Integrating Semantic Mathematical Documents and Dynamic Notebooks

Kai Amann, Michael Kohlhase, Florian Rabe, and Tom Wiesing

Computer Science, FAU Erlangen-Nürnberg

Abstract. Mathematical software systems offer two major paradigms for interacting with mathematical knowledge. One is static files with semantically annotated representations that define mathematical knowledge and can be compiled into documents (PDF, html, etc.), and the other dynamically build mathematical objects in interactive read-eval-print loops (REPL) such as notebooks. Many author-facing interfaces offer both features in some way. However, reader-facing interfaces usually show only one or the other.
In this paper we present an integration of the approaches in the context of the MMT system. Firstly, we present a Jupyter kernel for MMT which provides web-ready REPL functionality for MMT. Secondly, we integrate the resulting Jupyter notebooks into MathHub, a web-based frontend for mathematical documents. This allows users to context-sensitively open a Jupyter notebook as a dynamic subdocument anywhere inside a static MathHub document. Vice versa, any such highly interactive and often ephemeral notebook can be saved persistently in the MathHub backend at which point it becomes available as a static document. We also show how Jupyter widgets can be deeply integrated with the MMT knowledge management facilities to give semantics-aware interaction facilities.

1 Introduction

Mathematical software systems need 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. A "literate programming" version of notebooks which gives mathematical discourse structural precedence is possible in principle but has not been supported consistently at the system level.

A combination of both of these paradigms almost immediately leads to new applications. One such application is the interaction with document-based systems, such as MMT, within a REPL. The MMT tool ecosystem only really supported IDE-interaction with MMT libraries via Edit and (recently) IntelliJ IDEA plugins. While the MMT system provides a simple shell for interaction,

this was only used for configuration and setup of the MMT process. We anticipate that the REPL-like interaction will feel more natural for users of interactive theorem provers and computer algebra systems.

Goals and Challenges Static documents do not allow for interactivity, and notebook approaches require significant programming knowledge to use. Our goal is to overcome these restrictions to enable domain experts to created interactive documents declaratively. This leads us to two challenges

i) How can we combine the notebook and document paradigms?
ii) How can we support flexible interactions without forcing authors to program?

Traditionally, flexible interactions in (web) documents are handled by applets, small, document-embedded programs providing specialized functionality. Modern notebook systems such as Jupyter, which we introduce in more detail below, provide the concept of widgets which provide applet-like functionality, but their combination into interactions still requires non-trivial programming.

Contribution We present an integration of Jupyter Notebooks into the MathHub platform for hosting semantic, active documents. MathHub offers versioned persistent storage for semantically enhanced mathematical documents and knowledge representations. These are unified into the OMDoc/MMT format and loaded into a cross-document-format mathematical knowledge space managed by the MMT system (written in Scala). MathHub is a web frontend for showing OMDoc/MMT content as (largely static) mathematical documents. Jupyter offers a uniform interface to various computation facilities in the form of a read-eval-print loop (REPL), which can be seen as dynamic, ephemeral documents. The system consists of a general, feature-rich browser-based REPL interface that communicates to a system-specific backend, called a Jupyter kernel that supplies the computational capabilities. Such a kernel either connects the native system REPL via a generic Python kernel or uses language-specific network libraries.

Generally, the integration of MathHub and Jupyter consists of two challenges:

i) the integration of the document paradigms and user interfaces and
ii) 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. This paper describes progress in both integration challenges.

Overview 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 case 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 paper.

Acknowledgements We acknowledge financial support from the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541). 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

Jupyter notebooks consist of a sequence of cells; each of which contains either rich text or code that can be evaluated. The Jupyter user interface is implemented using TypeScript in the browser. The backend is implemented in Python and delegates the programming-language specific features to so-called kernels via a networking protocol. Each kernel works exactly like a REPL, that is to say they receive the user input of the code cells and produce output to be presented to the user. Additionally, kernels can implement custom interactions using widgets, consisting of re-usable user interface components that communicate directly with the kernel. Kernels for a specific programming language are typically implemented in that programming language, to ease implementation and make use of existing tool support. For details we refer the reader to [JD].

We designed and implemented a Jupyter kernel for MMT. The source code is available at [MMTJup17]. We describe the requirements of an MMT REPL in Section 2.1, its interface in Section 2.2, the implementation in Section 2.3 and our conversion between MMT data structures and notebook in Section 2.4. In Section 2.5, we describe and discuss our implementation of widgets within our kernel.

2.1 A REPL for MMT

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 to move knowledge between contexts. To adequately handle these subtleties, we systematically specified a new interface for Jupyter-style interactions with MMT.

Example MMT uses theory graphs to model mathematical knowledge (see Figure 1). This theory graph shows two kinds of inheritance mechanisms in MMT: commutative groups (theory CGrp) include monoids (Monoid), inheriting all Monoid objects (the universe U, the operationop, and the unit element e) and the unit axiom unit. Rings are formed by combining a (multiplicative) monoid with an (additive) commutative group. Inclusion roughly corresponds to class inheritance in object-oriented programming, while MMT structures duplicate material. Here the operation op from Monoid forms both addition ($+ =$ add/op) and multiplication ($* =$ mul/op) in a ring and the Monoid unit becomes both zero (add/e) and the one elements (mul/e) of the ring.

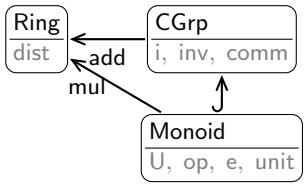

Fig. 1. Rings in MMT

The MMT system is usually used to answer queries such as computing particular, inherited ring axioms: $x+0=x$ and $x*1=x$ or determining the theorems and axioms of (i.e. inherited into) a theory.

2.2 Interface and Sessions

On top of the notebook abstraction, Jupyter interactions are managed in **sessions**: every browser page opening a notebook creates a new session.

MMT already has an abstraction that can closely model a notebook, called a document. In MMT terms, a document is a narrative construct that contains a sequence of declarations. For details, see the MMT documentation at [MMT]. Each input within the Jupyter session can be represented as a single declaration within the corresponding document; see Section 2.4 for further applications of this mapping.

Thus it makes sense to represent each session as an ephemeral MMT document. 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. This gives each session 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 come in three groups.

1. **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.
2. **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.
3. **Content commands** are the mathematically meaningful commands and described below.

The content commands are again divided into three groups:

1. **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.
2. **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.
3. **Interactive-commands** that create a new user interface component allowing the user to interactively read and write MMT content. In the Jupyter system these are implemented as widgets which extend the REPL-paradigm; see Section 2.5.

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.

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 2 builds two nested theories, where the inner one refers to the type `a` declared in the outer one. The right-hand side of Figure 2 shows the equivalent MMT surface syntax on the right. Semantically, there is no difference between entering the left-hand

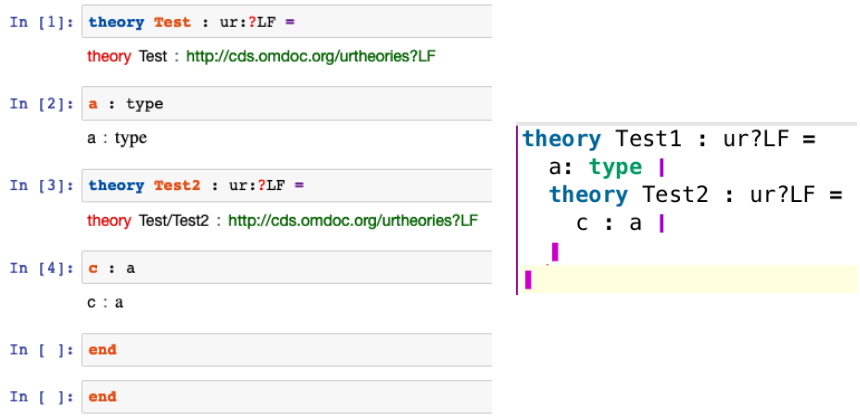

Fig. 2. Content Commands for Building Theory Graphs

side interactively via our new kernel or processing the write commands on the right with the standard MMT parser.

An additional 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:

1. **Admin messages** are strings returned in response to session management commands.
2. **New-element messages** return the declaration that was added by a write-command.
3. **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. A post-processing layer specific to Jupyter renders them in HTML5+MathML (presentation). That way, the core kernel functionality can be reused easily in frontends other than Jupyter.

2.3 Implementation

Generally, Jupyter emphasizes protocols that specify the communication between frontend and backend.

Executing the user commands requires a strong integration with the MMT system, which is implemented in Scala. Even though a Jupyter Scala kernel

exists, we implement the MMT kernel on top of the Jupyter Python kernel infrastructure which is by far the best developed one. We implement all Jupyter-specific functionality, especially the communication and management, in Python, while all mathematically relevant logic is handled in Scala.

Our implementation consists of three layers. The top layer (depicted on the left of Figure 3) 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 e.g., the IntelliJ IDE plugin for MMT. User commands are entered in the front-end 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 the top and bottom layer is handled by a middle layer which bridges between Python and MMT, formats results in HTML5, and adds interactive functionality via widgets.

This bridging of programming languages is a generally difficult problem. We chose to make use of the Py4J library [P4J], a Python-JVM bridge that allows seamless interaction between Python and any JVM-based language (such as Scala). 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 allows our kernel to directly and easily benefit from future improvements to the MMT backend, without needing to duplicate these improvements in kernel-specific code.

As Py4J works at the Java/JVM level, we provide a Python module that performs the bureaucracy of matching up advanced Python and Scala features. This is distributed along with the Jupyter Kernel.

2.4 Converting between Jupyter Notebooks and MMT Documents

Recall that we were to closely model each notebook as an MMT document. To integrate Jupyter notebooks and MMT documents, we make use of two fortunate design properties:

Firstly, the Jupyter notebook format is well-documented [JND]. We implemented an **OMDoc/NB** API in MMT that can extract the MMT content of a notebook and generate a notebook pre-filled with some MMT content.

Secondly, MMT abstracts from the file formats for MMT documents – e.g. MMT's native surface syntax, STEX, or prover libraries – and maintains a cross-format document space of any document that can be converted into OMDoc. The OMDoc/NB API to adds Jupyter notebooks into this. Thus, we can support the following workflow:

1. MMT content is written in any format and available as OMDoc.
2. A new interactive notebook is written, using some of that content.
3. The notebook is stored as a file and MMT extracts the relevant content as OMDoc.
4. Any other MMT document (including other notebooks) can now use this content.

2.5 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 directly in Scala using PY4J (in the top layer)

As our kernel's intelligence 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.

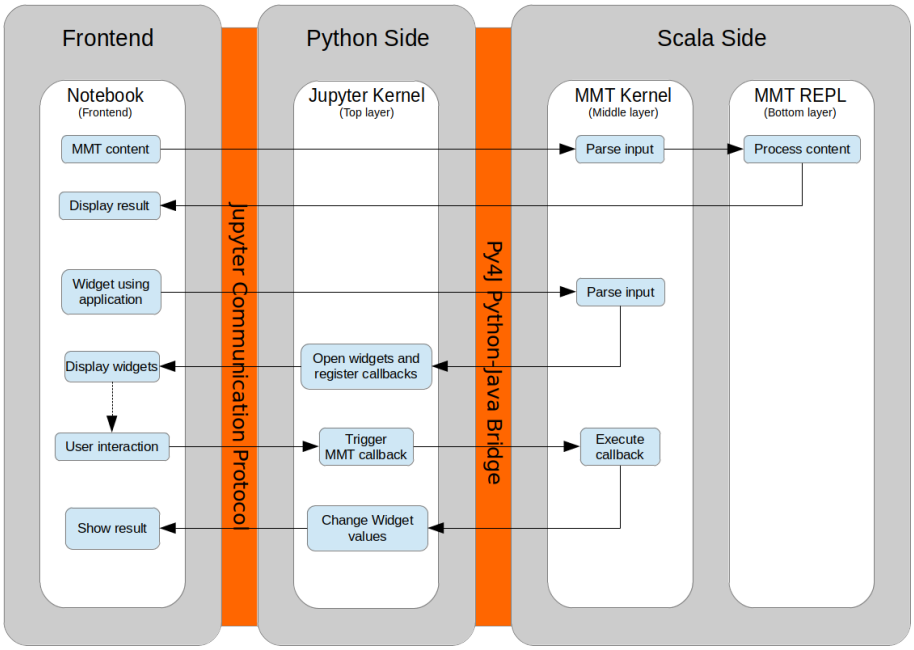

Fig. 3. Architecture diagram. Steps that simply forward data from one layer to the next are not shown explicitly.

Figure 3 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.

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 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 whose implementation makes use of the callbacks provided by the top layer. Thus, a user interaction triggers an MMT callback in the Python top layer, which is executed on the Scala side via Py4J, which in turn may call the Python callbacks exposed via Py4J.

Example: In-Document Computation We present an example of a GUI application inside of a notebook. We will later use this widget for active in-document computation. Figure 4 presents a simple example.

This notebook first defines a new theory (in `In[1]`), called AdditionExample. This theory makes use of the MMT implementation of real number arithmetics.

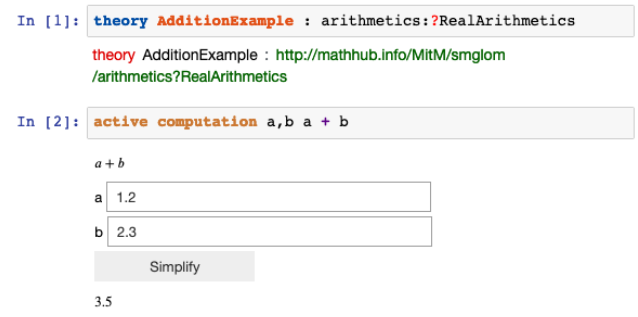

Fig. 4. An Active Computation widget in a Notebook via Jupyter/MMT Widgets

Our widget is then triggered in `In[2]` by the special command `active computation`. It takes two parameters, a list of variables (here a and b) and a term (here $a + b$).

These parameters are sent to the middle layer of our MMT kernel (see again Figure 3). This Scala code then parses the parameters (using the bottom layer), and instructs the middle layer to create a label and a text field for each variable. Furthermore, it also instructs the python code to create a button labeled `Simplify` and registers a callback inside the Scala code to be executed when the button is pressed. The labels, input fields and the button being used here are standard Jupyter widgets.

In our case the user has already entered some terms, 1.2 for a and 2.3 for b, and already clicked the `Simplify` button. This triggered the previously registered callback in the middle layer. The function first used the bottom layer to parse the terms inside the input fields. It then substituted the results into the original term $a + b$. The result of this substitution (in this case $1.2 + 2.3$) was then

simplified (again using bottom layer code). This resulted in the final output of 3.5.

The important take-away here is not the difficulty of the computation[1]; it is the seamless integration between the frontend, top, middle and bottom layer code. This example demonstrates that 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.

3 Jupyter Notebooks in MathHub

We now discuss the integration of Jupyter Notebooks into the MathHub system.

3.1 Overview

The Jupyter-extended MathHub system consists of four components:

1. A GitLab repository hosting server `https://gl.mathhub.info` that provides persistent storage of documents in any format, including their OMDoc representation.
2. A Jupyter Notebook server `https://jupyter.mathhub.info` provides web-based IDE for editing interactive documents
3. An MMT instance which uses the OMDoc representations to provides the shared knowledge space and provides a high-level API for it[2].
4. The MathHub frontend `https://mathhub.info` that serves as the main entry point and delegates some subtasks to the former. We have extended MathHub front-end with a new document type presenter for notebooks that gives access to the source, context, statistics, and metadata of notebooks, and provides a "preview" and "interact inline" views.

The Jupyter server is an out of the box installation of Jupyter except for additionally supporting our new MMT kernel and a small plugin enabling smoother opening of notebooks via a URL. Consequently, the integration between the Jupyter and the MathHub frontends is shallow: MathHub opens Jupyter Notebooks in separate tabs or iframes using URLs served by Jupyter.

[1] In our current implementation we compute using MMT, which models it of using term simplification. However in principle it is possible to use any kind of computation engine here. We want to integrate the active computation widget with our work on the Math-In-The-Middle paradigm (such as in [D6.518]) which would be ideally suited for further applications.

[2] Technically, each kernel has a separate MMT instance in addition to the primary one. Except for the ephemeral document representing each Notebook, these are identical to the main instance. These exist only to isolate different users from one another, and prevent scenarios where they could unintentionally break each others notebook sessions.

3.2 Notebooks as Parts of Semantic Documents

To interact dynamically with content in arbitrary MathHub documents, we can make use of the active computation widget presented in Section 2.5. For this purpose we implemented a new feature that creates a new ephemeral Jupyter Notebook and allows accessing it from the current document. Importantly, the new Notebook is pre-filled with an import of the current context.

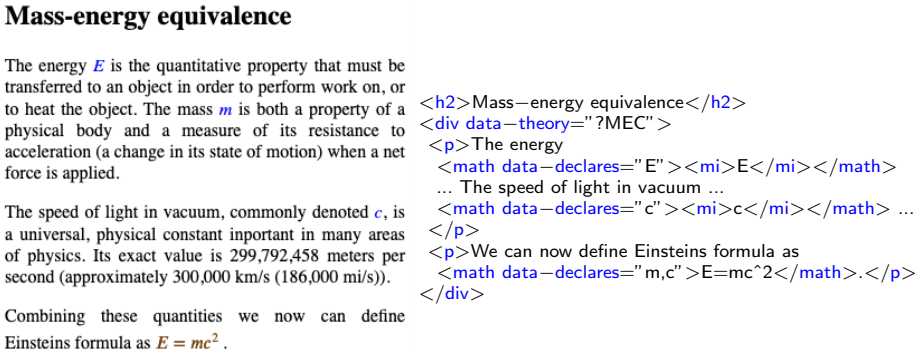

Fig. 5. A semantic HTML document and an abbreviated version of the source code

Figure 5 shows a (simplified) scientific HTML document (on the left) and an extract of its source code (on the right). The document contains the equation $E = mc^2$. The user can use the context menu to trigger the notebook generation on this formula.

This scientific document is semantically annotated. Most notably, the formula that the user can interact with defines the variables that the user might want to interactively change using the `data-declares` attribute. Futhermore, the document contains a reference to an MMT context (using the `data-theory` attribute). This gives semantic meaning to the formula.

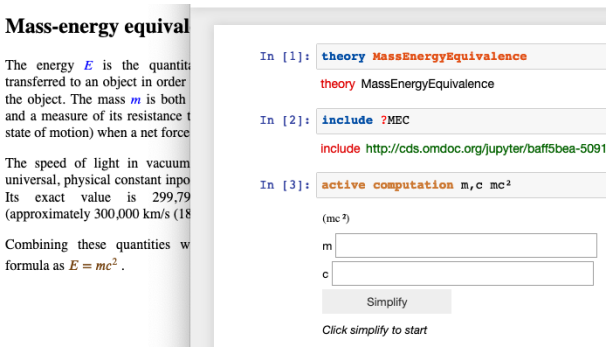

Fig. 6. The resulting Jupyter notebook/widget

The context menu is generated using JavaScript that picks up on these annotations. Currently the author has to manually create the formula and context annotations, but we are working on a mechanism to automatically create it from the document context. The data is then sent to our Jupyter installation using appropriate URL parameters.

Figure 6 shows the notebook created by our tool. This notebook starts with an `include` declaration of the document context. These are generated by our tool to obtain a minimal standalone MMT theory in which the respective formula is well-formed. The notebook then directly instantiates the active computation widget we presented above using the parameters extracted from the document.

In this demonstration we directly show the Jupyter Notebook to the user in a separate window. 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.

4 Applications

The immediate application of the Jupyter/MMT integration presented in this paper is interacting with MMT in a REPL. The MMT tool ecosystem only really supported IDE-interaction with MMT libraries via JEdit and (recently) IntelliJ IDEA plugins. While the MMT system provides a simple shell for interaction, this was only used for configuration and setup of the MMT process. We anticipate that the REPL-like interaction will feel more natural for users of interactive theorem provers and computer algebra systems. Even for the new MathScheme-style of specifying theory graph libraries via theory combinators [SR19] e.g.,

$$\mathsf{semigroup} = \mathsf{extend}\ \mathsf{magma}\ \mathsf{by}\ \{\mathsf{assoc}\colon \vdash \forall a, b, c : G.a \circ (b \circ c) = (a \circ b) \circ c\}$$

is well-suited to development/experimentation in a REPL followed by generating an OMDoc file from the recorded notebook.

4.1 Towards a Virtual Research Environment based on the Math-in-the-Middle Paradigm

Another direct application is in the context of the OpenDreamKit project, which integrates various independently developed computational engines into a mathematical virtual research environment following the Math-in-the-Middle (MitM) approach [Deh+16]. This 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[3] 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.

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

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.

```
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 |
```

Fig. 7. A Theory for LMFDB/MMT Interaction

```
In [ ]: theory Example : http://mathhub.info/MitM/Foundation?Logic
In [ ]: include http://mathhub.info/MitM/smglom/elliptic_curves?Elliptic_curve
In [ ]: include http://mathhub.info/MitM/smglom/elliptic_curves?Conductor
In [ ]: include http://lmfdb.org/db/elliptic_curves?curves
In [ ]: include http://lmfdb.org/db/transitivegroups?groups
In [ ]: mycurve: elliptic_curve = `http://lmfdb.org/db/elliptic_curves?ec_curves?11a1
In [ ]: c: int_lit = conductor mycurve
In [ ]: mygroup: group = `http://lmfdb.org/db/transitivegroups?groups?8T3
```

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

4.2 Domain specific applications e.g., 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 (usually in terms of physical principles), model knowledge (e.g., about suitable partial differential equations) with simulation (i.e., numerics/computing) knowledge. In current practice, this is resolved by intense collaboration between experts, which incurs non-trivial translation and communication overheads. With the infrastructure presented in this paper, we can do better. In fact, the MoSIS application was developed in parallel to our Jupyter/MathHub integration and MoSIS requirements helped inform the the

development. We have ported the original version [PKK18] to the new infrastructure, simplifying and extending it in the course. All in all, the interaction part of the MoSIS project would now be a straightforward software development exercise instead of a contribution of its own.

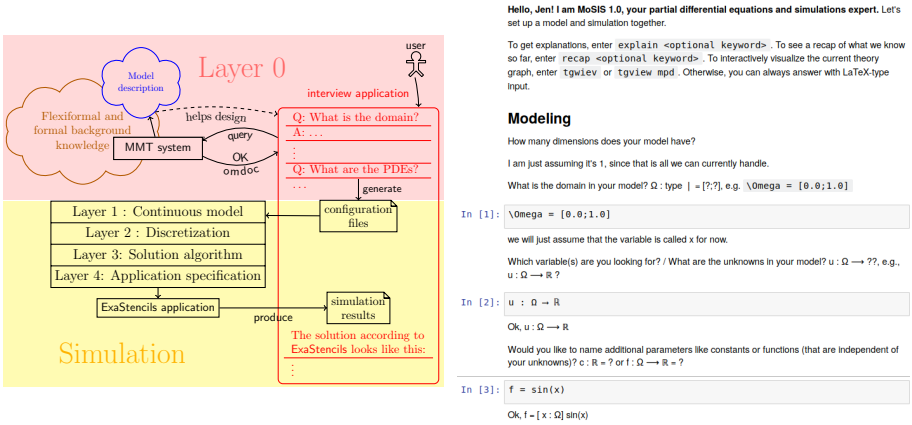

Fig. 9. MoSIS Information Architecture and Dialogue

Concretely, MoSIS uses a Jupyter notebook that has access to an MMT theory graph on MathHub.info. 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.

5 Conclusion and Future Work

We have presented an integration of two interaction paradigms in mathematical software systems: document-based and computation-oriented interactions. Concretely, we have implemented an integration of three systems: 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.

Even though the work presented in this paper lays the foundation towards an integration of the static/dynamic paradigms for the interaction with mathematical knowledge, at lot of practical enhancements remain for future work. We sketch the most important ones here:

Deeper MathHub/Jupyter Integration e.g., using Jupyter simply as a JavaScript library in MathHub. This would have been preferable to the current iFrame-based integration, but is infeasible because Jupyter is primarily designed as a monolithic system. Recent versions of Jupyter are working towards a Jupyter-as-a-module design, so we leave deep integration to future work.

IDE Support for Documents with Active Computation Currently, the semantic documents like the one in Figure 5 have to be manually extended by the pertinent semantic annotations. An extension of the sTEX framework would allow authors e.g., of educational documents to directly manage the annotations in the LaTeX sources.

REPL Cells/Documents as first-class citizens in MMT We already use the notebook-to-MMT-document isomorphism in our system. A first-class model of REPL cells in MMT – this will need a considerable language design effort – would allow to strengthen this isomorphism and refactor our system. We expect that first-class REPL cells in MMT would allow enhanced IDE support for MMT notebooks.

More Flexible Active Computation The current widget is relatively inflexible in terms of the objects it allows to change for computation. In principle, all variables and constants from the context could be used. We will need more user experience to generalize our current design.

TGView/Notebook Integration The MMT Jupyter kernel is fundamentally co-dependent on the background theory graph. Therefore we want to explore an integration of the TGView graph viewers [MKR] into the Jupyter front-end. Both Jupyter and TGView are based on REACT.JS, so this should be feasible

Mathematical Search on Notebooks Last, but not least, we want to extend mathematical search on MathHub to Jupyter notebooks by extending the MathWebSearch harvester accordingly.

References

[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.

[Deh+16] Paul-Olivier Dehaye et al. "Interoperability in the OpenDreamKit Project: The Math-in-the-Middle Approach". In: *Intelligent Computer Mathematics 2016*. Ed. by Michael Kohlhase et al. LNAI 9791. Springer, 2016. ISBN: 978-3-319-08434-3. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/CICM2016/published.pdf.

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

[JD] *What is Jupyter*. URL: http://jupyter-notebook-beginner-guide.readthedocs.org/en/latest/what_is_jupyter.html (visited on 08/22/2017).

[JND] *The Jupyter Notebook Format*. URL: https://nbformat.readthedocs.io/en/latest/ (visited on 03/13/2018).

[MKR] Richard Marcus, Michael Kohlhase, and Florian Rabe. "TGView3D System Description: 3-Dimensional Visualization of Theory Graphs". URL: https://kwarc.info/kohlhase/submit/tgview3D.pdf.

[MMT] Florian Rabe. *The MMT System*. URL: https://uniformal.github.io/doc/ (visited on 07/16/2014).

[MMTJup17] Tom Wiesing and Kai Amann. *mmt_jupyter_kernel: A Jupyter kernel for MMT*. Oct. 16, 2017. URL: https://github.com/UniFormal/mmt_jupyter_kernel (visited on 11/08/2017).

[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*. Ed. by Florian Rabe et al. LNAI 11006. Springer, 2018. ISBN: 978-3-319-96811-7. DOI: 10.1007/978-3-319-96812-4.

[SR19] Yasmine Sharoda and Florian Rabe. "Diagram Operators in MMT". In: *Intelligent Computer Mathematics (CICM) 2019*. Ed. by Cezary Kaliszyck et al. in preparation. 2019.