Benesh: a Programming Model for Coupled Scientific Workflows

Philip E. Davis*, Pradeep Subedi*, Shaohua Duan*, Lee Ricketson◊, Jeffrey A.F. Hittinger◊, Manish Parashar*

* Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ
◊ Lawrence Livermore National Laboratories, Livermore, CA
Introduction

• What I am talking about today is Benesh
  – a programming model for scientific workflows
  – makes it easier to combine existing codes into complex workflows
  – allows interactions be specified separately from the codes themselves

• What I am NOT talking about today
  – An architecture or performance model for Benesh

• Apologies on terminology (coupling, composition, workflow, …)
Programming In-situ Workflows

• What do I mean by in-situ workflows?
  – A single scientific computing experiment (simulation, analysis, viz, etc.)
  – Different programs (components) of the workflow run together, actively exchanging data

• Multiphysics workflow
  – Multiple simulations working together to solve some larger/more complex problem
  – E.g. split in space (WDMApp), split on physics (plasma + laser), split in medium (ice+sea)
  – Solutions are coupled in some mathematical regime
  – Codes must exchange data to accomplish this regime

• Common programming methods for workflows
  – ad hoc – directly code interactions into software packages; difficult to change and maintain
  – Orchestration – decompose workflow into many discrete tasks, orchestrator satisfies dependencies and launches new tasks; difficult to refactor legacy codes, loss of autonomy
Outline

1. Benesh Overview
2. Model Elements
3. Example
4. Future Work
Benesh

- A programming model for developing in-situ workflows
  - Take existing codes and make them work together
  - Abstractions aimed at supporting multiphysics use cases

- Programming-language hooks for preparing an existing code for use in a Benesh workflow

- Workflow description language for specifying the interactions of workflow components
  - Provide enough information about the workflow to make interactions flexible

- *(In progress)* Middleware for instantiating Benesh workflows efficiently
Goals

1. Reuse existing code

2. Rapid experimentation
   – Adding and removing components
   – Swapping components
   – Changing schemes

3. Consolidation of workflow language
   – Make interactions explicit
Benesh
Benesh
Benesh – Language Concepts

• Interface – What can be done by a component?
  – Each component implements an abstract interface that is declared as part of the workflow description. This encapsulation of components allows component functionality to be modularized.

• Data Target Rules – How should interactions occur?
  – Makefile-like rules define dependencies and ‘recipes’ for creating data objects
  – Invocation of target rules automatically invokes dependent rules.

• Touchpoints – When should interactions occur?
  – Points in the code where workflow interactions occur / dependencies to advance are satisfied
  – Touchpoints are where workflow and component-level tasks are synchronized.
  – Touchpoints signal when to invoke data target rules.
Benesh – Model Features

• Workflow is composed with a service-oriented architecture
  – Well-defined interface is implemented by each component
  – Borrow interface concepts from OO programming

• Workflow components are autonomous
  – Parallel (conceptual) execution tracks for workflow and component-level operations

• Workflow interactions are *choreographic*
  – Components have a shared understanding of the workflow
  – Components coordinate their activities when necessary

"Dancers Leaping" by Gabriel Saldana is licensed under CC by 2.0
Language Elements

- **Abstract Component Interface model**
  - Declare the interface that codes must provide in order to fill the role of a component in a workflow
  - Define the data objects that are visible at the workflow level
- **Workflow data model**
  - Define the global data domain and how this domain maps into workflow components
- **Workflow choreology**
  - Recipes for realizing component interactions
  - What kind of component state changes or data events trigger these interactions
- **Code Binding API**
Abstract Component Interface

• Interface in the OO sense
• Any component that can act in a particular role of a workflow provides this interface
  – Can simulate a plasma over the required domain in the required way
  – Can perform the required analysis
  – Can visualize the simulation in the desired way
• This is the abstraction of the codes in terms of how they interact, and the developer must instantiate this interface
• Function signatures and local data structures
• State transition markers from code (touchpoints)
Workflow Data Model

- Defines the parameters of a data domain
  - Will later be bound to interface data structures
  - Can be changed on a run-by-run basis in order to resize or redistribute the data domain between components
- Contains information to map data to components
  - Boundaries, overlaps, etc can be derived
- Does NOT contain information on mapping ranks to data
  - May be determined or changed after the workflow starts

```plaintext
domain <domain_name>:
  range: [<start>,<end>]
  periodic: (<val>,...<val>)
domain <subdomain_name>:
  range: [<start>,<end>]
  periodic: (<val>,...<val>)
```
Workflow choreology

• **Component declaration**
  – Components instantiate an interface using a particular program

• **Data rules**
  – A makeflow-type syntax that takes the form of
    target: dependencies
    procedure

• **State transition/synchronization**
  – Operations that take place when a component (or the workflow as a whole) reaches some defined location (this ties the user code state to the workflow state.) See examples.
Workflow choreology – component declaration

• Bind workflow components to a particular role (interface) that component interfaces provide.
• Interface refers to the abstract interfaces that already declared.
• Credential is provided by the component as part of the client bindings.
• Different credentials refer to different implementations of the same role.

```plaintext
component <comp_name>(<interface>)[<credential>]
component Generator1(ArrayBuilder1D)[random]
component Generator2(ArrayBuilder1D)[exponent]
component Analyzer(ArrayCompare1d)[maxdiff]
```
Workflow choreology – target generation rules

• ‘makeflow’-style target definition
  – Target name and dependencies, followed by a “recipe”

• Rules call interface methods

• Parameterize rules with %{} syntax (allow some manipulation of variables, e.g. %[[t-1]])
  – e.g. Generator2.x.1 depends on Generator2.x.0

```ini
<comp_name>.<var_name>.<version>: <dependencies>
<procedure>

Generator1.x.0:
Generator1.x.%{t}: Generator1.x.%[[t-1]]
  Generator1.build_array(seed) in=x.%[[t-1]] ; out= x.%{t}

Generator2.x.0:
Generator2.x.%{t}: Generator2.x.%[[t-1]]
  Generator2.build_array(seed) in=x.%[[t-1]] ; out= x.%{t}

Analyzer.diff.%{t}: Generator1.x.%{t} Generator2.x.%{t}
  Analyzer.x.%{t} < Generator1.x.%{t}
  Analyzer.y.%{t} < Generator2.y.%{t}
  Analyzer.compare_array()
```
Workflow choreology – touchpoint rules

- Rules for synchronizing component activity and state with workflow activity and state

- Touchpoints (identified by what’s to the right of ‘@’) are reached at certain points in the component code

```plaintext
<comp_name>@<touchpoint_name>:
  <target>

Generator1@ts.%{t}:
  Generator1.x.%{t}

Generator2@ts.%{t}:
  Generator2.x.%{t}

Analyzer@ts.%{t}:
  Analyzer.diff.%{t}
```
Code Binding API

- API calls to bind component code
- Binding syntax depends heavily on language capabilities (e.g. function pointers in C)
- Future work to create preprocessor
- Benesh will need to manage variable buffer(s), so a data accessor API is required in some languages.

```
bind_method(function_pointer, "method_name")
touchpoint("touchpoint_name")
bind_domain("domain_name", "variable_name", range)

bind_data(data_pointer, "data_name")

---or---

var_access_scalar(buffer_pointer, "variable_name", TYPE)
var_access_buffer(buffer_pointer, "variable_name", TYPE)
var_access_iter(iter_pointer, "variable_name", ...)
...
```
Example Workflow

- Solve a physical quantity \( u \) on a 3D domain
  - Unit cube
  - Split into left and right subdomains
- Suppose two different \( u \)-solvers
  - speedy
  - special
- Solve each subdomain with a separate simulation component, exchanging boundary data for consistency
ad hoc method - lockstep

```python
do_timestep():
    du = sxxx_calc_du(u)
    u = sxxx_advance(u, du)
    ... 
while(ts++):
    ...
    do_timestep()
    ...
do_timestep():
    wait_for_other()
    read_boundaries(u)
    du = sxxx_calc_du(u)
    u = sxxx_advance(u, du)
    write_boundaries(u)
    signal_other()
    ... 
while(ts++):
    ...
    do_timestep()
    ...
```
**ad hoc method - lockstep**

- Boundary exchanges hides complexity
  - Both codes must agree on boundary domain and data organization

- Swapping out codes requires maintaining this order of operations

- Changes in domain need to be coordinated

- Further splitting the domain would require more complex coordination operations

```python
do_timestep():
    wait_for_other()
    read_boundaries(u)
    du = sxxx_calc_du(u)
    u = sxxx_advance(u, du)
    write_boundaries(u)
    signal_other()

... while(ts++):
    ...
    do_timestep()
    ...
```
ad hoc Changing scheme

```python
do_timestep():
    wait_for_other()
    read_boundaries(u)
    du = sxxx_calc_du(u)
    u = sxxx_advance(u, du)
    write_boundaries(u)
    signal_other()

    ...  
while(ts++):
    ...
    do_timestep()
    ...
```

```python
do_timestep():
    write_boundaries(u)
    read_boundaries(u)
    du = sxxx_calc_du(u)
    u = sxxx_advance(u, du)
    signal_other()
    wait_for_other()

    ...  
while(ts++):
    ...
    do_timestep()
    ...
```
Ad hoc changing scheme

- Do we understand the data consistency requirements for the remainder code?

```python
do_timestep():
    write_boundaries(u)
    read_boundaries(u)
    du = sxxx_calc_du(u)
    u = sxxx_advance(u, du)
    signal_other()
    wait_for_other()
...
while(ts++):
    ...
    do_timestep()
    ...
```
Preparing the code for Benesh

do_timestep():
    du = sxxx_calc_du(u)
    u = sxxx_advance(u, du)
...
while(ts++):
    ...
    do_timestep()
    ...

benesh_bind_method("calc_du", sxxx_calc_du)
benesh_bind_method("advance", sxxx_advance)
benesh_bind_data("u", u)
benesh_bind_data("du", du)

...
Benesh Workflow Definition

```cpp
interface explicit3d:
    real<3> u
    real<3> du
    calc_du(real):
        in: u
        out: du
    advance(real):
        in: u, du
        out: u
    touchpoints:
        ts.%n
```

domain Global:
    periodic(a, a, a)
    range: [0,1] x [0,1] x [0,1]

domain Left:
    range: [0,.6] x [0,1] x [0,1]

domain Right:
    range: [.4,1] x [0,1] x [0,1]
Benesh Target Generation Rules - lockstep

```plaintext
real dt = .01
real t

Left.u.0:
    t = 0
Left.u.%{ts}: Left.u.%[[ts-1]] Right.u.%[[ts-1]]
    boundary(Left.u.%[[ts-1]] < Right.u.%[[ts-1]])
    Left.calc_du(t): in=u.%[[ts-1]] ; out=du.%[[ts-1]]
    Left.advance(dt): in=u.%[[ts-1]], du.%[[ts-1]] ; out=u.%{ts}

Right.u.0:
Right.u.%{ts}: Left.u.%{ts} Right.u.%[[ts-1]]
    boundary(Right.u.%[[ts-1]] < Left.u.%{ts})
    Right.calc_du(t): in=u.%[[ts-1]] ; out=du.%[[ts-1]]
    Right.advance(dt): in=u.%[[ts-1]], du.%[[ts-1]] ; out=u.%{ts}
    t = t + dt
```
Benesh Target Generation Rules – scheme change

```
real dt = .01
real t

Left.u.0:
    t = 0
Left.u.{ts} : Left.u.{{ts - 1}} Right.u.{{ts - 1}}
    boundary(Left.u.{{ts - 1}} < Right.u.{{ts - 1}})
    Left.calc_du(t) : in=u.{{ts - 1}} ; out=du.{{ts - 1}}
    Left.advance(dt) : in=u.{{ts - 1}} , du.{{ts - 1}} ; out=u.{{ts}}

Right.u.0:
Right.u.{ts} : Left.u.{{ts - 1}} Right.u.{{ts - 1}}
    boundary(Right.u.{{ts - 1}} < Left.u.{{ts - 1}})
    Right.calc_du(t) : in=u.{{ts - 1}} ; out=du.{{ts - 1}}
    Right.advance(dt) : in=u.{{ts - 1}} , du.{{ts - 1}} ; out=u.{{ts}}
    t = t + dt
```
Future Work – Development of Supporting Middleware

• Develop middleware to actualize Benesh workflows
  – Leverage and extend composable* data services

• Create performance model for workflow-level operations
  – Portable between execution spaces
Future Work – Binding Interface

• Interface binding is based on high-level features of the code

• It is desirable to minimize the work of preparing a code for Benesh

• Code annotation/pragma and introspection tools may be a better solution than runtime API hooks
Acknowledgements

• The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute.

• This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

• This work was also supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.