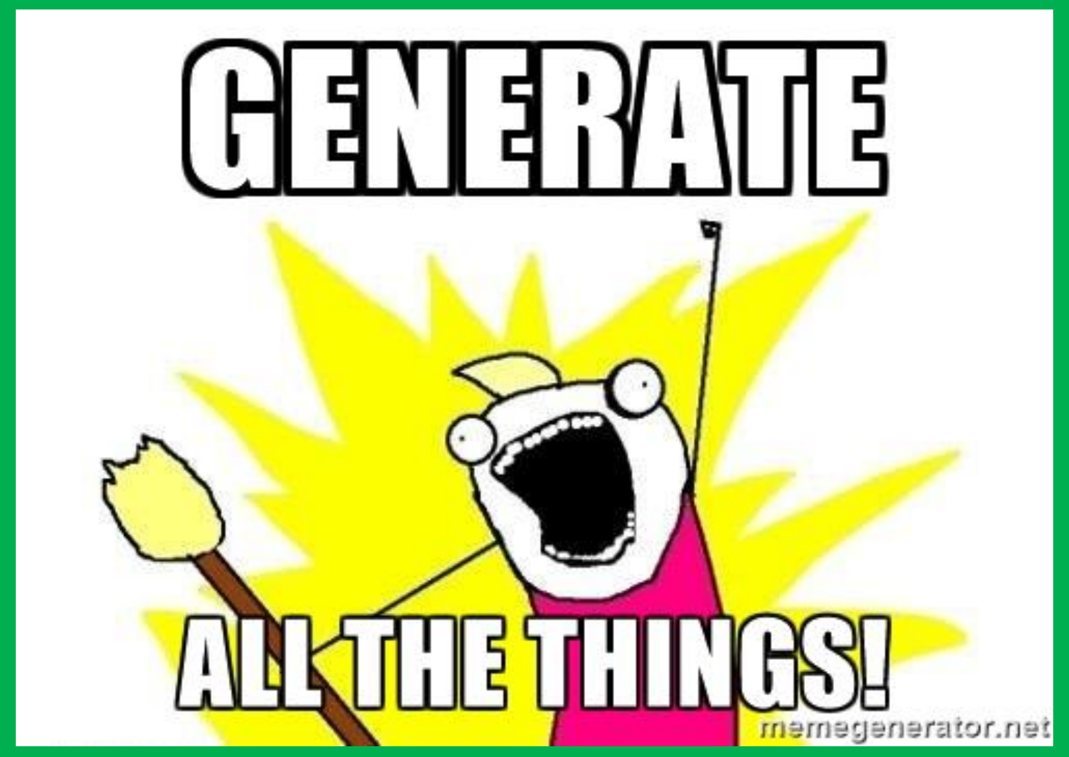
McMaster University

DRASIL: A Framework for Literate Scientific Software

Dan Szymczak, Steven Palmer, Jacques Carette, Spencer Smith
Department of Computing and Software, McMaster University

Sick of out of sync Software Artifacts?



Research Problem

Software artifacts contain a vast amount of knowledge duplication and transformation. A single piece of underlying knowledge should appear (in some form) in every artifact for a given piece of software.

In certain domains, like Scientific Computing, knowledge is duplicated across many different pieces of software. This duplication is almost exclusively performed manually, whether by copying/pasting from existing artifacts, or by transcription from another source such as a textbook. Artifacts tend to fall out of sync as the software is updated (especially code versus documentation), negatively impacting the overall software quality. This is most noticeable when it comes to maintainability, traceability, and verifiability.

Contact

Dan Szymczak: szymczdm@mcmaster.ca

Introduction

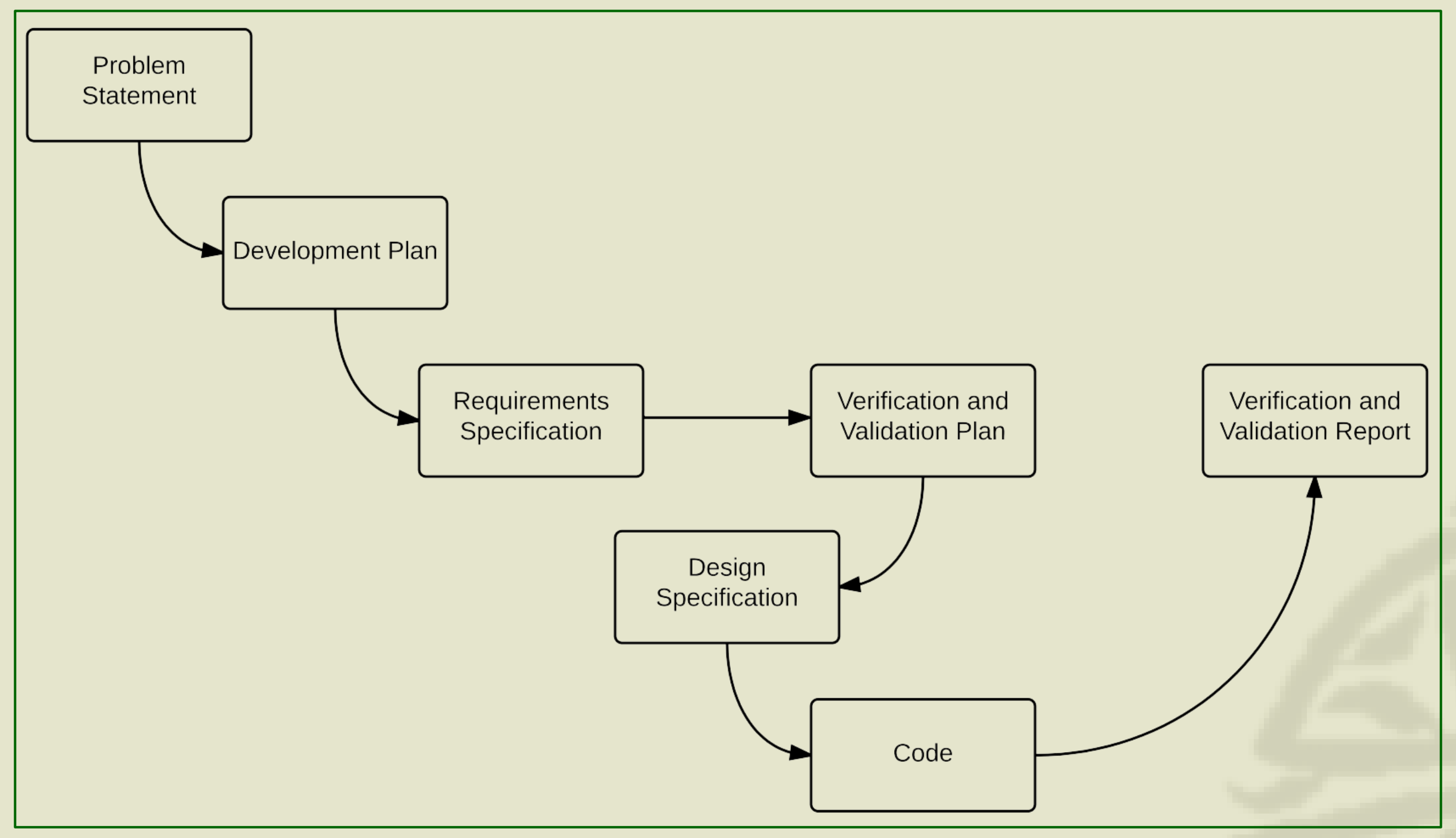

Problem Statement
Development Plan
Requirements Specification
Verification and Validation Plan
Verification and Validation Report
Design Specification
Code

Figure 1: A rational design process

Rational Design Process

- ❖ Leads to higher quality software *(Parnas & Clements 1986)*
- ❖ Tedious so ignored by many Scientific Computing (SC) developers *(Kelly 2007)*
- ❖ Necessary for certification
- ❖ Why not automate it?

Goals/Contributions

- ❖ The Drasil Framework
- ❖ Knowledge-based approach to SC Software Development
- ❖ Knowledge Capture
- ❖ Simplifying Software (Re)Certification

Figure 2: Knowledge Capture is necessary with Drasil

Approach

From Abstract Theories to Artifacts

I. Create a graph of the Abstract Theory
II. Obtain a graph of Instances by applying design choices to the Abstract Theory graph
III. Obtain a graph of the Executable Specification by applying execution choices
IV. Translate the Executable Specification to modules of Abstract Code
V. Render the final output

Note: Identifying the abstract theory and design/execution choices is a difficult process requiring the assistance of domain experts.

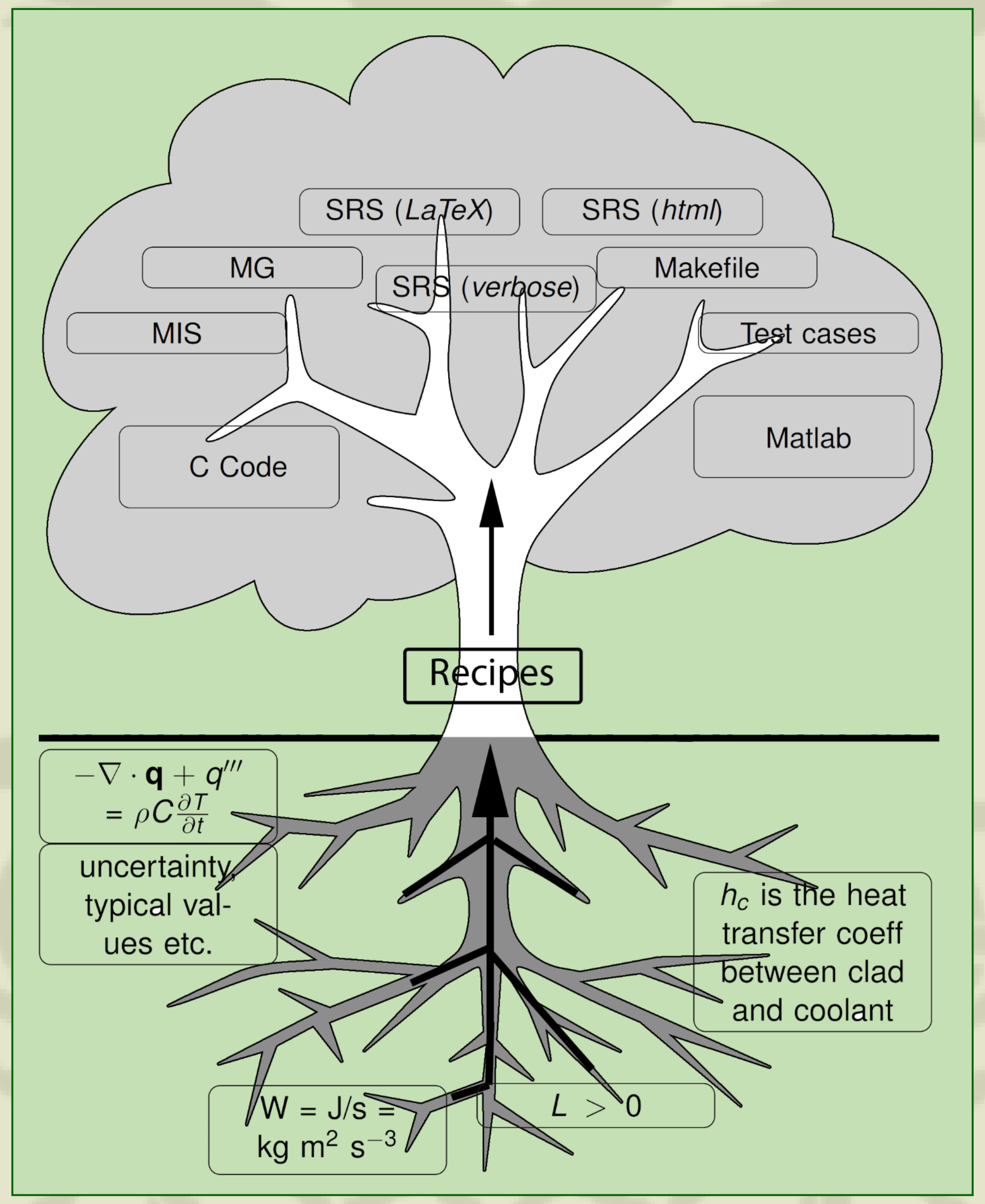

SRS (LaTeX)
SRS (html)
MG
SRS (verbose)
Makefile
MIS
Test cases
C Code
Matlab
Recipes
$-\nabla \cdot \mathbf{q} + q''' = \rho C \frac{\partial T}{\partial t}$
uncertainty, typical values etc.
h_c is the heat transfer coeff between clad and coolant
W = J/s = kg m^2 s^{-3}
$L > 0$

Figure 3: Our Knowledge-Based Approach – Recipes collect and transform knowledge into Software Artifacts

- ❖ Drasil is being developed through rapid iteration using a practical, example-driven approach
- ❖ Drasil is composed of a number of Domain-Specific Languages (DSLs) embedded in Haskell

Knowledge Capture

- ❖ Creating a knowledge base is central to the entire approach
- ❖ Knowledge is stored in *Chunks* and woven into software artifacts through *Recipes* (programs written using the DSLs)
- ❖ Complex chunks are built off simpler ones (Figure 4)

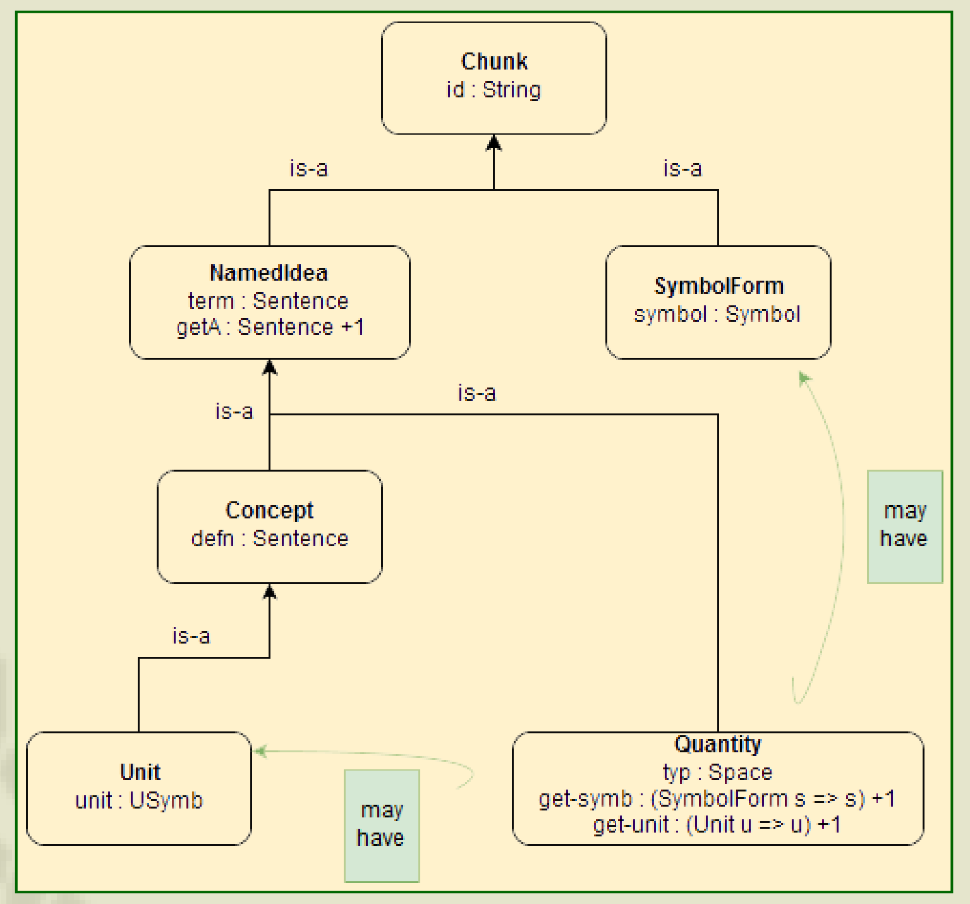


Figure 4: Chunk Hierarchy

Results

Code Generation

Incorporation of GOOL allows Drasil to generate output in a number of programming languages including:

❖ C++	❖ Java	❖ Objective C
❖ C#	❖ Lua	❖ Python

Document Generation

Drasil currently outputs documents to TeX and HTML formats. Our abstract recipes allow us to change formats on the fly, or produce versions in multiple formats simultaneously.

Disadvantages

- ❖ High initial time investment
- ❖ Adding knowledge is not simple; it requires domain experts

Advantages

- ❖ Internal consistency: necessary for (re)certification
- ❖ Common knowledge base promotes reuse; can be shared across many projects
- ❖ Pervasive bugs: much easier to find and fix