REPORT ON OpenDreamKitDELIVERABLE D4.2

Active/Structured Documents Requirements and existing Solutions

MICHAEL KOHLHASE

Due on	05/31/2016 (Month 6)
Delivered on	27/06/2016
Lead	Jacobs University Bremen (JacobsUni)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/91`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #91 ON 2017-01-09

- **WP4:** User Interfaces
- **Lead Institution:** Jacobs University Bremen
- **Due:** 2016-02-29 (month 6)
- **Nature:** Report
- **Task:** T4.6 (#74)
- **Proposal:** p. 47
- **Final report**

One of the most prominent features of a virtual research environment (VRE) is a unified user interface. The OpenDreamKit approach is to create a mathematical VRE by integrating various pre-existing mathematical software systems. There are two approaches that can serve as a basis for the OpenDreamKit UI: *computational notebooks* and *active documents*. The former allow mathematical text around the computation cells of a real-eval-print loop of a mathematical software system and the latter make semantically annotated documents semantic.

We report on two systems in the OpenDreamKit project: Jupyter – a notebook server for various kernels, and MathHub.info – a platform for active mathematical documents. We identify commonalities and differences and develop a vision for integrating their functionalities.

Related projects:

- MathBookXML: `http://mathbook.pugetsound.edu/`
- Jupyter notebook exporter for Sphinx:
  `https://github.com/sphinx-doc/sphinx/pull/2117`
- Jupyter javascript plugin for static sites:
  `https://github.com/oreillymedia/thebe`
  See also: `https://www.oreilly.com/ideas/jupyter-at-oreilly`
- ReST to IPython Notebook converter through pandoc and markdown:
  `https://github.com/nthiery/rst-to-ipynb/`

CONTENTS

1. INTRODUCTION

One of the most prominent features of a virtual research environment (VRE) is a unified user interface. The OpenDreamKitapproach is to create a mathematical VRE by integrating various pre-existing mathematical software systems. There are two approaches that can serve as a basis for the OpenDreamKitUI: "computational notebooks" and "active documents". The former allow mathematical text around the computation cells of a real-eval-print loop of a mathematical software system and the latter make semantically annotated documents semantic.

We report on two systems in the OpenDreamKitproject: Jupyter[Jup] – a notebook server for various kernels, and MathHub.info – a platform for active mathematical documents. We identify commonalities and differences and develop a vision for integrating their functionalitities.

The report is organized as follows: Sections 2 and 3 introduce the two systems individually. Section 4 relates them under a common perspective and uses that to develop a vison of a joint generalization that combines the advantages. Section 5 concludes the report.

2. JUPYTER

2.1. **Introduction**

Project Jupyter [Jup; JupDoc; JupGit] aims to support interactive data science and scientific computing across all programming languages. The notebook [JupNotebook] extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The Jupyter notebook combines two components:

(1) A web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.
(2) Notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

[JupLit] gives an overview over the best practices for writing (and using) Jupyter notebooks.

2.2. **What is there?**

2.2.1. *Introduction.* Jupyter is a web application that allows the user to create and share documents that contain live code, equations, visualizations and explanatory text.

This notebook has among its applications data cleaning and transformation, numerical simulation, statistical modelling and machine learning. It extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

The code that is suitable for this notebook can be written in more than 40 programming languages, including the most popular ones used in Data Science, such as Python, R, Julia and Scala.

2.2.2. *History.* Project Jupyter was born out of the IPython Project [IPh; WikiIPh] in 2014 as it evolved to support interactive data science and scientific computing across all programming languages. IPython continues to exist as a Python shell and a kernel for Jupyter, while the notebook and other language-agnostic parts of IPython moved under the Jupyter name.

IPython was a command shell for interactive computing in multiple programming languages, originally developed only for Python. It offered introspection, rich media, shell syntax, tab completion, and history. This interactive computing system provided the following features:

- Interactive shells (terminal and Qt-based).
- A browser-based notebook with support for code, text, mathematical expressions, inline plots and other media.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into one's own projects.
- Tools for parallel computing.

Jupyter evolved from this and was divided into three major components:

(1) A browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output. This application included and extended the features of IPython, leading to new ones:
    - In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
    - The ability to execute code from the browser, with the results of computations attached to the code which generated them.

- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library, can be included inline.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

(2) Notebook documents

A representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

Notebook documents contain the inputs and outputs of an interactive session as well as narrative text that accompanies the code but is not meant for execution. Rich output generated by running code, including HTML, images, video, and plots, is embedded in the notebook, which makes it a complete and self-contained record of a computation.

Notebooks consist of a linear sequence of cells. There are three basic cell types:

- Code cells: Input and output of live code that is run in the kernel
- Markdown cells: Narrative text with embedded LaTeX equations
- Raw cells: Unformatted text that is included, without modification, when notebooks are converted to different formats such as LaTeX via nbconvert.

Notebooks can be exported to different static formats including HTML, reStructeredText, LaTeX, PDF, and slide shows. Furthermore, any notebook document, available from a public URL qor GitHub, can be shared via "*nbviewer*". This service loads the notebook document from the URL and renders it as a static web page. The resulting web page may thus be shared with others without their needing to install the Jupyter Notebook.

(3) Kernels:

Through Jupyter's kernel and messaging architecture, the Notebook allows code to be run in a range of different programming languages. For each notebook document that a user opens, the web application starts a kernel that runs the code for that notebook. Each kernel is capable of running code in a single programming language and there are kernels available for languages including the following:

- Python: `https://github.com/ipython/ipython`
- Julia: `https://github.com/JuliaLang/IJulia.jl`
- R: `https://github.com/takluyver/IRkernel`
- Ruby: `https://github.com/minrk/iruby`
- Haskell: `https://github.com/gibiansky/IHaskell`
- Scala: `https://github.com/Bridgewater/scala-notebook`
- Node.js: `https://gist.github.com/Carreau/4279371`
- Go: `https://github.com/takluyver/igo`

The default kernel runs Python code. The notebook provides a simple way for users to pick which of these kernels is used for a given notebook.

Each of these kernels communicate with the notebook web application and web browser using a JSON over ZeroMQ/WebSockets message protocol.

2.2.3. *Communication Design.* This subsection explains the basic communications design and messaging specification for how the various Jupyter objects interact over a network transport. The current implementation uses the ZeroMQ library for messaging within and between hosts.

A single kernel can be simultaneously connected to one or more frontends. The kernel has three sockets that serve the following functions:

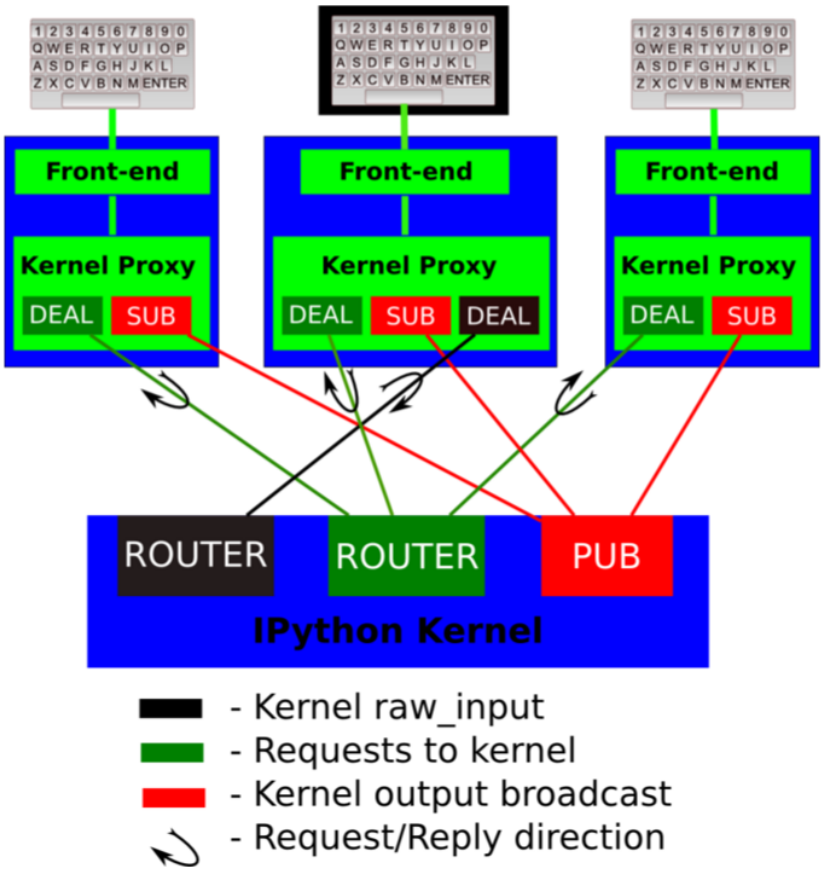

FIGURE 1. The basic design explained

(1) Shell: This single ROUTER socket allows multiple incoming connections from frontends, and this is the socket where requests for code execution, object information, prompts, etc. are made to the kernel by any frontend. The communication on this socket is a sequence of request/reply actions from each frontend and the kernel.

(2) IOPub: This socket is the ”broadcast channel” where the kernel publishes all side effects (stdout, stderr etc.) as well as the requests coming from any client over the shell socket and its own requests on the stdin socket. There are a number of actions in Python which generate side effects: *print()* writes to **sys.stdout**, errors generate tracebacks etc. Additionally, in a multi-client scenario, all desire from frontends to be able to know what each other has sent to the kernel (this can be useful in collaborative scenarios, for example). This socket allows both side effects and the information about communications taking place with one client over the shell channel to be made available to all clients in a uniform manner.

(3) stdin: This router socket is connected to all frontends, and it allows the kernel to request input from the active frontend when *raw-input()* is called. The frontend that executed the code has a DEALER socket that acts as a *”virtual keyboard”* for the kernel while this communication is happening (illustrated in figure 1 by the black outline around the central keyboard). In practice, frontends may display such kernel requests using a special input widget or otherwise indicating that the user is to type input for the kernel instead of normal commands in the frontend.

All messages are tagged with enough information for clients to know which messages come from their own interaction with the kernel and which ones are from other clients, so they can display each type appropriately.

(4) Control: This channel is identical to Shell, but operates on a separate socket, to allow important messages to avoid queueing behind execution requests (i.e. shutdown or abort).

Messages are dicts of dicts with string keys and values that are reasonably representable in JSON. The current implementation uses JSON explicitly as its message format, but this should not be considered a permanent feature. As the JSON has non-trivial performance issues due to excessive copying, it is plausible that in the future it could move to a pure pickle-based raw message format. However, it should be possible to easily convert from the raw objects to JSON, since there may be non-python clients (i.e. a web frontend). As long as it's easy to make a JSON version of the objects that is a faithful representation of all the data, a valid communication can be established with such clients.

2.3. **User Interface**

2.3.1. *Notebook Dashboard.* When the jupyter notebook is launched the first page encountered is the Notebook Dashboard.

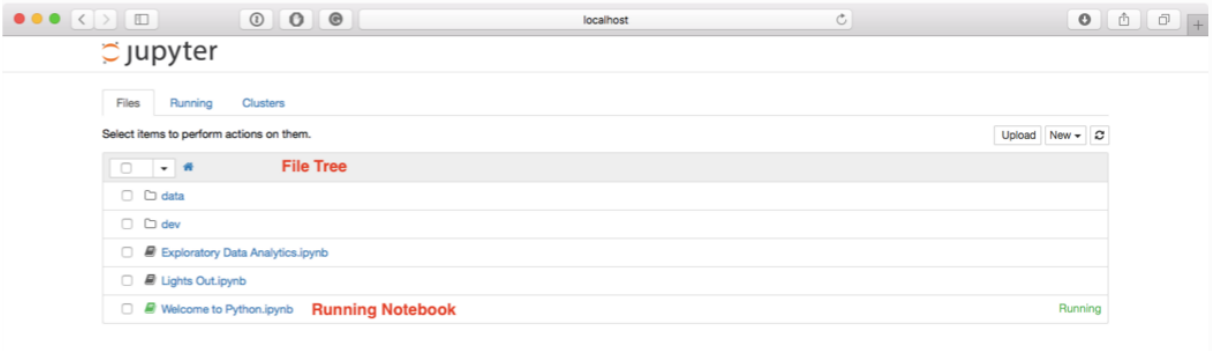

FIGURE 2. Jupyter Dashboard

2.3.2. *Notebook Editor.* Once the user selected a Notebook to edit, the Notebook will open in the Notebook Editor.

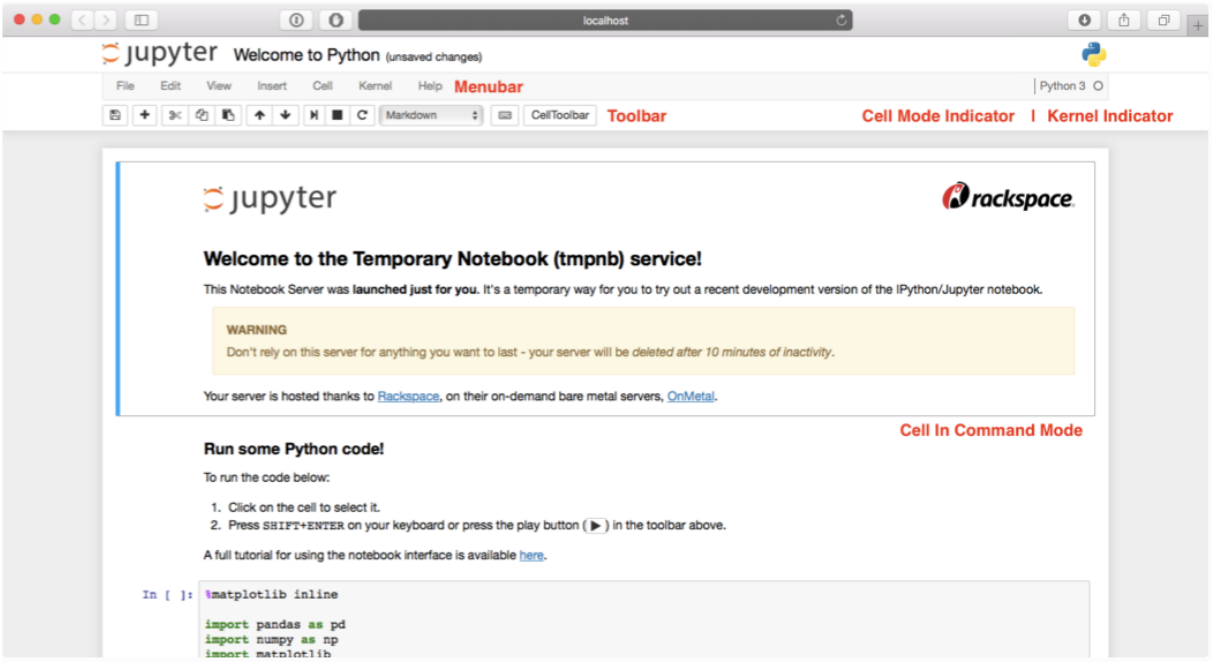

FIGURE 3. Jupyter Editor

2.3.3. *Interactive User Interface Tour of the Notebook.* If the user would like to learn more about the specific elements within the Notebook Editor, it is possible to take the User Interface Tour by selecting Help in the Menu Bar then selecting User Interface Tour.

2.3.4. *Edit Mode and Notebook Editor.* When a cell is in edit mode, the Cell Mode Indicator will change to reflect the cell's state. This state is indicated by a small pencil icon on the top right of the interface. When the cell is in command mode, there is no icon in that location.

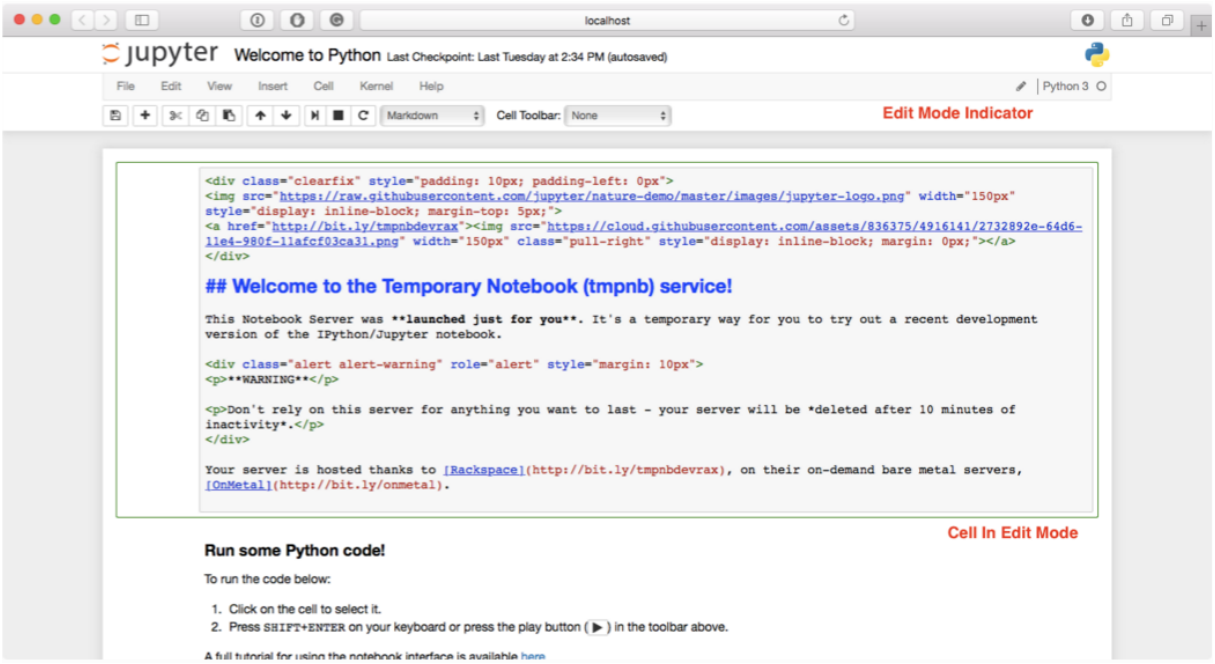

FIGURE 4. Jupyter Edit Mode

2.3.5. *File Editor.* Now let's say that the user has chosen to open a Markdown file instead of a Notebook file whilst in the Notebook Dashboard. If so, the file will be opened in the File Editor.

2.4. Future Plans for Jupyter

2.4.1. *Embracing web standards.* By being a pure web application using HTML, Javascript, and CSS, the Notebook can get all the web technology improvement for free. Thus, as browser support for different media extend, the notebook web app should be able to be compatible without modification.

This is also true with performance of the User Interface as the speed of Javascript VM increases.

The other advantage of using only web technology is that the code of the interface is fully accessible to the end user and is modifiable live. The project strives to keep its code as accessible and reusable as possible. This should allow the developers to develop with minimum effort small extensions that customize the behaviour of the web interface.

2.4.2. *Tampering with the Notebook application.* The first tool that is available to the user and that which should be aware of are the browser "developers tool". The exact naming can change across browser and might require the installation of extensions. But basically they can allow inspection/modifications to the DOM, and interact with the JavaScript code that runs the frontend.

2.4.3. *Extending the Notebook.* Certain subsystems of the notebook server are designed to be extended or overridden by users. These documents explain these systems, and show how to override the notebook's defaults with someone's own custom behaviour.

2.4.4. *Contents API.* This section describes the interface implemented by ContentsManager subclasses. We refer to this interface as the Contents API.

The Jupyter Notebook web application provides a graphical interface for creating, opening, renaming, and deleting files in a virtual filesystem.

The ContentsManager class defines an abstract API for translating these interactions into operations on a particular storage medium. The default implementation, FileContentsManager, uses the local filesystem of the server for storage and straightforwardly serializes notebooks into JSON. Users can override these behaviors by supplying custom subclasses of ContentsManager.

3. Active Documents

3.1. Introduction

We define the Active Documents as semantically annotated documents associated with a content commons that holds the corresponding background ontologies. An *Active Document Player* embeds user-visible, interactive services like program execution, computation, visualization, navigation, information aggregation and information retrieval to make documents executable. We call this framework the Active Documents Paradigm (ADP; see Figure 5), since documents can also actively adapt to user preferences and environment rather than only executing services upon user request. The ADP is implemented in the Active Documents Portal MathWeb.info [MH] building on standard components as an instance of the Planetary system [Koh12]; see Section 3.5 below.

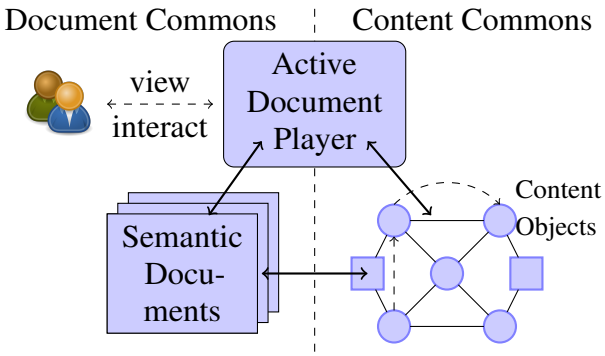

FIGURE 5. Active Documents

We present the ADP with a focus on three distinct annotation levels, *presentation structure*, *semantic* and *formal*.

3.2. Presentation Structure

The importance of the presentation structure level is that an active document player can turn legacy documents into active documents by transforming them into XHTML+MathML+SVG-encoded documents with semantic annotations in RDFa.

We have transformed over half a million articles from the Cornell ePrint arXiv to XHTML+MathML with LaTeXML, preserving properties like document and formula structures and embedded them into an instance the Planetary System, an active documents player.

The document structure can then be exploited for a FoldingBar service (see on the left in Figure 6) and for localizing discussions about document content to document structures and subformulae – e.g. for questions/answers, or reviewers' comments. In the situation in Figure 6 we have clicked on the formula, which pops up the IconMenu with three options: reporting errors in the content (bug icon), asking/answering a question (question mark icon), and accessing the discussion threads of this element (balloons icon). Here, a click on the question mark icon allowed us to pose a question and hope for an answer by other users in the forum. Figure 6 also shows the Planetary InfoBar with information markers on the right, which indicate the availability and state of the discussion threads pertaining to information objects in the line they are horizontally aligned with. Clicking them will highlight all items that have discussions. Localized discussions have proven a very valuable tool for community-based validation of papers, especially if they are coupled with a discussion subscription/trackback system for readers and personal notification system for authors.

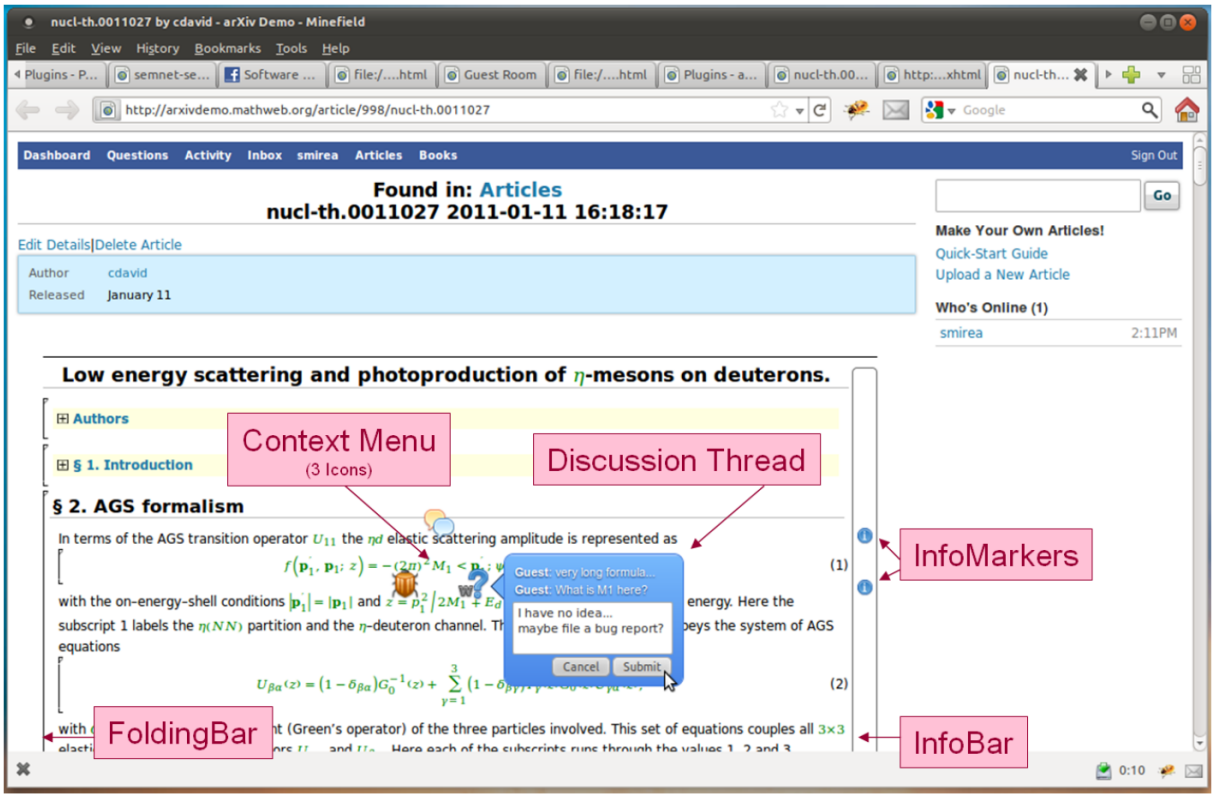

FIGURE 6. Interacting with an article

3.3. **Semantic Level**

We can considerably improve the user experience by extending the depth of semantic annotations. For this we employ OMDoc (Open Mathematical Documents), an XML-based content-oriented, semi-formal representation format for scientific and technical documents.

It builds on the OpenMath/MathML3 semantic representation format for mathematical formulae. OMDoc extends OpenMath with an infrastructure for context and domain models from Formal Methods, as well as a generic document infrastructure. At the semantic level Planetary is based on STEX documents, which can be transformed to OMDoc and via a user-adaptive and context-based presentation process further to XHTML+MathML+SVG+RDFa. The generated OMDoc documents are committed to an instance of the versioned XML database TNTBase that indexes them by semantic functional criteria, and can then perform server-side semantic services via user-defined XQuery queries.

TNTBase thus becomes a source of the user-adaptive, custom-generated documents forming Planetary's content commons. Many semantic services can directly be derived from this setup. At the semantic level, the IconMenu we know from Figure 6 can be extended, depending on the services available for the semantic item in focus. The book icon triggers definition lookup (see item i) below) and the graph and portrait icons prerequisites navigation (see iii)). Figure 7 shows results of these and other services.

(1) *Definition Lookup*: All technical terms and symbols in formulae presented in Planetary are linked to their semantic counterparts in the content commons, which in turn are linked to their definitions.

(2) *Semantic Folding*: If any explanations of the meanings of subformulae have been added as annotations, folding can use these instead of ": : :". In Figure 6/ii) the motion law above is semantically folded to

$$s_0 + s_v + s_a$$

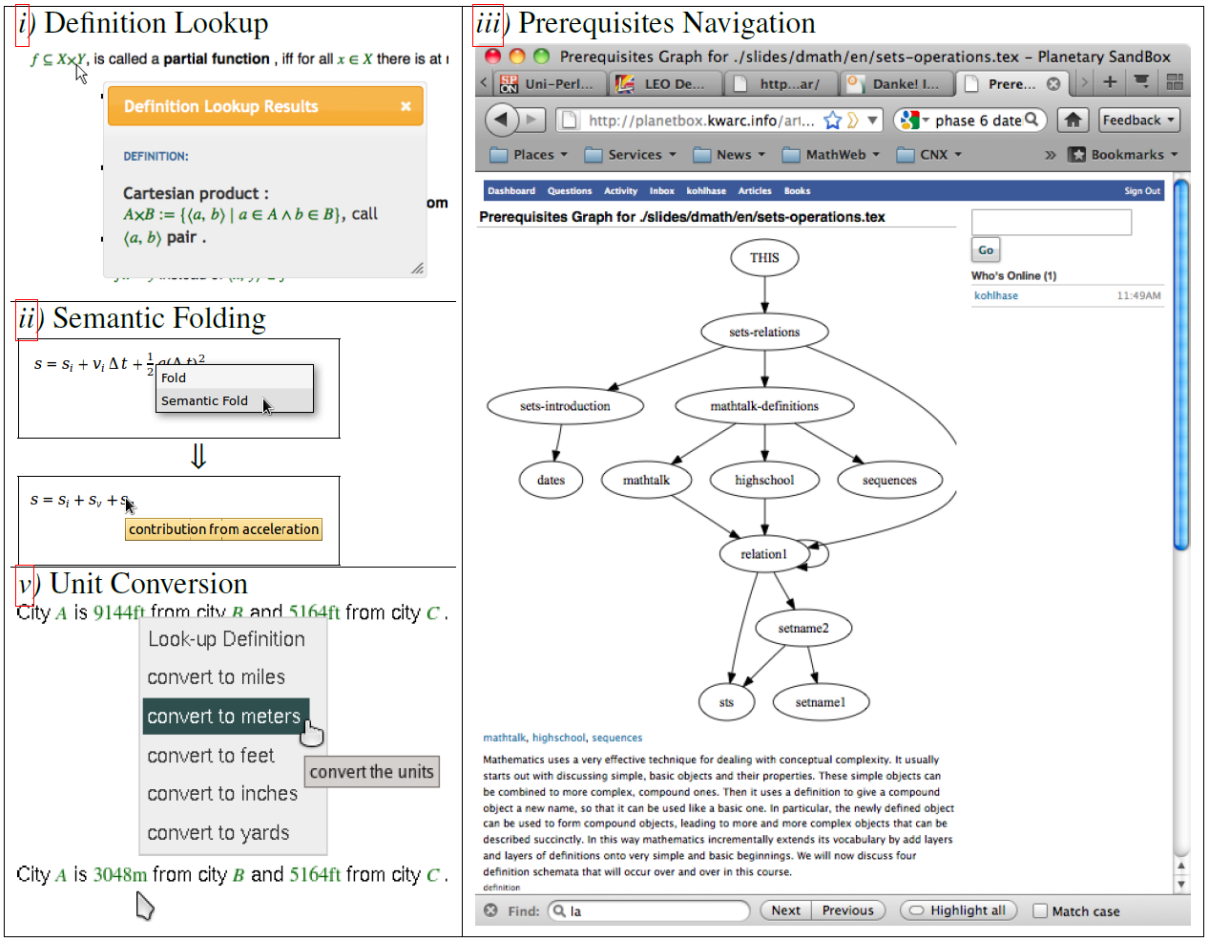

FIGURE 7. User Services at the Semantic Level

and the abbreviations

$$s_*$$

are explained via flyover help.

(3) *Prerequisites Navigation*: As the content commons has an inherent notion of semantic dependency, we can use that to show prerequisites leading to a concept. Currently Planetary supports two ways of dealing with prerequisites: i) a concept graph view, where the required concepts can be navigated on demand by clicking on concept nodes, and ii) guided tours, where the necessary content is generated in a coherent narrative.

(4) *Executable Formulae*: As the formulae in the documents are generated from OpenMath objects, we can export them to a computer algebra system like Mathematica (or in our case the open-source system GAP, which has been made available as an OpenMath-based Web service) for evaluation, graphing, or experimentation.

(5) *Unit Conversion*: In scientific papers, formulae often contain expressions for measurable quantities; these can be automatically converted to other unit systems.

Note that the underlying OMDoc format and the services based on it address a peculiarity of documents in the STEM disciplines. Here, documents are dynamic in the sense that they declare new concepts, definitions, model assumptions, terminology, notations, etc., as they go along, or else they explicitly (or implicitly) import such items from other documents. As a consequence, all knowledge items in STEM documents have a non-trivial context of declarations. This must be managed explicitly in our representations of these documents, in the content commons, and within user interaction in order for semantic services to be effective. This effect is especially pronounced in mathematical sciences (including Computer Science, Physics etc.), and somewhat

less so in Chemistry and the Life Sciences, where a global context in the form of external terminology and notation databases often suffices.

3.4. **Formal Level**

Finally, we can use Planetary as a frontend system for completely formal content. OMDoc is also a foundation-agnostic integration format for mathematical knowledge that can express web ontologies, program specifications and verifications, and even representations of logics in logical frameworks like LF. In formal systems, documents are dominated by complex formulae, and users need support in navigating, abstracting, and evaluating them in order to cope with this complexity. Figure 8 shows how fully formalized formulae can be adapted to user preferences, about, e.g., the level of brackets and the availability of inferred arguments or definitions. Further semantic services at the fully formalized level include access to automated theorem provers via the HETS system and argument reconstruction via the Twelf system.

It is important to note that programs (and program fragments) are also knowledge items at the fully formal level, as all their semantics can be recovered by parsing. Program fragments are not yet supported by Planetary.

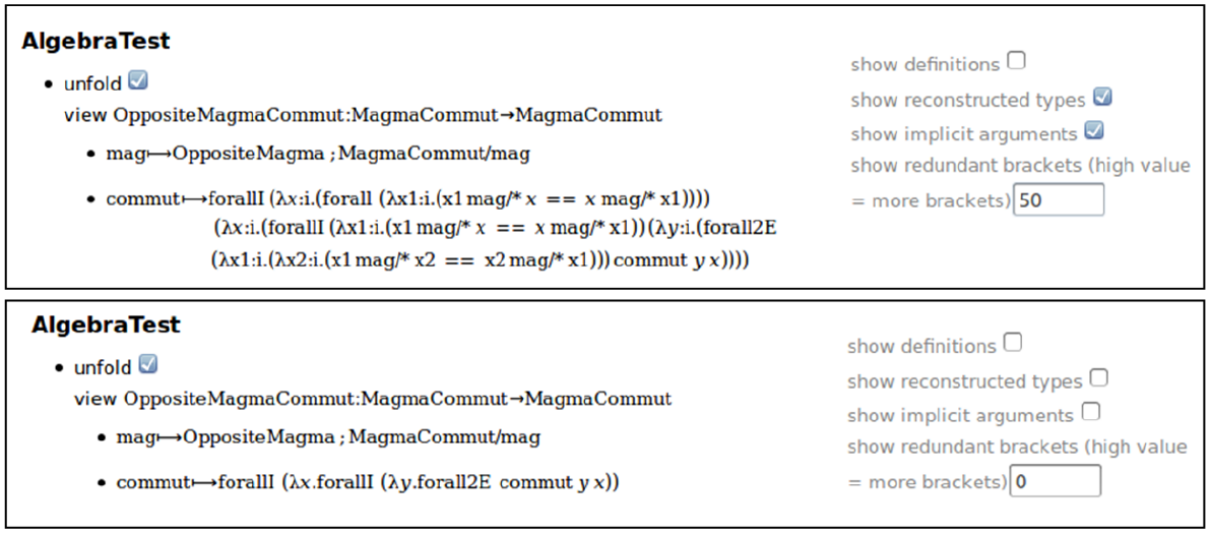

FIGURE 8. Formal Representations Adapted to Distinct User Settings (Customized via the DocDash Widget on the Right)

3.5. **Architecture**

MathHub.info has four main components (see Figure 9):

i) a versioned *backend* holds the libraries,
ii) the MMT API as the kernel tool understands the libraries provides semantic services for them,
iii) a web-based *frontend* makes the libraries and services available to users,
iv) a Javascript *plugin architecture* enriches document presentations with localized semantic services.

We use best-of-breed open source systems for the components going beyond MMT. In the backend, we use GIT for versioning, distribution, and user/rights management adapting the GitLab repository manager [GL], an open-source alternative to GitHub. For the frontend, we use the Drupal container management system.[1] For the Javascript library we use our JOBAD framework [GLR09; Koh12], which embeds semantic services into HTML documents and thus

[1]Drupal and similar systems self-describe as "content management systems", but they actually only manage the documents and their metadata – essentially document containers – without changing their internal structure.

makes them interactive and user-adaptive. Even though JOBAD is just a relatively thin layer of glue code that picks up on semantic annotations in the generated HTML5, its effect for the users is rather profound: It gives them access to added-value services "at the point of pain/interest", i.e., in the user interface. Figure 7 already shows JOBAD in action: it links fragments of the formula presentations with computations in the MMT system and makes both available to the user embedded in the document.

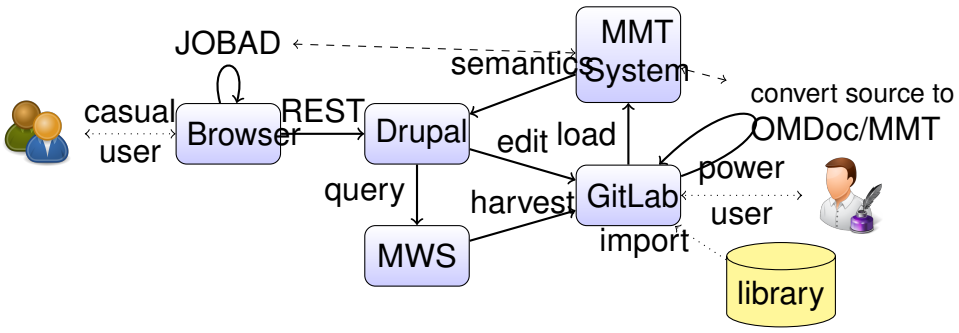

FIGURE 9. The modular MathHub.info Architecture

Figure 9 shows the detailed architecture. Here GitLab provides distributed versioned storage of the libraries and organizes them into repositories owned by users and groups. And Drupal supplies uniform theming, discussion forums, and a plugin infrastructure for adding interface functionality. Both systems provide user management, but we automatically synchronize the users and permissions between them, so that GitLab becomes invisible to the casual user.

This componentized architecture has the advantage that we can combine two methods for accessing the contents of MathHub.info: *i*) an online, web-based workflow for the casual user, and *ii*) an offline authoring workflow based on git working copies for power users and bulk edits. Users can fork or pull the relevant repositories from GitLab, edit them, and submit them back to MathHub.info either via a pull request to the repository masters or a direct commit/push. As the content is often highly interlinked and distributed across multiple interdependent repositories, we have developed tool support for managing multiple working copies across repository borders.

In the web-based system, semantic services (notation-based, presentation, definition lookup, relational navigation, dependency management, etc.) are provided by MMT and are made available to the user, primarily by dedicated JOBAD [GLR09] modules. The interactive functionalities in MathHub.info are based on the OMDoc/MMT representation of the libraries, but authors and users have to interact with them in the respective source language of the library. Both the source and OMDoc/MMT representations are versioned in GitLab and the respective source representations must be converted into OMDoc/MMT by language-specific custom exporters. Correspondingly library import is managed by MathHub.info at the level of the GitLab repository.

Then we can dedicate a specific GIT working copy together with an MMT instance to a user or a group that shares permissions. Thus, the MMT instance sees (and takes into account for its services) only the documents accessible to the group. If an authenticated user edits MathHub.info content, the changes are committed under his name into the specific working copy. This makes it easy to cope with multiple synchronous users, for which MathHub.info uses separate working GIT clones and MMT instances.

3.6. **Conclusion**

The Active Documents stand for semantically annotated documents connected to a content commons accessed through an adaptive document player , such as Planetary. The Planetary system is presented as an answer to the challenge of creating "executable papers". We have shown the initial feasibility of the concept in a variety of publicly available case studies.

Along the Planetary system, the Active Documents addresses all the following challenges:

- **Executability** is achieved by an extensible and configurable collection of semantic services that can be embedded into documents. In particular, code can be made executable via external computation machines, and computational experiments can be repeated and varied (via the same mechanism). Where the functionality of a service depends on ontologies, the user community can customize the service by customizing the ontology inside Planetary.
- **Short and long-term compatibility** is guaranteed by usage of open standards in representation formats and protocols (XML, RDF/RDFa, OpenMath, MathML, XQuery, SPARQL, XHTML, SVG) supporting a web service framework, and hence operating-system-independent. Of course, Planetary can export monographs, collections and even entire libraries both as PDF (inactive documents) or in the EPUB eBook format.
- **Validation** and in particular human refereeing and scientific validation can be facilitated via an in-text discussion feature. Moreover, documents can be automatically validated via semantic services, e.g. automated SI-dimensionality checking.
- **Copyright/licensing** is represented by fine-grained RDFa-based metadata annotations in STEX and OMDoc, which are maintained over the presentation process. So they can be used for filtering or attribution either on the backend storage level (TNTBase, RDF triple store) or in the frontend Planetary system. Together with the user management and permission system, Planetary can be extended to enforce compliance. In fact, as documents are assembled for the user at view-time they can be adapted to the license status of the user (e.g. it is possible to make a document license conforming by leaving out examples that are not licensed to her specific institution).
- **Systems** As the Planetary system is entirely based on web standards and communicates via RESTful interfaces, it is simple to wrap external systems into web services, if we can equip them with OMDoc, OpenMath, or RDF interfaces.
- **Size** Even though individual human-written documents are modest in size, journals and encyclopedias can get big – consider e.g. the Wikipedia or the arXiv. The Planetary system has been tested on the latter; the underlying data stores scale sufficiently for large document collections. Furthermore, the modular and semantic document formats accommodate the integration of external data stores via 'special' links, which the Planetary player can interpret on view, hence keeping the storage minimal and the experience optimal. To the best of our knowledge, the semantically transparent integration of data into a document player application is a new feature of the Planetary system.
- **Provenance** comes in various aspects. Data provenance can be specified by the techniques for semantic integration of data fields. For instance we can specify units (as OpenMath objects) and computations to obtain the displayed data from raw data, etc. As all of these are content representations in the documents or the content commons, they can be handled with semantic services. The system state provenance (i.e. what actions of the user led to the current state of interaction in the Planetary system), can be handled by recording system data ("who did what when") in the metadata store of Planetary. This can be opened to querying the system ontologies we have developed for semantically transparent system self-documentation. An encouraging aspect of this work is that document authors only need expertise in their own domain. In particular, no system-level programming is necessary for authors: the semantic representation formats involved act as a high level conceptual interface between content authors and system/service/interface developers. We are convinced that without such a separation of concerns, "the next generation of publishing" will not scale enough to become practical.

4. A Joint Perspective and Generalization of Jupyter and Active Documents

We will now highlight the features of the ADP and Jupyter notebooks with a view towards a possible unification of the systems.

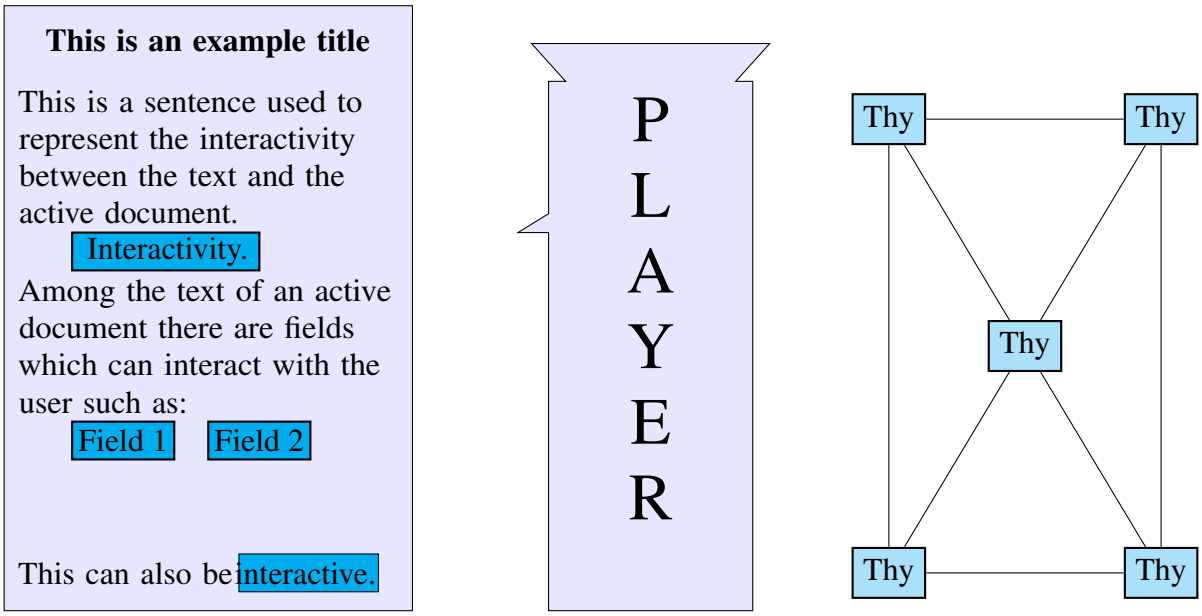

FIGURE 10. Active Documents

Active Documents need a Player process (e.g. the Planetary system) that makes them executable, gives access to provenance and copyright/licensing information, and supports various forms of validation. Figure 10 shows the situation in analogy to Figure 5: On the left we see an active document – a web page in a browser – as it is seen by the user: it contains text interspersed with regions that are interactive because they have been bound to semantic services, which are executed by the player system – in the middle – that interprets the represented content structures – the mathematical knowledge; here depicted by a theory graph on the right.

In *Jupyter* the situation is similar, the user interacts with a dynamic web page – the Jupyter notebook interface – in a browser that is a mathematical text interspersed with areas of interactivity: the computation cells. These can generate mathematical content and righ media output into the notebook upon user request. We see the notebook interface on the left of Figure 11. Again, we have a "player process" the Jupyter system and displays the text from the notebook and the computational kernel that runs the code for the notebook. Note that the notebook (source) is also a representation of mathematical knowledge; we see it on the right of Figure 11.

This already hints at a synthesis of the two systems; we make this explicit in Figure 12: We build a combined player system that combines the complementary features of both systems.

On the *user interface* side this combined player

(1) allows free-form mathematical documents with interactivity regions like in active documents, but also
(2) provides computational cells with read-eval-print style interaction with dedicated computational machines.

We have tried to indicate this in the mixed user interface in Figure 11.

On *the computational side*, it combines

(1) the generic semantic services of the MMT Tool based with
(2) dedicated computational machines running code in separate kernel processes.

Note that all of these need to share a notion of "mathematical state" so that the user interface can present a consistent view to the user.

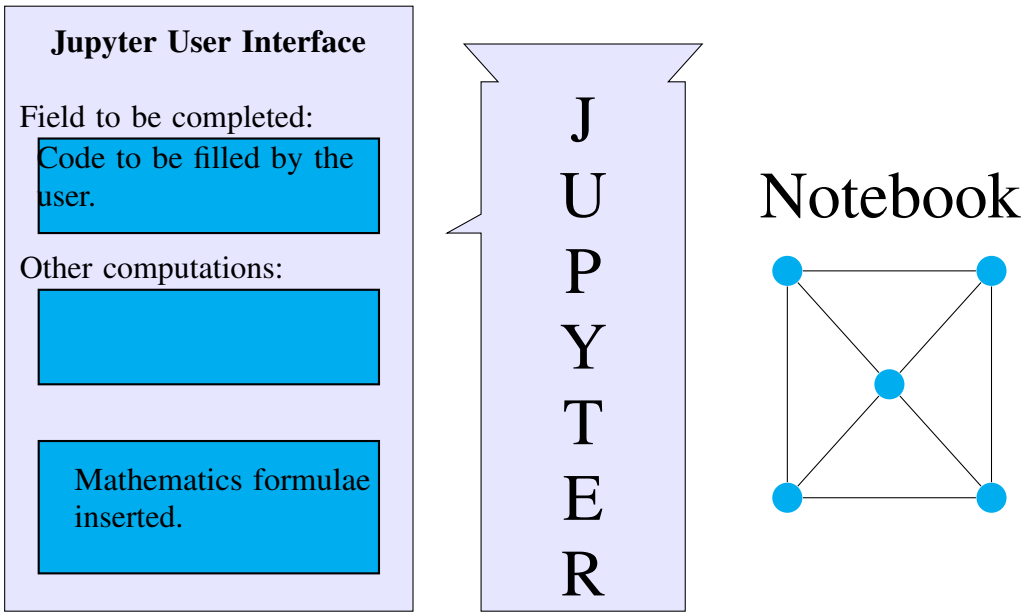

FIGURE 11. Jupyter Communication

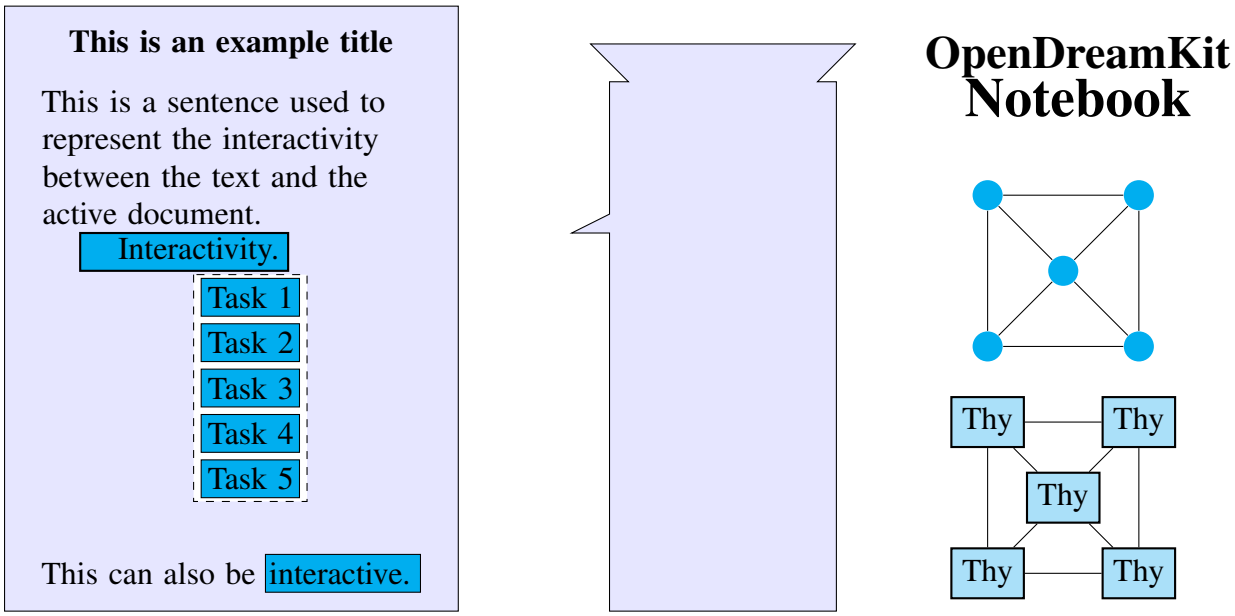

FIGURE 12. Jupyter combined with Active Documents

The new player process relies on the availability of three "declarative compondens" that also need to be in a consistent representation.

(1) mathematical document text
(2) theory-graph shaped content representations, and
(3) engine-specific code

We will call these tri-partite representations that combine the parts of Jupyter notebooks and OMDoc documents **OpenDreamKit Notebooks**.

5. Conclusion

We have surveyed the two document-formed user interfaces in the OpenDreamKitproject: Jupyter Notebooks and Active Documents and developed a joint perspective on them that allows us to propose a joint generalization oft the functionalities combines their respetive advantages. We feel that embedding computations in arbitrary places in traditional mathematical documents forms a natural generalization of the two existing user interfaces. In particular the appearance as "enhanced mathematical documents" may make the VRE more natural to mathematicians who have little experience with symbolic computation tools, as they naturally have experiences with paper articles and textbooks from their studies.

The survey and outlook provided by this report will be used as a basis for the discussion of an integrated user interface and in the OpenDreamKitproject (WP4). As integration of knowledge and computation (and interoperability between the various system involved) is a central theme of WP6, this will require interaction between the work packages.

Acknowledgements

The authors are grateful to Min RK from Simula for discussions on the inner workings of Jupyter and help with initial experiments with setting up a a MMT kernel for Jupyter. Florian Rabe, Dan Alistarh, and Tom Wiesing have worked on an initial integration of MMT with Jupyter that shaped and validated the vision for an integrated user interface for the OpenDreamKitVRE formulated in this report. Discussions with Nicolas M. Thiéry have refined the ideas presented here.

REFERENCES

[GL] *GitLab*. URL: http://gitlab.org (visited on 02/24/2014).

[GLR09] Jana Giceva, Christoph Lange, and Florian Rabe. "Integrating Web Services into Active Mathematical Documents". In: *MKM/Calculemus Proceedings*. Ed. by Jacques Carette et al. LNAI 5625. Springer Verlag, July 2009, pp. 279–293. ISBN: 978-3-642-02613-3. URL: https://svn.omdoc.org/repos/jomdoc/doc/pubs/mkm09/jobad/jobad-server.pdf.

[IPh] *IPython*. URL: http://jupyter.cs.brynmawr.edu/hub/dblank/public/Jupyter%20Notebook%20Users%20Manual.ipynb.

[Jup] *Project Jupyter*. URL: http://www.jupyter.org.

[JupDoc] *What is Jupyter*. URL: http://jupyter-notebook-beginner-guide.readthedocs.org/en/latest/what_is_jupyter.html.

[JupGit] *Git Repository Jupyter*. URL: https://github.com/jupyter.

[JupLit] *Jupyter Notebook Practice*. URL: http://www.svds.com/jupyter-notebook-best-practices-for-data-science.

[JupNotebook] *Jupyter Notebook*. URL: http://jupyter-notebook.readthedocs.org/en/latest/notebook.html#notebook-documents.

[Koh12] Michael Kohlhase. "The Planetary Project: Towards eMath3.0". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). Ed. by Johan Jeuring et al. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 448–452. ISBN: 978-3-642-31373-8. arXiv: 1206.5048 [cs.DL].

[MH] *MathHub.info: Active Mathematics*. URL: http://mathhub.info (visited on 01/28/2014).

[WikiIPh] *IPython Wikipedia*. URL: https://en.wikipedia.org/wiki/IPython.

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.