LINK: Augmenting Human Capabilities for Multi-Physics Model Generation

Final Report

December 15, 2021

Galois, Inc.

James LaMar, Yerim Lee, Chris Phifer,
Richard Jones (PL), Eric Davis (PI)

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112090064.
Distribution Statement: Approved for public release; distribution is unlimited.

1. Introduction

The CompMods program seeks the advance the state of the art in multi-physics modeling, addressing core challenges faced by scientists, engineers, and other decision makers and practitioners in digital modeling domains. Galois has addressed these challenges through the **open source** LINK framework, which introduces a new paradigm for specifying, developing, implementing, maintaining, and sharing multi-physics computational models. The results of this early proof-of-concept is an implementation that improves flexibility, prevents legacy lock-in to languages and frameworks, and promotes collaboration.

Prior to the Computable Models program, authoring single- and multi-physics models was a difficult, manual, and often expensive process that resulted in artifacts that were directly bound to non-physics implementation choices made at the start of the effort, resulting in the ossification of knowledge artifacts in forms that have low reusability, compatibility, and are difficult to leverage for additional scientific knowledge. This results in one-, or few-time use models that can be fundamentally incompatible with other computational models due to language, framework, or even the choice of problem formulation and underlying assumptions.

Physicists, and those that use the outcome of their research and development, currently must be experts in multiple disciplines, being experts in the domain science being represented; mathematicians and modelers so they can formulate their system correctly; data scientists capable of using ground truth observations to calibrate, fit, and assess their models; and software engineers capable of authoring high performance source code in their chosen language and framework. Even when these processes are carried out faithfully, and with full ability, their results are difficult to optimize, encode, and generalize to other adjacent use cases. The result is an end product for which correctness is neither assured, nor easy to ascertain.

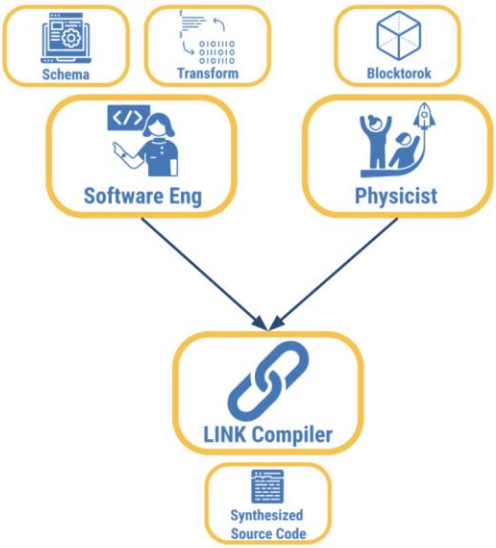
The process of generating multi-physics modeling is usually a collaborative one that can involve experts from multiple domains of physics, domain science, and mathematics. The process of multi-physics model construction is more realistically framed as a collaborative scientific endeavor that must first begin at assumption uncovering and alignment, followed by agreeing on methods and processes that result in compatible, comparable, and composable computational objects.

2. LINK Implementation

LINK attempts to advance the state of the art by fundamentally altering the process of single- and multi-physics model generation. The core hypothesis of LINK is that human-augmentation with automated reasoning can result in both a lower level of overall effort on the part of the domain scientist when generating single- and multi-physics models. As shown in the figure to the left, LINK subdivides tasks for single- and multi-physics model generation into three separate concern areas: the concerns of the **software engineer**, the concerns of the **physicist**, and the concerns of solver generation which are handled by LINK's own automated reasoning.

Physicists in this new model are responsible for a language-agnostic, and largely solver-agnostic, representation of their problem as a **Blocktorok** specification. Specifications are much like JSON objects: A set of key:value pairs where the value can be of any type. A blocktorok specification is tied only to a given problem in the physics domain, but it is agnostic to the backend in which the solver is to be implemented, and can be reused when implementing

a model with a new transform, for example, when a new solver is written that accelerates the runtime of physics modeling. Software engineers are responsible for defining a **schema** that informs a physicist on the details of their problem they must encode, and a **transform** that provides instructions to LINK on implementing the problem domain defined by a schema. A given schema is specific to a given physics domain, but reusable for multiple problems involving this sort of physical system, and for multiple backends. It is only necessary to write this once per problem domain. The transform contains the instructions for LINK to map input from a Blocktorok specification utilizing a given schema, to a given backend. As such it is highly reusable for any problem that works with a given schema, and backend, regardless of the Blocktorok specification provided.



Once all three components have been defined for a physics problem, LINK is able to synthesize backends to solve the specified problem and domain. More importantly, LINK allows low effort reimplementation of models using alternative methods, languages, and backends allowing a user to evaluate choices in solution technique, to examine the impact of differing methods or assumptions, and to better understand sensitivity to certain methods, simplifications, or assumed physical processes.

2.1 Initial Design of LINK

The original design of LINK focused on lessons learned from conversations with the IV&V teams, which highlighted the practical workflow challenges that result from modeling exercises. Currently, it's common for the different pieces of physics in a multiphysics problem to be solved separately, where results between different models (that represent part of the overall problem) are manually passed back and forth between representations or solution techniques. Individual physics models can be created by entirely different teams of expertise from distinct physics domains, so to take a single coupled approach can require early decision making and a lot of coordination.

The initial architecture for LINK focused on solving this problem with a single user-facing domain specific language (DSL) used to encode multiphysics problems. LINK took these results, and automated code generation from the definitions provided by a user. Users specified individual models, and the information flow process between these models in the LINK DSL. For incompatible models, LINK's first implementation created alerts to the incompatibility. If compositions respected assumptions, the LINK compiler then resulted in an orchestrated solution of multiple solvers. Our initial DSL design focused on providing a form of universal intuitive syntax that allowed the user to define multiple models to represent the different physics involved in a multiphysics problem as well as the information flow process between the models. In order to correctly compose these models, however, LINK needed to understand the assumptions inherent to both models, which we found were often encoded in different and

incompatible ways. The dimensions of these incompatibilities come in different forms, such as semantic types, units, and data storage requirements. Grounded in a shared representation, the hope was for LINK to check these assumptions defined in different physics models. Ultimately, however, we found this approach to be untenable due to the lack of common assumptions between solvers, and the high degree of solver incompatibility witnessed in practice.

In practice, our research has shown that scientists develop and implement these multiphysics problems by decomposing them into individual physics problems and encoding them as separated models and solution frameworks with differing assumptions that must be learned manually. Each domain of physics involved in the problem is modeled independently, and necessary information is passed between the models. Legacy code bases associated with these implementations made it difficult to both design a universal DSL for all physics domains, and difficult to automate the information exchange between models, and as such the information exchange was required to be handled manually. Hence, the current state of the engineering practices at most research institutions engaged in multi-physics modeling, such as NASA Langley, does not include coupled fluid flow and body heat transfer simulations. With these limitations, it would take practitioners months to years of development to create a simulation tool with coupling capability.

2.2 Redesign of LINK

The results of our initial proof-of-concept implementation with LINK led to a fundamental redesign of the way LINK approaches automated single- and multi-physics code generation. This decision was made due to the following core reasons:

- Different back-end solvers have wildly different assumptions, and encodings of these assumptions, making it difficult to target multiple backends effectively from a single DSL. This realization led directly to the development of LINK's new **transformer** functionality, approach, and language, which asks developers of frameworks to produce small sets of instructions on implementing a given physics domain in their schema. The level of effort associated with this development is quite low, and a one-time, reusable effort for any model using this backend, and requires approximately 20-30% of the level of effort required to develop a single-physics model in the same framework.
- There currently does not exist a universal DSL for physics, or even the initial building blocks, as exists in neighboring domains such as systems biology and chemistry. In lieu of such a representation, we've designed LINK to allow for multiple representations supported by the common **Blocktorok** specification. Authors of back-end frameworks and solvers can instead design a **Schema** that physicists and other scientists can use to instantiate their model in a back-end and solver agnostic manner.
- Separation of concerns: in working with IV&V teams we've come to better understand the way cognitive load and concerns impede the process of physics model generation. In many cases, simply learning the back-end solver's limitations, assumptions, and requirements is itself a monumental task. Our chief engineer tasked with this portion of

the effort spent approximately 75% of her time on the project learning OpenFoam, SU2, and the other solver frameworks utilized by this program. By instead focusing on methods for the solver's own authors to expose their functionality in a more generalized way, we hope to improve collaborative opportunities, and to speed the development of correct and accurate multi-physics simulators using legacy solvers.

At a high level, the LINK workflow can be broken down as follows:

1. Data layouts are described through schemas written in the `blocktorok-schema` language. See Appendix A.1 for detailed documentation;
2. Transformations to output formats are described through templates written in the `blocktorok-transform` language. See Appendix A.2 for detailed documentation;
3. Data formatted in the `blocktorok` metalanguage is fed, together with the schema it is intended to satisfy and the appropriate transformer, to the LINK compiler, which uses the schema to validate the data and the transformer to produce output. See Appendix A.3 for an example schema, transform and metalanguage inputs for a single-physics problem that uses either SU2 or OpenFOAM as the backend with identical front-end inputs.

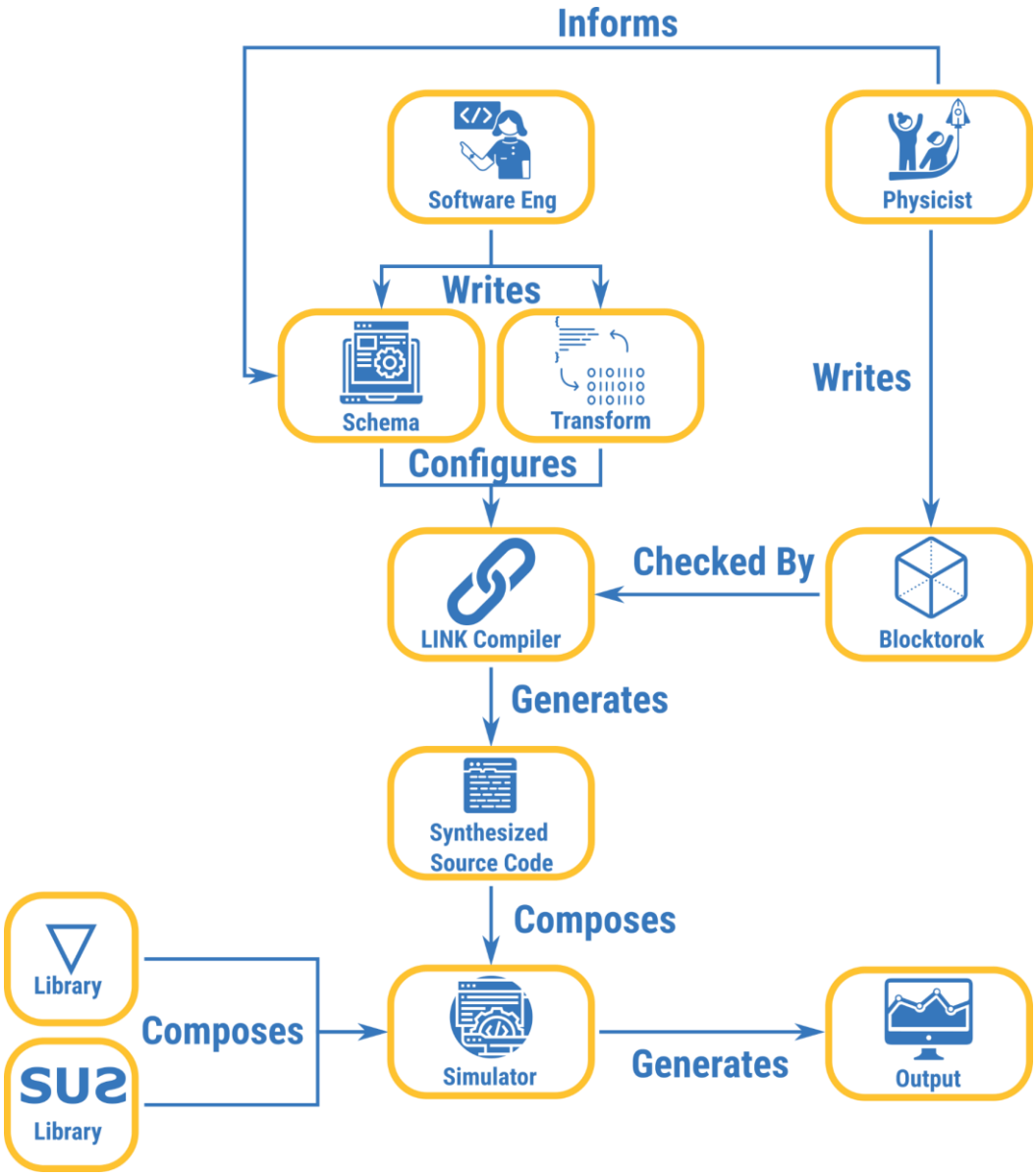


Figure 1: LINK Architecture

Figure 1 captures this intended `blocktorok` workflow at a high level, and illustrates the overall architecture of the LINK framework. It is worth noting that, on a problem-by-problem basis, not all of the user-provided components (schema, transform, and blocktorok specification) need to be adjusted: It is our intention that software engineers write schemas and transformers once, while physicists and domain users (lab technicians or simulation writers) produce blocktorok specifications for the problems they are simulating. In short: the software engineer produces write-one-use-many artifacts, while the physicist produces problem-specific artifacts leveraging the reusable work done by the software engineer.

Blocktorok is a language ecosystem for expressing data schemas, encoding data that matches these schemas, and transforming these data into arbitrary targets defined by the user. Blocktorok features simple syntax, static analysis, and the flexibility to express data across many domains in ways natural to practitioners in those domains.

The design of Blocktorok was motivated by problems in the usability of tools for modeling physical phenomena: Labs often use custom tools, making collaboration between groups difficult and tedious. With Blocktorok, a common language can be agreed upon and individual labs can continue to use their own tools simply by defining a transformer from the common language to those tools' input languages. In particular, it is possible, through the use of multiple transformers, for a single expression of a physics problem to be translated to the language of sophisticated (but challenging to use) systems such as SU2 and OpenFoam.

Blocktorok is available as open source software here: https://github.com/GaloisInc/blocktorok

2.2 NASA Challenge Problem

Hampering our attempts to work with the core challenge problem presented by our IV&V partner was the lack of a state of the art implementation. This makes it difficult to meaningfully compare accuracy, performance, and other characteristics. As such, we have generated our own reference implementations with NASA in absence of an existing implementation of the challenge problem. Working with the NASA team, we defined, implemented and tested LINK on the following challenge problems:

- Single Physics: Heat transfer along a rod with both SU2 and OpenFOAM backends
- Multi-Physics with single backends: Three cylinders in a fluid with SU2 or OpenFOAM backends for all of the physics domains

This redirection allowed Galois to focus on innovation in the toolchain necessary to achieve reliable multi-physics modeling outcomes instead of spending considerable time and effort getting backends working on complex problems.

2.3 Results of the NASA Challenge Problem

We have successfully written the necessary schemas and transforms for two different backends (SU2 and OpenFoam) to solve the NASA challenge problem. As part of our experiments, we required the schemas for the two problems to be similar enough, to allow reimplementation from a single blocktorok specification. A single specification was written to model the problem of heat transfer along a rod, with the schema, transformation, and blocktorok specification files given in the appendices of this report.

In addition, we successfully demonstrated LINK's use for multi-physics problems by creating the blocktorok specifications for three heated cylinders in a cool fluid, involving fluid flow, and conjugate heat transfer. The results of these experiments are also given in the appendix of this report.

3. The LINK Model for Program Synthesis

In our revised architecture, LINK is now more capable of leveraging artifacts created by physicists and software engineers defining model schemas, specifications, and transforms; which allow LINK to easily, and with a low level of effort, reimplement a single or multi-physics simulation in a new backend, either for validation purposes, or to take advantage of a new

framework with additional capabilities. Towards this end, the schemas and specifications used by Blocktorok are **backend agnostic**, and define the problem in a general form, implementable on many backends. The transformer, by comparison, gives LINK a blueprint for implementing specifications that conform to a given schema for a particular backend.

Blocktorok

```
boundary_conditions: [ Isothermal { marker: "left" value: 573.0 }
                     , Isothermal { marker: "right" value: 273.0 }
                     , Heatflux { marker: "bottom" value: 0 }
                     , Heatflux { marker: "top" value: 0 }
                     ]
```

Schema

```
block bc_val {
  marker: string;
  value: float;
}

union boundary_conditions {
  Isothermal bc_val;
  Heatflux bc_val;
}
```

Figure 2: Example of a Blocktorok specification and schema, showing isothermal and isoflux boundary conditions, with color coded mappings showing corresponding sections of each definition.

Figure 2 shows an example of a fragment of a Blocktorok specification (upper) and a fragment of a Blocktorok schema (lower). These fragments define a boundary condition, which the schema tells the scientist can be either an isothermal boundary condition (the temperature at the boundary is constant) or an isoflux boundary condition (the heat flux at the boundary is constant).

Blocktorok

```
boundary_conditions: [ Isothermal { marker: "left" value: 573.0 }
                     , Isothermal { marker: "right" value: 273.0 }
                     , Heatflux { marker: "bottom" value: 0 }
                     , Heatflux { marker: "top" value: 0 }
                     ]
```

Transform

```
render ::boundary_conditions.Isothermal
  |  ${marker}
  |  {
  |    type  fixedValue;
  |    value uniform ${value};
  |  }
...
  |boundaryField
  |{
  |${vjoin(simulation.boundary_conditions)}
...
```

Synthesized Source Code

```
boundaryField
{
  left
  {
    type  fixedValue;
    value uniform 573.0;
  }
  right
  {
    type  fixedValue;
    value uniform 273.0;
  }
...
```

Figure 3: Example of a Blocktorok specification and transform, for the same schema shown in Figure 2. This transform tells LINK how to render specified boundary conditions in OpenFoam, with color coded mappings showing corresponding sections of each definition.

Figure 3 shows the implementation of this boundary condition in OpenFoam with a transform. Transforms provide instructions for mapping the contents of a specification that conform to a schema the transform is written for, onto a back end, in this case synthesizing OpenFoam code automatically.

While this is useful in isolation, as it separates the concerns of model synthesis to those users with the appropriate skill sets, and that are likely to make the least errors, it is also leveraged to lower the level of effort associated with redevelopment of models for new frameworks.

Blocktorok

```
boundary_conditions: [ Isothermal { marker: "left" value: 573.0 }
                     , Isothermal { marker: "right" value: 273.0 }
                     , Heatflux { marker: "bottom" value: 0 }
                     , Heatflux { marker: "top" value: 0 }
                     ]
```

Transform

```
render ::boundary_conditions.Isothermal |${marker}, ${value}
...
render ::domain
  ...
  |MARKER_ISOTHERMAL= ( ${join(", ", boundary_conditions.Isothermal)} )
```

Synthesized Source Code

```
...
MARKER_ISOTHERMAL= ( left, 573.0, right, 273.0 )
MARKER_HEATFLUX= ( bottom, 0.0, top, 0.0 )
...
```

Figure 4: *Example of a Blocktorok specification and transform, for the same schema shown in Figure 2. This transform tells LINK how to render specified boundary conditions in SU2, with color coded mappings showing corresponding sections of each definition.*

Figure 4 shows the implementation of this boundary condition in SU2 with a new transform. It is important to note that neither the **schema** or the **specification** need to change in any way to produce this transform. All that is required is that a software engineer implement a novel transform for a given physics area, utilizing previously authored schemas and specifications.

4. Metrics

The LINK framework seeks to lower the overall level of effort associated with the development of single- and multi-physics models, to better separate concerns on the basis of expertise and skills, and to ensure high levels of reusability and compatibility to smooth over barriers to collaboration between scientists. LINK seeks to lower this level of effort not only at the point of initial authoring, but during the entire process of the model lifecycle.

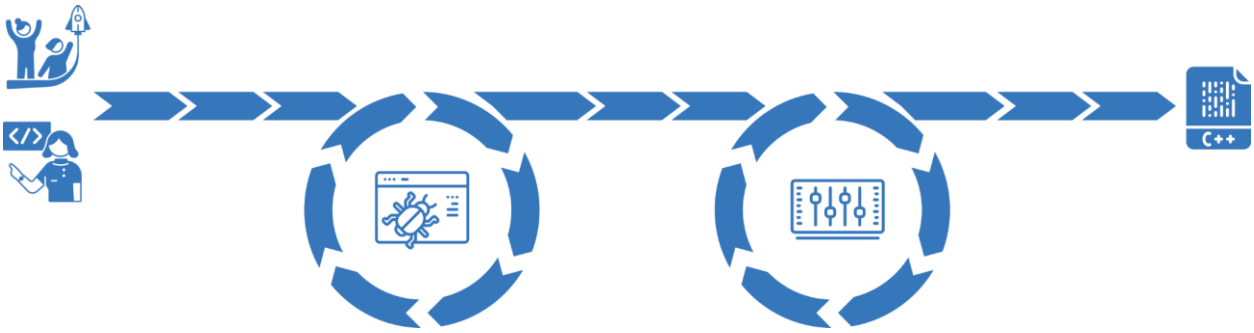


Figure 5: An illustration of the model development lifecycle.

Figure 5 shows a typical model development lifecycle, wherein a joint team of domain experts, subject matter experts, and software developers work together to first derive a specification (typically informal) of the model that the scientists want to produce, its requirements, and then select a back-end framework within which to implement a hopefully high performance version of the solver. Unfortunately, these decisions are made very early on, and result in the "ossification" of knowledge products as code that is bound to a set of assumptions that have more to do with the framework and language used to implement the solver, than the needs of the problem domain itself.

Once a framework has been selected, and committed to, the model must then be debugged, parameterized and optimized, and then can be used to solve one very specific problem. With luck, the implementation will be easily adapted to nearby problems in the same domain, but this is rarely the case in practice.

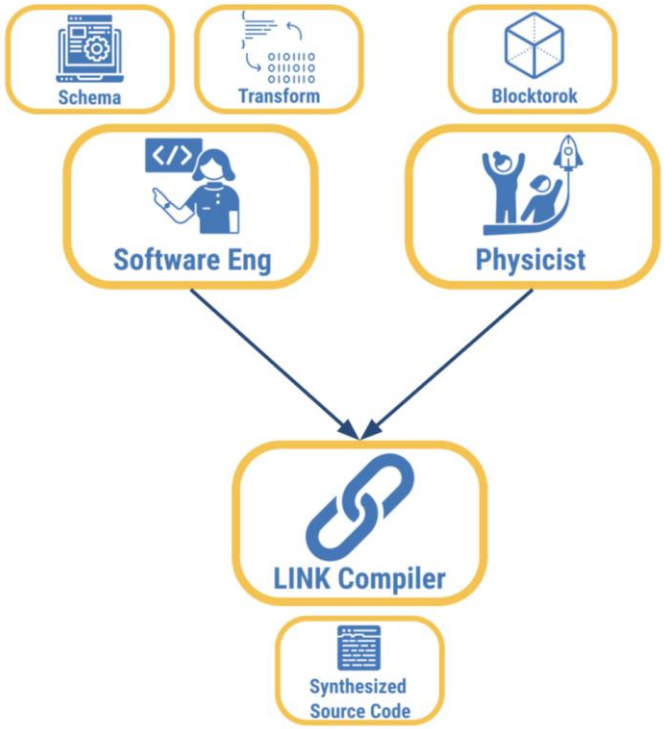


Figure 6: Separation of concerns and level of effort in LINK.

LINK attempts to reduce the level of effort for single- and multi-physics code by impacting the entire lifecycle process, reducing the level of effort on any one performer in the development process, lowering the complexity of single and multi-physics model development and shortening the model development cycle; while also lowering the overall sum of the level of effort required to implement the model. The primary strategy employed by the LINK framework is to separate the concerns of model development, according to those shown in Figure 6. The software engineer is responsible for the development of the schema and transform, while the physicist or SME is responsible for authoring the Blocktorok specification of the problem. The LINK compiler then takes over the task of authoring the code for a given backend, ideally resulting in an overall reduction of the level of effort, and a separation of concerns that limits the human effort involved, and divides tasks according to expertise and knowledge.

A major achievement of the LINK program of research was also the high reusability of the Schemas, Transforms, and Blocktorok specifications. We conducted further experiments documented in section 4.10 with NASA to better understand the usability and extent to which this code reuse could be leveraged in practice.

4.1 Measuring Level of Effort

We quantify and measure the level of effort (LoE) associated with the development of single- or multi-physics models by finding proxy measures for program complexity. We explore two primary measures of this complexity in this section, total number of variable assignments that must be manually handled by the physicist and software engineering teams, and Kolmogorov Complexity.

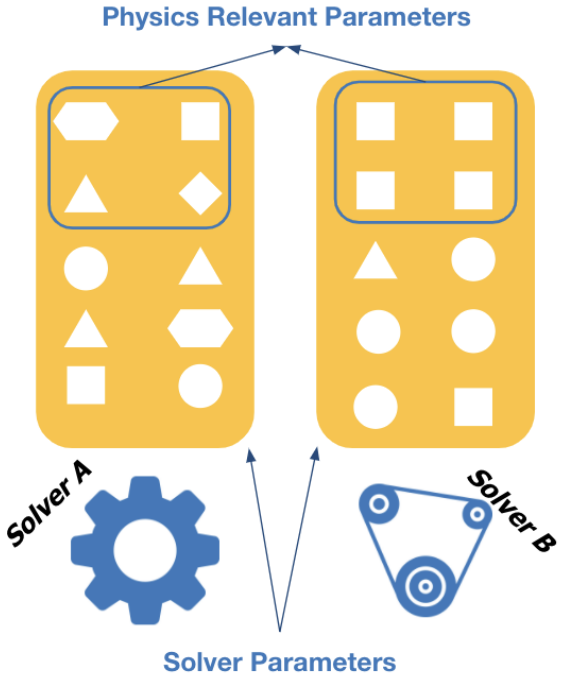


Figure 7: Example of parameter reduction opportunity

Figure 7 shows an abstract representation of two solvers, A and B, and the parameters they require a user to understand, parameterize, and account for. More concretely, for the NASA conjugate heat transfer problem, there are a total of 165 variables that must be accounted for, and assigned manually, when implementing the program in OpenFoam. We discovered that a large number of these parameters, while critically important to the solver's correct function, could be instantiated automatically with LINK via the transform. In this case, these parameters deal with specifics of the solver method being utilized.

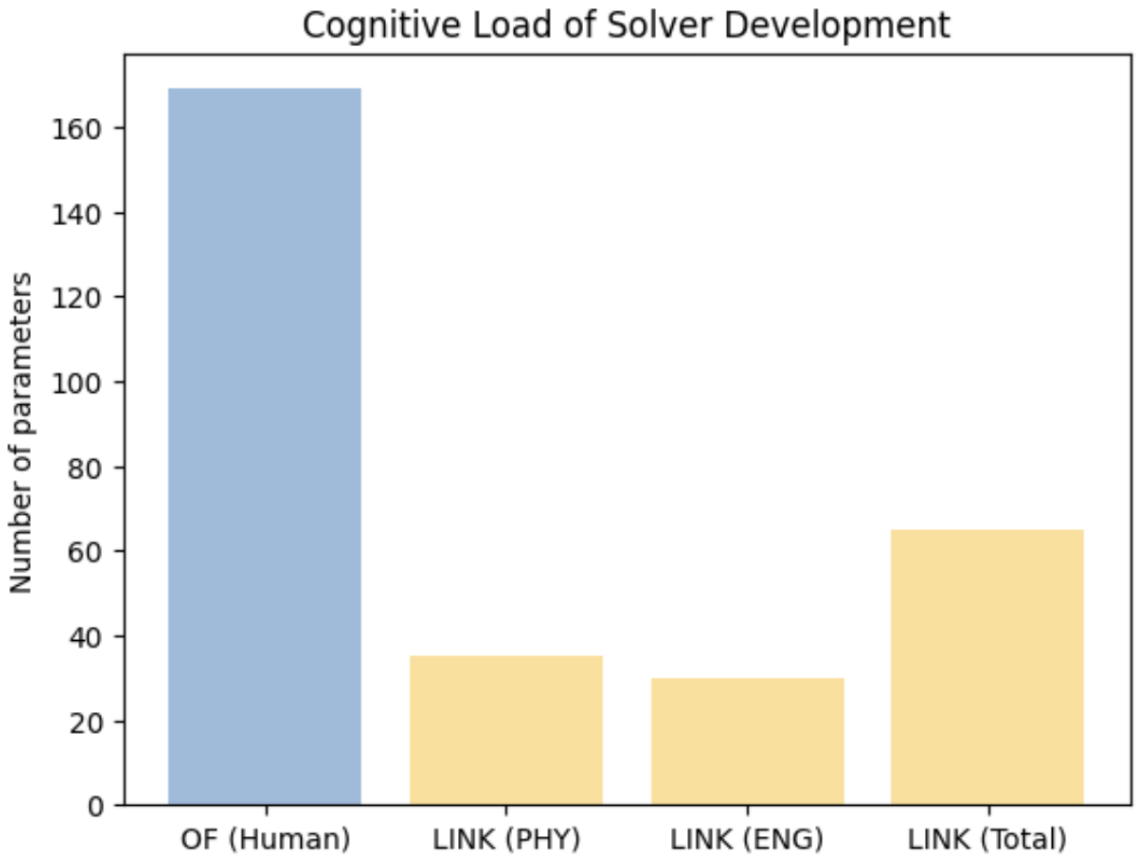


Figure 8: Cognitive load reduction associated with total manual parameter requirements of traditional vs. LINK implementations.

As shown in Figure 8, we found that while a solution in native OpenFoam requires a user to manually understand and set 169 parameters, of these only 30 need to be set by the software engineer for the given domain of physics associated with the schema, and 35 need to be set by the physicist to encode the problem. This reduces the number of variables that need to be understood, and accounted by the physicist to just 18% of those in a traditional monolithic implementation. A total of 65, or 38%, are required to be understood and set by any human. The remaining 104 are handled by LINK automatically.

We propose that LoE is related to program complexity, which in turn is related primarily to the size of the program. To a first approximation, a longer program takes longer to write, to debug, and to properly parameterize and validate. The challenge then becomes finding a way to quantify the length of a program in a way that is somewhat predictable despite variations in

programmer style and preference, for example the presence or absence of comments and whitespace and use of short versus long identifier names.
The Kolmogorov complexity[1] metric suggests a solution which is independent of choice of representation: find the shortest program which emits the text of the program to be measured. While Kolmogorov complexity is not computable[2], the compressibility of the program text serves as an approximate upper bound on the value expected of the the text's Kolmogorov complexity. By compressing the program *P* with some method yielding *C(P)*, and implementing the decompression algorithm in the chosen language as *D*, and concatenating it to the compressed program, we can measure the resulting string of decompressor and compressed program *D + C(P)* to achieve a proxy measure of Kolmogorov complexity which we call K(s).

4.2 Caveats

Kolmogorov complexity (and our computable proxy: compressibility) considers the entire program text, including comments (which are informative to the reader, but do not contribute to the behavior of the program) and translation directives of various kinds (e.g. compiler directives, pragmas and makefiles). In our initial analysis we compute K(s) via this method, not discounting any comments, or identifier lengths. In our future work, we will write parsers to more accurately measure K(s), but the numbers provided in this report are a hard upper bound with significant reductions, proving the feasibility of our approach.

4.3 Alpha-conversion

When using compressibility as a proxy for Kolmogorov complexity, the resulting metric is strongly affected by identifier length. We will address this in future implementations of LINK by renaming user-defined symbols to have the minimum number of characters needed, based upon the number of such symbols in the program (using techniques such as Huffman Coding). Likewise, while literal strings are essential to the proper function of a program, they do not (absent use of an eval-like function) contribute to the essential complexity of a program; we will normalize all strings to a fixed length and content. Symbols which are essential to describe the interpretation of a program, such as library function names and reserved words in the language, are not subjected to alpha-conversion. Keyword argument names are considered an essential part of function invocation and are not subjected
to alpha-conversion.

4.4 Keyword normalization

The set of reserved words and directives varies according to programming language. After alpha-conversion, we will assume that remaining unconverted symbols are specific to the language and convert these to tokens of uniform length.

4.5 Other Considerations

The implementation of the compression method used as a proxy for Kolmogorov complexity will affect the compressibility metric. This will be particularly true in the case of short program texts, as the size of the dictionary (which is part of the compressed text) may, for example, be padded

[1] Li, Ming; Vitányi, Paul (1997). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer. ISBN 978-0387339986.
[2] Wallace, C. S.; Dowe, D. L. (1999). "Minimum Message Length and Kolmogorov Complexity". *Computer Journal*. **42** (4): 270–283

to some minimum size for the convenience of the decompression algorithm. Implementations of a given compression method may differ in their encodings while still being considered to be the same method; we don't know how significant these differences may be as regards a compressibility measure. We will attempt to vet these risks and concerns during our future work on the program.

4.6 Validation of Approach

We tested the approach of combined alpha-conversion and keyword normalization on a set of four C programs having identical source code except for the order in which functions are defined. An identical set of function declarations at the head of each variant program allowed for the function definitions to be reordered without otherwise changing code. Our tests demonstrate that the order of function definition changes the Kolmogorov complexity estimate by approximately one percent; we take this as confirmation that insignificant differences in source texts will have an insignificant effect upon the complexity estimate. Our tests also illustrate the need for alpha-conversion in order to impose a consistent length of user-defined identifier.

4.7 Other Approaches

We attempted to apply Latent Semantic Indexing as a measure of document similarity. We thought of this not as LOE for a specific program text, but rather as a measure of the effort required to transform one text into a larger, more complex text (e.g. Blocktorok to OpenFoam). We quickly discovered that the method is not useful when scaled down to using a single document as a corpus.

We considered and rejected the use of edit distance (e.g. Levenshtein distance) to measure document similarity. The edit distance is sensitive to spelling variations; transforming a program by simply renaming all of its identifiers will result in a nonzero edit distance even though the two variations are functionally identical. We have considered the use of structural metrics, however measurement of these metrics depends upon the ability to build an AST from the source text.

4.8 Results

Figure 9 shows the results from our initial level of effort experiments. The leftmost figure shows the traditional development path, resulting in a monolithic simulator in OpenFoam with little to no reusable parts. The total level of effort calculated was **76,181 K(s)**. The right most figure shows the same process in the LINK framework. In this case the software engineer develops the schema, calculated at **19,457 K(s)**, and the transformer, calculated at **20,037 K(s)**. The total LoE for the software engineer is thus **39,494 K(s)**. The physicist in this case is responsible only for the blocktorok specification, calculated at **19,025 K(s)**. The total combined level of human effort in the LINK framework of our first release is thus **58,519 K(s)**.

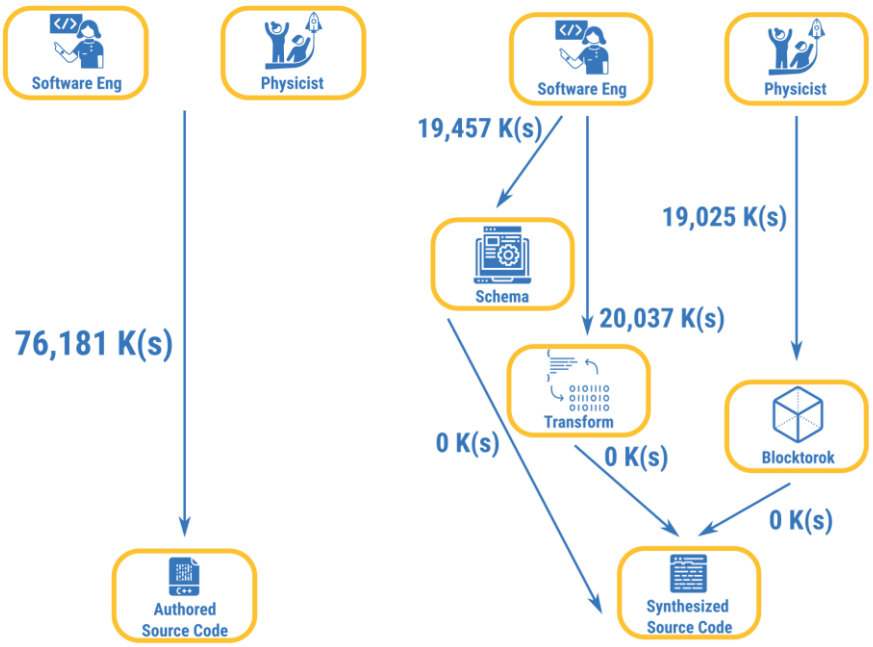


Figure 9: LoE for the development of the Steel Rod problem in OpenFoam

This results in an overall level of human effort that is a 23% reduction when compared to the current state of the art. The full power of LINK, however, results from the reusability of the artifacts generated by the separation of concerns. Writing a model in LINK involves authoring three primary artifacts, that the LINK compiler uses to synthesize code for the solver, the schema, transform, and blocktorok specification.

- **Schema** - The schema is specific to a given physics domain, but reusable for multiple problems involving this sort of physical system, and for multiple backends. It is only necessary to write this once per problem domain.
- **Transform** - The transform contains the instructions for LINK to map input from a Blocktorok specification utilizing a given schema, to a given backend. As such it is highly reusable for any problem that works with a given schema, and backend, regardless of the Blocktorok specification provided.
- **Blocktorok specification** - The specification in Blocktorok is likely the least reusable part, as it specifies a given problem in the physics domain, but it is agnostic to the backend in which the solver is to be implemented, and can be reused when implementing a model with a new transform, for example, when a new solver is written that accelerates the runtime of physics modeling.

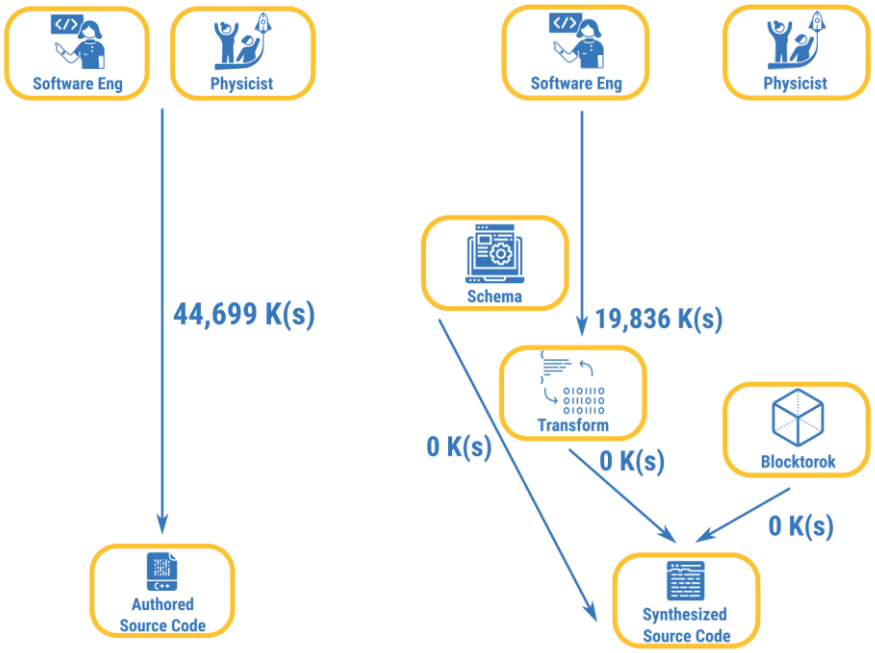


Figure 10: LoE for the redevelopment of the Steel Rod problem in SU2.

Figure 10 shows this sort of reimplementation workflow for our steel rod challenge problem in SU2, building on the prior work shown in Figure 9. As can be seen in the leftmost part of the figure, the traditional SU2 implementation of this challenge problem has a complexity of **44,699 K(s)**. The right most diagram in the figure shows the LoE associated with the LINK architecture to generate the same code. In this case both the original schema and blocktorok specification are fully reusable. The only new code that must be written is a new transform by the software engineer, at **19,836 K(s)**. This results in a 55% improvement over the state-of-the-art when reimplementing in multiple backends.

4.9 Accuracy

As a proof-of-concept for the high-level forms of accuracy checking we believe to be possible with the LINK architecture, we have fully implemented a system in which unit-bearing quantities can be statically checked for compatibility and converted to other dimensionally consistent units during transformation. Figure 9 demonstrates at a high-level the kind of accuracy checking our framework provides.

In schemas, a software engineer may indicate that a float-typed field also bears a unit (see Appendix A.2 for the notation used); this serves both to indicate the dimension of the field in question (e.g. velocity, time, distance, energy, etc) and to specify what units the quantity will be expressed in upon transformation if no explicit conversion is performed.

In data specifications, a physicist may specify, for a dimension-bearing field, the units of their data (see Appendix A.1 for the notation used). Upon compilation, a conversion to the unit

specified in the schema will be attempted; if this is impossible due to a dimensional mismatch, an error will be reported in the data input (Figure 11).

Conversion can also be performed explicitly in a transform using the `convert[<unit>](<expr>)` form of expression. Again, the compiler checks that the provided unit is compatible with that of the schema; if not, an error will be reported in the transformer (Figure 11).

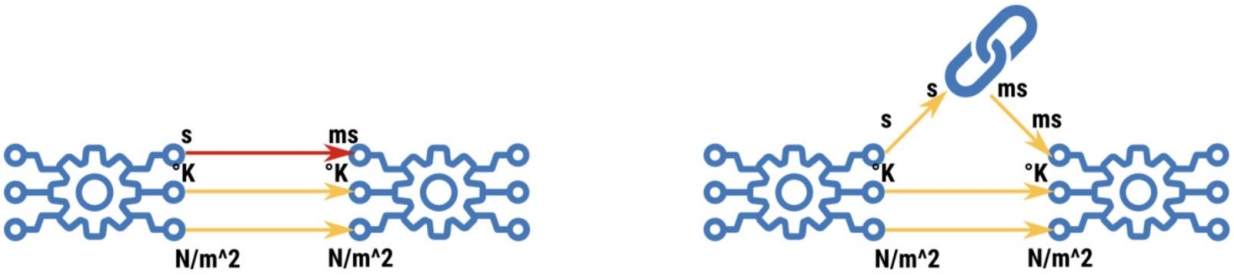


Figure 11: Correctness Corrections by the LINK Framework.

```
/Users/cphifer/Develop/steel-thread-model/test_cases/basic_units/input.blok:3:14-3:15: 'g' has a different dimension than 's'
```

Figure 12: Example error converting data unit to schema unit.

```
/Users/cphifer/Develop/steel-thread-model/test_cases/basic_units/transform.oct:7:23-7:27: Cannot convert from 's' to 'g'
```

Figure 13: Example error converting data unit explicitly in transform.

This approach has the potential to save on Level of Effort, as well as accuracy, composability, and conservation, as problems of this nature which can either be corrected automatically and do not require additional effort from the domain experts and scientists to account for, and correct, differing assumptions on units; or they can be corrected with machine assistance, providing domain specific feedback for the domain expert, or software engineers, to use in locating the conflict and determining either a proper resolution, or deciding a given multi-physics composition is infeasible. We believe that there is a lot of potential to expand this thinking about units to include other domain-specific accuracy checks, such as stiffness properties that arise when working with multi-scale models.

4.10 NASA/IV&V Implementation

The LINK team worked closely with Matthew O'Connell of the Computable Models IV&V team to apply LINK to a pair of proprietary NASA physics simulation backends. Implementation was done in a "pair programming" style, with Matt writing the schema, transform, and blocktorok code while Galois engineers observed and advised accordingly.

Over the course of three sessions of about an hour each, Matt was able to encode his problem in LINK's languages such that it produced usable output and it was evident how this approach could be extended to other NASA backends. Matt's opinion after using the tool suggests that one of the major things that is valuable about LINK's approach is to help with workflows at NASA where several different simulation backends are applied to the same problem and the

results are subsequently compared. Normally this is a manual, error-prone process, but Matt sees some potential in LINK to automate translation of the same problem into different simulation frameworks. Matt also found the ability to schematize the problem and to have units be explicit and checked to be a valuable feature.

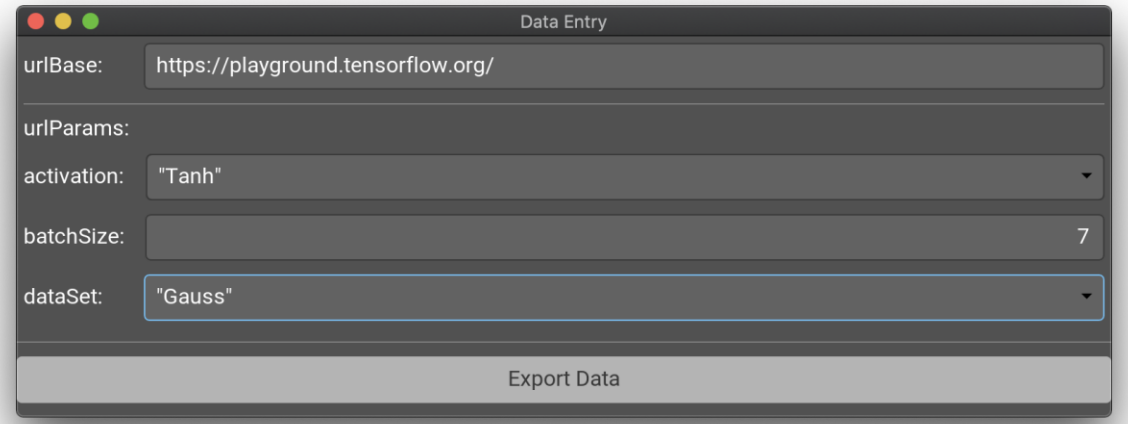
5. Recommendations for Future Research and Development

In this section we discuss our recommendations for future programs in computable models, and discuss our plans for future work with the LINK framework.

5.1 Level of Effort and Human Usability Considerations

Below is a list of Blocktorok usability ideas and thoughts that could be considered when further reducing the time and expertise required to use various back-end solvers.

- A way to visualize (probably using paraview) the problem described by LINK. A form of validation for the users regarding the LINK input;
- More expression support in the transformer language (let people do math), and/or support for schematization of expression languages (let people define the kind of expressions they want to work with);
- Schema-bound data invariants; kind of like ‘require’ in the transformer language, but at the type-level/statically checked. Basically, a fancier type system (as a bonus, validation of transformer definitions against schemas);
- Use schemas to generate graphical interfaces for data entry; generally, move away from fully text-based solutions. Here’s a screenshot of what that might look like:



- Generate more composability/correctness/conservation cases beyond `units`, allowing automated error correction and more explainable error messages for a broader spectrum of problem specification functionality and model incompatibilities;
- Improved tooling in editors (e.g. VSCode) through syntax highlighting, code completion, and other language server conveniences;

- It's unclear that `render` is a good feature given the conceptual difficulty people have in understanding it; are there other ways to make this metaphor clearer? More generally, if we explicitly pitch the transform language to programmers, does that change its design at all?
- Much of the blocktorok setup lends itself well to the building of - in some ways it could almost be thought of as an IDL. This would allow parts of the LINK workflow to be defined as APIs. For example: a schema could easily describe a data structure which can be validated and subsequently converted to blocktorok or have transforms applied to to it directly - similarly one could imagine writing a transform in a general purpose programming language instead of using our transform language
- In one way LINK is a tool for building abstract descriptions of problems that can be translated into simulation backend input, but in another way LINK could be seen as a tool for transforming backends. Is there a way to think about the problem from this angle that allows us to work around the impedance differences between different solvers while still allowing for physics-specific (ex. conservation laws) checking to be implemented? Put another way, it seems to be impossible to project physics in the abstract into the space of simulation, but is it possible to project simulation into the space of physics?

5.2 Mesh Conversion Challenges

One of the main challenges in mesh conversion is that there is not a single representation for meshes. Unfortunately, this lack of standardization is a recurring problem for unifying physics representation in different backends as well. Refer to section 5.3.

Different backends have different mesh representations which makes mesh conversion between backends challenging. As a particular example of challenging representation differences, consider SU2 and OpenFoam. SU2, geometrically speaking, uses a point- and surface-oriented view of a mesh; connectedness and closure is, in some sense, explicit. Conversely, OpenFoam uses a face-oriented view of meshes. Connectedness is significantly more implicit, and surfaces are described indirectly by the faces comprising them. Conversion from one of these views to the other is possible through geometric reasoning, but there was not sufficient time to do so before delivering our final result. This particular challenge highlights a theme in the broader challenges faced throughout the program: Currently, there is little to no agreement on common standards between different physical simulation engines, especially around assumptions and geometric concepts such as meshes. Lack of a standard is a problem in both meshes and the backends.

5.3 Challenges Unifying Different Backends

Our initial attempt of unifying different simulation backends led to the realization that different simulation engines have different representations and simplifying assumptions. As such, our attempt at unifying different physics simulation engines was virtually impossible without any prior knowledge of the backends.

LINK targets SU2 and OpenFoam, two well-known and established computational fluid dynamics simulation engines. Despite the fact that they both target the same physics domain (computational fluid dynamics), there are large discrepancies in how the backends operate that led to the creation of blocktorock, a backend agnostic language framework.

The lack of a unifying theme between SU2 and OpenFoam starts from problem descriptions to numerical schemes calculated during simulation run. In OpenFoam, problems are specified at the level of variables for the equation that's currently being solved (pressure, velocity, temperature, …). On the other hand, SU2 problems are specified at a higher level using keywords. For instance, OpenFoam requires the user to set the numerical scheme for each term that appears in the solver, where each solver corresponds to a particular governing equation being solved. The terms that can be set include gradient terms, divergence terms, laplacian terms, interpolation schemes, and time derivatives. However, numerical schemes in SU2 are specified using keywords divided between central and upwind schemes with brief descriptions for each keyword (e.g. Jameson-Schmidt-Turkel scheme, Classic Roe scheme, ...). OpenFoam was designed to maximize flexibility and freedom for the user while SU2 was designed with ease of use in mind. This discrepancy means that there are problems that can be described in OpenFoam that do not have an equivalent description in SU2.

This problem also extends to boundary conditions as well. Similarly, boundary conditions are specified at the individual variable level in OpenFoam while SU2 relies on built-in keywords. For example, the far field boundary condition is set in SU2 using `MARKER_FAR` followed by the location on the mesh to apply the boundary condition: `MARKER_FAR = ( farfield )`. Far field boundary conditions in OpenFoam must be broken into the component variables. One potential valid way to implement the far field boundary condition in OpenFoam would be to use the Dirichlet condition, known as boundary condition type `fixedValue`, for velocity and temperature at the inflow, and the Neumann condition set to zero, `zeroGradient` in OpenFoam, for pressure at the outflow boundary.

This leads to the next challenge in unifying different backends: lack of documentation and implicit assumptions. For example, the far field boundary condition in our example above does not have an exact mathematical definition, and there are multiple ways of implementing the far field boundary condition (and different solvers implement the same boundary condition differently). Hence, the far field boundary condition implemented in SU2 by `MARKER_FAR = ( farfield )` may not be equivalent to the way it was implemented in OpenFoam. Due to the lack of documentation, the only way to verify its implementation is by digging into the SU2 code to determine exactly how the far field boundary condition is implemented in SU2.

This led to the creation of blocktorock. Blocktorock separates problem description from the target backend. Since there is no common framework to represent the problem, blocktorock is ignorant of the backend by design. Refer to section 2.2 for a more in depth discussion.

5.4 Surrogate Modeling

Often times, single- and multi-physics modeling directly solving these models proves too computationally complex. We hope to address this with future versions of LINK through the use of a variety of surrogate modeling methods, which can then be parameterized and instantiated given appropriate training data. Surrogate modeling is an engineering method often employed when the mechanistic model describing the process is too complex to solve in the available time, or cannot be traditionally constructed. Surrogate models form an approximate metamodel, or emulator, of the process of interest mimicking the behavior of the process (or a mechanistic model for the process) as closely as possible while also providing computational efficiency during evaluation. Surrogate models are constructed using data-driven, bottom up approaches, specific to each surrogate model type. While the internals of a surrogate model can accurately mimic the input/output behavior of the system in question. They can also provide explainability through the evaluation of feature importance, and input/output causality relationships.

Koopman operator theory shows that the evolution of any set of observables in a dynamical system can be expressed through the action of an infinite dimensional linear operator known as the Koopman operator. This operator forms a canonical representation of any dynamical system and, in principle, can allow both the efficient solution of otherwise complex non-linear systems and the application of linear analysis methods on non-linear systems. Prior work has shown that when properly utilized, Koopman operators can even infer properties of dynamical systems that are either partially or completely unknown, or that are simply too complex to express using standard methods of analysis.

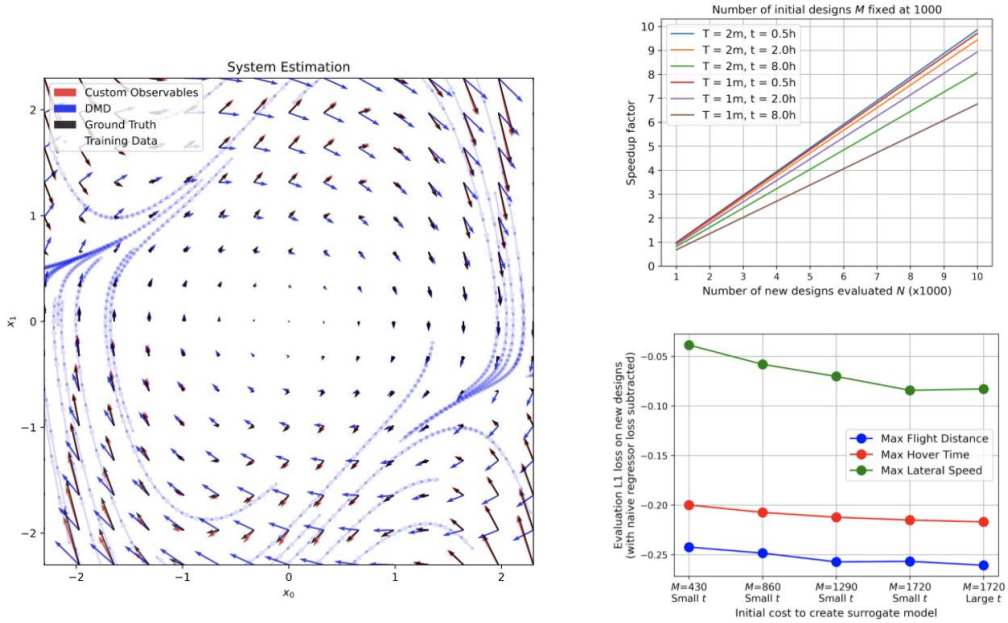


Figure 14: Surrogate Modeling

Figure 14 shows some of our initial results of fluid flow simulations which we are currently using to compute aerodynamic properties related to lift, drag, etc; for UAV designs. Utilizing surrogate

models, we are able to more rapidly evaluate candidate designs, models, and algorithms; with a speedup of over 10x. We hope to incorporate these features into LINK in future releases.

5.4 Future Research on the LINK Framework

Our research and development on the LINK framework has resulted in an open source proof of concept, along with open source releases of our individual components, such as blocktorok. Our hope is that LINK will lead to a paradigm shift in how single- and multi-physics models are authored in the future, resulting in a more open, collaborative, and more easily maintained environment of scientific software and computable models. Currently, models created by tools such as SU2 and OpenFoam cannot be used in other platforms without significant rework. The choice of language and framework is necessarily made at the start of a project, and results in artifacts that are, to some degree, “ossified” in forms that capture the scientific knowledge generated, and lead to it becoming obsolete, hard to update and maintain, and incompatible for reasons other than basic scientific or mathematical ones.

In the same way LINK was utilized to simplify the time necessary to create a model, it is envisioned LINK will provide physics model users the ability to transform models from one tool to another, preventing the loss of valuable scientific knowledge due to the loss of support of viability of legacy backends. Galois has initiated discussions on integrating LINK into commercially offered physics based modeling solutions to consume and utilize varied modeling formats. For the DoD and commercial users of these models, this simplifies the aggregation and use of these disparate models from varied vendors. Galois will continue to seek internal research and development, venture capital and other non-dilutive sources of funding to complete the producitization, extend model input formats to other tools, and improve usability and transparency of the tool results.

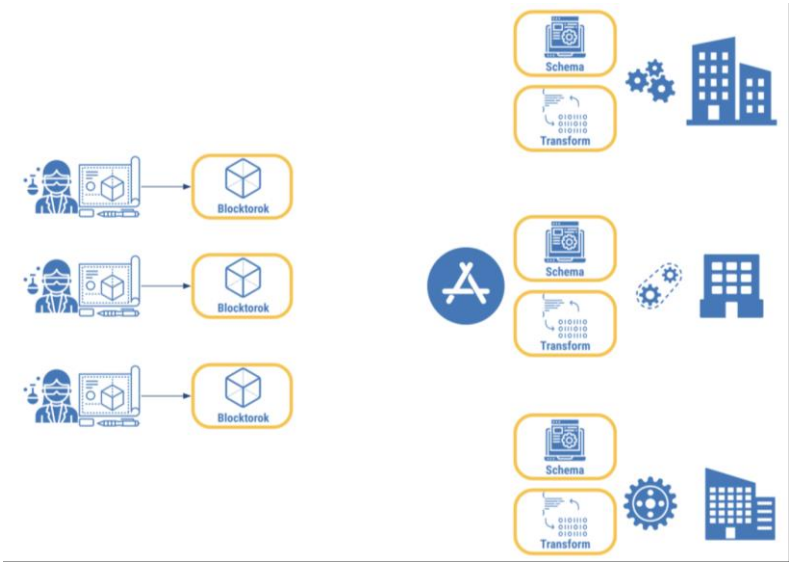


Figure 15: Envisioned LINK Future

We envision an eventual state of LINK similar to that illustrated in Figure 15. In this environment, LINK provides impedance matching between scientists who have programs codified as backend-agnostic blocktorok specifications, and vendors of various simulation technologies produce schemas (which can potentially be shared across vendors) and transforms. This allows the creation of a “market-place” or “app store” of solutions with compatibility, requirements, and assumptions clearly stated. At a lower level of effort, scientists can quickly adapt their specifications to potential backends, ask LINK to implement them automatically, and compare the results in terms of performance, accuracy, correctness, etc; much as was done in our case study with NASA.

Appendices

The latest versions of appendix content can be found in the Blocktorok open source software repository: https://github.com/GaloisInc/blocktorok

A.1. Data Documentation

Data at a glance

`blocktorok` supports a number of common (and not-so-common) data types:

- Numbers (integer and floating-point)
- Quantities (numbers annotated with a unit)
- Strings
- Lists
- Blocks (records)*
- Tagged unions*

*The structure of these types is determined by the schema author; see that documentation for more details.

Blocks are much like JSON objects: A set of key:value pairs where the value can be of any type (except the type of the block itself.)

Tagged unions represent a finite set of alternatives; if you’re familiar with algebraic data types in languages such as Haskell or OCAML, these should be straightforward to understand - For a deeper explanation of such types, please see the schema language documentation.

Here’s a small snippet of Blocktorok-formatted data:

```
hero: {
  name: "Phrobald the Halfling"
  hp: Fixed 50
```

```
  damage: Dice { sides: 6  number: 2 }
}
```

This is a block with three fields: *name*, of type string, *hp*, which is of a union type carrying a single integer, and *damage*, another union-typed value carrying a block with two integer fields, *sides* and *number*.

The top-level object(s) in a data file are labeled the same as any other block field, hence the `hero:` on the outside of the outermost braces. When we look at the full example later, we'll see this more clearly (this snippet is actually taken from a field of the outermost structure.)

Data in detail

Because `blocktorok` is such a simple language, it's worth giving a precise definition of its grammar:

$$
\begin{array}{lll}
\langle START \rangle & ::= & \langle blockContents \rangle \\
\langle blockContents \rangle & ::= & \langle blockElement \rangle * \\
\langle blockElement \rangle & ::= & \langle ident \rangle \text{ `:` } \langle value \rangle \\
\langle value \rangle & ::= & \text{`\{`} \langle blockContents \rangle \text{ `\}`} \\
 & | & \langle number \rangle \text{ `(`} \langle unit \rangle \text{ `)`} \\
 & | & \langle number \rangle \\
 & | & \text{`[`} \langle value \rangle * \text{ `]`} \text{ (* Comma-separated *)} \\
 & | & \text{`"`} \langle char \rangle * \text{ `"`} \\
 & | & \langle ident \rangle \ [\langle value \rangle] \\
\langle ident \rangle & ::= & \text{A valid identifier: The first character must be alphabetic} \\
 & & \text{or an underscore, following characters may be alphabetic,} \\
 & & \text{numeric, or underscores} \\
\langle number \rangle & ::= & \text{Any integral or floating-point number} \\
\langle unit \rangle & ::= & \text{Any SI unit (e.g. `m/s`)} \\
\langle char \rangle & ::= & \text{Any single character}
\end{array}
$$

Notation

The left-hand sides are *non-terminals*, the grammatical objects of the language which are given definitions after the '::=' symbol. The special '<START>' non-terminal defines the top-level of the grammar.

When non-terminals appear on the right-hand side of '::=', that means anything satisfying the definition of that non-terminal may appear in that position.

Vertical pipes (e.g. in the <value> rule) denote alternatives; they should be read as 'or'.

An unquoted asterisk means "zero or more occurrences." Unquoted square braces surrounding an item mean "optional".

Quoted items are literal; they're the actual characters/strings we expect to see while parsing.

The delimiters '(*' and '*)' mark comments separate from the quantitative/qualitative description of a non-terminal's meaning.

How schemas fit in

A schema author constrains this syntax further by specifying what identifiers are valid and what type of data they're associated with. Additionally, the schema provides an entrypoint to understanding the data layout via the `root` definition - This tells us what, at the top-level, our data should look like.

From there, it is possible to follow the type definitions and reconstruct a template for the data layout; this is, in fact, implemented in the LINK compiler, which can dramatically speed up data entry tasks by reducing the amount of boilerplate. Future versions of the template generation system will use metadata in documentation annotations to generate full example data rather than blank input templates.

The main takeaway here is that schemas further restrict the syntactic validity (through what is essentially a type system) of Blocktorok data by providing a definition of *which* identifiers are allowed *where*, and what type of data they must be associated with. This makes Blocktorok similar to JSON, if there was built-in support for authoring schemas and validating data against them.

A.2. Schema Documentation

Schemas at a glance

Schemas are constructed out of a series of *type definitions* and a *root specification*. All types must be defined before use, and the root specification appears last, below all other type definitions.

The two ways of introducing new types are the *union* and the *block*. If you're familiar with languages that support algebraic data types such as Haskell or OCAML, these are essentially sum and product (record) types that are not allowed to be recursive.

If you're *not* familiar with such languages:

- A *union* is a type that has multiple alternative value shapes to choose from. As an example, the typical `Bool` type is a very small union containing only two values, `True` and `False`. Unions in `blocktorok-schema` tend to be more complex, and they look like this:

  ```
  union value {
     [-- Roll a die with `sides` sides `number` times --]
     Dice dice;

     [-- A fixed amount --]
     Fixed int;
  }
  ```

  This defines a type named `value` which may be either a `Dice` (which carries data of type `dice`) **or** a `Fixed` (which carries a single integer value.)

- A *block* is a type made up of many parts, much like a record. Blocks in `blocktorok-schema` look like this:

  ```
  block creature {
    [-- The number of hitpoints a creature has  --]
    hp: value;

    [-- The amount of damage a creature does --]
    damage: value;
  }
  ```

  This defines a type named `creature` that has two fields, `hp` and `damage`, both of which have type `value` (the union type defined previously.)

These two type introduction methods are the meat and potatoes of `blocktorok-schema`; the only other major component is the *root specification*, which is essentially a special block type definition that indicates that it is the top-level structure of the schema. It looks something like this:

```
root {
  battle: battle;
}
```

Which says that a valid input file is composed of a single block that looks like this (omitting the details of what a `battle`-typed block contains):

```
battle: {
  …
}
```

This particular example will be looked at more closely and completely as a complete example of what a schema looks like.

Schemas in detail

This section details the syntax and semantics of schemas, primarily focused on types (since this is ultimately what a schema becomes: a type environment used to drive the validation of inputs and specify transformations.)

Types

Primitive

There are a small number of primitive types that can be used in specifying block layouts. These are mostly familiar, and closely resemble the types of data supported in JSON.

- `int`: The type of signed integer numbers
- `float`: The type of floating-point numbers (double-precision)
- `string`: The type of strings of characters

The type `float` may optionally be decorated with a unit, which simultaneously defines the dimension (e.g. length, mass, time) of the quantity and determines the default unit to render the quantity as in transformers. Explicit control of the output unit is also possible; see the transformer documentation for more details. In practice, defining these field schemas looks like:

```
block unitblock {
  velocity: float (m/s);
  time: float (s);
}
```

Here, we are saying that the field `velocity` is numeric, and has the default unit of meters per second.

List types are also supported, but use a special syntax - See the section about *globbed types* below.

Unions

Union types, as mentioned previously, represent a 'sum' of alternatives. Every union type has at least one alternative to choose from. Alternatives may carry additional information, but it's possible that they carry no data other than their names, which we call *tags*.

A new union type is introduced with the `union` keyword followed by a name for the type and a curly-brace delimited list of variants which are terminated by semicolons.

A variant is an alphanumeric string of characters (underscores allowed) where the first character is alphabetic or an underscore. This is called a 'tag', and is followed by a type. Variants may be optionally preceded by a documentation annotation, which will be discussed later.

To demonstrate this syntax, here is a union type representing an integer value that may not be present:

```
union mInt {
  Nothing;
  Just int;
}
```

There are two variants, one of which carries no data other than its name.

Blocks

Block types, in contrast to unions, represent a collection of data defining a whole, much like a record. A block may be empty, in which case it functions similarly to a union variant that carries no additional data.

A new block type is introduced with the `block` keyword followed by a name for the type and a curly-brace delimited list of field declarations which are terminated by semicolons.

Each field declaration has the form `<identifier>: <type>`.

Like union variants, block fields are optionally preceded by a documentation annotation.

To demonstrate this syntax, here is a block type representing some basic information about a person:

```
block person {
  name: string;
  age: int;
  hobbies: string*;
}
```

Globbed types

Types in block schemas can be annotated to indicate how many instances of that field there might be in the block we're defining, or whether it is a list of values.

If this concept is a bit unclear, the guided example later should make it make more sense.

We call these annotations *globs*, after the similar bash syntax of the same name. They are essentially a very lightweight set of regular expression modifiers:

- If a block field declaration is not decorated, the schema is asserting that exactly one of that field should appear.
- When decorated with a '`?`', the schema is asserting that the field appears at most once (i.e. is optional.)
- When decorated with a '`+`', the schema is asserting that the field appears at least once.
- When decorated with a '`*`', the schema is asserting that the field appears zero or more times. This is the 'glob' for which these decorated types are named.

In practice, these glob annotations look like this:

```
block battle {
  hero: hero*;
  orc: creature*;
  minotaur: creature*;
}
```

The details of the `hero` and `creature` types are not important; what matters is that we know that a battle contains zero or more fields labeled `hero, orc,` and `minotaur`. Changing the first field to `hero: hero+;` would indicate that the sub-block must appear at least once.

More about blocks

An additional feature that blocks will support is *extension*, which adds a convenient shorthand for defining block types that simply add more fields to block types previously defined.

Extending existing blocks

NOTE: This feature is not yet implemented!

Suppose we have the following (simplified) block type definition:

```
block creature {
  hp: int;
  damage: int;
}
```

Now, suppose we want to define a new block type, `hero`, that is just like `creature` except it has an additional field, `name`.

With the tools we have now, we'd need to write this new type out explicitly, duplicating the work done to define `creature`, i.e.

```
block hero {
  name: string;
  hp: int;
  damage int;
}
```

This is a fairly common pattern: We have some block schema definition, and need to *extend* it as a new block type that contains additional information. To reduce redundancy of this nature, `blocktorok-schema` includes the following convenient definition form that introduces a new block type based on an existing one. Here, there must be at least one additional field in the braces, and it cannot have the same name as a field in the block type being extended. For the example above, we could write:

```
block hero extends creature {
  name: string;
}
```

Much cleaner!

Root specifications

Since a root specification is simply an unnamed block type that indicates the top-level structure of input files, there isn't much more to say here that hasn't already been said about introducing block types - the root does not extend other block types or have a name, but the same globbed type syntax is used for its field declarations.

Documentation

`blocktorok-schema` aims to be something that can produce readable documentation for a user constructing input files that satisfy the defined schema. There is a command available in

the `blocktorok` tool to generate such documentation from a schema; please see the README for more information on utilizing this feature.

There are two places documentation annotations may appear: Before union variants and before block fields. These are simply text delimited by the symbols `[--` and `--]`; this text can contain any characters and span multiple lines, which will influence the way documentation is rendered when that feature is invoked.

If you're familiar with other programming languages, these are essentially comments. They should inform a reader what the variant/field being described means in the world of ideas. In terms of documenting the language defined by the schema, these annotations are invaluable in terms of the additional context or constraints not expressible in the schema itself that they provide.

A.3. Transform Documentation

Transformers at a glance

Transformers are built out of *declarations*; specifically, these declarations consist of *rendering rules*, *symbol definitions*, *file outputs*, *subtemplates*, and *requirements*.

Symbol definitions are just like variable definitions in other programming languages; they allow you to give a name to something in order to refer to it later by that name, and have a familiar syntax:

```
output = file(“orcs.py”)
```

A symbol can be bound to any type of expression: A call to the functions `join`, `vjoin`, or `file`, a bar string containing interpolated expressions (see below), a literal, a `for` loop, an `if` expression, a unit conversion, or a selector. They are useful for simplifying verbose code and avoiding unnecessary duplication.

File output declarations are similarly simple, specifying what data should be sent as output to a previously opened file. File output is notated in a way reminiscent of C++ stream operators:

```
output << battle
```

We can output to as many files as we’d like; simply bind symbols for each file (just like `output = file(“orcs.py”)`), and add as many uses of `<<` as you need. Relative paths will begin from the directory from which the `blocktorok` tool is invoked.

Subtemplates allow for a certain form of *scoped rendering*, and will be explained later after rendering is studied in more detail.

Requirements allow for the specification of custom error reporting when certain conditions (e.g. selectors are present/not present) are met; they look like:

```
require foo.bar!? “foo must not have a bar!”
```

Which means that, if the selector `foo.bar` is non-empty, an error should be thrown containing the provided message. At this time, the error will focus on the requirement in the transformer, rather than the data file. Future versions of the language may change this to be more informative to data authors.

The real interesting parts of transformers are the rendering rules, which define how the types outlined in the schema should be translated when producing file outputs. Rendering rules are based on similar templating systems for front-end web development and metaprogramming;

they mainly consist of string literals, but these string literals may contain embedded expressions that evaluate to string literals based on other rendering rules and calls to primitive functions.

Since rendering rules are more complex, detailed discussion is saved for the following section. So the syntax isn't completely surprising, though, this is what a typical rendering rule might look like:

```
render ::battle.hero |Creature("${name}", ${hp}, ${dmg})
```

Transformers in detail

This section details the syntax and semantics of transformers, mostly focused on rendering rules as these ultimately control the translation performed by the compiler.

Expressions

Symbol definitions (described above) and bar string interpolations (described below) both have a notion of *expression*; that is, a syntactic item which is evaluated to some value. `blocktorok-transform` supports the following forms of expression:

- Selectors
- Bar strings
- Conditionals
- For-loops
- String literals
- Unit conversions
- Sequences (lists)
- Primitive function calls
- Selector predicates (empty, non-empty)
- Boolean operators (not, and, or)

Selectors and bar strings are explained in detail below in the discussion of rendering rules.

Conditionals are similar to other programming languages, allowing for some Boolean condition to be checked to make a decision between two arbitrary expressions. In a transformer, a conditional expression looks like:

```
if battle.orc? {
  |orcs = [${join(", ", battle.orc)}]
} else {
  |
}
```

Note that additional branches may be included using the usual `} else if { … }` paradigm.

The condition `battle.orc?` returns true if and only if the selector is non-empty, i.e. there are orcs in the `battle` block. The corresponding predicate `battle.orc!?` returns true if and only if the selector *is* empty.

The usual notation for Boolean operations (prefix `!`, `&&`, `||`) is used to combine two Boolean expressions (typically, the selector predicates described in the preceding paragraph in `require` declarations.)

For-loops use an iterator model similar to Python; that is, rather than specifying a counter update scheme, a collection is explicitly iterated over, like so:

```
for hero in battle.hero {
  |${hero.name} the Hero
}
```

Which results in a new collection containing the specified bar strings for each hero selected by `battle.hero`. It helps to think of this for-loop as a `map`; the resulting collection is always equal in length to the collection iterated over.

String literals are written as usual: Double-quote delimited characters.

Unit conversions use a special function-like syntax; if you're familiar with languages supporting parametric polymorphism and type instantiation, it is appropriate to think of the following syntax as denoting an instantiation of `convert` at the desired target unit and taking as input a quantity:

```
convert[cm/s](velocity)
```

This will, of course, result in an error if the target unit is not of a compatible dimension to the input value.

Sequences are also denoted as in most languages: Square-bracket delimited, comma-separated expressions, e.g. `["foo", "bar", "baz"]`

Finally, the primitive functions: `join`, given a separator string and collection of strings, concatenates the strings by interspersing the separator between them. `vjoin` takes no separator, and instead concatenates the collection of strings by interspersing newline characters. `file` opens a file (creating it and all of its parent directories, barring permission issues), returning a handle that can be written to using the `<<` declaration form. Files are automatically closed by the compiler.

Rendering rules

Selectors

Selectors are used to refer to particular parts of a block, or union variants.

Take, for example, this block definition from the schema used as an example in that documentation:

```
block battle {
  [-- The dauntless heroes in the battle --]
  hero: hero*;

  [-- The fell orcs opposing the heroes --]
  orc: creature*;

  [-- The forbidding minotaurs opposing the heroes --]
  minotaur: creature*;
}
```

We can select for the `hero` field by writing `::battle.hero`; this access can go as deep as we want, by appending the selectors defined on `hero`, for example `::battle.hero.damage` selects the `damage` field of the `hero` block within `battle`.

You might wonder, though, since the `hero` field has a globbed type, which hero does the aforementioned selector select for? The answer is: All of them! Selectors refer to all parts of the data tree matching the selector. Where things get tricky is selector *expressions* vs. selectors as used in rendering declarations. This distinction is explained further below, and there are some syntactic distinctions that help keep the differences clear.

Bar Strings

To keep things clean, `blocktorok-transform` uses a unique syntax for multi-line string literals that supports a form of *interpolation* common to templating languages.

Each line of a multi-line string begins with a pipe (i.e. '`|`') character, and these lines are vertically concatenated by the compiler (in other words, the lines are joined together with newline/return characters.) Quotation marks are *not* used to delimit multi-line strings of this form: Everything from the initial pipe to the end of the line is considered part of the string. One immediate perk of this is the ability to include quote characters in the string literals without escaping them.

Here is a simple example of this idea, what we call a *bar string*:

```
|This is the first line of the string.      "This is part of it, too."
|Part of the same string, separated by a newline from the above.
```

```
|The quotes on line 1 are part of the string, too!
```

Within a given line of a bar string, it is possible to embed some expression to be evaluated; these expressions may be any of the things described in the aforementioned section talking about expressions.

Think of this the same way as format strings in Python, C/C++, or Rust. An embedded expression is written in a bar string as `${...}`, where the dots are replaced with the expression to render and embed.

Embedding a selector causes the rendering rule for that selection to be invoked at that position; we'll look at an example of this soon, as it is how we can keep our transformers modular and avoid duplicated code for similar entities.

As of this writing, the only combinator/primitive functions supported for string manipulation/rendering are `join`, which takes a string literal separator and a sequence-like expression, producing a string that is the result of rendering each element of the sequence and concatenating with the separator in-between each element, and `vjoin`, which takes a sequence-like expression, producing a string that is the result of rendering each element of the sequence separated by a newline. These are useful for handling elements of the schema that have been given a glob type such as `*` or `+`.

Here is an example of what embedded expressions look like in bar strings:

```
|heroes = [${join(", ", hero)}]
```

Note: Don't worry about what the significance of this rendering target is; it will be made clearer in the full worked example at the end of this document.

To understand the translation better, we can break this line down into three parts: An initial string chunk, an embedded expression that needs to be evaluated to some other string chunk, and a final string chunk, i.e.:

```
"heroes = ["  +  <evaluation of join(", ", hero)>  +  "]"
```

Recall the prior discussion of selectors referring to underlying types when defining rendering rules; here we see that, when used as expressions, the selector instead refers to the data itself, including the full type information provided by the schema. This is more clearly illustrated with a full example of schema/transformer/data, which can be found at the end of this document.

Putting it together

With these tools, it is now possible to write a full rendering rule, which has this basic structure:

```
render <schema selector> <bar string>
```

The selector here should be interpreted in the first way selectors were explained, while any selectors appearing in embedded expressions within the bar string should be interpreted in the second way. To drive this distinction home, when you see (the schema selector will be preceded by `::`):

```
render <schema selector> …
```

The selector is referring to a type/schema - Read this as “define a rendering rule for everything that looks like `<selector>`.” In contrast, when you see:

```
render … |… ${<data selector>} …
```

The selector is referring to the actual data to be transformed, i.e. whatever string/integer/etc appears in the data file. This may, of course, be a collection of data, if the underlying type is globbed with `*` or `+`.

Note that a schema selector changes the *context* of the embedded expressions, such that all of the parts of what is being selected are in scope. To be more concrete, suppose we have a small schema like this:

```
block foo {
  bar: int;
  baz: float;
}

root {
  foo: foo;
}
```

To define the rendering, we might write something like:

```
render ::foo |foo blocks contain a bar: ${...} and a baz: ${...}
```

What should go in those embedded expressions in order to properly render the fields of the block? Based on what was said about selectors changing the context, we could write:

```
render ::foo |foo blocks contain a bar: ${bar} and a baz: ${baz}
```

This is allowed since, in the scope of a `foo` block, we know there are fields with the names `bar` and `baz`. To put it another way, we don’t need to write out fully qualified selectors (e.g. `foo.bar`) all the time, and can instead rely on the compiler knowing about the fields thanks to the provided schema. Note that this is a context *replacement*, so higher-level fields become

inaccessible. Future versions of the language may provide alternative forms of render declaration that allow reference to other parts of the document.

Something that has not been mentioned yet are unions and their variants - How do they fit into this discussion of selectors and rendering rules?

The answer is, rendering rules for these constructs use the same notation as blocks. Recall this union defined in the schema language documentation:

```
union value {
  Dice { number: int, sides: int };
  Fixed int;
}
```

Rendering a `value` requires rules for rendering each of the variants. These are written like this:

```
render ::value.Dice  |Roll(${number}, ${sides})
render ::value.Fixed |Fixed(${value})
```

But wait - what is `${value}`? For variants carrying data that isn't a block (e.g. `Fixed`), the data is given the special name `value` to be referred to in defining renderers - It's possible that this name will change in future versions of the language to prevent confusion.

Scoped rendering

In an effort to reduce certain kinds of transformer code duplication, `blocktorok-transform` provides a mechanism to temporarily 'enter' a selector and write declarations as if that selector is the root of the data input.

Since this is a somewhat complex idea, we demonstrate it with an example. Suppose that, in addition to generating the Python script simulating a battle, we would like to generate simple character sheets for all of the `hero`es in the battle. With only what has already been introduced, this isn't really possible, since we define a rendering rule for heroes and have no mechanism to introduce alternative modes of rendering.

This is where the `in` declaration form comes in. You can write:

```
in <selector> {
  … <declarations> …
}
```

Which, between the braces, enters the scope of the schema selector and allows the definition of rendering rules, local variables, and file outputs that go out of scope once outside again. For our particular use case, we can imagine augmenting the guided example below with the following `in` declaration in order to generate the described character sheets:

```
in battle.hero {
  filename = |${name}.charsheet
  cs_out = file(filename)

  render ::value.Dice  |${number}d${sides}
  render ::value.Fixed |${amount}

  cs_out << |${name}
            |  hp: ${hp}
            |  damage: ${damage}
}
```

Note that these rendering rules for `value` do not conflict with what we have already defined earlier! They only control rendering such data within the braces, and will not be used when rendering the Python code for the primary transformer output.

A guided example

For convenience, here is the full example schema used both here and in the schema language documentation:

```
block dice {
  number: int;
  sides: int;
}

union value {
  [-- Roll a die with `sides` sides `number` times --]
  Dice dice;

  [-- A fixed amount --]
  Fixed int;
}

block creature {
  [-- The number of hitpoints a creature has  --]
  hp: value;

  [-- The amount of damage a creature does --]
  damage: value;
}

block hero extends creature {
  [-- The hero's bold name --]
```

```
  name: string;
}

block battle {
  [-- The dauntless heroes in the battle --]
  hero: hero*;

  [-- The fell orcs opposing the hero --]
  orc: creature*;

  [-- The forbidding minotaurs opposing the hero --]
  minotaur: creature*;
}

root {
  battle: battle;
}
```

And here is the corresponding transformer, which defines rules to turn data matching this schema into Python code which simulates the battle being described. Note that the generated Python depends on some libraries written ahead of time:

```
schema "schema.ocs"

output = file("orcs.py")

render ::value.Dice      |Roll(${number}, ${sides})
render ::value.Fixed     |Fixed(${amount})

render ::battle.hero     |Creature("${name}", ${hp}, ${dmg})
render ::battle.orc      |Creature("Orc", ${hp}, ${dmg})
render ::battle.minotaur |Creature("Minotaur", ${hp}, ${dmg})

render ::battle
  |from battle import *
  |heroes = [${join(", ", hero)}]
  |orcs = [${join(", ", orc)}]
  |minotaurs = [${join(", ", minotaur)}]
  |battle(heroes, orcs + minotaurs)

output << battle
```

First, the schema to be transformed is loaded using the `schema …` directive. This brings into scope the typing environment defined by the schema, which in turn brings into scope the selector names used to define rendering rules.

Next, the symbol `output` is assigned to a file handle returned by creating and opening a new file named `orcs.py`; this will be written to later on.

Next are the rendering rules. Note that there are rules for each union variant and each sub-block of the root `battle` type - No deeper selection or selection on the constituent block types (e.g. `creature`) needs to be done, since the fields of these blocks are either of primitive type (for which rendering rules are already defined in the compiler) or covered by other rendering rules.

Finally, output (which is ultimately determined by the real data provided to the compiler filling in the gaps represented by expression interpolations) is written to the file opened previously. To test understanding, here is some real data and the output produced by the transformer for it - convince yourself that this is indeed how the data would be rendered according to the transform just described:

The data:

```
battle: {
  hero: {
    name: "Phrobald the Halfling"
    hp: Fixed 50
    damage: Dice { sides: 6  number: 2 }
  }

  orc: {
    hp: Dice { number: 1  sides: 6 }
    damage: Fixed 2
  }

  orc: {
    hp: Dice { number: 1, sides: 6 }
    damage: Fixed 2
  }

  orc: {
    hp: Dice { number: 1  sides: 6 }
    damage: Fixed 2
  }

  minotaur: {
    hp: Dice { number: 4  sides: 8 }
    damage: Dice { number: 1  sides: 8}
  }
}
```

The transformed output (spacing for readability):

```
from battle import *
heroes = [Creature("Phrobald the Halfling", Fixed(50), Roll(6, 2))]
orcs = [ Creature("Orc", Roll(6, 1), Fixed(2))
       , Creature("Orc", Roll(6, 1), Fixed(2))
       , Creature("Orc", Roll(6, 1), Fixed(2))
       ]
minotaurs = [Creature("Minotaur", Roll(8, 4), Roll(8, 1))]
battle(heroes, orcs + minotaurs)
```

A.4. Example: Heat Transfer in a Rod with Multiple Backends

Schema

First, we present the schema that input data must conform to. As mentioned earlier, this is intended to be written once by an advanced user; as we'll see, this same schema (and, therefore, data inputs) can be used to define multiple transformers (to vastly different backend formats):

```
block thermal_diffusivity {
  [-- Right hand side of the equation --]
  density: float;

  heat_capacity: float;

  thermal_conductivity: float;

  [-- Left hand side of the equation --]
  dt: float;
}

block criterion {
  min_value: float;
  start_iter: int;
  cauchy_elements: int;
  cauchy_epsilon: float;
}

union convergence {
  [-- Maximum number of iterations before the simulation ends --]
  Max_iterations int;

  [-- Terminates when a field reaches a certain threshold. --]
  Criterion criterion;
}

block transient {
  dt: float;
  max_time: float;
  start_time: float;
  inner_iter: int
  outer_iter: int;
}

union time {
```

```
  [-- Steady state simulation --]
  Steady_state;

  [-- Transient simulation --]
  Transient transient;
}

union output_format {
  CSV;
  TECPLOT;
}

union mesh_format {
  SU2;
  CGNS;
}

block input_output {
  [-- Output solution frequency in iterations for su2 --]
  write_interval_iter: int;

  [-- Output solution frequency in seconds for openfoam --]
  write_interval_sec : float;

  [-- Name of output file is optional (OpenFoam does not have one) --]
  output_file: string;

  [-- Output file format --]
  output_format: output_format;

  mesh_filename: string;

  [-- The format of the mesh file --]
  mesh_format: mesh_format;
}

block physics {
  [-- Params such as turbulence and thermophysical models --]
  physical_models: thermal_diffusivity;
}



block bc_val {
  marker: string;
```

```
  value: float;
}

union boundary_conditions {
  Isothermal bc_val;

  Heatflux bc_val;
}

block domain {
  [-- Physics parameters --]
  physics: physics;

  [-- Initial temperature --]
  initial_temp: float;

  [-- Simulation termination conditions --]
  convergence: convergence;

  [-- Input output parameters (mesh files are included here) --]
  input_output: input_output;

  [-- Transient or steady state parameters are defined here --]
  time: time;

  [-- Boundary conditions --]
  boundary_conditions: boundary_conditions*;
}

root {
  simulation: domain;
}
```

Data

Next, the data satisfying the above schema and specifying the simulation to be performed. As mentioned, this same description of the simulation can be used to generate both an SU2 *and* an OpenFOAM configuration, demonstrating the lifting of burdens currently on simulation authors to learn and understand complex frameworks when simple configuration data is sufficient:

```
simulation: {
  physics: {
    physical_models: {
      density: 19300
      heat_capacity: 130
      thermal_conductivity: 318
```

```
      dt: 0.000127
    }
  }

  initial_temp: 273

  convergence: Criterion {
    min_value: -19
    start_iter: 10
    cauchy_elements: 100
    cauchy_epsilon: 1E-6
  }

  input_output: {
    write_interval_iter: 250
    write_interval_sec: 0.2
    output_file: "flow"
    output_format: CSV
    mesh_filename: "mesh_solid_rod.su2"
    mesh_format: SU2
  }

  time: Transient {
    dt: 0.005
    max_time: 3
    start_time: 0
    inner_iter: 200
    outer_iter: 600
  }

  boundary_conditions: [ Isothermal { marker: "left" value: 573 }
                       , Isothermal { marker: "right" value: 273 }
                       , Heatflux {marker: "bottom" value: 0}
                       , Heatflux {marker: "top" value: 0}
                       ]
}
```

Transformer: SU2

Next, a transformer defining how data should be translated to work in the SU2 simulation engine. The author of such a transform would need extensive knowledge of SU2, but as mentioned earlier, these transformers are write-once-use-many components of the workflow:

```
schema "schema.ocs"

output = file("v3_solid_rod.cfg")

render ::thermal_diffusivity
```

```
 |SOLID_DENSITY= ${density}
 |SPECIFIC_HEAT_CP= ${heat_capacity}
 |SOLID_THERMAL_CONDUCTIVITY= ${thermal_conductivity}

render ::convergence.Criterion
 |CONV_RESIDUAL_MINVAL= ${min_value}
 |CONV_STARTITER= ${start_iter}
 |CONV_CAUCHY_ELEMS= ${cauchy_elements}
 |CONV_CAUCHY_EPS= ${cauchy_epsilon}

render ::convergence.max_iterations
 |ITER= ${max_iterations}

render ::time.Steady_state
 |TIME_DOMAIN= NO
render ::time.Transient
 |TIME_DOMAIN= YES
 |TIME_STEP= ${dt}
 |MAX_TIME= ${max_time}
 |INNER_ITER= ${inner_iter}
 |TIME_ITER= ${outer_iter}

render ::input_output
 |MESH_FILENAME= ${mesh_filename}
 |MESH_FORMAT= ${mesh_format}
 |TABULAR_FORMAT= ${output_format}
 |CONV_FILENAME= history
 |VOLUME_FILENAME= flow
 |SURFACE_FILENAME= surface_flow
 |OUTPUT_WRT_FREQ = ${write_interval_iter}
 |SCREEN_WRT_FREQ_TIME = 1

render ::physics
 |SOLVER= HEAT_EQUATION
 |MATH_PROBLEM= DIRECT
 |RESTART_SOL= NO
 |OBJECTIVE_FUNCTION= TOTAL_HEATFLUX

render ::boundary_conditions.Isothermal
 |${marker}, ${value}
render ::boundary_conditions.Heatflux
 |${marker}, ${value}

render ::mesh_format.SU2
 |SU2
```

```
render ::mesh_format.CGNS
 |CGNS

render ::output_format.CSV
 |CSV
render ::output_format.TECPLOT
 |TECPLOT

conduction_condition =
 |INC_NONDIM = DIMENSIONAL
 |SOLID_TEMPERATURE_INIT = ${simulation.initial_temp}
 |${simulation.physics.physical_models}

render ::domain
 |${physics}
 |${time}
 |MARKER_ISOTHERMAL= ( ${join(", ", boundary_conditions.Isothermal)} )
 |MARKER_HEATFLUX= ( ${join(", ", boundary_conditions.Heatflux)} )
 |MARKER_PLOTTING= ( ${join( ", "
                          , boundary_conditions.Isothermal.marker
                          )}
                    , ${join( ", "
                            , boundary_conditions.Heatflux.marker
                            )} )
 |${conduction_condition}
 |NUM_METHOD_GRAD= GREEN_GAUSS
 |LINEAR_SOLVER= FGMRES
 |LINEAR_SOLVER_PREC= ILU
 |LINEAR_SOLVER_ILU_FILL_IN= 0
 |LINEAR_SOLVER_ERROR= 1E-15
 |LINEAR_SOLVER_ITER= 5
 |${convergence}
 |${input_output}

output << simulation
```

Transformer: OpenFOAM

Finally, a second transformer targeting OpenFOAM. This transformer operates on data satisfying the same schema as the SU2 transformer above, with nothing extra required from the data author. Note the ability to output to arbitrarily complex file/directory structures:

```
schema "schema.ocs"
```

```
output_var = file("0/T")
output_transport = file("constant/transportProperties")
output_control = file("system/controlDict")
output_schemes = file("system/fvSchemes")
output_solution = file("system/fvSolution")
blockMeshDict = file("system/blockMeshDict")

render ::boundary_conditions.Isothermal
 |    ${marker}
 |    {
 |        type            fixedValue;
 |        value           uniform ${value};
 |    }

render ::boundary_conditions.Heatflux
 |    ${marker}
 |    {
 |        type            zeroGradient;
 |    }

output_var <<
 |FoamFile
 |{
 |    version     2.0;
 |    format      ascii;
 |    class       volScalarField;
 |    object      T;
 |}
 |dimensions      [0 0 0 1 0 0 0];
 |internalField   uniform ${simulation.initial_temp};
 |boundaryField
 |{
 |${vjoin(simulation.boundary_conditions)}
 |}

render ::thermal_diffusivity
 |FoamFile
 |{
 |    version     2.0;
 |    format      ascii;
 |    class       dictionary;
 |    location    "constant";
 |    object      transportProperties;
 |}
 |DT              DT [0 2 -1 0 0 0 0] ${dt};
```

```
output_transport << simulation.physics.physical_models

render ::time.Transient
 |deltaT          ${dt};
 |endTime         ${max_time};
 |startTime       ${start_time};

output_control <<
 |FoamFile
 |{
 |    version     2.0;
 |    format      ascii;
 |    class       dictionary;
 |    location    "system";
 |    object      controlDict;
 |}
 |application     laplacianFoam;
 |startFrom       latestTime;
 |stopAt          endTime;
 |${simulation.time.Transient}
 |writeControl    runTime;
 |writeInterval   ${simulation.input_output.write_interval_sec};
 |purgeWrite      0;
 |writeFormat     ascii;
 |writePrecision  6;
 |writeCompression off;
 |timeFormat      general;
 |timePrecision   6;
 |runTimeModifiable true;

output_schemes <<
 |FoamFile
 |{
 |    version     2.0;
 |    format      ascii;
 |    class       dictionary;
 |    location    "system";
 |    object      fvSchemes;
 |}
 |ddtSchemes
 |{
 |    default         Euler;
 |}
 |gradSchemes
```

```
 |{
 |    default         Gauss linear;
 |    grad(T)         Gauss linear;
 |}
 |divSchemes
 |{
 |    default         none;
 |}
 |laplacianSchemes
 |{
 |    default         none;
 |    laplacian(DT,T) Gauss linear corrected;
 |}
 |interpolationSchemes
 |{
 |    default         linear;
 |}
 |snGradSchemes
 |{
 |    default         corrected;
 |}

output_solution <<
 |FoamFile
 |{
 |    version     2.0;
 |    format      ascii;
 |    class       dictionary;
 |    location    "system";
 |    object      fvSolution;
 |}
 |solvers
 |{
 |    T
 |    {
 |        solver          PCG;
 |        preconditioner  DIC;
 |        tolerance       1e-06;
 |        relTol          0;
 |    }
 |}
 |SIMPLE
 |{
 |    nNonOrthogonalCorrectors 2;
 |}
```

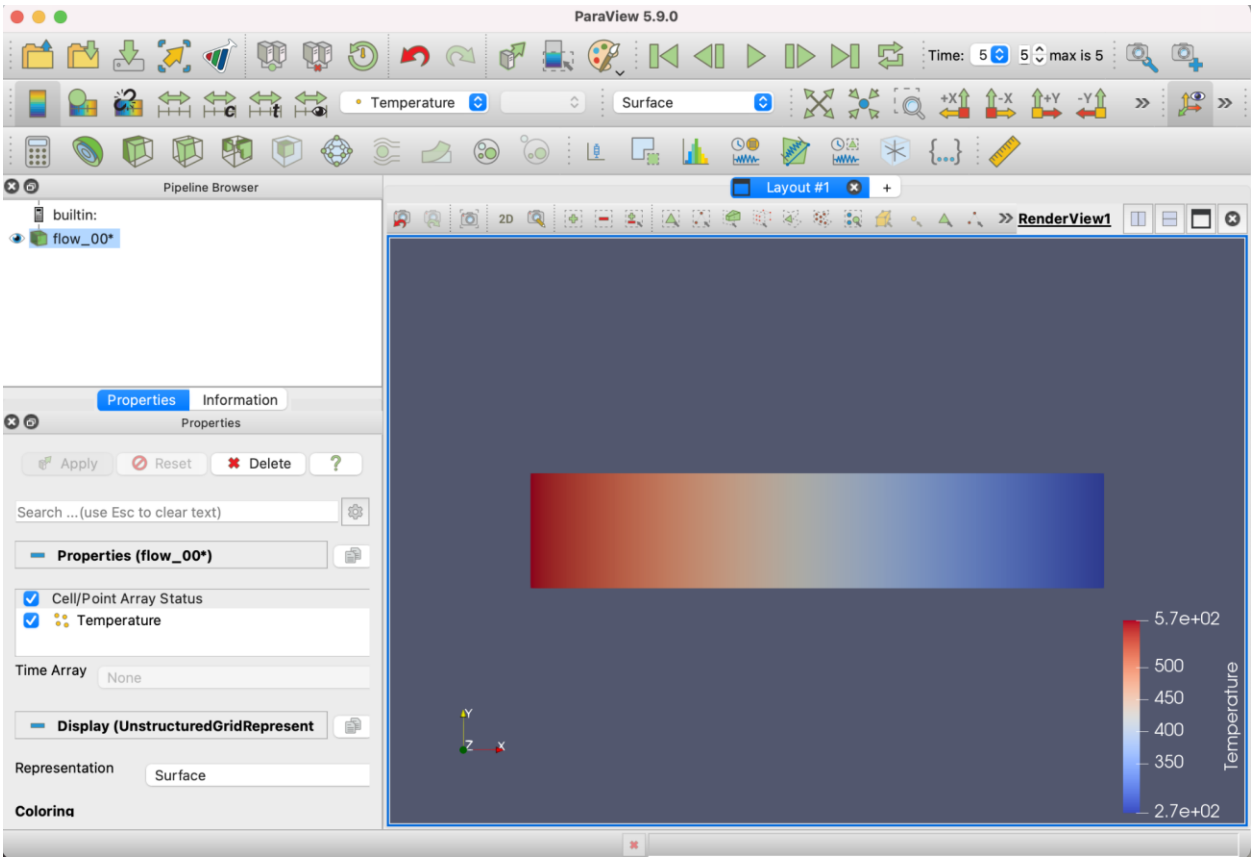
```
blockMeshDict <<

 |FoamFile
 |{
 |    version     2.0;
 |    format      ascii;
 |    class       dictionary;
 |    object      blockMeshDict;
 |}
 |convertToMeters 0.1;
 |vertices
 |(
 |    (0 0 0)
 |    (5 0 0)
 |    (5 1 0)
 |    (0 1 0)
 |    (0 0 0.1)
 |    (5 0 0.1)
 |    (5 1 0.1)
 |    (0 1 0.1)
 |);
 |blocks
 |(
 |    hex (0 1 2 3 4 5 6 7) (100 20 1) simpleGrading (1 1 1)
 |);
 |edges
 |(
 |);
 |boundary
 |(
 |    top
 |    {
 |        type patch;
 |        faces
 |        (
 |            (3 7 6 2)
 |        );
 |    }
 |    bottom
 |    {
 |        type patch;
 |        faces
 |        (
 |            (1 5 4 0)
```

```
|        );
|    }
|    left
|    {
|        type patch;
|        faces
|        (
|            (0 4 7 3)
|        );
|    }
|    right
|    {
|        type patch;
|        faces
|        (
|            (2 6 5 1)
|        );
|    }
|    frontAndBack
|    {
|        type empty;
|        faces
|        (
|            (0 3 2 1)
|            (4 5 6 7)
|        );
|    }
|);
|mergePatchPairs
|(
|);
```

Simulation results

- Steel rod results:



A.5. MultiPhysics Example: Heated Cylinders With Conjugate Heat Transfer

Schema

```
block components {
  density: float;
  heat_capacity: float;
  thermal_conductivity: float;
}

union thermal_diffusivity {
  [-- Right hand side of the equation --]
  components components;

  [-- Left hand side of the equation --]
  dt float;
}

block incompressible_ideal_gas {
  specific_heat_capacity: float;
  molecular_weight: float;
}

union fluid_model {
  incompressible_ideal_gas incompressible_ideal_gas;

  standard_air;
}

union density_model {
  [-- variable density: appropriate fluid model must be selected --]
  variable fluid_model;

  constant float;

  boussinesq;
}

block sutherland {
  reference_viscosity: float;
  reference_temp: float;
  sutherland_constant: float;
```

```
}

union viscosity_model {
  [-- dynamic viscosity --]
  sutherland sutherland;

  [-- constant viscosity --]
  constant float;
}

block constant_prandtl {
  laminar_prandtl_number: float;
  turbulent_prandtl_number: float;
}

union thermal_conductivity_model {
  [-- constant Prandtl number --]
  constant_prandtl constant_prandtl;

  [-- constant conductivity --]
  constant_conductivity float;
}

union energy_equation {
  use float;
  not_used;
}

union governing_equations_fluid {
  compressible_navier_stokes;
  incompressible_navier_stokes;
  compressible_euler;
  incompressible_euler;
}

union turbulence_model {
  [-- There's more, this is an incomplete list --]
  NONE;
  SA;
}

block velocity {
  x: float;
  y: float;
  z: float;
```

```
}

union initial_value {
  density float;
  velocity velocity;
  temperature float;
}

block bc_val {
  marker: string;
  value: float;
}

union boundary_conditions {
  isothermal bc_val;

  heatflux bc_val;

  farfield string;
}

block fluid {
  [-- The governing equation/solver for the problem --]
  governing_equation: governing_equations_fluid;

  [-- Turbulent model (NONE, SA) --]
  turbulence: turbulence_model;

  [-- variable density: appropriate fluid model must be selected --]
  density_model: density_model;

  viscosity_model: viscosity_model;

  thermal_conductivity: thermal_conductivity_model;

  [-- If energy_equation, need initial temperature --]
  energy_equation: energy_equation;

  initial_values: initial_value*;

  [-- Boundary conditions --]
  boundary_conditions: boundary_conditions*;
}

union governing_equations_solid {
```

```
  heat_conduction;
}

block solid {
  [-- The governing equation/solver for the problem --]
  governing_equation: governing_equations_solid;

  conduction: thermal_diffusivity;

  initial_values: initial_value*;

  [-- Boundary conditions --]
  boundary_conditions: boundary_conditions*;
}

block criterion_with_start {
  min_value: float;
  start_iter: int;
}

union convergence {
  [-- Maximum number of iterations before the simulation ends --]
  max_iterations int;

  [-- Simulation terminates when a field reaches a certain threshold.
      The list of possible fields depends on the solver --]
  criterion float;

  criterion_with_start criterion_with_start;
}

block transient {
  dt: float;
  max_time: float;
  inner_iter: int;
  outer_iter: int;
}

union time {
  [-- Steady state simulation --]
  steady_state int;

  [-- Transient simulation --]
  transient transient;
}
```

```
union output_format {
  CSV;
  TECPLOT;
  RESTART;
  PARAVIEW;
  SURFACE_TECPLOT;
  SURFACE_PARAVIEW;
}

union mesh_format {
  SU2;
  CGNS;
}

block input_output {
  [-- Output solution frequency in iterations for su2 --]
  write_interval_iter: int;

  [-- Output solution frequency in seconds for openfoam --]
  write_interval_sec : float;

  [-- Name of output file is optional (OpenFoam does not have one) --]
  output_file: string?;

  [-- Output file format --]
  output_format: output_format*;

  [-- The mesh file is optional (OpenFoam may not have a mesh file if
      using blockMeshDict) --]
  mesh_filename: string?;

  [-- The format of the mesh file --]
  mesh_format: mesh_format?;
}

block domain {
  [-- Description of the problem --]
  name: string;

  fluid: fluid*;
  solid: solid*;

  [-- Simulation termination conditions --]
  convergence: convergence;
```

```
  [-- Time dependency: transient or steady state parameters are
      defined here --]
  time: time;
}

block coupling_interface {
  interface1: string;
  interface2: string;
}

block coupled_domains {
  [-- Pairwise coupled domains --]
  domain1: string;
  domain2: string;

  [-- Pairwise interfaces between domains --]
  interface: coupling_interface*;
}

block simulation {
  [-- multiphysics simulation has at least one coupled domains
      (currently only allowed for conjugated heat transfer) --]
  coupling: coupled_domains*;

  domain: domain*;

  [-- Input output parameters (mesh files are included here) --]
  input_output: input_output;

  [-- couple iter is not required if single_physics
  Number of coupled iterations --]
  couple_iter: int;

  [-- Simulation termination conditions --]
  convergence: convergence;
}

root {
  simulation: simulation;
}
```

Data

```
simulation: {
  coupling: {
    domain1: "solid1"
    domain2: "fluid"

    interface: {
      interface1: "cylinder_outer1"
      interface2: "cylinder_inner1"
    }
  }

  coupling: {
    domain1: "solid2"
    domain2: "fluid"

    interface: {
      interface1: "cylinder_outer2"
      interface2: "cylinder_inner2"
    }
  }

  coupling: {
    domain1: "solid3"
    domain2: "fluid"

    interface: {
      interface1: "cylinder_outer3"
      interface2: "cylinder_inner3"
    }
  }

  domain: {
    name: "solid1"
    solid: {
      governing_equation: heat_conduction
      conduction: components { density: 0.00042
                               heat_capacity: 1004.703
                               thermal_conductivity: 0.1
                             }
      initial_values: temperature 288.15
      boundary_conditions: isothermal { marker: "core1"
                                        value: 350
                                      }
```

```
  }
  time: steady_state 10
  convergence: criterion_with_start { min_value: -20
                                      start_iter: 10
                                    }
}

domain: {
  name: "solid2"
  solid: {
    governing_equation: heat_conduction
    conduction: components { density: 0.00042
                             heat_capacity: 1004.703
                             thermal_conductivity: 0.1
                           }
    initial_values: temperature 288.15
    boundary_conditions: isothermal { marker: "core2"
                                      value: 350
                                    }
  }
  time: steady_state 10
  convergence: criterion_with_start { min_value: -20
                                      start_iter: 10
                                    }
}

domain: {
  name: "solid3"
  solid: {
    governing_equation: heat_conduction
    conduction: components { density: 0.00042
                             heat_capacity: 1004.703
                             thermal_conductivity: 0.1
                           }
    initial_values: temperature 288.15
    boundary_conditions: isothermal { marker: "core3"
                                      value: 350
                                    }
  }
  time: steady_state 10
  convergence: criterion_with_start { min_value: -20
                                      start_iter: 10
                                    }
}
```

```
  domain: {
    name: "fluid"
    fluid: {
      governing_equation: incompressible_navier_stokes
      turbulence: NONE
      density_model: variable incompressible_ideal_gas {
specific_heat_capacity: 1004.703

molecular_weight: 28.96
                                                        }
      viscosity_model: constant 1.7893e-05
      thermal_conductivity: constant_prandtl { laminar_prandtl_number:
0.72

turbulent_prandtl_number: 0.90
                                               }
      energy_equation: use 288.15
      initial_values: [ density 0.00042
                      , velocity { x: 3.40297
                                   y: 0
                                   z: 0
                                 }
                      ]
      boundary_conditions: farfield "farfield"
    }
    time: steady_state 1
    convergence: criterion_with_start { min_value: -20
                                        start_iter: 0
                                      }
  }

    input_output: {
      write_interval_iter: 15000

      write_interval_sec: 0.2

      output_format: [ RESTART, TECPLOT, PARAVIEW, SURFACE_TECPLOT,
                       SURFACE_PARAVIEW ]

      mesh_filename: "mesh_cht_3cyl_ffd.su2"

      mesh_format: SU2
    }

    couple_iter: 15000
```

```
    convergence: criterion -20

  }
```

Transformer: SU2

```
schema "schema.ocs"

cht = file("cht_fluid_solid.cfg")

render ::thermal_diffusivity.components |SOLID_DENSITY= ${density}
                                        |SPECIFIC_HEAT_CP=
${heat_capacity}
                                        |SOLID_THERMAL_CONDUCTIVITY=
${thermal_conductivity}

render ::convergence.criterion          |CONV_RESIDUAL_MINVAL=
${value}
render ::convergence.criterion_with_start
                                        |CONV_RESIDUAL_MINVAL=
${min_value}
                                        |CONV_STARTITER= ${start_iter}

render ::convergence.max_iterations     |ITER= ${max_iterations}

render ::time.steady_state              |TIME_DOMAIN = NO
                                        |INNER_ITER = ${value}
render ::time.transient                 |TIME_DOMAIN = YES
                                        |TIME_STEP = ${dt}
                                        |MAX_TIME = ${max_time}
                                        |INNER_ITER = ${inner_iter}
                                        |TIME_ITER = ${outer_iter}


render ::input_output                   |MESH_FILENAME=
${mesh_filename}
                                        |MESH_FORMAT= ${mesh_format}
                                        |OUTPUT_FILES= ( ${join(", ",
output_format)} )
                                        |CONV_FILENAME= history
                                        |VOLUME_FILENAME= flow
                                        |SURFACE_FILENAME=
surface_flow
                                        |OUTPUT_WRT_FREQ =
${write_interval_iter}
```

```
                                        |SCREEN_WRT_FREQ_TIME = 1

render ::governing_equations_fluid.incompressible_navier_stokes
  |INC_NAVIER_STOKES
render ::governing_equations_fluid.compressible_navier_stokes
  |NAVIER_STOKES
render ::governing_equations_fluid.compressible_euler
  |EULER
render ::governing_equations_fluid.incompressible_euler
  |INC_EULER

render ::governing_equations_solid.heat_conduction
  |HEAT_EQUATION

render ::initial_value.density          |${value}
render ::initial_value.velocity         |( ${x}, ${y}, ${z} )
render ::initial_value.temperature      |${value}

render ::governing_equations_solid.heat_conduction
                                        |HEAT_EQUATION

render ::solid
  |SOLVER = ${governing_equation}
  |WRT_ZONE_HIST = YES
  |HISTORY_OUTPUT = (ITER, RMS_RES, HEAT)
  |INC_NONDIM= DIMENSIONAL
  |${conduction}
  |LINEAR_SOLVER_ITER= 5

render ::turbulence_model.NONE          |NONE
render ::turbulence_model.SA            |SA

render ::fluid_model.incompressible_ideal_gas
  |FLUID_MODEL= INC_IDEAL_GAS
  |SPECIFIC_HEAT_CP= ${specific_heat_capacity}
  |MOLECULAR_WEIGHT= ${molecular_weight}
render ::fluid_model.standard_air
  |FLUID_MODEL = STANDARD_AIR

render ::density_model.variable         |INC_DENSITY_MODEL= VARIABLE
                                        |${value}
render ::density_model.constant
  |FLUID_MODEL = CONSTANT_DENSITY
  |SPECIFIC_HEAT_CP = ${specific_heat_capacity}
```

```
render ::viscosity_model.constant
  |VISCOSITY_MODEL= CONSTANT_VISCOSITY
  |MU_CONSTANT= ${value}
render ::viscosity_model.sutherland
  |VISCOSITY_MODEL= SUTHERLAND
  |MU_REF= ${reference_viscosity}
  |MU_T_REF = ${reference_temp}
  |SUTHERLAND_CONSTANT = ${sutherland_constant}

render ::thermal_conductivity_model.constant_prandtl
  |CONDUCTIVITY_MODEL= CONSTANT_PRANDTL

  |PRANDTL_LAM= ${laminar_prandtl_number}

  |PRANDTL_TURB= ${turbulent_prandtl_number}
render ::thermal_conductivity_model.constant_conductivity
  |KT_CONSTANT= ${molecular_thermal_conductivity}

render ::energy_equation.use
  |INC_ENERGY_EQUATION = YES
  |INC_TEMPERATURE_INIT= ${value}
render ::energy_equation.not_used       |INC_ENERGY_EQUATION = NO

render ::fluid
  |SOLVER = ${governing_equation}
  |KIND_TURB_MODEL = ${turbulence}
  |WRT_ZONE_HIST= YES
  |HISTORY_OUTPUT= (ITER, RMS_RES, HEAT)
  |${density_model}
  |${energy_equation}
  |INC_NONDIM= DIMENSIONAL
  |${viscosity_model}
  |${thermal_conductivity}
  |LINEAR_SOLVER_ITER= 10
  |CONV_NUM_METHOD_FLOW= FDS
  |MUSCL_FLOW= YES
  |SLOPE_LIMITER_FLOW= NONE
  |TIME_DISCRE_FLOW= EULER_IMPLICIT

render ::boundary_conditions.isothermal |${marker}, ${value}
render ::boundary_conditions.heatflux   |${marker}, ${value}
render ::boundary_conditions.farfield   |${value}

render ::mesh_format.SU2                |SU2
render ::mesh_format.CGNS               |CGNS
```

```
render ::output_format.CSV               |CSV
render ::output_format.TECPLOT           |TECPLOT
render ::output_format.RESTART           |RESTART
render ::output_format.PARAVIEW          |PARAVIEW
render ::output_format.SURFACE_TECPLOT   |SURFACE_TECPLOT
render ::output_format.SURFACE_PARAVIEW  |SURFACE_PARAVIEW

in simulation.domain {
  filename = |${name}.cfg
  cfg_out = file(filename)

  domain_specific_vars =
    if fluid? {
      |INC_VELOCITY_INIT= ${fluid.initial_values.velocity}
      |INC_DENSITY_INIT= ${fluid.initial_values.density}
      |MARKER_FAR= (${join(", ", fluid.boundary_conditions.farfield)})
    }
    else if solid? {
      |SOLID_TEMPERATURE_INIT = ${solid.initial_values.temperature}
      |MARKER_ISOTHERMAL= (${join(", ",
                              solid.boundary_conditions.isothermal)} )
    }
    else {
      |
    }

    cfg_out << |${fluid}
               |${solid}
               |${domain_specific_vars}
               |${time}
               |NUM_METHOD_GRAD= GREEN_GAUSS
               |CFL_NUMBER= 50.0
               |CFL_ADAPT= NO
               |CFL_ADAPT_PARAM= ( 1.5, 0.5, 10.0, 10000.0 )
               |RK_ALPHA_COEFF= ( 0.66667, 0.66667, 1.000000 )
               |LINEAR_SOLVER= FGMRES
               |LINEAR_SOLVER_PREC= ILU
               |LINEAR_SOLVER_ILU_FILL_IN= 0
               |LINEAR_SOLVER_ERROR= 1E-15
               |${convergence}
}

render ::coupling_interface           |${interface1}, ${interface2}
```

```
master =
  |SOLVER = MULTIPHYSICS
  |MATH_PROBLEM= DIRECT
  |RESTART_SOL= NO
  |CONFIG_LIST = (fluid.cfg, solid1.cfg, solid2.cfg, solid3.cfg)
  |MARKER_ZONE_INTERFACE = ( ${join(", ",
                                simulation.coupling.interface)})
  |MARKER_CHT_INTERFACE = ( ${join(", ",
                               simulation.coupling.interface)})
  |OBJECTIVE_FUNCTION = TOTAL_HEATFLUX
  |OBJECTIVE_WEIGHT= 1.0
  |OUTER_ITER = ${simulation.couple_iter}
  |${simulation.input_output}
  |${simulation.convergence}

-- Skipping the free-form deformation parameters and design variable
parameters
cht << master
```

Transformer: OpenFoam

The transformer for the OpenFoam backend is incomplete and untested due to lack of a mesh for the OpenFoam backend. We were unable to finish our effort to convert the SU2 mesh to OpenFoam mesh in time. The transformer in this section generates most of the files needed to run the multiphysics problem in the OpenFoam backend. The missing components include the system and the controlDict files, and the names of the boundary markers.

```
schema "schema.ocs"

initial_fluid_T = file("0/fluid/T")
initial_fluid_U = file("0/fluid/U")
initial_fluid_p = file("0/fluid/p")
initial_fluid_p_rgh = file("0/fluid/p_rgh")

properties_fluid_g = file("constant/fluid/g")
properties_fluid_thermo =
file("constant/fluid/thermophysicalProperties")
properties_fluid_turbulence =
file("constant/fluid/turbulenceProperties")

regions = file("constant/regionProperties")

fluid_schemes = file("system/fluid/fvSchemes")
fluid_solution = file("system/fluid/fvSolution")
```

```
require (simulation.domain.fluid? && simulation.domain.solid?)
  "Need at least one fluid and one solid domain."

-- TODO: systems

------------------------------------------------------------
-- boundary conditions
------------------------------------------------------------

bc = simulation.domain.boundary_conditions

pressure =
if bc.farfield? {
  |farfield
  |{
  |    type            zeroGradient;
  |}
}
else {
  |
}

temp =
if bc.farfield? {
  |farfield
  |{
  |    type            fixedValue;
  |    value           \$internalField;
  |}
}
else {
  |
}

init = simulation.domain.initial_values

velocity =
if bc.farfield? {
  |farfield
  |{
  |    type            fixedValue;
  |    value           uniform ( ${init.velocity.x},
${init.velocity.y}, ${init.velocity.z} );
  |}
}
```

```
else {
  |
}

interfaces_temp =
  for d in simulation.domain
    |fluid_to_${d.name}
    |     {
    |         type
compressible::turbulentTemperatureCoupledBaffleMixed;
    |         value           \$internalField;
    |         Tnbr            T;
    |     }

interfaces_velocity =
  for d in simulation.domain
    |fluid_to_${d.name}
    |{
    |    type            noSlip;
    |}

// ************************************************************************* //
-- 0/fluid section
// ************************************************************************* //
initial_fluid_p <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       volScalarField;
  |    location    "0/fluid";
  |    object      p;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |dimensions      [ 1 -1 -2 0 0 0 0 ];
  |
  |internalField   uniform 0;
  |
  |boundaryField
  |{
  |    ${pressure}
  |}
  |
  |// ************************************************************************* //
```

```
initial_fluid_p_rgh <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       volScalarField;
  |    location    "0/fluid";
  |    object      p_rgh;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |dimensions      [ 1 -1 -2 0 0 0 0 ];
  |
  |internalField   uniform 0;
  |
  |boundaryField
  |{
  |    ${pressure}
  |}
  |
  |// ************************************************************************* //

initial_fluid_T <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       volScalarField;
  |    location    "0/fluid";
  |    object      T;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |dimensions      [ 0 0 0 1 0 0 0 ];
  |
  |internalField   uniform ${init.temperature};
  |
  |boundaryField
  |{
  |    ${temp}
  |    ${interfaces_temp}
  |}
  |
  |// ************************************************************************* //
```

```
initial_fluid_U <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       volVectorField;
  |    location    "0/fluid";
  |    object      U;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |dimensions      [ 0 1 -1 0 0 0 0 ];
  |
  |internalField   uniform (0 0 1e-3);
  |
  |boundaryField
  |{
  |    ${velocity}
  |    ${interfaces_velocity}
  |}
  |
  |// ************************************************************* //

-------------------------------------------------------------
-- 0/solid section
-------------------------------------------------------------

initial_solid_p =
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       volScalarField;
  |    object      p;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |dimensions      [1 -1 -2 0 0 0 0];
  |
  |internalField   uniform 0;
  |
  |boundaryField
  |{
  |    ".*"
```

```
  |    {
  |        type            calculated;
  |        value           \$internalField;
  |    }
  |}
  |
  |// ************************************************************* //

in simulation.domain {
  filename = |0/${name}/p
  in solid {
    cs_out = file(filename)
    cs_out << initial_solid_p
  }
}

in simulation.domain {
  filename = |0/${name}/T
  in solid {
    cs_out = file(filename)
    cs_out <<
      |FoamFile
      |{
      |    version     2.0;
      |    format      ascii;
      |    class       volScalarField;
      |    object      T;
      |}
      |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
      |
      |dimensions      [ 0 0 0 1 0 0 0 ];
      |
      |internalField   uniform ${initial_values.temperature.value};
      |
      |boundaryField
      |{
      |    solid1_to_fluid
      |    {
      |        type
compressible::turbulentTemperatureCoupledBaffleMixed;
      |        value           uniform
${boundary_conditions.isothermal.value};
      |        Tnbr            T;
      |    }
      |}
```

```
      |
      |// ********************************************************* //
  }
}

-------------------------------------------------------------
-- constant/fluid section
-------------------------------------------------------------

properties_fluid_g <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       uniformDimensionedVectorField;
  |    location    "constant/fluid";
  |    object      g;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |dimensions      [0 1 -2 0 0 0 0];
  |value           (0 0 0);
  |
  |// ********************************************************* //

turb =
  if simulation.domain.fluid.turbulence.NONE? {
    |simulationType  laminar;
  }
  else if simulation.domain.fluid.turbulence.SA? {
    |simulationType  SA;
  }
  else {
    |
  }

properties_fluid_turbulence <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       dictionary;
  |    location    "constant";
  |    object      turbulenceProperties;
  |}
```

```
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |${turb}
  |
  |// ********************************************************* //

render ::viscosity_model.constant              |mu
${value};

render ::thermal_conductivity_model.constant_prandtl |Pr
${laminar_prandtl_number};

render ::fluid_model.incompressible_ideal_gas
  |    specie
  |    {
  |        nMoles          1;
  |        molWeight       ${molecular_weight};
  |    }
  |    thermodynamics
  |    {
  |        Cp              ${specific_heat_capacity};
  |        Hf              0;
  |    }

transport =
if simulation.domain.fluid.viscosity_model.constant? {
  |transport       constant; -- if constant dynamic viscosity
}
else {
  |
}

vis =
if simulation.domain.fluid.viscosity_model.constant? {
  |transport       constant; -- if constant dynamic viscosity
}
else {
  |
}

fm =
if simulation.domain.fluid_model.incompressible_ideal_gas? {
  |    equationOfState incompressiblePerfectGas; -- if
incompressible_ideal_gas
}
```

```
else {
  |
}

fluid_template =
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       dictionary;
  |    location    "constant";
  |    object      thermophysicalProperties;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |thermoType
  |{
  |    type            heRhoThermo;
  |    mixture         pureMixture;
  |${transport}
  |${vis}
  |    thermo          hConst;
  |${fm}
  |    specie          specie;
  |    energy          sensibleInternalEnergy;
  |}
  |
  |dpdt yes;
  |
  |air
  |{
  |
${simulation.domain.fluid.density_model.variable.incompressible_ideal_
gas}
  |    transport
  |    {
  |        ${simulation.domain.fluid.viscosity_model};
  |        ${simulation.domain.fluid.thermal_conductivity};
  |    }
  |}
  |
  |// ************************************************************* //

properties_fluid_thermo << fluid_template

-------------------------------------------------------------
```

```
-- constant/solid
-------------------------------------------------------------

render ::thermal_diffusivity.components
  |    transport
  |    {
  |        kappa   ${thermal_conductivity}; // Thermal conductivity
[W/(m.K)]
  |    }
  |    thermodynamics
  |    {
  |        Hf      0;
  |        Cp      ${heat_capacity}; // Specific heat capacity
[J/(kg.K)]
  |    }
  |    equationOfState
  |    {
  |        rho     ${density}; // Density [kg/m3]
  |    }

solid_thermo =
  |thermoType
--  a model for a solid material with constant properties
  |{
  |    type            heSolidThermo;
  |    mixture         pureMixture;
  |    transport       constIso;
  |    thermo          hConst;
  |    equationOfState rhoConst;
  |    specie          specie;
  |    energy          sensibleEnthalpy;
  |}
  |
  |mixture
  |{
  |    specie
  |    {
  |        molWeight   12; // molar weight, [g/mol]
  |    }
  |${simulation.domain.solid.conduction}
  |}
  |
  |// ************************************************************* //

in simulation.domain {
```

```
  filename = |constant/${name}/thermophysicalProperties
  in solid {
    cs_out = file(filename)
    cs_out << solid_thermo
  }
}

solid_rad =
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       dictionary;
  |    location    "constant";
  |    object      radiationProperties;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |radiation off;
  |
  |radiationModel none;
  |
  |// ************************************************************************* //


in simulation.domain {
  filename = |constant/${name}/radiationProperties
  in solid {
    cs_out = file(filename)
    cs_out << solid_rad
  }
}

------------------------------------------------------------------
-- constant/regionProperties
------------------------------------------------------------------

fluid_names =
for d in simulation.domain
  if d.fluid? {
    |${d.name}
  }
  else {
    |
  }
```

```
solid_names =
for d in simulation.domain
  if d.solid? {
    |${d.name}
  }
  else {
    |
  }

regions <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       dictionary;
  |    location    "constant";
  |    object      regionProperties;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
  |regions
  |(
  |    fluid       (${join(" ", fluid_names)}
  |    solid       (${join(" ", solid_names)})
  |
  |);
  |
  |// ************************************************************* //

------------------------------------------------------------------
-- system/fluid
------------------------------------------------------------------

fluid_schemes <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       dictionary;
  |    location    "system/fluid";
  |    object      fvSchemes;
  |}
  |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
  |
```

```
  |ddtSchemes
  |{
  |    default         Euler;
  |}
  |
  |gradSchemes
  |{
  |    default         Gauss linear;
  |}
  |
  |divSchemes
  |{
  |    default         none;
  |
  |    div(phi,U)      Gauss upwind;
  |    div(phi,K)      Gauss linear;
  |    div(phi,h)      Gauss upwind;
  |    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
  |}
  |
  |laplacianSchemes
  |{
  |    default         Gauss linear corrected;
  |}
  |
  |interpolationSchemes
  |{
  |    default         linear;
  |}
  |
  |snGradSchemes
  |{
  |    default         corrected;
  |}
  |
  |// ************************************************************* //

fluid_solution <<
  |FoamFile
  |{
  |    version     2.0;
  |    format      ascii;
  |    class       dictionary;
  |    location    "system/fluid";
  |    object      fvSolution;
```

```
|}
|// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
|
|solvers
|{
|    rho
|    {
|        solver          diagonal;
|    }
|
|    rhoFinal
|    {
|        \$rho;
|    }
|
|    p_rgh
|    {
|        solver           GAMG;
|        smoother         symGaussSeidel;
|        tolerance        1e-7;
|        relTol           0.01;
|    }
|
|    p_rghFinal
|    {
|        \$p_rgh;
|        relTol           0;
|    }
|
|    "(U|h)"
|    {
|        solver           PBiCGStab;
|        preconditioner   DILU;
|        tolerance        1e-7;
|        relTol           0.1;
|    }
|
|    "(U|h)Final"
|    {
|        \$U;
|        relTol           0;
|    }
|}
|
|PIMPLE
```

```
  |{
  |    momentumPredictor   yes;
  |}
  |
  |relaxationFactors
  |{
  |    equations
  |    {
  |        h               1;
  |        U               1;
  |    }
  |}
  |
  |// ************************************************************* //

------------------------------------------------------------
-- system/solid
------------------------------------------------------------

in simulation.domain {
  filename = |system/${name}/fvSchemes
  in solid {
    cs_out = file(filename)
    cs_out <<
      |FoamFile
      |{
      |    version     2.0;
      |    format      ascii;
      |    class       dictionary;
      |    location    "system/solid"; --TODO: fix location
      |    object      fvSchemes;
      |}
      |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
      |
      |ddtSchemes
      |{
      |    default         Euler;
      |}
      |
      |gradSchemes
      |{
      |    default         Gauss linear;
      |}
      |
      |divSchemes
```

```
      |{
      |    default         none;
      |}
      |
      |laplacianSchemes
      |{
      |    default         none;
      |    laplacian(alpha,e) Gauss linear corrected;
      |}
      |
      |interpolationSchemes
      |{
      |    default         linear;
      |}
      |
      |snGradSchemes
      |{
      |    default         corrected;
      |}
      |
      |// ************************************************************* //
  }
}

in simulation.domain {
  filename = |system/${name}/fvSolution
  in solid {
    cs_out = file(filename)
    cs_out <<
      |FoamFile
      |{
      |    version     2.0;
      |    format      ascii;
      |    class       dictionary;
      |    location    "system/solid"; --TODO: fix location
      |    object      fvSolution;
      |}
      |// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
      |
      |solvers
      |{
      |    e
      |    {
      |        solver           GAMG;
      |        smoother         symGaussSeidel;
```

```
        |         tolerance        1e-6;
        |         relTol           0.1;
        |     }
        |
        |     eFinal
        |     {
        |         \$e;
        |         relTol           0;
        |     }
        |}
        |
        |PIMPLE
        |{
        |     nNonOrthogonalCorrectors 0;
        |}
        |
        |// ************************************************************************* //
    }
}
```

Simulation results: SU2

- Results for Multiphysics, with three heated cylinders in a cool fluid that involves fluid flow and conjugate heat transfer: