ROSMOD: A Domain Specific Tool-suite (DSTS) for Distributed CPS

William Emfinger, Pranav Srinivas Kumar, and Gabor Karsai

Kumar, P.S.; Emfinger, W.; Karsai, G.; Watkins, D.; Gasser, B.; Anilkumar, A. ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS. *Electronics* **2016**, *5*, 53.

This work was supported by DARPA under contract NNA11AB14C and USAF/AFRL under Cooperative Agreement FA8750-13-2-0050, and by the National Science Foundation (CNS-1035655). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, USAF/AFRL, or the NSF.

Outline

- Motivation
- Solution: ROSMOD
  - Why WebGME
- Design Principles
  - Meta-model / language
  - Functionality
  - Interface
- Implementation / Demo
  - How to structure the Meta-model
  - How to structure the WebGME components
- Lessons Learned

Motivation

- Distributed CPS are hard to design, develop, analyze, deploy, and manage
- Integration of these key requirements into an IDE would make these processes
  - Easier to teach
  - Less error prone
  - Faster / More repeatable
- Some IDEs are heavy and complicated to install / set-up
  - Require training as well
  - Need maintaining to ensure proper versions and roll-out of updates/bug-fixes
  - Need to be cross-platform
- Not every system (or type of system) can or should conform to the same meta
  - Even within the same class of "Distributed CPS"
  - Not everyone wants or needs the context of very explicit / fine-grained network specification

Solution

- Need an extensible, modular tool-suite that allows users / developers to create or swap feature sets for different deployments
- Need to be able to easily change the language associated with the modeling to tightly fit the class of systems being developed
- Need to make installation, training, and updating of the IDE easy for end-users
  - Especially when the IDE interacts with a lot of back-end infrastructure, e.g. for CI builds/tests or deployment
- All of these concerns leads us to:
  - WebGME: yes it can be better than eclipse.

Design Principles: Meta (1/2)

- Need to capture only the abstract concepts related to the design and development of the class of systems in question
- Must balance between creating abstractions for everything (generalization to more systems) and implementation details (coupling to a technology / system)
  - Generalization is good, but can come at the cost of usability / user-friendliness
  - Implementation details can require proliferation of meta-models to account for variations in the implementation, but can improve usability / user-friendliness
- More than just **what** is captured in the meta is important; it matters **how** and **where** in the meta the concepts exist
  - This translates to *containment* / *inheritance* of the objects and how their *attributes* are represented
- **The design of the meta is the most important part.**

Design Principles: Meta (2/2)

- When designing the meta you must also think about:
  - Is the meta something that should be extensible by users?
  - Should users even see the meta?
- The meta is the main user-interface a modeler will see, and the one they will interact with the most.
  - Good meta → happy users
  - Can be worked around with some good visualizers, but use sparingly
    - A visualizer / decorator only changes one aspect of the interaction, you'd need to change the other webgme components as well, e.g. *TreeBrowser*, *PartBrowser*, *AttributePanel*, etc.
- Finally, anything required for the class of systems that is not in the meta will not be available to the users but can be controlled in you components
  - E.g. the connection mechanisms to target nodes, or the middleware library in use for communication
  - Need to draw the line between the platform (infrastructure) and the model

Design Principles: Functionality (1/2)

- Once you've designed the meta, you need to make sure its design facilitates the creation of models
  - Especially important for **sets** and **pointers**:
    - Do you want a pointer to be a pointer or a *connection*?
    - Is a set actually a set, or just a collection of connections?
  - Similarly for containment:
    - Just because the domain you're modeling has a hierarchy / tree-structure, doesn't mean your meta/model should. Heavily dependent on what kind of user-interaction you forsee.
- Now that you've designed your meta and made sure it's painless to make models, what else do you want your domain specific tool-suite (*DSTS)* to do?

Design Principles: Functionality (2/2)

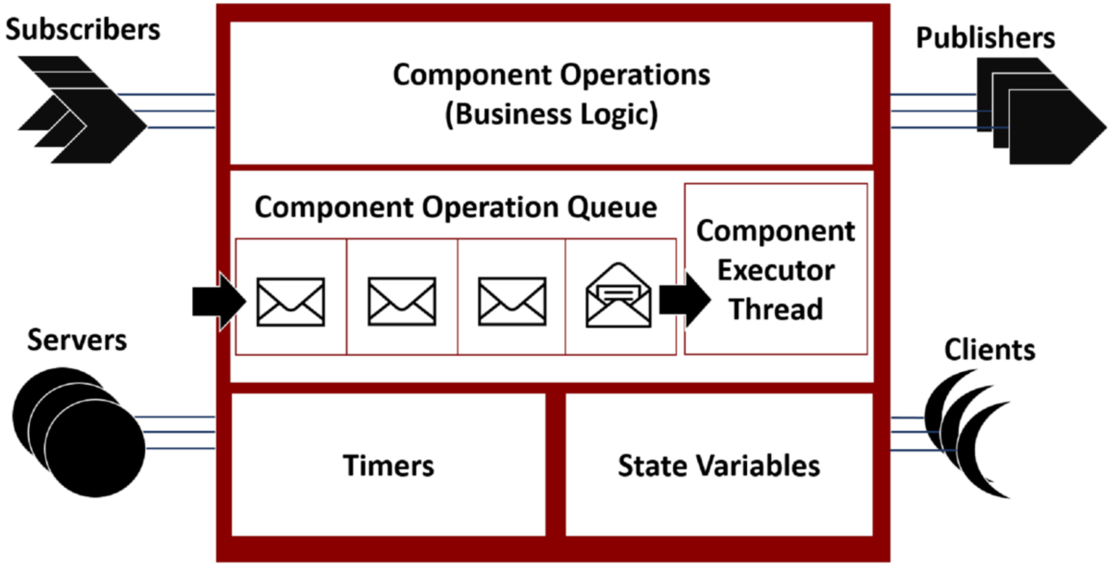
- Generally, you're wanting to
  - Get some parts of the model
  - Perform some transformation
  - (Rendering) -- visual, textual, etc.
  - (Update) -- write-back
- Do you want it as a **Plugin**, **Visualizer**, **Addon**? Depends on:
  - How you want the users to trigger and interact with it
  - What functional dependencies your functionality has (e.g. file-system, user-interaction, remote computing, etc.)
  - How many users/instances can be active?
- Also influenced by how reusable your code-base is and how dependent it is on your meta
  - Similar trade-off as meta design: modular, meta-agnostic code (generalization) versus meta-specific code that only works with your current meta model
  - Generalization is good (for everyone) if done right, but can be a challenge to achieve
  - Implementation specific code changes every time your meta changes (which may be frequently)

Design Principles: Interface

- Final part (that you should do last)
  - But unfortunately you should think about it during the rest of the processes
- Visual styling of the meta/models
  - Icons
  - Decorators
  - Visualizers
  - Themes (color schemes)
- These improve quality of life, but are likely to change during development of the rest of the DSTS
  - As you're developing model transformation / analysis code, you're still likely to update the meta as you find elements you've forgotten or not represented in a manageable way

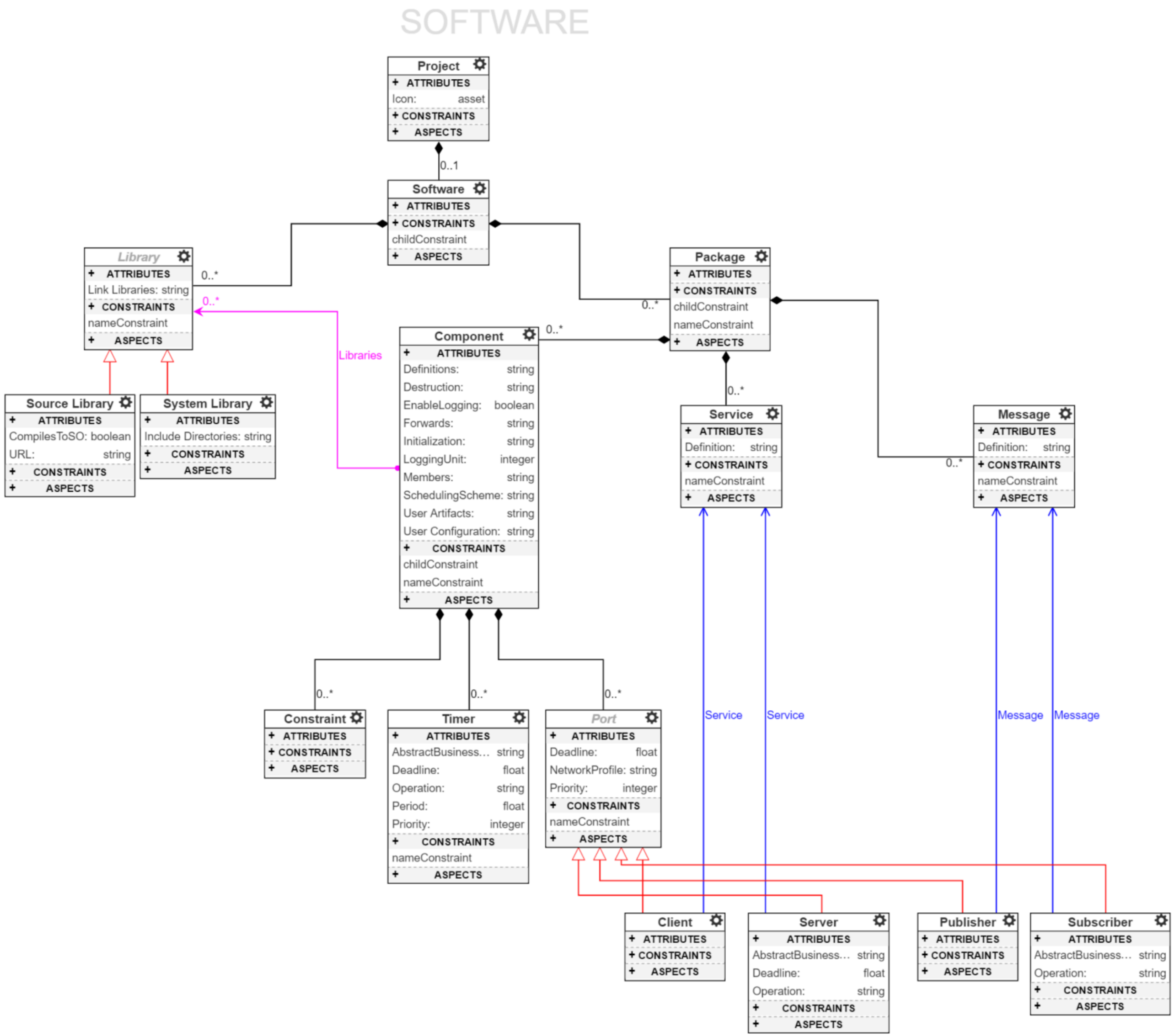
ROSMOD Specifics

- Meta contains everything needed to specify
  - Distributed, component-based software
  - Networked embedded systems
  - Deployments of software onto hardware
  - Experiment executions and their results
- Plugins enable
  - Code generation/compilation
  - Functional (timing) model analysis
  - Documentation generation
  - Experiment deployment / execution
  - Experiment Teardown and results aggregation
- Visualizers enable
  - Project browser with relevant descriptions and identification
  - Deployment visual inspection and call chain tracing
  - Execution trace log visualization



Implementation: Meta

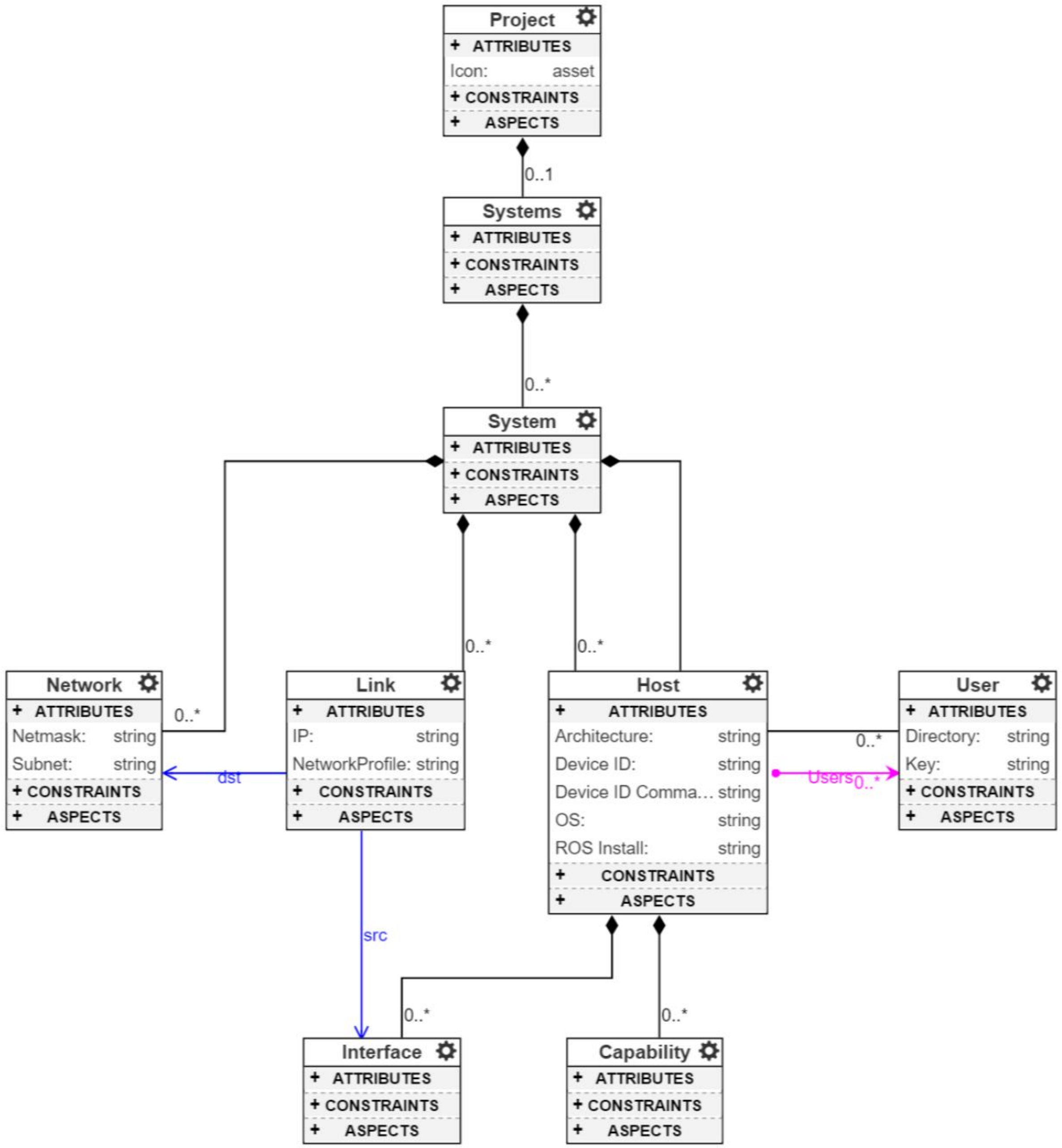
- Software contains generic concepts like libraries, operations, definitions, etc.
  - Mostly language agnostic; can be C/C++, python, or any other language which supports these concepts
- *Message* and *Service* pointers are not connection objects, don’t’ drag / draw between objects to establish connection
  - Trade-off for ease of specification with simplicity and out-of-tree specification
  - Currently cross-cuts are not able to do what we need them to (and further complicate interface)



Implementation: Meta

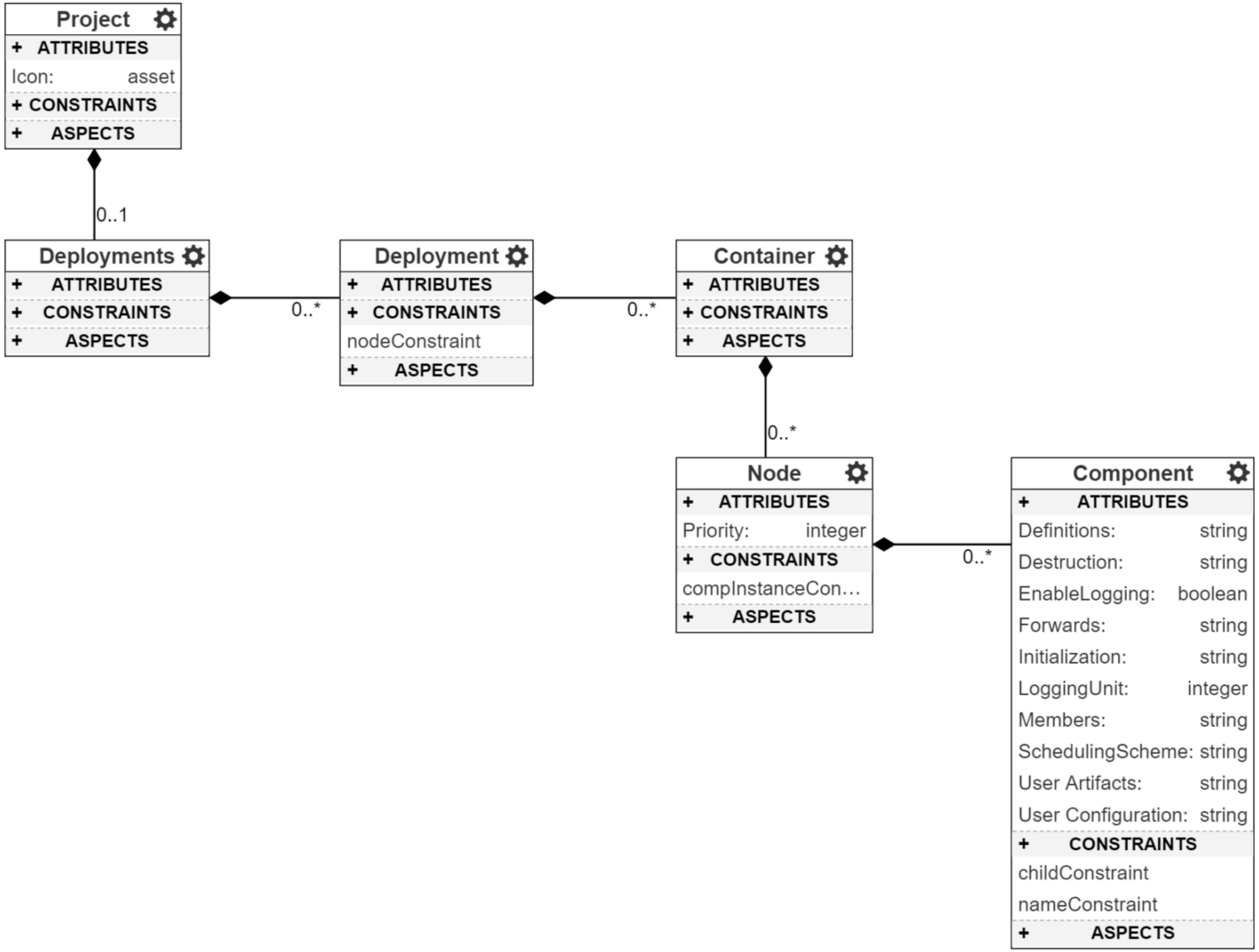
- This is a case of meta trying too hard to be like the domain it represents
  - Why did I have a *Link* between an *Interface* and a *Network*? I like Zelda, but no, I was just following the domain.
  - Means I have to create an Interface object inside a Host before drawing a link between its interface *port* and the network
    - Would be much simpler to just have the link between the *Host* and the *Network*
- Some attributes like Key are a little too implementation specific, but unavoidable

SYSTEMS



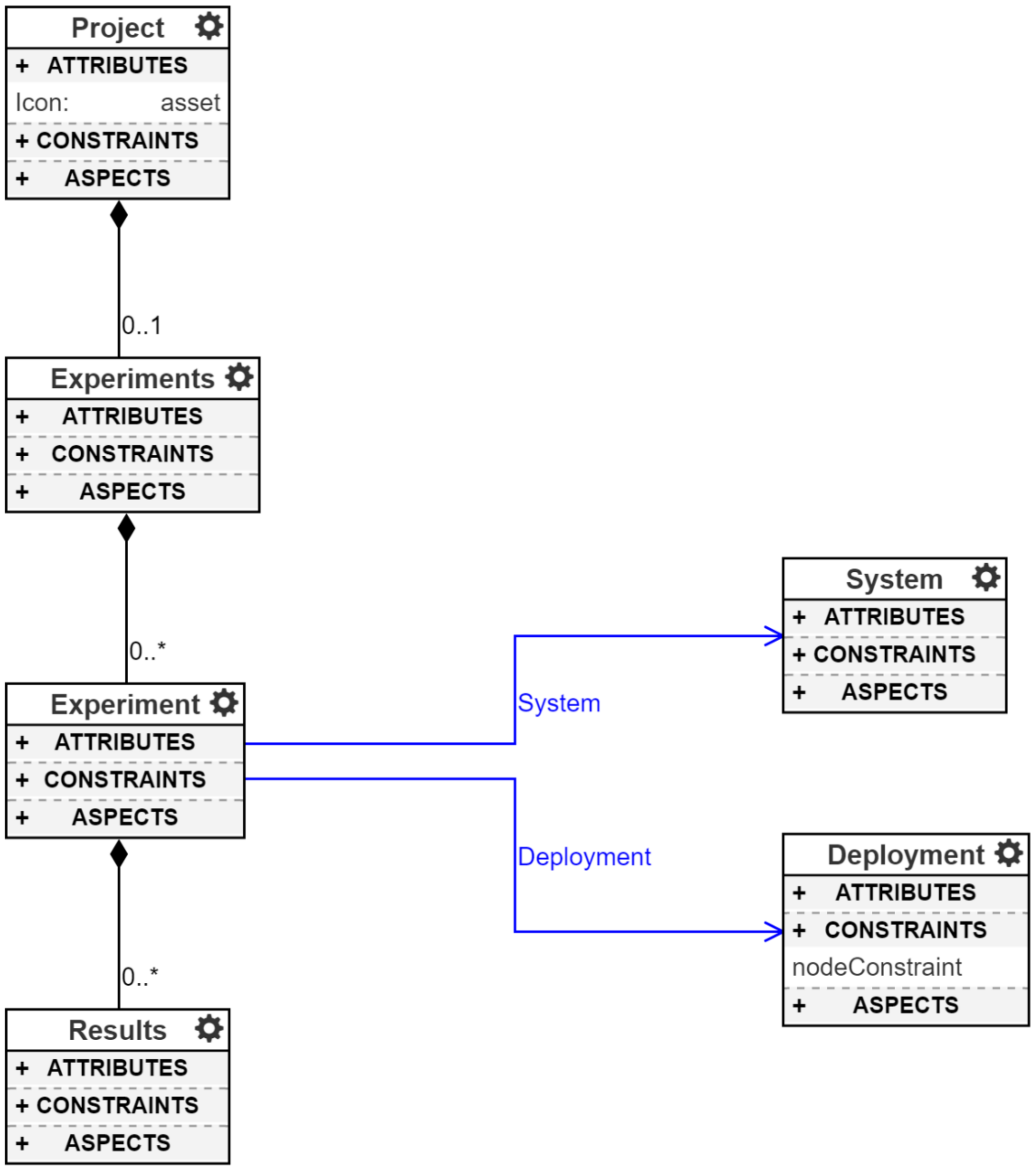
Implementation: Meta

- Meant to be a generic collection of components in to nodes, nodes into containers
- Nice from a meta-modeling perspective, and lets us re-use these deployment specifications across systems
- But has been a source of confusion when training new users
  - Most don't really see the point of it
- Moral to this story: if you're going to generalize something, make sure it **simplifies** the concepts and the training required as much as possible
  - If you can't describe it simply, it may not be worth it.



Implementation: Meta

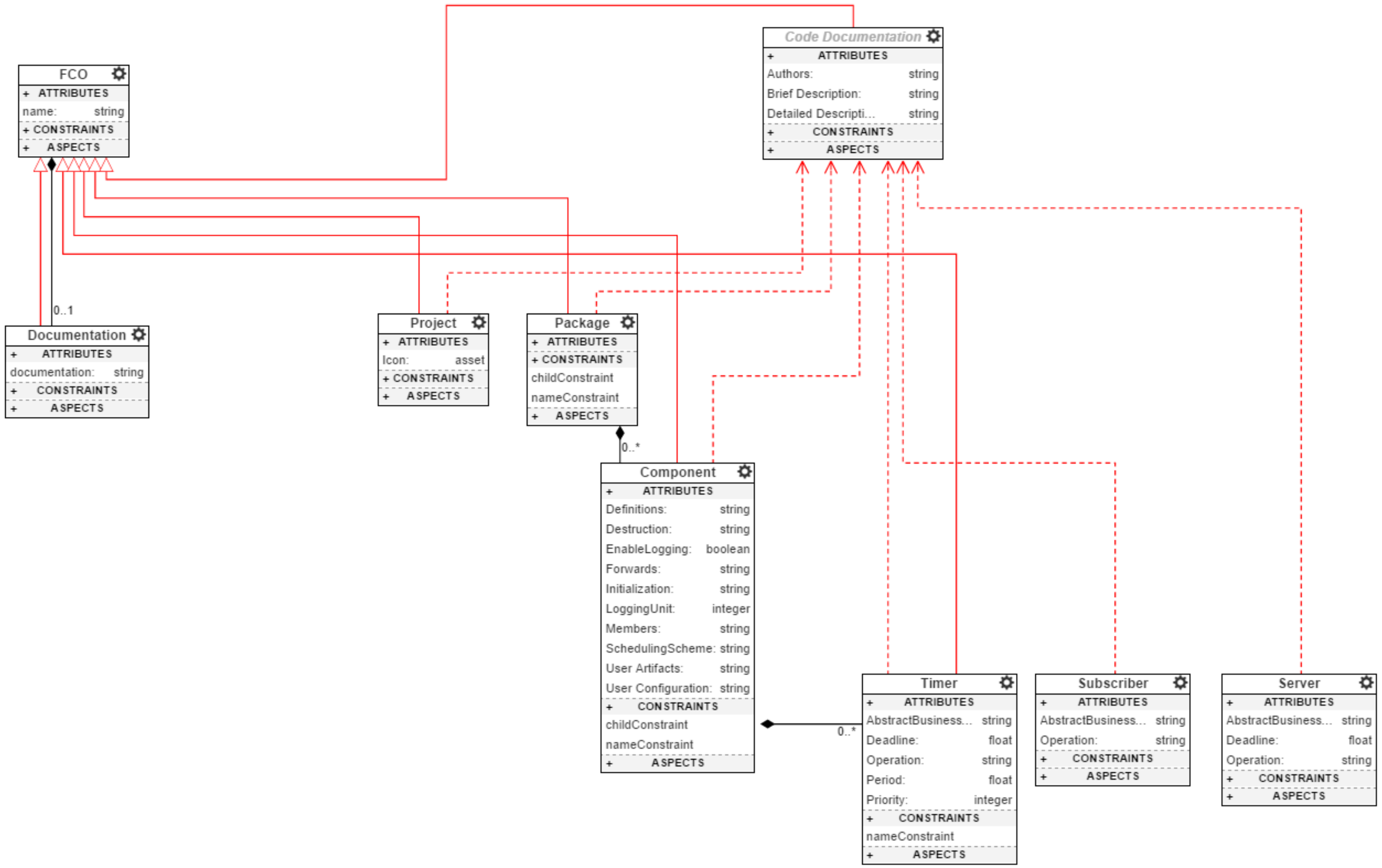
- Exhibits another downside of default pointer visualization, specification:
  - Can't easily see exactly which system or deployment an experiment is configured with.
- Need a way to visualize the composition of these two objects into one
  - Sounds like a job for a visualizer!
- However, the purpose of this relation was to have our infrastructure ***automatically*** calculate the mapping based on parameters that are **not modeled** because they are variable
  - E.g. current resource utilization or network connectivity



Implementation: Meta

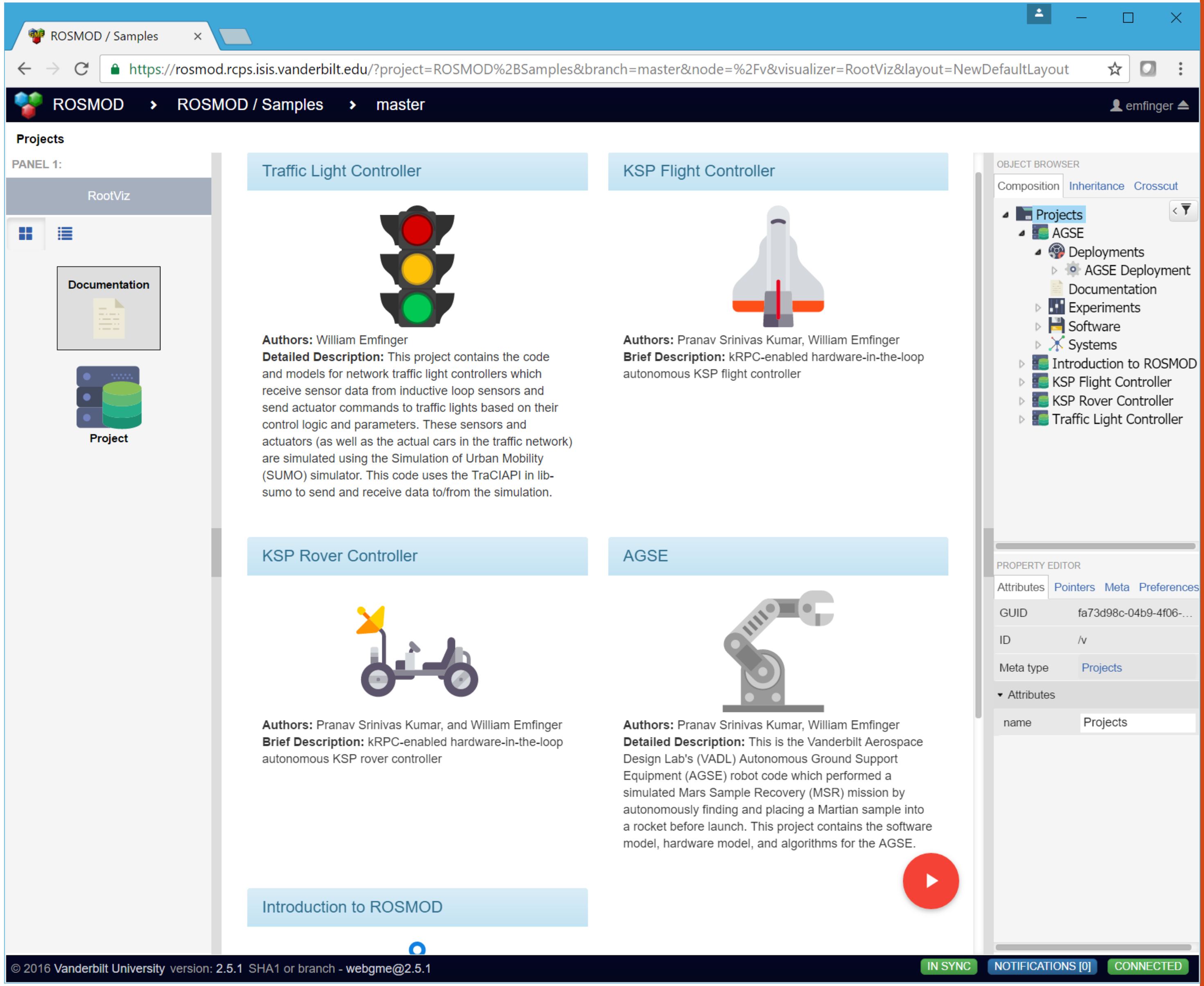
- Don't forget about **Mixins**, they allow multiple inheritance and nice object-oriented design of your meta!
- I can easily specify and change which objects have the same attributes and what the default attributes are
  - This also lets me generalize my plugins / visualizers a little more because now I can just tell other developers if they want to use my components for these purposes, they simply need to add a mixin with these attributes to their meta
- Only downside is that you can't see directly in the objects' attributes their inherited attributes

DOCUMENTATION



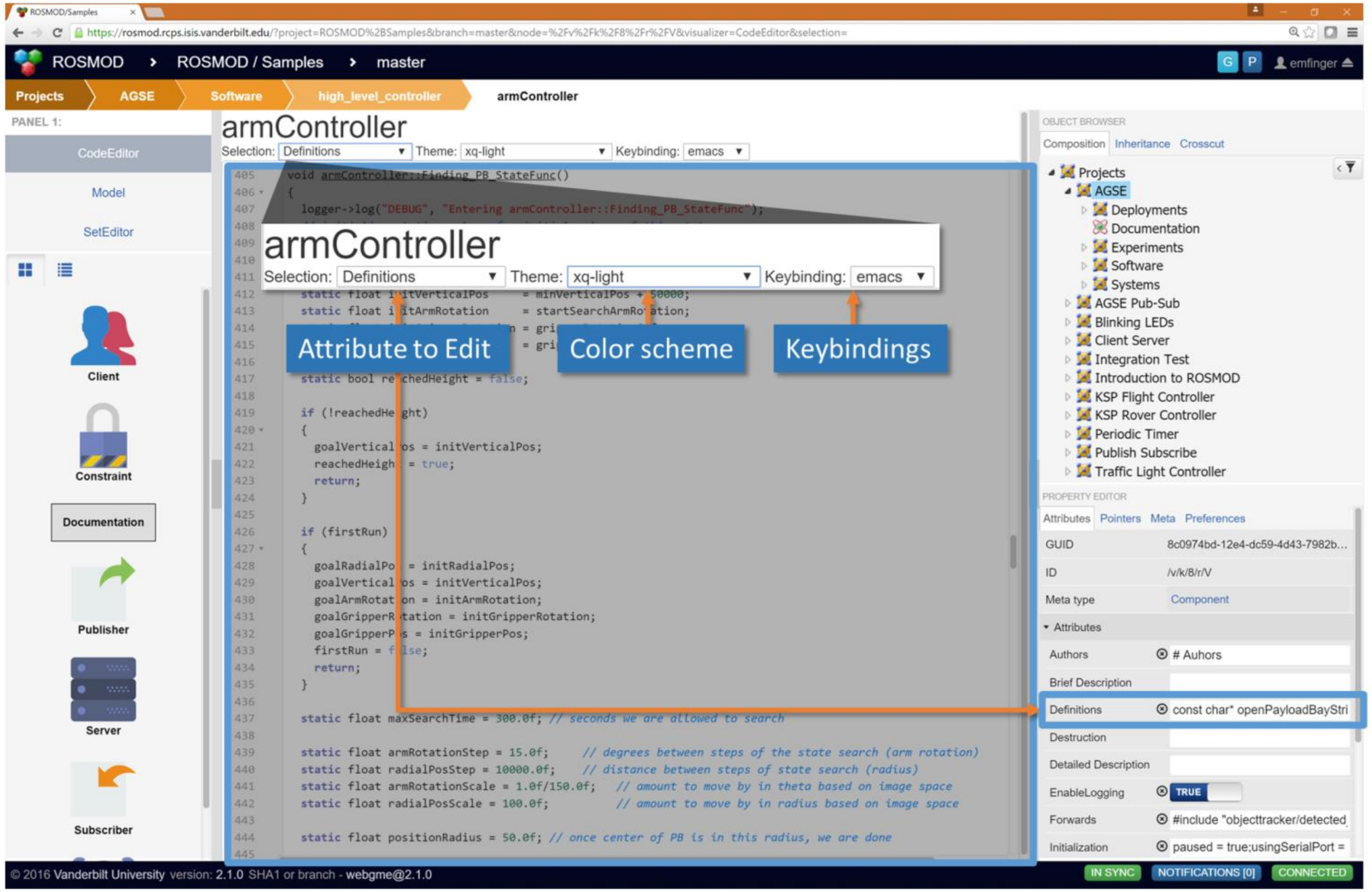
Implementation: WebGME Components

- Project Browser
  - Allows users to easily navigate and select which project they want to work on
  - Gives more contextual information than standard decorators/visualizers
- Implemented without decorator
  - All in the visualizer
- Is this the right way to do this?
  - Depends on what your goal is ☺
- Getting some measure of *default behavior* is still a little tricky / undocumented
  - E.g. drag and drop



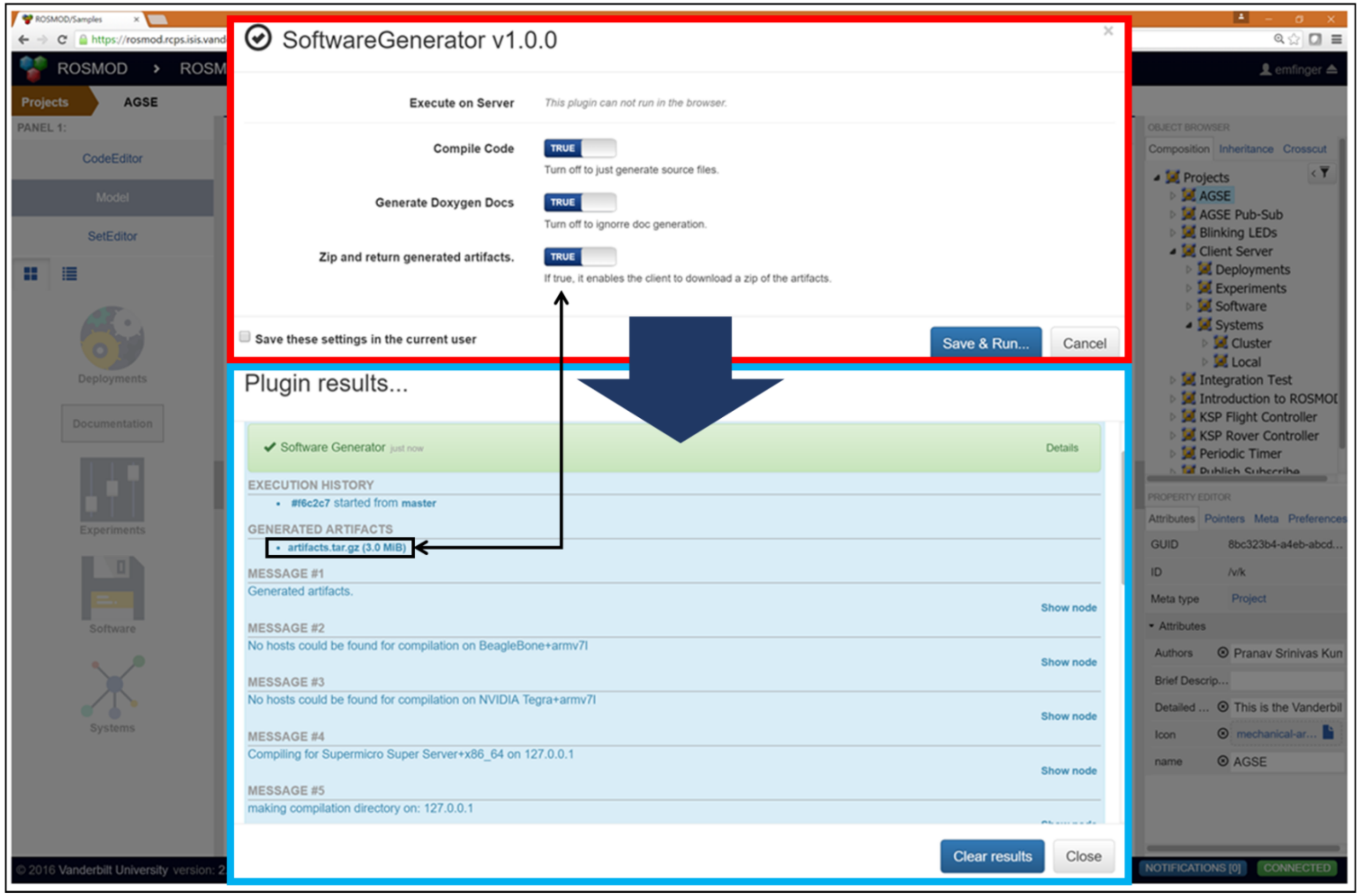
Implementation: WebGME Components

- Code Editing in the browser; need features of an IDE:
  - Syntax highlighting
  - Code completion
  - Code folding
  - Theming ☺
  - Keybindings (don’t start a vi(m) vs. emacs flame-war here…)
  - Multiple buffers
- The CodeEditor component is generic, and can be added to any WebGME project with configuration though ./config/components.json



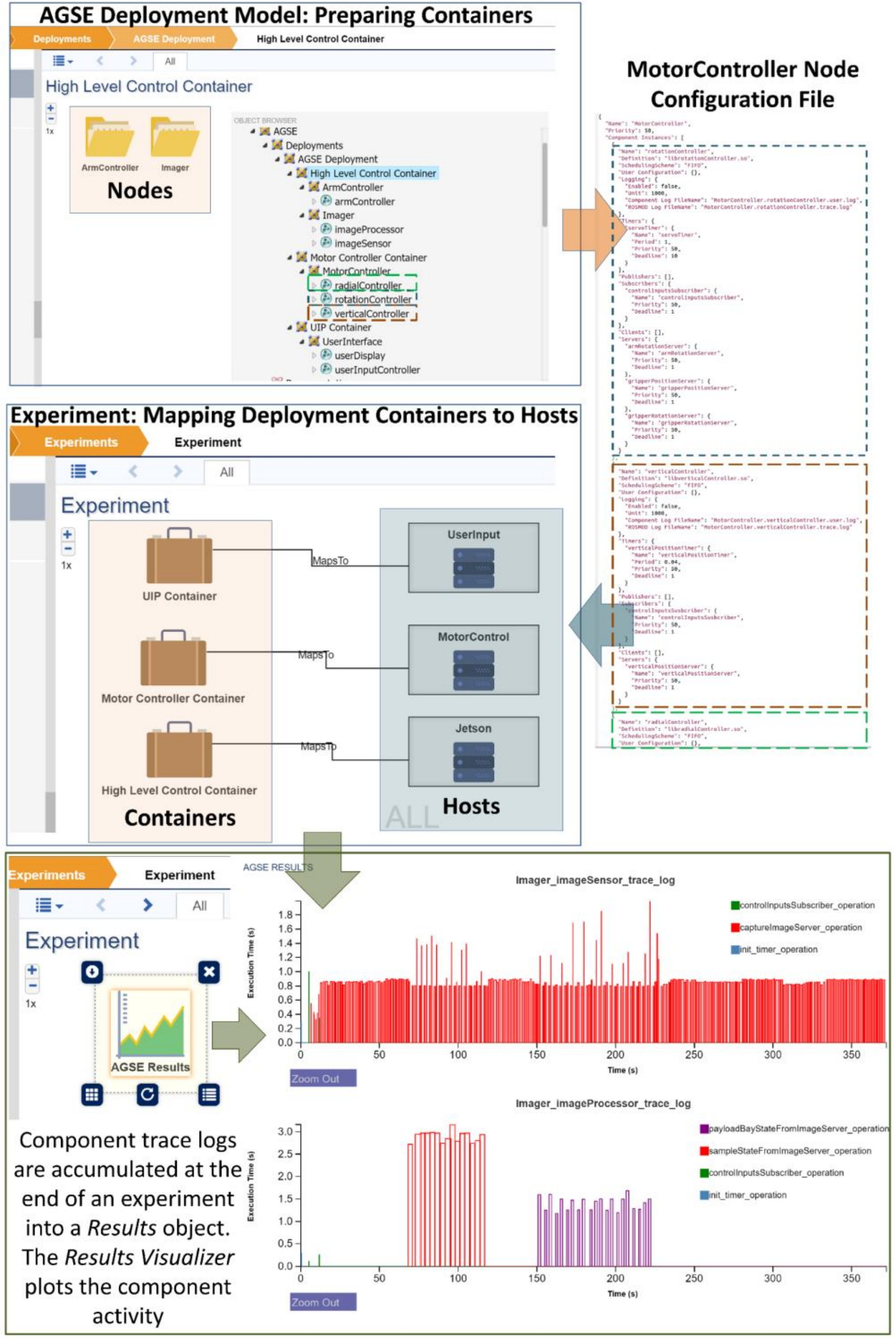
Implementation: WebGME Components

- Software Generation and Compilation
  - All the code is either
    - Contained within the model,
    - Generatable from the model, or
    - Located in a repository as part of a library which is referenced in the model
  - This means that users don't have to touch the generated code ➔ large quality of life improvement over previous systems we've used in the past
  - **Code templates should be as agnostic of the generation code as possible**
- Compilation (when required) must use the filesystem and call compilers which cannot run in the browser ☹
  - The compilation runs on the server, which is good for two reasons:
    - Users don't have to install or manage the compilers or their dependencies
    - Updates to the compilers can be managed in a centralized fashion by sys-admins



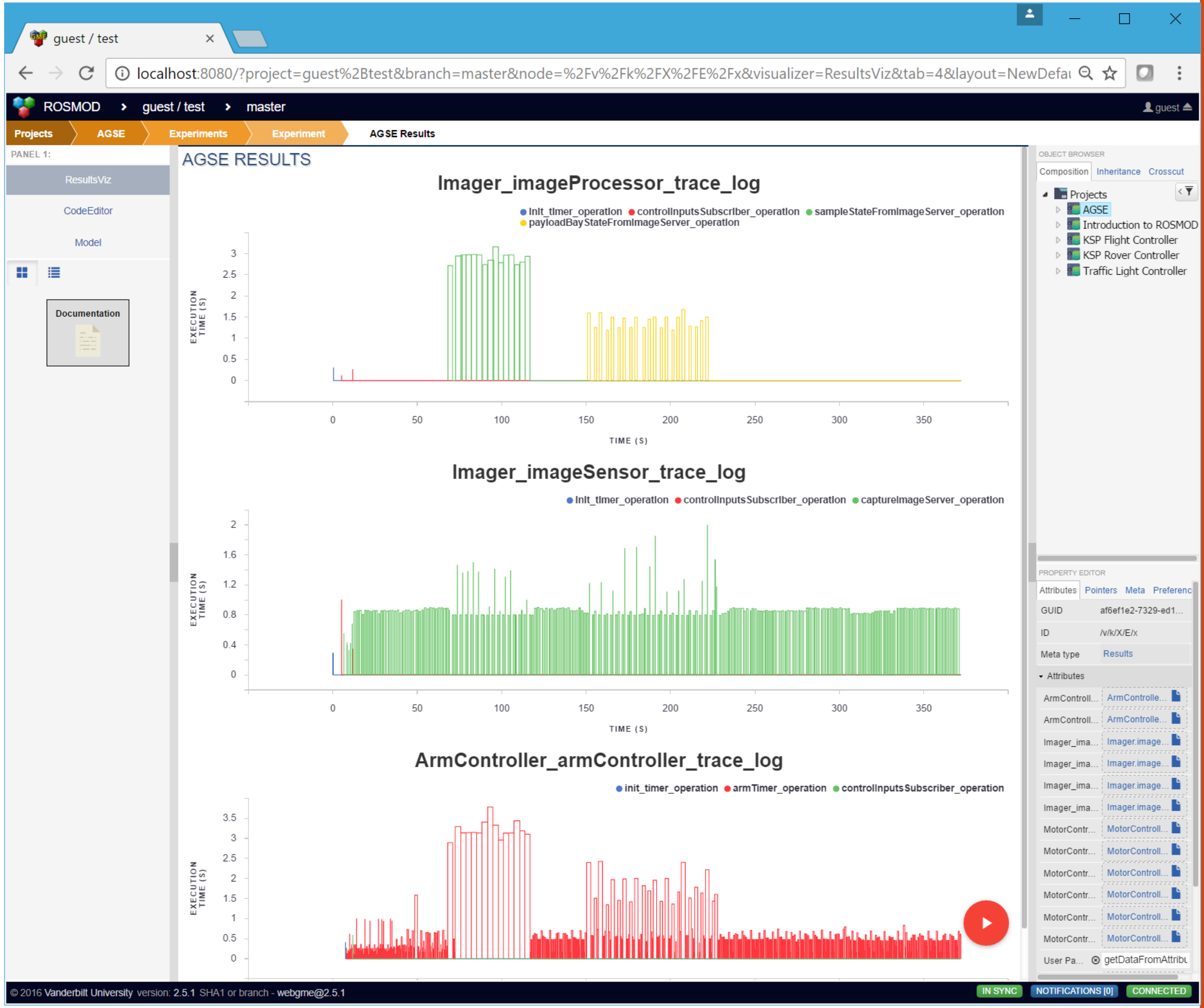
Implementation: WebGME Components

- Experiment Deployment and Execution
  - Like the software generation/compilation, must run on the server, since it actually moves the binaries and configuration files over to the distributed systems
  - Automatically queries the systems described in the model to determine which have available resources for running the experiment
  - Updates the model to create a map that the user can see (and that the other components can interact with) which specifies the exact mapping that the plugin calculated
    - Map is useful because user who starts the experiment may leave, and another user may need to stop it; good to have the state stored in the model
- When Experiment is stopped, the map is removed from the model and the results of the experiment are returned to the user and saved in the model
  - As **assets**, since these results may be large, don't want to bog down the UI by loading a lot of data that may not be used; load it on demand.



Implementation: WebGME Components

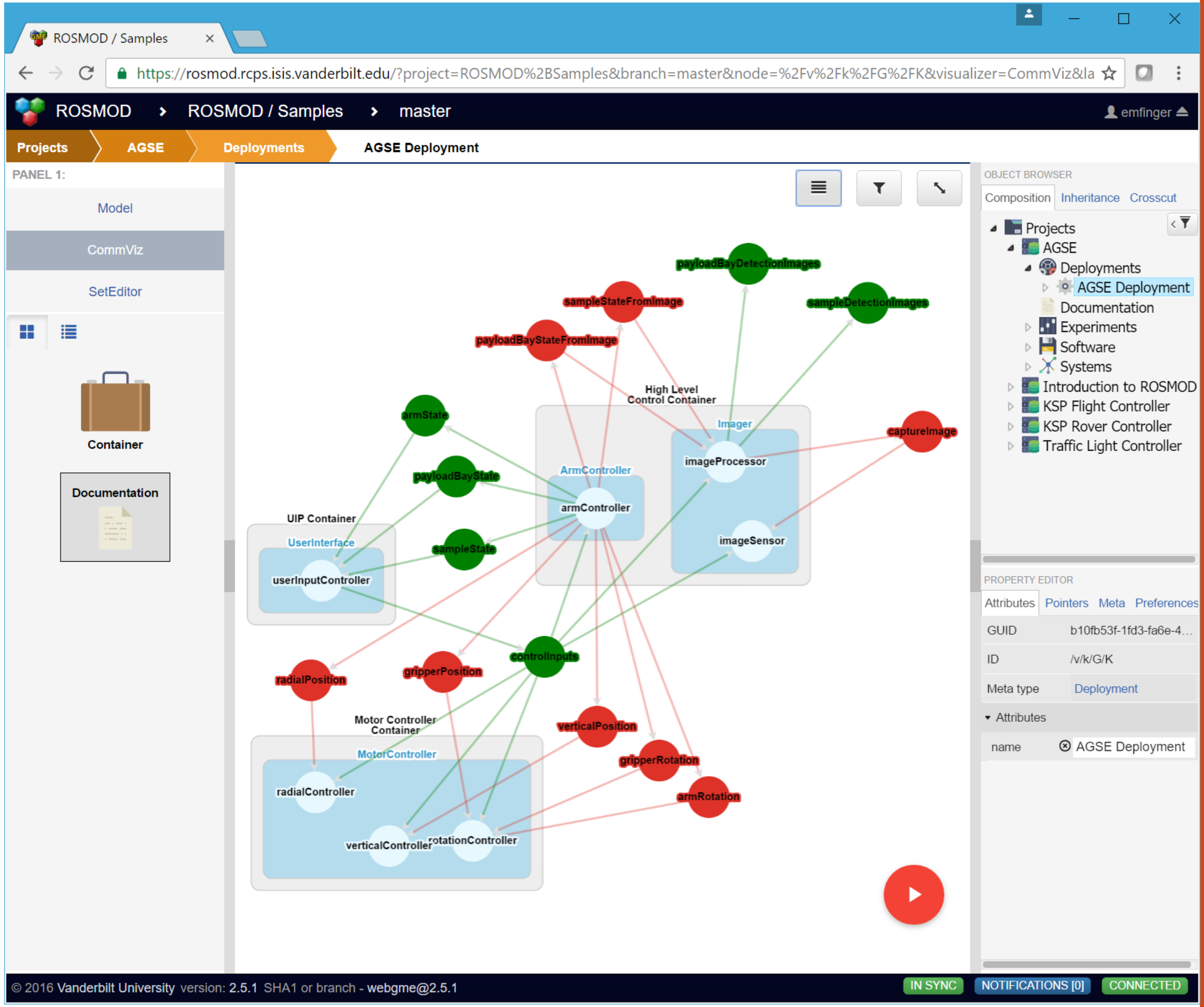
- Results Visualization is important for distributed systems
  - If users have to look through tons of text logs from different processes on different nodes, they will not use your system.
  - Visualization lowers the difficulty and time it takes to find execution errors in your code/system
- Interactivity is key
  - Static plots look nice, but have limited utility when actually analyzing or debugging the system.
  - Need easy methods for users to massage the plots/data into something more meaningful for their current context.
    - Remove extra plots / data
    - Zoom x/y/x&y
    - Pan



ROSMOD

Implementation: WebGME Components

- Users want to know what configuration is actually running in a deployment
  - Since not all software components may be used, and the components in use may not be correctly configured
- The actual deployment may be large and difficult to visualize with many connections
  - So again, interactivity is key
- Being able to let the user select their current *context/focus* is important as the scale of the models/systems increases



ROSMOD

Let's look at ROSMOD and its Code!

ROSMOD: rosmod.rcps.isis.vanderbilt.edu

CODE: github.com/rosmod/webgme-rosmod

(note: some components, like the CodeEditor, are dependencies maintained in their own repositories, just look at package.json to figure out where they come from)

Lessons Learned

- Meta specification is an evolving creature, depending on the changing needs of your platform, the target domain(s), and user experience
  - Iterative testing between over-generalization and over-dependence on implementation specifics
- As always: *documentation is important*
- Can't always just rely on WebGME built-in components
  - But also don't try to just make everything from scratch; many libraries exist that can help you do what you want to do
- The line between platform and (meta)model may shift over time and in some cases is actually more of a gray area
- When developing your code, better to err on the side of generalization / re-use since you don't know how your meta might change or what other projects may want to replicate your functionality
  - Be nice to the open-source community, since they've been so nice to you ☺
- Standardize your coding style and make use of libraries when possible to keep your code readable, e.g. **Q** for *promises* (instead of relying on callback functions

Thank you!

Questions?

Many acknowledgements to Brian Broll and Patrick Meijer, who were both quite patient and helpful in teaching me JS and WebGME.