Creating and Operationalizing Justification Models Using jPipe

Sébastien Mosser
mossers@mcmaster.ca
McMaster University, McSCert
Hamilton, Ontario, Canada

Nirmal Chaudhari
chaudn12@mcmaster.ca
McMaster University, McSCert
Hamilton, Ontario, Canada

Cass Braun*
braunc8@mcmaster.ca
McMaster University, McSCert
Hamilton, Ontario, Canada

Kai Sun*
sunk15@mcmaster.ca
McMaster University, McSCert
Hamilton, Ontario, Canada

ABSTRACT

Justification models are a lightweight approach to supporting accreditation, validation, or certification. Usually, when engineers work on pipelines (e.g., continuous integration/deployment, machine learning, notebooks), their primary focus is on the pipeline itself, and the justification of why this pipeline is the right one for their software is, at best, part of the documentation. This leads to operational/maintenance problems: Is your machine learning pipeline reusable? What is the purpose of that "*weird*" step in your continuous integration pipeline that you have no idea why it is there, but the pipeline fails if you remove it? With jPipe, we assume that justifying software should be easy and support both the initial modelling of a system and its incremental evolution. In this tutorial, we will present how the jPipe compiler can be used to model a justification, how composition algorithms can be used to support incremental/iterative evolution, and how the compiler's modular nature allows one to integrate it into one's own system. The tutorial will illustrate these key points of jPipe by using a family of good practices to validate a data science notebook automatically. It will guide the audience through (1) the definition of justification models to validate notebooks, (2) their organization into composable artifacts, (3) their operationalization into CI/CD pipelines through code generation and (4) the integration of these justification models in a standalone Java application.

KEYWORDS

Modelling, Justification, Pipeline, Data Science, Compiler

ACM Reference Format:
Sébastien Mosser, Nirmal Chaudhari, Cass Braun, and Kai Sun. 2024. Creating and Operationalizing Justification Models Using jPipe. In *Proceedings of ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (Tutorial) (MODELS'24).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

*Author only; Will not be presenting

MODELS'24, September 22–27, 2024, Linz, AU

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 PROPOSAL

1.1 Authors

Dr. Sébastien Mosser. Sébastien received his PhD in Computer Science in 2010 at Université de Nice (France). He is an Associate Professor at McMaster University (Canada), which he joined after several years as a research scientist/professor at SINTEF (Norway), Université Côte d'Azur (France) and Université du Québec à Montréal (UQAM). He is an executive member of the *McMaster Centre for Software Certification* (McSCert), an award-winning research centre established in 2009 to ensure that software is safe, secure and dependable. He is also Associate Chair of the Computing and Software Department. His research interests are centred on software engineering at scale, focusing on the definition of composition techniques to support the separation of concerns in various domains. He regularly gives tutorials in high-quality venues, from academic conferences [6–8] to industrial forums (*e.g.*, DevLog, General Motors, Amadeus Global Tech Forum). He has served as conference chair of MODELS for two editions (2020 and 2022).

Nirmal Chaudhari. Nirmal studies Software Engineering at McMaster University and is a member of the McMaster Centre for Software Certification (McSCert), which he initially joined in 2023 as a summer intern (with a federal scholarship). Since then, he has worked as a research assistant on the jPipe tool suite, designing and implementing composition algorithms to support modularity in the language and supporting developers by creating a language server one can use to interact with the language smoothly. In addition to his study and research project, he is a key member of McMaster's EcoCar Team, where he is using his knowledge in modelling and software engineering to engineer the next generation of battery electric vehicles (adapting a 2023 Cadillac)

Cass Braun. Cass studies Software Engineering at McMaster University and is a member of the McMaster Centre for Software Certification (McSCert). She recently joined the centre as a summer intern (federal scholarship) and is working on improving the user experience for developers when using the jPipe language. In her free time, she is the vocal director of McMaster Engineering Musical. She implemented the latest langauge server for the jPipe language using the Langium platform.

Kai Sun. Kai received his MEng in Software Engineering from McMaster University in February 2024 after obtaining a Bachelor's in Maths and Statistics at the University of Waterloo. He co-founded Dazi in January 2024, a software development company in Toronto,

Ontario. As part of his MEng degree, he modelled notebooks best practices by extracting them from the state-of-practice in Data Science.

1.2 Format

- Length: 1.5 hours or 3 hours;
- Level: beginner (introductory level);
- Prerequisites:
  - Basic knowledge of graph
  - Understanding of object-oriented programming

2 DESCRIPTION

The tutorial can be implemented as a 90-minute or 180-minute one. This description covers a 180-minute setup. An incompressible part concerns the presentation of (1) justification languages and (2) the JPipe tooling, which covers the first 30 minutes. If the timeslot is a 90-minute one, considering the MODELS audience, the tutorial will focus on the modelling dimension and model composition operators' definition and usage. The integration of JPipe in an existing toolchain through code generation will be demonstrated in a slideware way rather than with hands-on exercises.

2.1 Validating Notebook Best Practices

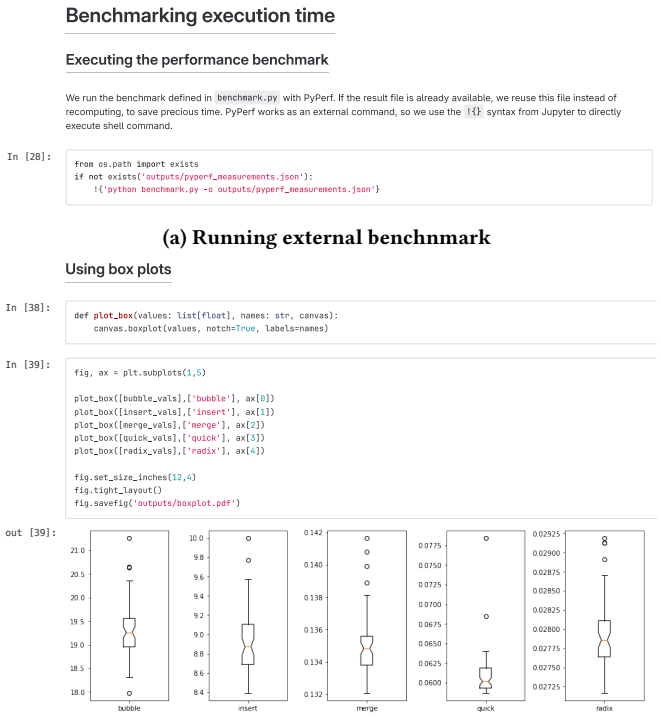
The tutorial will start by briefly describing Jupyter notebooks and how they are classically used in Data Science to implement experiments, from data processing to immediate visualization of results.

Fig 1 is an example of such a notebook. A notebook can be summarized as an implementation of the literate programming paradigm [5], where programmers (here, data scientists) mix documentation and code into a consistent unit. For example, Fig 1a is a mix of textual description (using Markdown) followed by a *cell* that contains executable code running a script ("`In [28]`" on the left margin means "*input to the Python interpreter*", cell 28).

Writing a notebook is a simple task. One can open Jupyterlab, a web-based system that will run in their browser and start writing Python code to process data and plot results (*e.g.*, Fig. 1b). However, making such notebooks shareable is more challenging: data scientists might not have the discipline required to engineer software, and, as such, makes it deployment-ready. This triggers reproducibility issues for experimental work developed in notebooks. The reproducibility crisis is not a specific data science thing, as a recent Nature survey reported that out of 1,500 researchers (not only data scientists), 75% were not able to reproduce experiments, and, more specifically, 50% were not able to reproduce their own experiments [2].

Even if the notebook presented in Fig. 1 is simple (it benchmarks some off-the-shelf sorting algorithms and plots their execution times), it must follow good practices to be shareable. Among all the best practices identified in the literature, we can focus on the following two:

- **Linear execution**. In our example, the code executing the benchmark is provided in cell 28, and the code used to generate the plots in cells 38 – 39. However, from a data science point of view, nothing prevents a user from pulling the plots at the beginning of their notebook to present them first, as they are the key result. When doing so, to reproduce the results, one would have to first go to the end of the notebook to execute the code and then come back to the first cell to draw the plot. This is not maintainable, so a good practice is to ensure that a notebook is *linear*, *i.e.*, can be executed from top to bottom automatically.
- **Coding standards (PEP8)**. To ensure code understandability, the Python community relies on the following assumption: "*code is read much more often than written*". As such, they have defined an extensive coding style that prescribes how Python code must be written to ensure readability and, transitively, maintenance. In Fig. 1b, we can identify a violation of the PEP8 standard, with a missing whitespace inside function calls in cell 29 (the instruction `plot_box([bubble_vals],['bubble'],␣ax[0])` should be `plot_box([bubble_vals],␣['bubble'],␣ax[0])`).



(a) Running external benchnmark

(b) Plotting benchmarked results

Figure 1: Example of Jupyter notebook

> **Participants involvment:** The audience will be presented with the use case used as an example and given a quick demonstration of how a notebook works in JupyterLab. An example notebook will be provided as part of the tutorial starter kit, as the tutorial focuses on the justification of notebooks rather than the notebooks themselves.

2.2 Justification Diagrams and JPipe Tooling

The first part exemplified how notebooks work and how sharing such artifacts is challenging. In this second part, we will introduce justification models as a way to capture *why* things are the way they are in a given piece of software. To support this task, we are

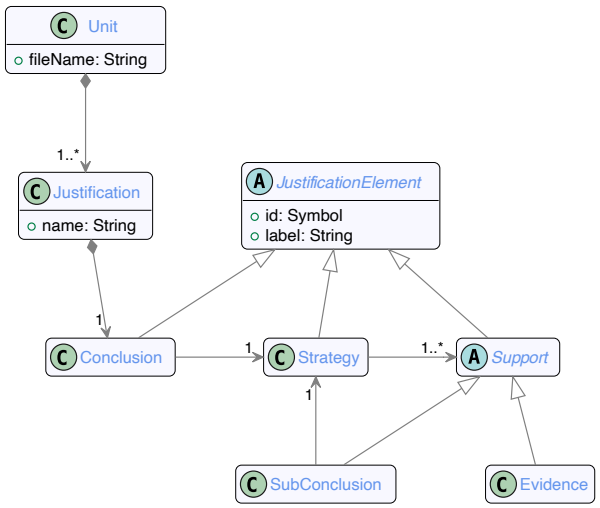


Figure 2: jPipe justification metamodel

using a justification diagram metamodel to capture the essence of a justification: starting from concrete `Evidence`, the model progresses towards a final `Conclusion` by using `Strategy`(ies) to reach `SubConclusions`, used to feed subsequent strategies. This metamodel (Fig. 2)is a simplification of the initial justification diagram proposed in 2016 [9], and can be seen as a simplification of the *Goal Structuring Notation* (GSN) [4].

The objective is to provide the simplest metamodel to capture the essence of Toulmin's model [10] identified in 1958, while removing the complexity engineers can see as accidental when discovering these models. When users are comfortable with these notions and realize complex cases require more complex tools, they can migrate to more expressive tools and language, such as WF+ or Astah Safety. The tool is available as a standalone compiler and a VS Code extension, as depicted in Fig. 3. In this picture, the left panel is where the data scientist is modelling their justification. The right panel provides a graphical representation of their justification. Finally, the bottom panel allows one to invoke the compiler manually and get helpful feedback on the model (e.g. inconsistencies or errors).

> **Participants involvment:** The audience will have the opportunity to install jPipe on their computer (it only requires a recent JDK) or to use a provided image containing all the necessary software in a containerized environment.

2.3 Modelling a Justification

jPipe is implemented as a textual domain-specific language (DSL) to instantiate models conforming to the previously described metamodel. The tool can compile a textual model into a graphical representation using the graphical syntax initially described by Duffau *et. al* [3]. Fig 4a describes how one can model the situation presented in the previous section using the textual DSL, modelling *why* a given notebook is shareable. Fig 4b is the graphical representation obtained by compiling such a model.

```
composition complete {
 justification shareable is reproducible with fair
}
```

Listing 1: Directive to compose independent models

> **Participants involvment:** The audience will work with an incomplete justification model to run the compiler, understand the error message, and fix the model by providing the missing information.

2.4 Merging Justification Models

Building justification models often aggregates multiple concerns. In the previous case, one can consider two different concerns from two stakeholders: (1) code quality concerns, coming from developers, and (2) reproducibility of the experiment, coming from a data scientist. Even if, in the case of notebook development, these two roles can be merged into the data scientist role, from a software engineering point of view, one can still make a distinction, as their objectives/goals are different.

jPipe supports the separation of concerns paradigm by defining independent justification models and using a composition algorithm to build the final justification by merging its input. As such, one can express a justification model targetting the *reproducible* concern (in a model named `reproducible`), and another stakeholder can do the same for the *code fairness* one (in a model named `fair`). One can then compose both concerns into a model named `shareable`, automatically generating, as a result, the model we manually built in the previous step.

> **Participants involvment:** The audience will take two existing models (provided) and write the composition directive to merge them. They will also add a third concern to their shareability justification to practice (1) writing justification models and (2) their automated composition.

2.5 Structuring Justification Models

One can assemble elements using automatic merge, but the final structure depends on how each concern is modelled. In cases where a given structure needs to be ensured, the open-ended merge approach might not be suitable.

Consequently, jPipe supports the user when modelling so-called *patterns* to implement *abstract* justifications. These justifications contain abstract supporting steps that need to be concretized when instantiating the pattern. Such a pattern to support both reproducibility and code fairness is represented in Fig. 5. The pattern, named `shareable_P`, models that a notebook can be shared when two quality gates are met. However, how these quality gates are concretely realized (the two leaves) depends on the team, project, and product. One can then instantiate the pattern when writing a justification, and the compiler will reject the model if it does not provide a concrete justification for each abstract support defined at the pattern level.

> **Participants involvment:** The audience will take the provided pattern and refactor the justification modelled in the previous step

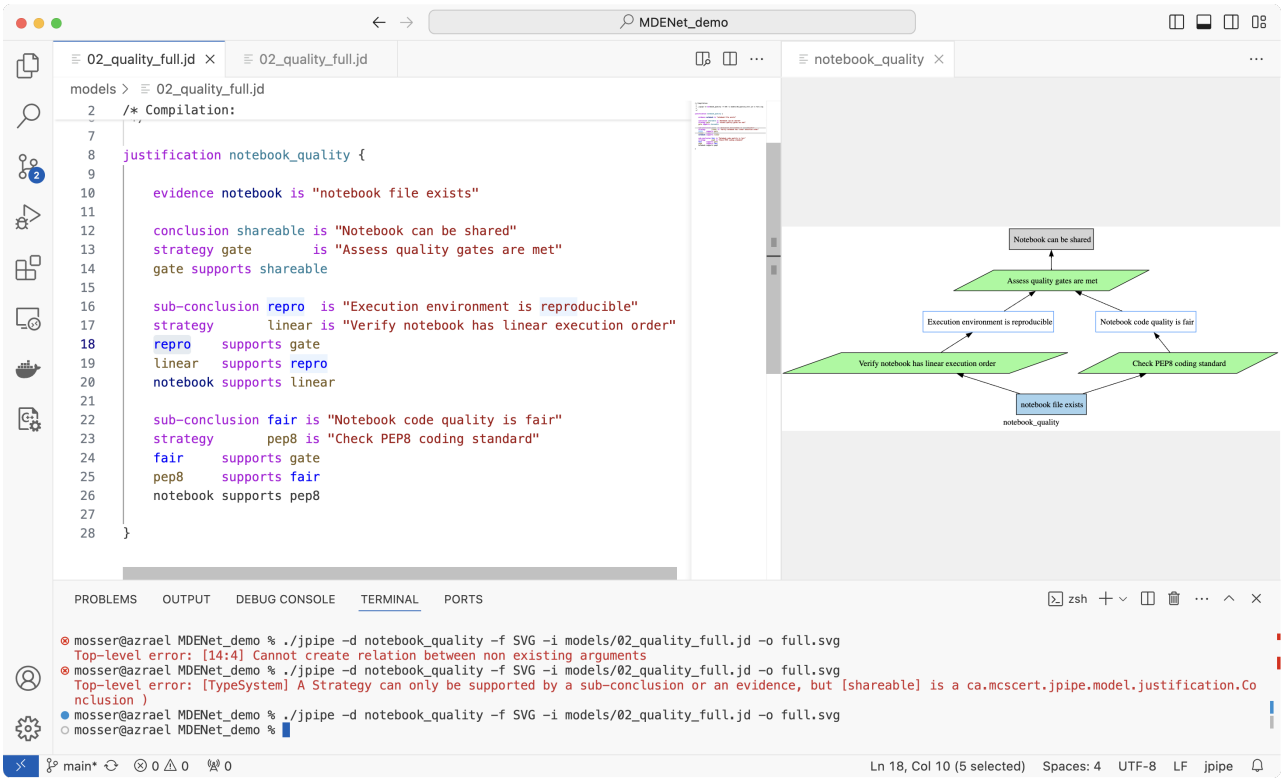


Figure 3: jPipe environment (VS Code extension)

into a pattern instantiation. They will experiment with the compiler completeness check when a pattern is only partially realized.

2.6 Generating Validation Code – Optional

So far, the models we developed are only descriptive. This part of the tutorial will demonstrate how to enrich the model with so-called *implementation concerns* to provide a concrete way of supporting the validation of the requirements they modelled.

For example, the strategy *"Verify notebook has linear execution order"* can be automatically verified using a tool named `pynblint`[1], a linter for Jupyter notebooks containing Python code.

The tutorial will demonstrate how one can enrich a justification by associating to it an *implementation*, *i.e.*, a set of meta-data that implements the justification for a given context. One can then use model transformations (provided in the compiler) to generate validation code from the justification model and its implementation meta-data.

Participants involvment: The audience will have access to an implementation file containing *complicated* commands dedicated to notebooks. They can enrich it with more common knowledge (e.g., check that a file exists) and use the completed implementation meta-data to generate a Github Action continuous integration pipeline automatically.

[1]https://github.com/collab-uniba/pynblint

2.7 Integrating into an existing tool – Optional

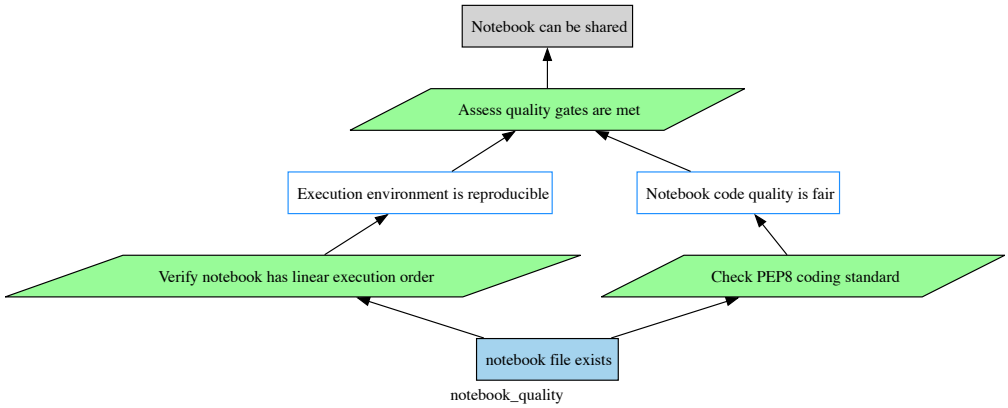
The final step of this tutorial is to demonstrate how jPipe is designed as an extensible platform. One can use the provided library to load justification models from text files and then (1) create their own composition algorithm and (2) use their own model transformation to generate proprietary code or integrate it into service-based systems, for example.

For this final step of the tutorial, we will consider a library of "off-the-shelf" justification for best practices. According to a product line philosophy, one can select the best practice they want to enforce in their notebook, and we will demonstrate how jPipe can be called programmatically to assemble the final justification and scaffold the development of a notebook to conform to the selected practices.

Participants involvment: The audience will load a REPL-like (*Read-Eval-Print-Loop*) application that allows one to navigate among standard best practices for notebooks and select the one they want. They will have to write approximately 40 lines of Java code as part of a provided skeleton to call jPipe programmatically and generate scaffolding code to create a new notebook and the associated validation code based on their selection. The effort will be to demonstrate the integration with external tooling, and notebook-specific code will be provided. People not comfortable with Java will also have access to the solution if they do not feel comfortable coding.

```
justification notebook_quality {

 evidence notebook is "notebook file exists"

 conclusion shareable is "Notebook can be shared"
 strategy gate        is "Assess quality gates are met"
 gate supports shareable

 sub-conclusion repro  is "Execution environment is reproducible"
 strategy       linear is "Verify notebook has linear execution order"
 repro    supports gate
 linear   supports repro
 notebook supports linear

 sub-conclusion fair is "Notebook code quality is fair"
 strategy       pep8 is "Check PEP8 coding standard"
 fair     supports gate
 pep8     supports fair
 notebook supports pep8

}
```

(a) Textual DSL



(b) Graphical representation

Figure 4: Domain-specific language(s) used in JPipe

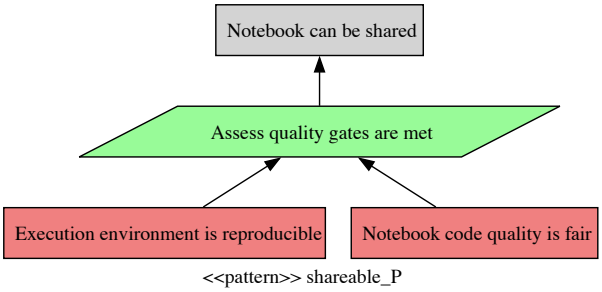


Figure 5: Justification Pattern example

3 NOVELTY

The tutorial aligns with the MODELS conference's core goal, as it targets a modelling approach to support software engineers. The novelty factor relies on its focus on *"modelling justifications"*, to help the audience understand and practice how to model such dimension of a given project using a lightweight tool like JPipe.

More specifically, quoting the MODELS 2024 call for paper, the tutorial will provide the audience insights on the following aspects of modelling:

- Presentation of a lightweight paradigm to support justification modelling at scale ("*New paradigms, formalisms, applications, approaches, frameworks, or processes for model-based engineering such as low-code/no-code development, digital twins, etc.*")
- Hands-on approach using the provided open-source compiler and/or the visual code extension ("*Tools, meta-tools, and language workbenches for model-based engineering, including model management and scalable model repositories.*")
- Demonstration of applicability of the tool and language on real-life examples, modelling best practices created by domain experts ("*Modeling in software engineering, e.g., applications of models to address common software engineering challenges.*")
- Hands-on exercises using data science notebooks as target ("*Modeling with, and for, novel systems and paradigms in*

fields such as [...], data analytics, data science, machine learning, [...].")

3.1 Previous MODELS editions (last ten years)

The MODELS conference series has a long tradition of tutorials, which can cover foundational topics as well as very practical tooling. In this regard, this tutorial will cover both dimensions, with an introduction to justification modelling followed by hands-on exercises using existing open-source tooling.

Concerning previous tutorials, the closest tutorial would be *"Assurance of Complex Software-Intensive Systems using WF+ (T5)"* given in 2020, which focused on assurance case modelling using WF+ (another tool developed and maintained at McSCert [1]). Contrarily to WF+, jPipe does not focus on safety-critical systems and is defined as a lightweight approach to democratize justification-based models. As we are using jPipe to model justification as a basis to validate requirements, it can also be linked to *"Model-Driven Requirements Engineering (T6)"* in 2014. From a bird-eye point of view, as it uses an operational DSL as an interface, the tutorial builds on top of our previous community effort on making DSL technologies accessible, such as: *"Compositional Modeling Languages in Action: Engineering and Application of Heterogeneous Languages: compositional dimension/modularity"* (2023), *"Agile, Web-Centric, Model-Driven Development of Real Systems Using Umple: code generation, product line approach"* (2022), *"Shadow Models: Incremental Model Transformations as an Enabler for Language Implementations. Incremental evolution and transformation based approach."* (2019), *"Managing the Co-Evolution of Domain-Specific Languages and Models"* (2018), and *"Applying Model Driven Engineering Technologies in the Creation of Domain Specific Modeling Languages"* (2016, 2015, 2014).

3.2 Required Infrastructure

The tutorial will alternate between a "lecture"- like period where the presenter(s) deliver new material to the audience and a "hands-on" period where participants can practice what they have just learned. Consequently, we will need a video projector to project the training material and access to a Wi-Fi network so that participants can download the training material on their computer at the beginning of the tutorial.

In the case of virtual delivery, the tutorial can easily be moved to a remote platform such as Zoom or MS Teams by using screen sharing to share material and having dedicated breakout rooms to support participants while doing the practical part.

3.3 Sample slides

See page 7-9 for some sample slides related to jPipe. These slides were initially designed for a talk given in the context of the MDENet Research Demonstration Series on June 6th, 2024. The talk is available on the MDENet Youtube channel[2]. Additional material was developed as internal training for McSCert and the NSERC/MITACS Alliance project "*DevOps for Software Defined Network*". This project involves participants from TELUS (a major telco company in Canada), Queen's University and École de Technologie Supérieure (ETS Montréal).

[2] https://www.youtube.com/@mdenet2528

REFERENCES

[1] Nicholas Annable, Thomas Chiang, Mark Lawford, Richard F. Paige, and Alan Wassyng. 2023. Lessons Learned Building a Tool for Workflow+. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*. IEEE, 140–150. https://doi.org/10.1109/MODELS58315.2023.00032

[2] Monya Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature News* 533, 7604 (May 2016), 452. https://doi.org/10.1038/533452a

[3] Clément Duffau, Thomas Polacsek, and Mireille Blay-Fornarino. 2018. Support of Justification Elicitation: Two Industrial Reports. In *Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10816)*, John Krogstie and Hajo A. Reijers (Eds.). Springer, 71–86. https://doi.org/10.1007/978-3-319-91563-0_5

[4] Tim Kelly. 2004. The Goal Structuring Notation – A Safety Argument Notation. https://api.semanticscholar.org/CorpusID:18921431

[5] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. http://dblp.uni-trier.de/db/journals/cj/cj27.html#Knuth84

[6] Sébastien Mosser and Jean-Michel Bruel. 2018. Reconciling Requirements and Continuous Integration in an Agile Context (Tutorial). In *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018*, Guenther Ruhe, Walid Maalej, and Daniel Amyot (Eds.). IEEE Computer Society, 508–509. https://doi.org/10.1109/RE.2018.00076

[7] Sébastien Mosser and Jean-Michel Bruel. 2021. Requirements Engineering in the DevOps Era (Tutorial). In *29th IEEE International Requirements Engineering Conference, RE 2021, Notre Dame, IN, USA, September 20-24, 2021*. IEEE, 510–511. https://doi.org/10.1109/RE51729.2021.00079

[8] Sébastien Mosser and Jean-Michel Bruel. 2024. Modern Teaching of Requirements Engineering and Business Analysis (Tutorial). In *32nd IEEE International Requirements Engineering Conference, RE 2024, Reykjavik, Iceland, June 24-28, 2024*. IEEE.

[9] Thomas Polacsek. 2016. Validation, accreditation or certification: A new kind of diagram to provide confidence. In *Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 1-3, 2016*. IEEE, 1–8. https://doi.org/10.1109/RCIS.2016.7549297

[10] Stephen E. Toulmin. 2003. *The Uses of Argument*. Cambridge University Press. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0521534836

Safety?

- **Definition** from Merriam-Webster:
  - *"The **condition of being safe** from undergoing or causing **hurt**, **injury**, or **loss**"*
- Lots of work on **safety-critical systems**:
  - Very **sharp** research **expertise**.
- What does it mean for a **Software Defined Network**?
  - **24-hour cell network** outage in Toronto, resulting in **911 not being reachable**
- What does it mean for the **Mental Health Natural & Language Processing?**
  - Your **most intimate thoughts are sent to an LLM** and used for training.

Anatomy of a Robotaxi Crash: Lessons from the Cruise Pedestrian Dragging Mishap
Philip Koopman

So we are not talking about Safety-Critical Systems

(and maybe not even mixed-criticality systems)

But that's not an excuse to ignore these situations!

(We need to a way to express and validate these "requirements")

Explaining is not Justifying

explain verb — *"Documentation"*

Save Word

ex·plain | \ ik-splān \
explained; **explaining**; **explains**

Essential Meaning of *explain*

1 : to make (something) clear or easy to understand
// I need a lawyer to *explain* this contract to me.
// The professor *explained* the poem to the class.
See More Examples

2 : to tell, show, or be the reason for or cause of something
// Scientists could not *explain* the strange lights in the sky.
// I don't know how to *explain* the dog's strange behavior.
See More Examples

justify verb — *"Actionable requirements validation"*

Save Word

jus·ti·fy | \ ˈjə-stə-ˌfī \
justified; **justifying**

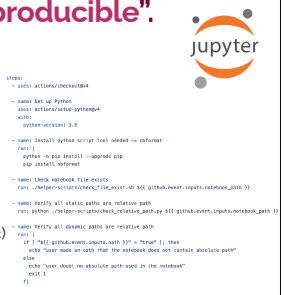
Essential Meaning of *justify*

1 : to provide or be a good reason for (something) : to prove or show (something) to be just, right, or reasonable
// He tried to *justify* his behavior by saying that he was being pressured unfairly by his boss.
// The fact that we are at war does not *justify* treating innocent people as criminals.
// It's hard to *justify* the cost of a new car right now. = It's hard to *justify* spending money on a new car right now.

2 : to provide a good reason for the actions of (someone)
// Why should I have to ***justify myself*** [=to provide an explanation for my actions] when it was their fault?

"The experiment shall be reproducible".

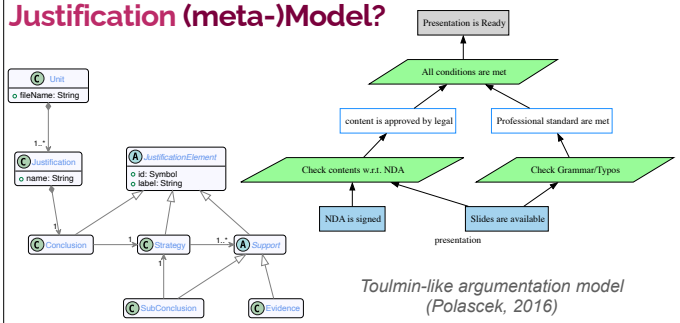
- **Context**:
  - **Working with NLP researchers** since 2019
- **Worst** case:
  - **No info**. Fingers crossed, hope for the best.
- **Best** case:
  - Notes in a **meeting's report** (e.g., minutes, doc)
- **Real life**(c):
  - info encoded in **cryptic shell commands**/APIs



We need to defend that the way we're building software is

Just, Right, Reasonable

Justification (meta-)Model?



Toulmin-like argumentation model (Polascek, 2016)

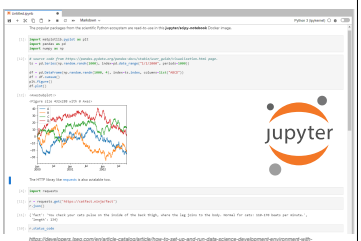
Keep it simple, stupid!

Why Justif. Diagrams (JD) and not others?

- **Lots of work** exists in this realm (e.g., **GSN, WF+**)
  - Toulmin's work is from the 50s, after all.
- Extensive work on **Assurance Cases**:
  - "*A **structured set of arguments** and a body of evidence showing that **an information system satisfies specific claims** with respect to a given quality attribute.*" [NIST]
- Let's consider **JD** as "**poor man's language.**"
  - **Poor man** here being "***average soft. developer***"
  - *It's **not as powerful as safety/assurance cases**, but easier to apprehend.*

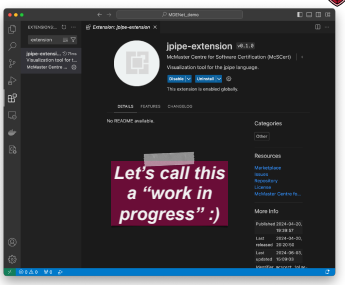
Notebook Quality?

- **Jupyter** Notebooks
  - ***Literate Programming***
  - **Massively used** in ML/Data Science
- **Software Engineering support**:
  - **Best Practices**/Guidelines
  - **Linter** (-ish) tooling
- **Problem?**
  - ***Can I share a given notebook?***



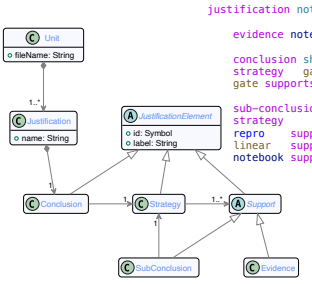
Getting jPipe on your computer

- **Standalone compiler**
  - Java & **ANTLR**
- **VS Code Extension**
  - **Language server** using LEVER
  - **Basic LS support**:
    - *Syntax highlighting*
    - *Code completion*
    - *Fold/unfold*



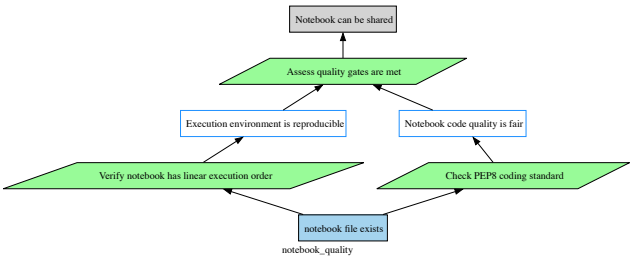
https://marketplace.visualstudio.com/items?itemName=mcscert.jpipe-extension

A Textual DSL to express justifications models



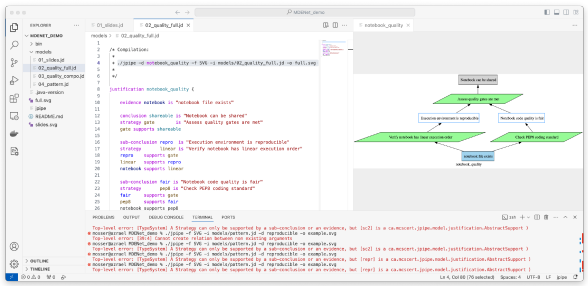

```
justification notebook_quality {

    evidence notebook    is "notebook file exists"

    conclusion shareable is "Notebook can be shared"
    strategy   gate      is "Assess quality gates are met"
    gate supports shareable

    sub-conclusion repro  is "Execution environment is reproducible"
    strategy       linear is "Verify notebook has linear execution order"
    repro    supports gate
    linear   supports repro
    notebook supports linear

    sub-conclusion fair is "Notebook code quality is fair"
    strategy        pep8 is "Check PEP8 coding standard"
    fair     supports gate
    pep8     supports fair
    notebook supports pep8

}
```

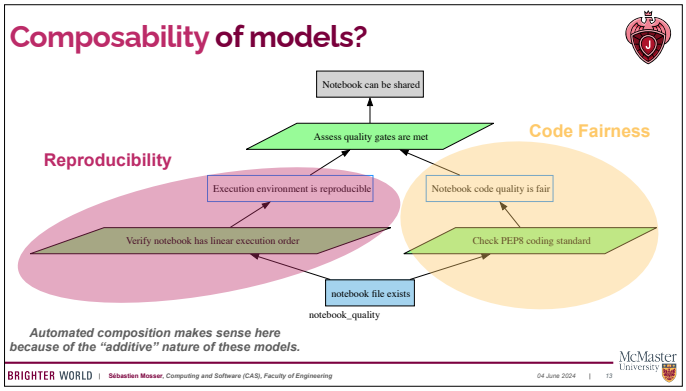

Using the standalone compiler

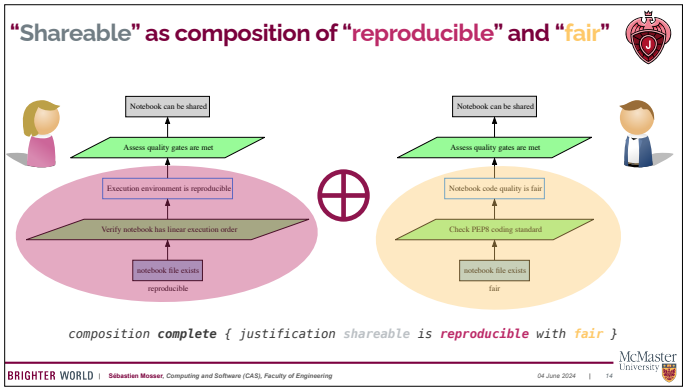


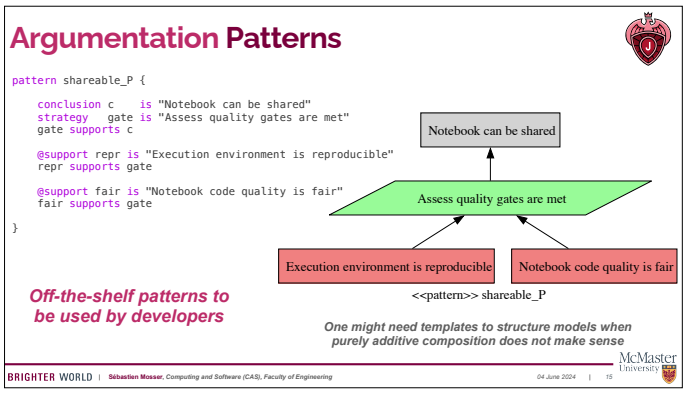

```
./jpipe -d notebook_quality -f SVG -i models/02_quality_full.jd -o full.svg
```

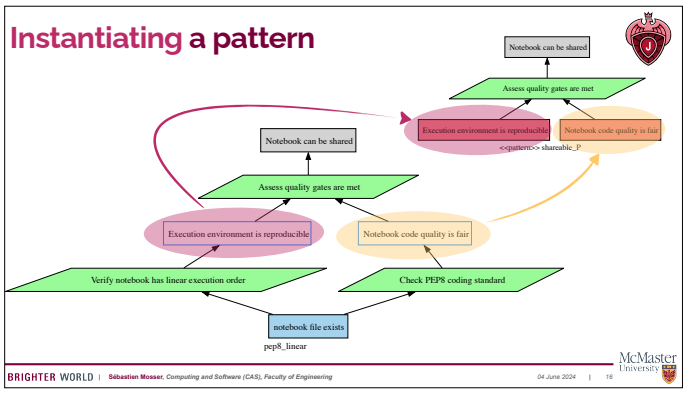

Using the VS Code extension

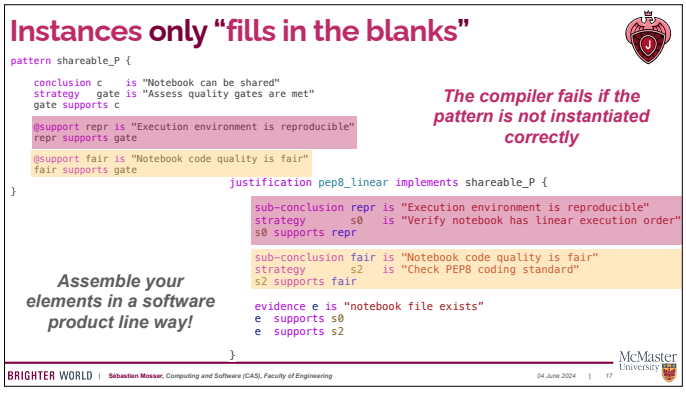


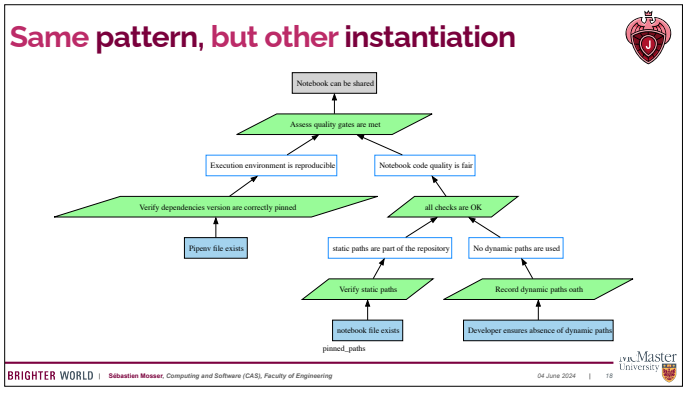

Composability of models?
Notebook can be shared
Assess quality gates are met
Reproducibility
Code Fairness
Execution environment is reproducible
Notebook code quality is fair
Verify notebook has linear execution order
Check PEP8 coding standard
notebook file exists
notebook_quality
Automated composition makes sense here because of the "additive" nature of these models.
BRIGHTER WORLD | Sébastien Mosser, Computing and Software (CAS), Faculty of Engineering 04 June 2024 | 13
McMaster University


"Shareable" as composition of "reproducible" and "fair"
Notebook can be shared
Assess quality gates are met
Execution environment is reproducible
Verify notebook has linear execution order
notebook file exists
reproducible
Notebook can be shared
Assess quality gates are met
Notebook code quality is fair
Check PEP8 coding standard
notebook file exists
fair
composition complete { justification shareable is reproducible with fair }
BRIGHTER WORLD | Sébastien Mosser, Computing and Software (CAS), Faculty of Engineering 04 June 2024 | 14
McMaster University


Argumentation Patterns
pattern shareable_P {
conclusion c is "Notebook can be shared"
strategy gate is "Assess quality gates are met"
gate supports c
@support repr is "Execution environment is reproducible"
repr supports gate
@support fair is "Notebook code quality is fair"
fair supports gate
}
Notebook can be shared
Assess quality gates are met
Execution environment is reproducible
Notebook code quality is fair
<<pattern>> shareable_P
Off-the-shelf patterns to be used by developers
One might need templates to structure models when purely additive composition does not make sense
BRIGHTER WORLD | Sébastien Mosser, Computing and Software (CAS), Faculty of Engineering 04 June 2024 | 15
McMaster University


Instantiating a pattern
Notebook can be shared
Assess quality gates are met
Execution environment is reproducible
Notebook code quality is fair
<<pattern>> shareable_P
Notebook can be shared
Assess quality gates are met
Execution environment is reproducible
Notebook code quality is fair
Verify notebook has linear execution order
Check PEP8 coding standard
notebook file exists
pep8_linear
BRIGHTER WORLD | Sébastien Mosser, Computing and Software (CAS), Faculty of Engineering 04 June 2024 | 16
McMaster University


Instances only "fills in the blanks"
pattern shareable_P {
conclusion c is "Notebook can be shared"
strategy gate is "Assess quality gates are met"
gate supports c
@support repr is "Execution environment is reproducible"
repr supports gate
@support fair is "Notebook code quality is fair"
fair supports gate
}
The compiler fails if the pattern is not instantiated correctly
justification pep8_linear implements shareable_P {
sub-conclusion repr is "Execution environment is reproducible"
strategy s0 is "Verify notebook has linear execution order"
s0 supports repr
sub-conclusion fair is "Notebook code quality is fair"
strategy s2 is "Check PEP8 coding standard"
s2 supports fair
evidence e is "notebook file exists"
e supports s0
e supports s2
}
Assemble your elements in a software product line way!
BRIGHTER WORLD | Sébastien Mosser, Computing and Software (CAS), Faculty of Engineering 04 June 2024 | 17
McMaster University


Same pattern, but other instantiation
Notebook can be shared
Assess quality gates are met
Execution environment is reproducible
Notebook code quality is fair
Verify dependencies version are correctly pinned
all checks are OK
Pipenv file exists
static paths are part of the repository
No dynamic paths are used
Verify static paths
Record dynamic paths oath
notebook file exists
Developer ensures absence of dynamic paths
pinned_paths
BRIGHTER WORLD | Sébastien Mosser, Computing and Software (CAS), Faculty of Engineering 04 June 2024 | 18
McMaster University