragan-Kelley *et al.*, "Collaborative cloud-enabled tools allow rapid, reproducible biological insights," *The ISME journal*, vol. 7, no. 3, p. 461, 2013.

[16] F. Perez and B. E. Granger, "Project Jupyter: Computational narratives as the engine of collaborative data science," Grant Proposal, 2015.

[17] E. Syriani *et al.*, "AToMPM: A web-based modeling environment," in *International Conference on Model Driven Engineering Languages and Systems*, 2013, pp. 21–25.

[18] A. Tabard, W. E. Mackay, and E. Eastmond, "From individual to collaborative: the evolution of Prism, a hybrid laboratory notebook," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*. ACM, 2008, pp. 569–578.

[19] B. Victor, "Explorable explanations," http://worrydream.com/ExplorableExplanations/, 03 2011.

[20] J. De Lara, E. Guerra, and J. S. Cuadrado, "A-posteriori typing for model-driven engineering," in *18th International MODELS Conference*. IEEE, 2015, pp. 156–165.

[21] B. Denckla and P. J. Mosterman, "Formalizing Causal Block Diagrams for modeling a class of hybrid dynamic systems," in *Proceedings of Conference on Decision and Control*. IEEE, 2005, pp. 4193–4198.

[22] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss, "FTG+PM: An integrated framework for investigating model transformation chains," in *International System Design Languages Forum*. Springer, 2013, pp. 182–202.

[23] Y. Van Tendeloo and H. Vangheluwe, "The Modelverse: a tool for multi-paradigm modelling and simulation," in *2017 Winter Simulation Conference (WSC)*. IEEE, 2017, pp. 944–955.