Medical Simulation Markup Language

A tutorial introduction

Nicolai Schoch [*] Markus Stoll [†] Stefan Suwelack [‡]
Alexander Weigl [§]

November 16, 2014

Abstract

1 Introduction

We first mentioned the idea from an unified and mighty workflow system for biomechanical engineering in [1]. In the meantime we realize the Medical Simulation Markup Language (MSML).

You can see MSML as a build tool, like *GNU Make*, except the offered operations are settled in the pre- and postprocessing of three dimensional data after the biomechanical simulation. The simulation is the heart of every workflow and is outsourced to modern simulation frameworks, like SOFA[1] or Hiflow3[2], and MSML cares about the messy details of the simulation framework. You get an unified interface.

Content In this paper we show the various power of MSML in different scenarios (sections **??**). Before the tutorials we create a view on MSML in section 2. This chapter defines the ground terms and nomenclatures in MSML. Openness is one power of MSML. You can extend it in various kinds, such topics are discovered in section 5.

Acknowledgement MSML was developed in the SFB TRR 125 Cognition Guided Surgery project, funded by the DFG.¶[3]

[*] `nicolai.schoch@iwr.uni-heidelberg.de` IWR EMCL

[†] `m.stoll@dkfz-heidelberg.de`, DKFZ

[‡] `suwelack@kit.edu`, Humanoids and Intelligence Lab, Karlsruher Institute for Technology

[§] `Alexander.Weigl@student.kit.edu`, Humanoids and Intelligence Lab, Karlsruher Institute for Technology

[1] http://www.sofa-framework.org/

[2] http://hiflow3.org

[3] *weigl:* add english name here

2 MSML Architecture

We start with the terms, give an overview of the pipeline and end this section with the MSMLFile datastructure.

Terms An *Operator* is a function, that can be used within the workflow. The function can be written in C/C++ or Python. We maintain additional meta data of the arguments and output in a separate XML file. If an Operator is used, it get instantiated with the given references to other instances or variables. We call instances of Operator a *Task*. *Elements* define additional features, like materials or constraints, to your objects in the scene. The *Alphabet* contains both, Elements and Operators, with their XML meta data. You can add new Operators or Elements to MSML by creating new Alphabet entries. The simulation is outsourced to several simulation frameworks. The interface between MSML and the frameworks is done within several specific *Exporter*. Currently we support Sofa, Hiflow, FeBio and Abaqus.

Pipeline Figure 1 gives an overview about the process. MSML takes a data structure as shown in figure 2. The data structure can allocated from XML or with Python. Defining in Python offers more control over the pipeline and is more flexible, as you can use parameterized the creation. XML is easy, understandable but more fixed.

The first step analyze the data structure for finding errors and creating a build graph. The build graph is a directed acyclic graph[4]. The generated build graph is checked for consistency of types. We try to solve type errors by injecting conversions in the graph. Such a graph is in figure **??**.

[4]You mustn't have cycles.

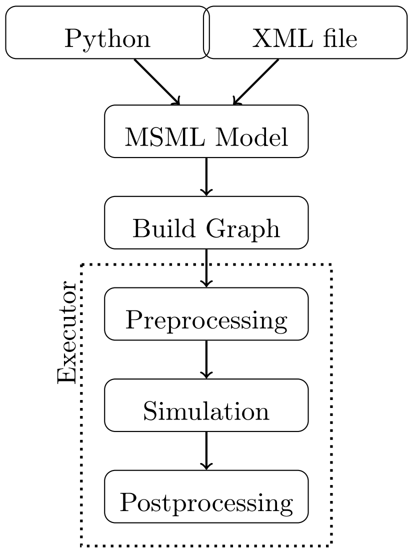

Figure 1: *MSML Pipeline*

An *Executor* takes control over the pre-, postprocessing and simulation. It executes the build graph in the correct order. Currently We are offering three Executors:

LinearSequence is a simple execution of the task in topological order.

Parallel executes a set of independently task in process or thread pool.

Phase gives you control, to disable certain phases, task or exporter.

The distinction between the three phases is a logical separation. The build graph only knows nodes, that are tasks, variables or exporter.

Data structure The *MSML File* data structure is the central interface between you and MSML. It gathers all information for making a simulation:

Workflow holds the list of Tasks to be executed

Environment is the set of parameters and configuration for the simulation system.

Variables are reusable placeholders, that allow to create parameterized workflows.

Scene Object encapsulates an object in the simulation, that can be enriched by material regions, constraints or output requests.

The structure is visible in the first example ?? and figure 2 shows the cardinality.

Sort logic

3 Everyone has the bunny, we too.

4 Have someone say: Optimization?

5 Extend MSML

MSML is a platform, that can be extended in multiple fashions. The terms explain in section 2

5.1 Operator

An operator in MSML is a function that perform an operation. At least the function has to executable within Python. For C/C++ we provide a CMAKE build environment and wrapping with SWIG. Additional we provide operator adapters for external programs or shared object (ctype).

```
import vtk
def cp(mesh, ref):
  locator = vtk.vtkPointLocator
      ()
  ugrid = read_ugrid(mesh)
  locator.SetDataSet(ugrid)

  index = locator.
      FindClosestPoint(ref)
  point = ugrid.GetPoint(index)
  distance = distance(ref,
      point)
  return {'index': index,
          'point': point,
           'dist': distance}
```

Figure 3: *Python snippet of a function for calculating the nearest point in `mesh` from `ref`*

Let's assume we want to provide `cp(mesh , ref)` from figure 3 in MSML. The function calculates the closest point in `mesh` to a given reference vector `ref`. Save the function in Python module, make sure MSML can import this module by setting `PYTHONPATH` or using the `--operator-dir` on the command line. The next step is to create the entry in the Alphabet. Figure 4 shows an accuarate one. The runtime gives the type of the operator. For a Python Operator you need to specify the module and the function name. Input, output and parameters contains list of args. The order of args determines the order in which the arguments are given in the function call. Here we define one input argument, that should be given as a VTK object, and one parameter as a list of floats values. This operators delivers three different output values. Once you added the XML file (figure 4) to the alphabet search path or add `--alphabet-search-dir` on the command line, you should be able to call this operator within MSML `<closestPoint id="c"mesh="${mesh}"ref="2.2␣3.5␣6"/>`

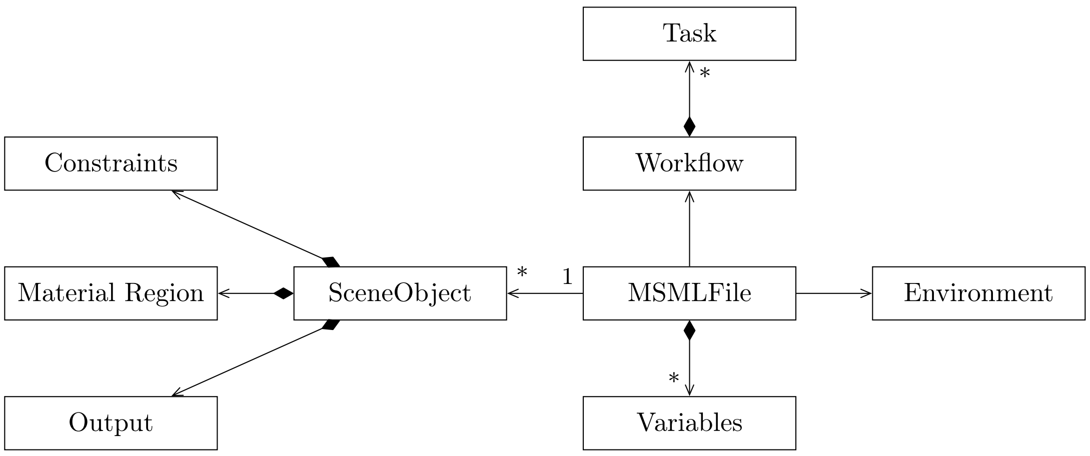

Figure 2: *MSML Model*

You can access to every output with `c.distance`, `c.point` and `c.index`.

5.2 Element

An Element defines the semenatics of an entry under constraint, material region or output. They should be a disambiguous description of their use. Every exporter handles the elements on his own. An introduction of an element leads to an adaption of the exporters. The Exporter features determines the supported elements. Figure 5 shows an element definition. It consists from a name, category, description and parameters. The definition of parameters are the same as for operators.

```
<element name="surfacePressure"
      category="constraint">
    <description>
        Add pressure load to
            surface
    </description>

    <parameters>
        <arg name="indices"
            physical="vector.
            float" />
        <arg name="pressure"
            physical="float"/>
    </parameters>
</element>
```

Figure 5: *Operator Definiton Example*

5.3 Exporter

The Exporter are a central point of the system. Their task is to create the input for the specific simulation environment, run the simulation and provide the calculated output back into the Workflow. You can see the Exporter like special operators, that sees the whole MSML file data structure and has dynamically input and output slots.

The creation of an Exporter begins with deriving from `msml.exporter.Exporter` (figure 6). The Exporter need to call `initialize` given:

- the MSML File data structure,
- an exporter name,
- supported features
- logical and physical mesh sort and
- a dictionary describing types of element parameters.

The initialization takes care of creating the appropriate input and output slots. The supported features are a subset of string. If you set physical types for every input and output slot, you will get the correct on access via `get_value_from_memory()`.

The next step is `render()`. This method is called first from the Executor. It should transfer the scene structure into an appropriate representation for the simulation environment. You should not modify the workflow memory or write absolute filenames. Everything executed by the Executor has it's working directory in the given output directory. The changes you want to make should be memorized and returned by `execute()`. There are two ways to access the referenced values from the workflow memory: by calling `get_value_from_memory()` with the correct nodes from the scene, or by using the `datamodel`. `datamodel` is a mirror data structure of scene, where the referenced values are injected.

`execute()` is executed directly after `render()`, and should execute the simulation and converts the results into appropriates formats. The later step is only needed if there is no automatically conversion defined. Figure 7 gives an sketch of an exporter.

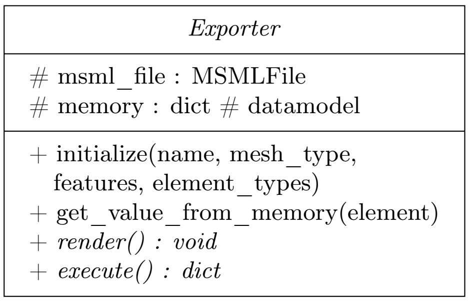
Exporter
# msml_file : MSMLFile
# memory : dict # datamodel
+ initialize(name, mesh_type, features, element_types)
+ get_value_from_memory(element)
+ *render() : void*
+ *execute() : dict*

Figure 6: *UML Class Exporter*

6 Conclusion

We hope to give you an introduction in the world of MSML. MSML is more than just a build or workflow tool. The project offers easily to use Python and C++ function.

MSML is driven towards an semantic an intelligent system.

References

[1] Stefan Suwelack, Markus Stoll, Sebastian Schalck, Nicolai Schoch, Rüdiger Dillmann, Rolf Berndl, Vincent Heuveline, and Stefanie Speidel. The medical simulation markup language - simplifying the biomechanical modeling workflow. *Studies in Health Technology and Informatics*, 196:396–400, April 2014.

```
<operator name="closestPoint">
    <runtime>
        <python module="closestpoint" function="cp"/>
    </runtime>

    <input>
        <arg name="mesh" logical="Mesh" physical="vtk"/>
    </input>

    <output>
        <arg name="index" logical="Index" physical="int"/>
        <arg name="point" logical="Point3D" physical="vector.float"/
            >
        <arg name="dist" logical="Distance" physical="float"/>
    </output>

    <parameters>
        <arg name="ref" physical="vector.float" logical="Point3D"/>
    </parameters>
</operator>
```

Figure 4: *Operator Definiton Example*

```
from msml.exporter import Exporter

class ExporterSkeleton(Exporter):
    def __init__(self, msmlfile):
        self.initialize(msmlfile, name = "myexporter",
                        features = supported,
                        mesh_sort = ...,
                        output_type_of_elements = ...)

    def render(self):
        self._update = {}
        for sceneobject in self.datamodel:
            # process object
            sceneobject.id
            sceneobject.mesh
            for constraints in sceneobject.constraints:  pass
            for regions in sceneobject.materials: pass
            for output in sceneobject.output: pass

        # write simulation input file

    def execute(self):
        # execute the simulation
        subprocess.call(["run_simulation" ...])
        # return the new memory values
        return self._update
```

Figure 7: *Operator Definiton Example*