REPORT ON OpenDreamKit DELIVERABLE D4.11
Notebook Import into MathHub.info (interactive display)

KAI AMANN, MICHAEL KOHLHASE, FLORIAN RABE, TOM WIESING

Due on	31/08/2018
Delivered on	3/09/2018
Lead	Friedrich-Alexander Universität Erlangen/Nürnberg (FAU)
Progress on and finalization of this deliverable has been tracked publicly at: https://github.com/OpenDreamKit/OpenDreamKit/issues/85	

ABSTRACT.
A comprehensive distributed virtual research environment (VRE) for mathematics as envisioned in the OpenDreamKit project must integrate many disparate existing systems. Often these are developed independently, for different reasons and by different communities. In the absence of a joint vision for an overarching VRE, they tend to remain isolated and incompatible.

The OpenDreamKit project has supplied such a joint vision, and this report considers three of these systems: the Jupyter user interface system, the MathHub document hosting system, and the MMT knowledge representation system. Jupyter offers a uniform interface to the computation facilities of OpenDreamKit systems in the form of a Read-Eval-Print Loop. MathHub offers versioned persistent storage of mathematical documents based on git. MMT provides semantics-aware knowledge management for mathematical objects and as such serves as the center of Math-in-the-Middle (MitM) infrastructure developed in OpenDreamKit (see D6.5). Our ultimate goal is to integrate these three standalone systems as functional units of a VRE: respectively, its frontend, backend, and semantic kernel.

Concretely, we present two practical steps towards this goal. Firstly, we have designed and implemented a Jupyter kernel for MMT. That allows using Jupyter as a frontend for MitM operations in MMT. We also show how Jupyter widgets can be deeply integrated with the MMT knowledge management facilities to give semantics-aware interaction facilities. Secondly, we show how to combine the advantages of Jupyter Notebooks and MathHub documents. In particular, we dynamically employ the highly interactive and often ephemeral Jupyter Notebooks as subdocuments of MathHub documents such as static HTML pages generated from scientific articles.

We evaluate the integration in two case studies. The first one serves as a stepping stone towards the OpenDreamKit VRE: MitM-based computation via MMT using a Jupyter Notebook that is maintained by MathHub. The second one applies our results to a concrete problem outside of mathematics: a knowledge-based specification dialog for modeling and simulation.

Thus this report answers the need for interface integration of documents and REPL interfaces identified in Deliverable Report **D4.2**, and implements and integrates the designs and prototypes from **D4.3** and **D4.9** into a coherent prototype.

CONTENTS

1. Introduction

A mathematical Virtual Research Environment (VRE) needs to support two kinds of user interface paradigms. Firstly, mathematical *documents* have been very successful for presenting mathematical knowledge. While there have been efforts to make them modular and interactive, they predominantly remain in the mode of archiving and transporting knowledge in Mathematics. Secondly, *notebook* interfaces focus on REPL (Read/Eval/Print Loop) interaction leading to documents consisting of a sequence of computational cells within which the mathematical discourse is interspersed in the form of rich comments.

In this report we present a prototypical integration of Jupyter Notebooks into the MathHub document hosting platform. In addition to versioned persistent storage, MathHub offers an interface for reading, writing, and interacting with mathematical documents. Jupyter offers a uniform interface to the computation facilities of the OpenDreamKit VRE toolkit in the form of a read-eval-print loop (REPL). The tension between these systems has previously been explored in OpenDreamKit Deliverable D4.2 [D4.216], and the concept of in-document computation in OpenDreamKit Deliverable D4.9 [ODK17]. In both cases, the integration was incomplete, since it lacked a full integration of the underlying knowledge management and computation services.

Generally, the integration of MathHub and Jupyter consists of two challenges: *a*) the integration of the document paradigms and user interfaces and *b*) the integration of the knowledge management and computation services. The latter requires defining the semantics of the mathematical knowledge maintained in the user interfaces, and both Jupyter and MathHub are parametric in this semantics. In Jupyter, a separate kernel must be provided for each concrete language, in particular there are separate kernels for all computation systems used in OpenDreamKit. In MathHub, the determination of the semantics is delegated to the MMT system. Moreover, MMT is also used in OpenDreamKit as the center of the Math-in-the-Middle based system integration (see OpenDreamKit Deliverable D6.5 [D6.518]).

This reports describes progress in both integration challenges.

In Section 2, for the integration of services, we present an MMT kernel for Jupyter. This not only makes the MMT functionality available at the Jupyter level, but also deeply integrates Jupyter widgets with the MMT Scala level. Widgets are a key Jupyter feature that reaches far beyond the standard REPL interaction. For instance, the Jupyter community has developed a large array of widgets for interactive 2D and 3D visualization of data in the form of charts, maps, tables, etc.

In Section 3, for the integration of document paradigms, we first show how to extend MathHub with a Jupyter server that allows viewing notebooks stored in MathHub. Then we present a MathHub feature that allows using interactive, ephemeral Jupyter Notebooks as subdocuments of static mathematical documents, e.g., HTML pages generated from scientific articles.

In Section 4, we present two cases studies that evaluate our results: in-document computing facilities in active documents and a knowledge-based specification dialog for modeling and simulation. Section 5 concludes the report.

Acknowledgements. The authors gratefully acknowledge the support of the Jupyter team and in particular the advice of Benjamin Ragan-Kelly. The MoSIS system was developed in collaboration with Theresa Pollinger [PKK18].

2. JUPYTER NOTEBOOKS FOR MMT

We designed and implemented a Jupyter kernel for MMT. We describe its interface in Section 2.1 and the implementation in Section 2.2. In Section 2.3, we extend both to graphical interfaces via Jupyter widgets.

2.1. **Interface**

MMT differs from typical computational engines in Jupyter in that it does not only (and not even primarily) perform computation but also handles symbolic expressions with uninterpreted function symbols, whose semantics is described by logical axioms. Another important difference is how MMT handles context and background knowledge. Kernels for (mathematics-oriented or general purpose) programming languages, as typical in Jupyter, build and maintain a dynamic context of declarations with imperative assignment and stack-oriented shadowing and rely on a fixed — often object-oriented — background library of computational functionality. MMT, on the other hand, uses graphs of inter-connected theories to represent a multitude of possible contexts and background libraries and to move knowledge between contexts. To adequately handle these subtleties, we systematically specified a new interface for Jupyter-style interactions with MMT.

Sessions. Jupyter interactions are managed in **sessions**: every browser page opening a notebook creates a new session. Sessions are represented as (ephemeral[1]) MMT documents, which gives them a unique MMT URI, which in turn allows full referencing of all document components. All commands executed within a session manipulate the associated document, most importantly by interactively creating new theories and then calling MMT algorithms on them. The latter include but are not limited to computation.

Input. The possible inputs accepted by the MMT kernel are divided into three groups.

- **Global management commands** allow displaying and deleting all current sessions. In practice, these commands are typically not available to common users, which should only have access to their own session.
- **Local management commands** allow starting, quitting, and restarting the current session. These are the main commands issued by the frontend in response to user action.
- **Content commands** are the mathematically meaningful commands and described below.

The content commands are again divided into two groups:

- **Write-commands** send new content to the MMT backend to build the current MMT document step by step. The backend maintains one implicit, ephemeral MMT document for each session, and any write command changes that document.
- **Read-commands** retrieve information from the backend without changing the session's document. These include lookups (both in the session document and in any other accessible document) or computations.

A write-command typically consists of a single MMT declaration roughly corresponding to a line in a typical MMT source file. However, the nesting of declarations is very important in MMT. This is in contrast to many programming language kernels where nesting is often optional, e.g., to define new functions or classes; for many current kernels, it makes sense to simplify the implementation by requiring that the entire top-level command, including any nesting, be contained in a single cell.

[1]We call an MMT document **ephemeral**, iff it is (at least initially; it can be serialized and saved) created only in memory in the MMT process; apart from this, it behaves like any other MMT document

In our MMT kernel, all declarations that may contain nested declarations (most importantly all MMT documents and theories) are split into parts as follows: the header, the list of nested declarations, and a special end-of-nesting marker. Each of these is communicated in a separate write-command. The semantics of MMT is carefully designed in such a way that *i*) any local scope arising from nesting has a unique URI, and *ii*) if a well-formed MMT document is built incrementally by appending individual declarations to a currently open local scope, any intermediate document is also well-formed. This is critical to make our implementation feasible: the MMT kernel maintains the current document as well as the URI of the current scope; any write-command affects the current scope, possibly closing it or creating new subscopes. This ensures that all nested declarations are parsed and interpreted in the right scope.

For example, the sequence of commands on the left of Figure 1 builds two nested theories, where the inner one refers to the type `a` declared in the outer one. The right-hand side of Figure 1 shows the equivalent MMT surface syntax on the right. Semantically, there is no difference between entering the left-hand side interactively via our new kernel or processing the write commands on the right with the standard MMT parser.

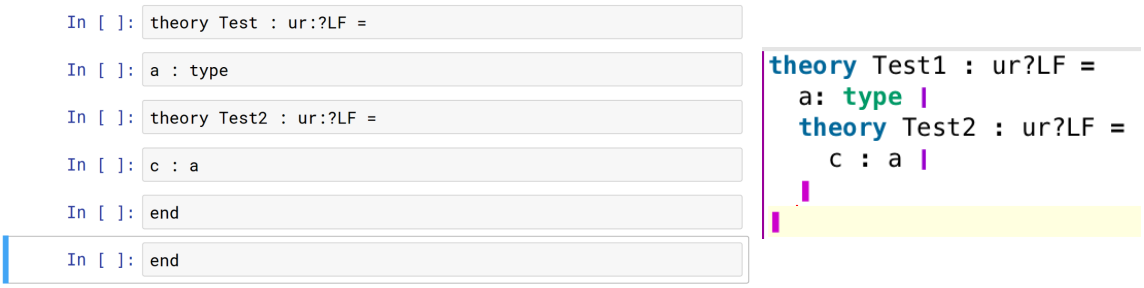

FIGURE 1. Content Commands for Building Theory Graphs

A special write-command is `eval T`. It interprets `T` in the current scope, infers its type `A`, computes its value `V`, and then adds the declaration `resI:A=V` to the current theory, where `I` is a running counter of unnamed declarations. This corresponds most closely to the REPL functionality in typical Jupyter kernels.

While write-commands correspond closely to the available types of MMT declarations, the set of read-commands is extensible. For example, the commands `get U` where `U` is any MMT URI returns the MMT declaration of that URI.

Output. The kernel returns the following kinds of return messages:

- **Admin messages** are strings returned in response to session management commands.
- **New-element messages** return the declaration that was added by a write-command.
- **Existing-element messages** return the declaration that was retrieved by a `get` command.

Like read-commands, the set of output messages is extensible.

The new-element and existing-element messages initially return the declaration in MMT's abstract syntax. And a post-processing layer specific to Jupyter renders them in HTML+presentation MathML. That way, the core kernel functionality can be reused easily in other frontends than Jupyter.

2.2. Implementation

Overview. Generally, Jupyter emphasizes protocols that specify the communication between frontend (i.e., usually Jupyter notebooks) and backend (i.e., kernels implemented in various programming languages). This requires a certain duplication of implementation and, critically, maintenance, e.g., when implementing xeus, xwidgets and similar libraries for C++. But the Python infrastructure for kernels is by far the best developed one, especially when it comes to

Jupyter widgets. Therefore, it makes sense to implement our kernel on top of Python. However, actually executing the user's commands requires a strong integration with the MMT implementation, which uses Scala. That made it advisable to implement all Jupyter-specific functionality, especially the communication and management, in Python, while all mathematically relevant logic is handled in Scala.

Therefore, our implementation consists of three layers. The top layer (depicted on the left of Figure 2) is a Python module that implements the abstract class for Jupyter kernels. The bottom layer is a Scala class adding a general-purpose REPL to MMT that handles all the logic of MMT documents. This can be reused easily in other frontends. User commands are entered on the client and sent to the top layer, which forwards all requests to the bottom layer and all responses from the bottom layer to the client. The communication between top and bottom layer is handled by a middle layer. Its main purpose is to bridge between Python and MMT, format results in HTML, and add interactive functionality via widgets.

This bridging of programming languages is a generally difficult problem. After some experiments with different solutions (e.g., HTTP communication) and discussion within the OpenDreamKit community, we identified the Py4J library [P4J] as the best choice. This is a Python-JVM bridge that allows seamless interaction between Python and any language (such as Scala) that compiles to the JVM. Thus, our Python kernel can call MMT code directly. Valuable Py4j features include callbacks from MMT to Python, shared memory (by treating pointers to JVM objects as Python values), and synchronized garbage collection. That makes our kernel very robust against bit rot and allows benefiting from future improvements to the MMT backend.

Py4J is only JVM-specific, not Scala-specific. That means that some Scala-specific constructs are not readily exposed to Python. For example, both Python and Scala allow magic methods for treating any object as a function, but the JVM does not; moreover, the magic method is called `__call__` in Python and `apply` in Scala. Similarly, Scala collections like lists are not automatically seen as their counterparts in Python. Therefore, we wrote a Python module (which is distributed with MMT[2]) that performs the bureaucracy of matching up advanced Python and Scala features.

2.3. **Graphical User Interfaces via Jupyter Widgets**

Jupyter widgets are interactive GUI components (e.g., input fields, sliders, etc.) that allow Jupyter kernels to provide graphical interfaces. While the concept is general, it is most commonly used to refer to the Python-based widget library developed for the Python kernel. A widget encapsulates state that is maintained in an instance of a Python class on the server and displayed via a corresponding Javascript/HTML component on the client. A major advantage of our kernel design is that we can reuse these widgets (via the to layer).

As our kernel's intelligence is maintained is maintained in MMT and thus Scala, we had to write some middle layer code to allow our kernel to create widgets. This code uses Py4J to expose the widget-management functionality of the top layer to the lower layers. This is done via a class of callback functions C that are passed along when the former calls the latter.

Figure 2 shows the details of the communication. The upper part shows the simplest (widget-less) case: MMT content is entered in the frontend and forwarded to the bottom layer, and the response is forwarded in the opposite direction. (Steps that simply forward data from one layer to the next are not shown explicitly.)

The lower part shows a more complex widget-based interaction. First of all, we add special management commands that are not passed on to the GUI-agnostic bottom layer. Instead, they are identified by the middle layer, which responds by delegating to a GUI application. This application then builds its graphical interface by calling the callbacks passed along by the top

[2]This is currently at `https://github.com/UniFormal/MMT/blob/devel/src/python-mmt/example/mmt.py` but may be moved in the future.

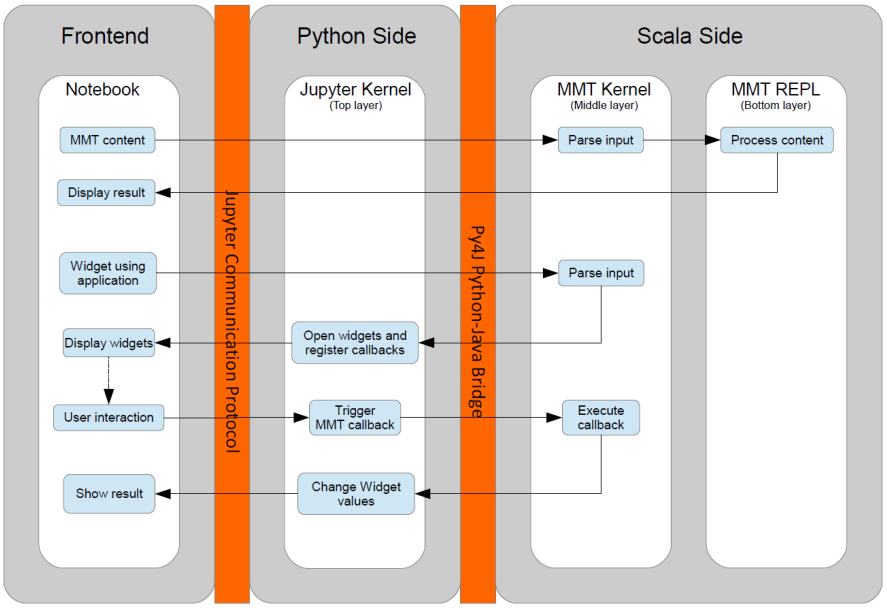

FIGURE 2. Architecture diagram

layer. This results in a widget object in Python that is returned to the top layer and then forwarded to the frontend.

As usual, GUI components may themselves carry callback functions for handling events that are triggered by user interaction with the GUI in the frontend. While conceptually straightforward, this leads to an unusually deep nesting of cross-programming language callbacks. When creating a widget, the Scala-based GUI application may pass Scala callbacks D whose implementation makes use of the C-callbacks provided by the top layer. Thus, a user interaction triggers an MMT callback D in the Python top layer, which is executed on the Scala side via Py4J, which in turn may call the Python callbacks C exposed via Py4J.

Our design makes it very easy to build and deploy simple GUI applications for MMT — we still have the full power of Jupyter widgets at our fingertips.

Example: In-Document Computation. We present a simple example of a GUI application for in-document computation as specified D4.9 [ODK17]. It is triggered by the special command `active computation` and builds a GUI consisting of a few standard Jupyter widgets: a label, a button (labeled *compute*), three text input fields, and one button widget. A concrete example can be seen in Figure 3. This shows a notebook in which our application is returned as the response to cell `In[1]`.

The three text input fields contain values linked by an equation, in this case $E = mc^2$. The user can edit these fields and press the button to compute the other values. In that case, a the button carries the callback D, which results in a call to our application on the Scala side. It uses MMT to perform the computation and then calls the C callbacks to update the values in the widgets. No additional work is needed to implement the synchronization between the Python top layer and the HTML frontend as this is a standard feature of Jupyter widgets.

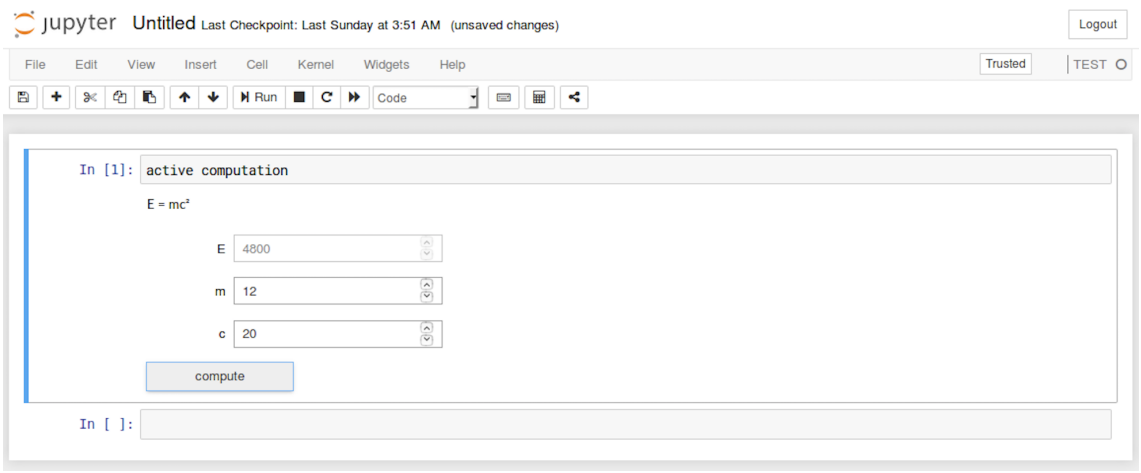

FIGURE 3. Active Computation in Jupyter Notebooks via Jupyter/MMT Widgets

3. JUPYTER NOTEBOOKS IN MATHHUB

We present the design and partial implementation[3] of a new user interface that combines the advantages of Jupyter Notebooks and MathHub documents.

3.1. **Architecture**

Our integrated VRE consists of four components: a Jupyter Notebook server[4], a MathHub Git repository hosting server[5], an MMT instance[6], and our new MathHub frontend[7] that serves as the main entry point for users and delegates some subtasks to the former. The Jupyter server is an out of the box installation of Jupyter except for additionally supporting our new MMT kernel. Jupyter Notebooks are stored (and versioned) as regular files in MathHub.

The MathHub frontend provides special interaction functionality for individual document types. This allows making Jupyter Notebooks a new document type; see Figure 4 for an example. When displaying a known document type, the frontend shows multiple tabs. For notebooks, these are the following:

i) view gives a preview of the notebook, essentially the computation cells without output, pre-rendered for static serving without involving Jupyter at all.

ii) run/edit opens the respective notebook on the Jupyter server for execution and editing. Any changes to the notebook can be committed back to the Git repository.

iii) metadata (this is the tab open in Figure 4), shows the metadata provided by the Jupyter kernel and the repository.

iv) source provides access to the document source; here simply a link to the notebook file in the Git repository.

[3]We had originally planned to realize this user interface as an extension of the existing frontend to the MathHub system. However, this interface was based on Drupal, which led to a major system vulnerability when Drupal was repeatedly targeted by hackers. Therefore, we are developing a new MathHub frontend from scratch, which has the advantage that the integration of Jupyter and MMT can be considered from the start. It employs docker-based orchestration of services and a React.JS based frontend but is not publicly deployed just yet. We expect a fully-featured beta version until the OpenDreamKit Review end of November 2018. A development preview server is available at http://new.mathhub.info, but cannot be considered stable in any way.

[4]http://jupyter.mathhub.info

[5]http://gl.mathhub.info

[6]http://mmt.mathhub.info

[7]To be deployed at http://mathhub.info; development preview at http://new.mathhub.info

v) statistics shows statistical information about the notebook, its corresponding MMT document, and its connections with other MMT documents in the background knowledge base.

vi) graph links to graph-based visualizations of the document including the theory graph, declaration graph, and dependency graph, using our TGView system, a canvas-based in-browser visualizer for knowledge graph information [RKM17].

This integration combines the interactive features of the Jupyter server with the knowledge management facilities on MathHub. In the future, we plan to integrate the notebook diff/patch nbdime developed in OpenDreamKit to extend the knowledge management facilities.

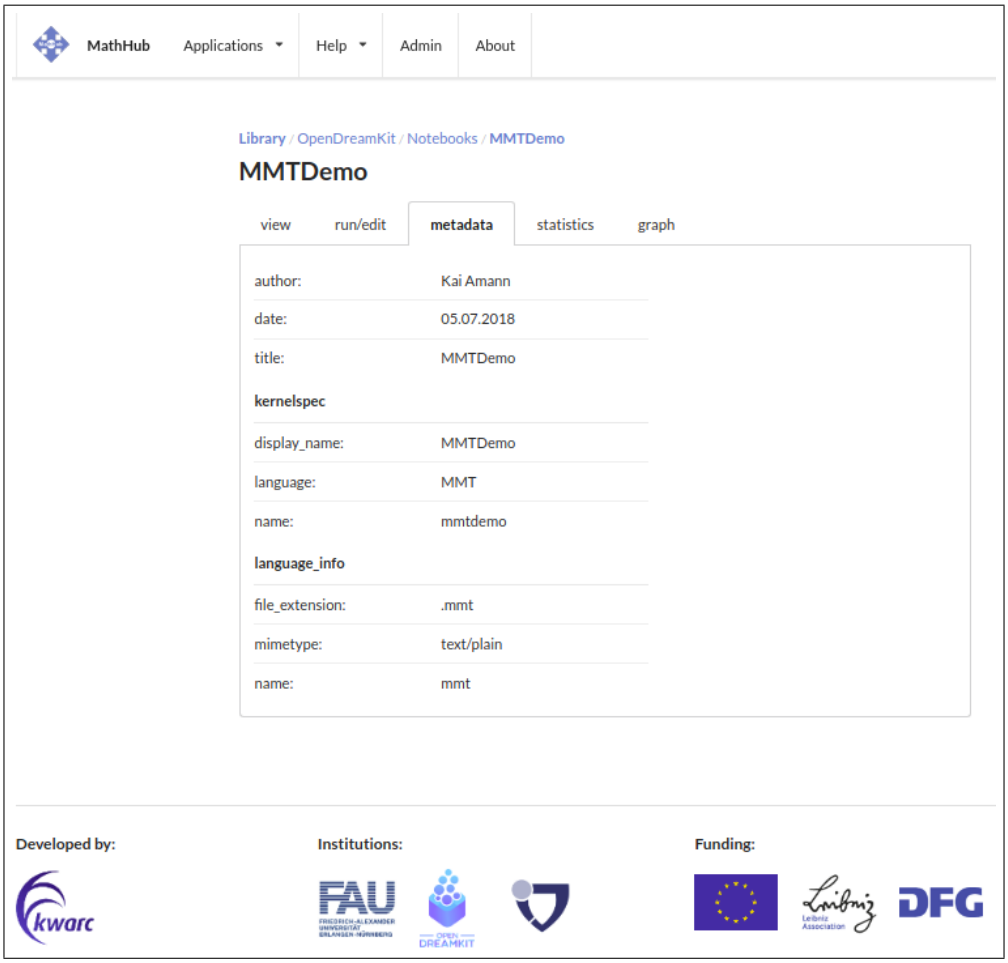

FIGURE 4. A Jupyter Notebook in MathHub.info (Metadata)

3.2. **Document/Notebook Integration**

To achieve full document/notebook integration, as envisioned in D4.2 [D4.216], we have developed a tool that generates ephemeral Jupyter Notebooks dynamically and embeds them into HTML presentation of MathHub documents. Here we can feed the document context information into the interior notebook as described in [ODK17] (this part of the integration can be re-used directly).

As a running example, we revisit the active computation example from the previous section. Figure 5 shows an example of a scientific HTML document that contains the equation $E = mc^2$. The user can use the context menu to trigger the notebook generation on this formula. The context menu is generated using Javascript that picks up on annotations of formulas with specific CSS classes. Currently the author has to manually annotate the formulas, but we are working on a mechanism to automatically create it from the document context.

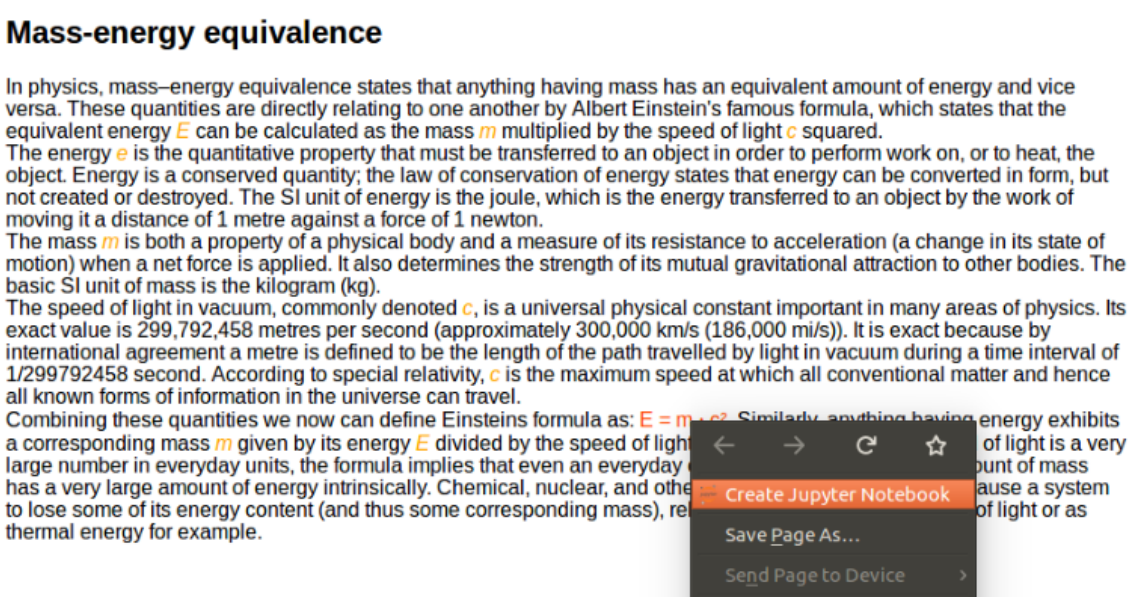

FIGURE 5. HTML document and the context menu for converting

Figure 6 shows the notebook created by our tool. If desired, the notebooks can be easily uploaded to the Jupyter server, stored persistently in the repository server, or evaluated in a locally deployed version of the system per drag-and-drop.

Note that the generated notebook starts with several `include` declarations that import the context of the formula. These are generated by MathHub to obtain a minimal standalone MMT theory in which the respective formula is well-formed.

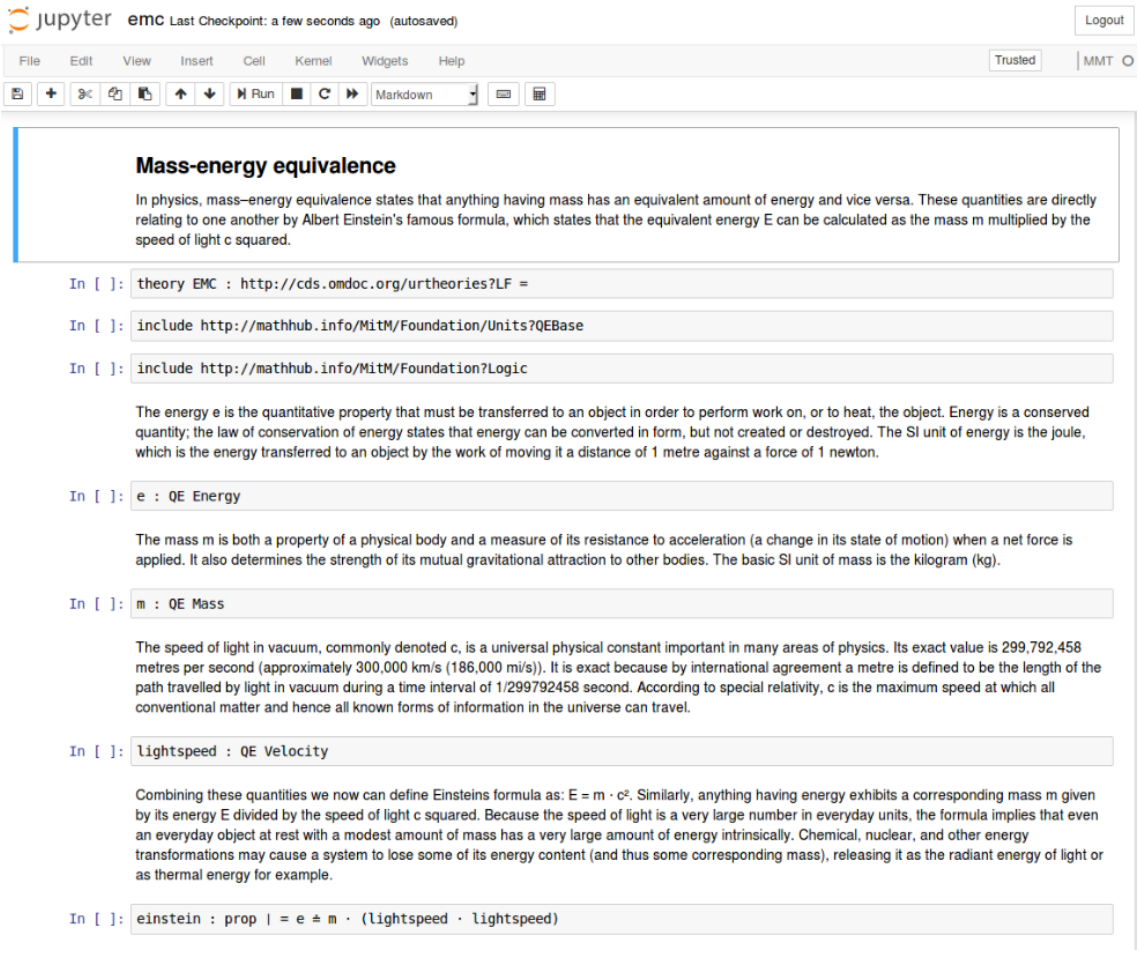

FIGURE 6. The resulting Jupyter notebook

4. APPLICATIONS

4.1. **Towards a MitM-Based VRE**

The Math-in-the-Middle (MitM) approach developed in OpenDreamKit WP 6 uses the MMT language for formalizing mathematical background knowledge (which we store in MathHub documents of type MMT) and the MMT system for integrating computation tools. Therefore, Jupyter-MMT notebooks can serve as a unified user interface for MitM systems.

For example, consider the theory[8] in Figure 7, which serves as our standard example for the interaction between MMT and LMFDB (a large database of mathematical objects that was integrated with MMT in previous deliverables of OpenDreamKit). We can now rewrite it as a notebook.

A screenshot of the resulting notebook, as displayed by a Jupyter server running our MMT kernel, is shown in Figure 8. The selected declaration of `mycurve` accesses the elliptic curve `11a1` that is stored in LMFDB. When the Jupyter kernel for MMT processes this command, the bottom layer of the kernel dynamically retrieves this curve from LMFDB and builds from it an object of type `elliptic_curve` in the MitM ontology. More complex MitM computations (e.g., as described in Deliverable **D6.5** [D6.518]) can be handled accordingly.

```
theory Example : MitM:/Foundation?Logic =
 include MitM:/smglom/elliptic_curves?Elliptic_curve |
 include MitM:/smglom/elliptic_curves?Conductor |

 include db/elliptic_curves?curves |
 /T get elliptic curve 11a1 from LMFDB |
 mycurve: elliptic_curve | = `db/elliptic_curves?curves?11a1 |
 /T let c be its conductor (as stored in the LMFDB) |
 c: int_lit | = conductor mycurve |

 include db/transitivegroups?groups |
 /T get transitive group with label 8T3 from LMFDB |
 mygroup: group | = `db/transitivegroups?groups?8T3 |
 /T let d be its number of automorphisms (as stored in the LMFDB) |
 d : ℕ | = automorphisms mygroup |

 /T Same for a Hilbert Newform |
 include db/hmfs?hecke |
 hecketest : hilbertNewform | = `db/hmfs?hecke?4.4.16357.1-55.1-c |
 dimtest : ℕ | = dimension hecketest |

|
```

FIGURE 7. A Theory for LMFDB/MMT Interaction

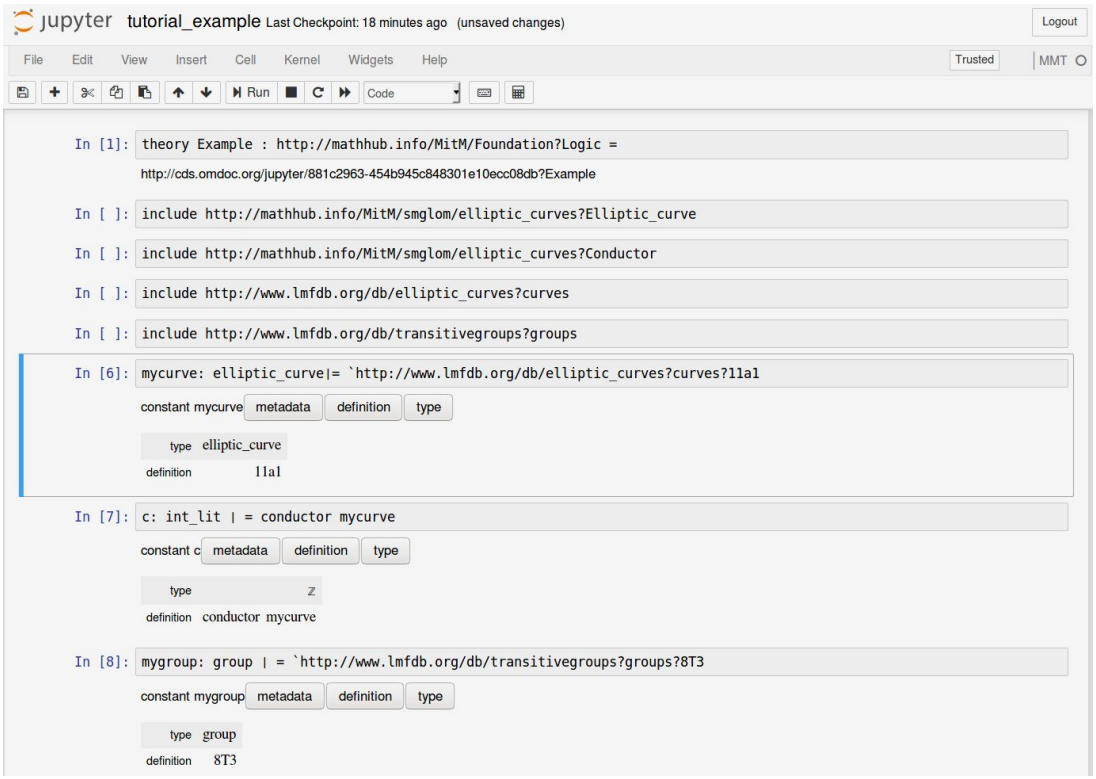

FIGURE 8. The Beginning of the Notebook for the theory from Figure 7

4.2. **Domain specific applications: MoSIS**

Our second case study addresses a *knowledge gap* that is commonly encountered in computational science and engineering: To set up a simulation, we need to combine domain knowledge

[8]Available at https://gl.mathhub.info/ODK/lmfdb/blob/master/source/schemas/tutorial_example.mmt

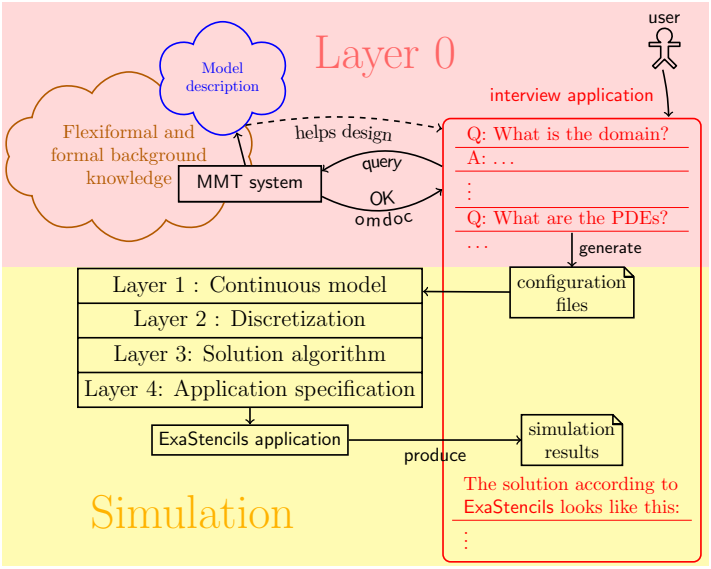

FIGURE 9. MoSIS Information Architecture

(usually in terms of physical principles), model knowledge (e.g., about suitable partial differential equations) with simulation (i.e., numerics/computing) knowledge. In pre-VRE practice, this is resolved by intense collaboration between experts, which incurs non-trivial translation and communication overheads. In OpenDreamKit, we propose an alternate solution based on mathematical knowledge management techniques. A detailed description was published as [PKK18].

Concretely, we use a Jupyter notebook that has access to an MMT theory graph on MathHub.info. Figure 11 shows this theory graph of background knowledge about mathematical models and partial differential equations. Our Jupyter/MMT/Mathhub integration enabled building an interview application that hides these mathematical details from the user.

Based on this theory graph, we built a targeted knowledge acquisition dialog that supports the formalization of domain knowledge, combines it with simulation knowledge and finally drives a simulation run — all integrated into a Jupyter Notebook. Figure 9 shows the general architecture: The left side shows the simulation engine ExaStencils [EXA] and the MMT system that acts as the theory graph interface. The right hand side shows the interview — a Jupyter notebook — as the active document and how it interacts with the MMT kernel. The user only sees the notebook. She answers the knowledge acquisition questions presented by MoSIS until MoSIS can generate a configuration file for ExaStencils. The latter builds efficient code from it through the ExaSlang layers and computes the results and visualizations, which MoSIS in turn incorporates into the notebook. Figure 10 shows a screenshot of the notebook.

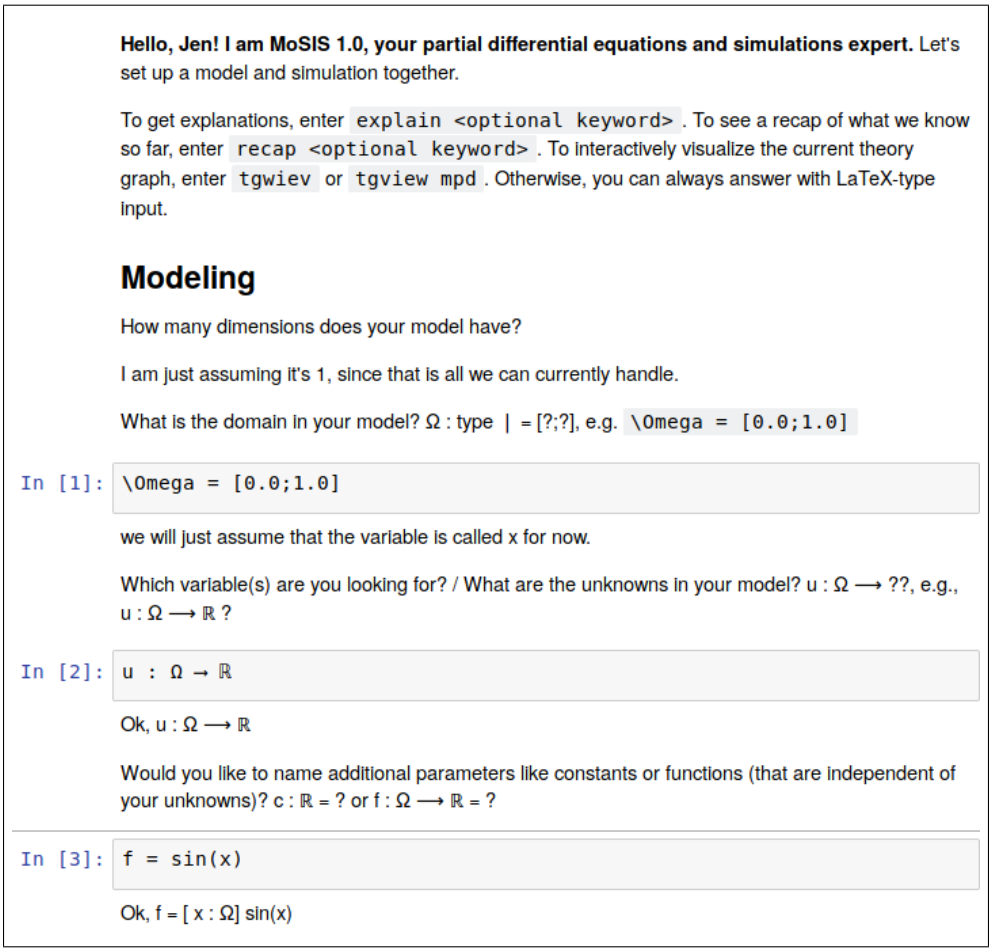

FIGURE 10. Beginning of a Dialogue in MoSIS

5. CONCLUSION AND FUTURE WORK

We have presented an integration of three systems in the OpenDreamKit project: Jupyter for computation/experimentation in notebooks and MathHub for interactive mathematical documents as well as MMT for describing the semantics of the knowledge contained in the former. We have evaluated the reach of the evaluation in several case studies.

The resulting system serves as an intermediate step towards the full OpenDreamKit Virtual Research Environment. While this report focuses on the integration of documents and notebooks in a unified system, other deliverables have built other components of the envisioned environment: the Math-in-the-Middle ontology [D6.818] and the instantiation of the MitM framework with concrete mathematical computation and database systems [D6.518]. Future work will focus on merging and scaling up these development strands.

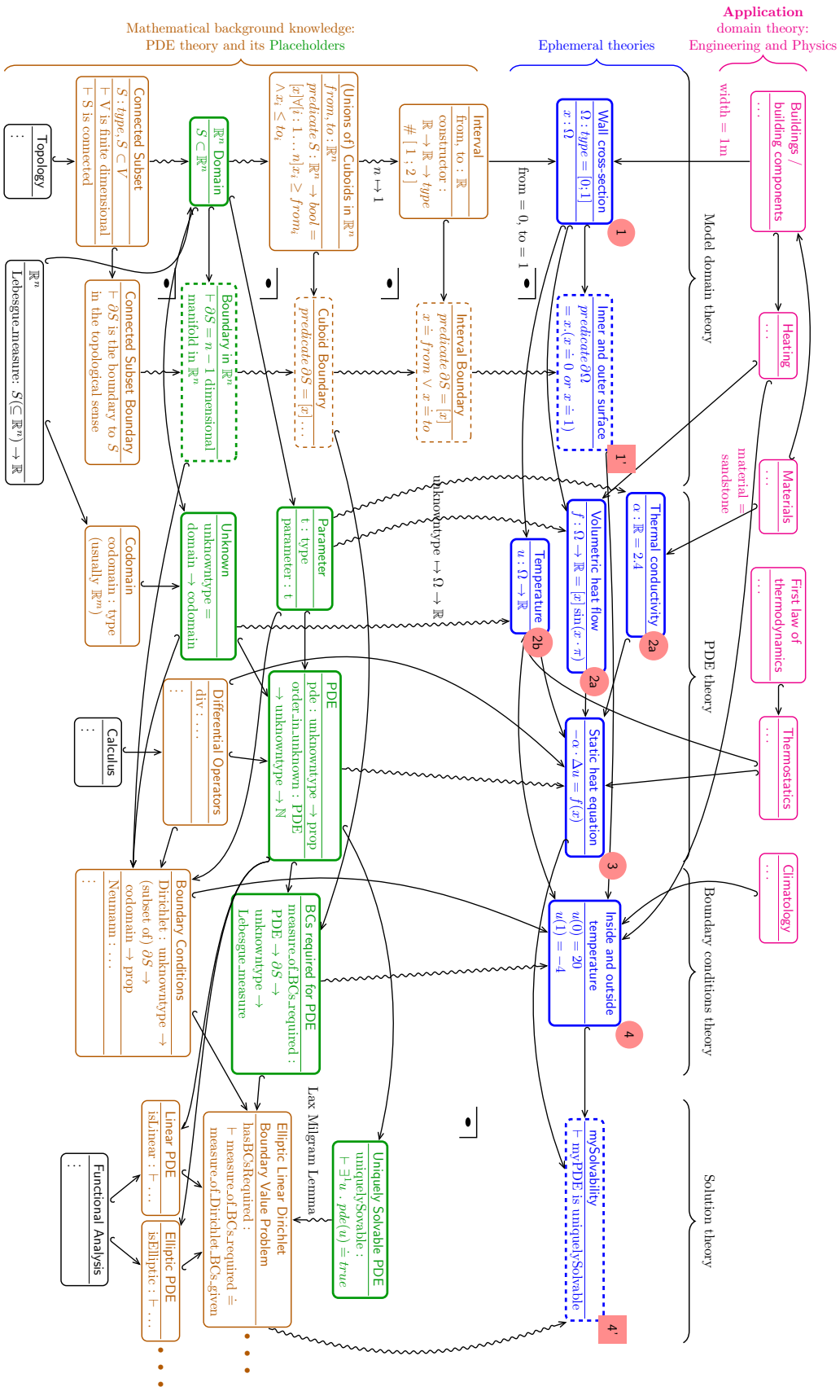

FIGURE 11. Theory Graph for the MoSIS Case Study

REFERENCES

[D4.216] Michael Kohlhase. *Active/Structured Documents Requirements and existing Solutions*. Deliverable D4.2. OpenDreamKit, 2016. URL: https://github.com/OpenDreamKit/OpenDreamKit/raw/master/WP4/D4.2/report-final.pdf.

[D6.518] John Cremona et al. *Report on OpenDreamKit deliverable D6.5: GAP/SAGE/LMFDB Interface Theories and alignment in OMDoc/MMT for System Interoperability*. Deliverable D6.5. OpenDreamKit, 2018. URL: https://github.com/OpenDreamKit/OpenDreamKit/raw/master/WP6/D6.5/report-final.pdf.

[D6.818] John Cremona et al. *Report on OpenDreamKit deliverable D6.8: GCurated Math-in-the-Middle Ontology and Alignments for GAP/SAGE/LMFDB*. Deliverable D6.8. OpenDreamKit, 2018. URL: https://github.com/OpenDreamKit/OpenDreamKit/raw/master/WP6/D6.8/report-final.pdf.

[EXA] *Advanced Stencil-Code Engineering (ExaStencils)*. URL: http://exastencils.org (visited on 04/25/2018).

[ODK17] Michael Kohlhase and Tom Wiesing. *In-place computation in active documents (context/computation)*. Deliverable D4.9. OpenDreamKit, 2017. URL: https://github.com/OpenDreamKit/OpenDreamKit/raw/master/WP4/D4.9/report-final.pdf.

[P4J] *Py4J*. URL: https://www.py4j.org/ (visited on 07/16/2018).

[PKK18] Theresa Pollinger, Michael Kohlhase, and Harald Köstler. "Knowledge Amalgamation for Computational Science and Engineering". In: *Intelligent Computer Mathematics (CICM) 2018*. Conferences on Intelligent Computer Mathematics. Ed. by Florian Rabe et al. LNAI. in press. Springer, 2018. URL: http://kwarc.info/kohlhase/papers/cicm18-mosis.pdf.

[RKM17] Marcel Rupprecht, Michael Kohlhase, and Dennis Müller. "A Flexible, Interactive Theory-Graph Viewer". In: *MathUI 2017: The 12th Workshop on Mathematical User Interfaces*. Ed. by Andrea Kohlhase and Marco Pollanen. 2017. URL: http://kwarc.info/kohlhase/papers/mathui17-tgview.pdf.

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.