A Qualitative Reasoning Model of a simple (?) Container System

Project Report

Jonsson, Haukur 11137304 | Rajamanickam, Santhosh 11650702 | Rapp, Max 11404310

October 27, 2017

1 Introduction

Everyday life requires engaging with countless systems of varying complexity. Successful engagement necessitates robust approximate prediction of their behaviour. However, understanding the dynamics of even the simplest of these systems would require solving a set of differential equations of prohibitive complexity.

Rather than precise modelling, humans therefore engage in *Qualitative Reasoning* to understand and predict the behaviour of systems surrounding them. Qualitative models are built by using simplifying *assumptions* to construct a simplified ontology of the system including its objects and the causal relations between them as well as discretizing the quantities of the objects and the magnitudes of the relations. Such models are surprisingly powerful in predicting possible system outcomes.

It is threfore arguable that understanding a domain is a function of one's qualitative model of that domain. In this paradigm, human (and machine) learning and teaching can then be understood in terms of building and adapting qualitative models of the world.

The computer assisted learning paradigm tries to leverage this approach by developing *Human Level AI* that assists and interacts with the learner in building better qualitative models.

The goal of this project is to implement a qualitative model of a container system that could serve in an interactive learning software by taking various inputs and returning a state graph and trace of the development of the system.

2 Three Container Models

The system to be modelled can be described as follows: A **tap** is used to regulate the influx of water into an open **container** with a **drain** through which water escapes.

The basic ingredients needed to build a model for this system are the *entities* which describe the basic objects the model contains. An entity may have one or more *quantities* describing its magnitude. The quantities each have a magnitude constrained by a *quantity space* containing its allowed values as well as a *derivative* with quantity space (-,0,+) describing the current change in quantity. Between quantities *causal relations* obtain. In our case these include *influences* (I^+, I^-) and *proportionalities* (P^+, P^-). Influences are positive or negative relationships from the magnitude of a quantity to the derivative of another. Proportionalities are positive or negative relationship from one derivative to another.

Using these ingredients, depending on one's assumptions, models of varying complexity of the container system can be devised. Here we describe three.

2.1 Model I

Ontology:

- Entities: Tap, Container, Drain
- Quantities(Spaces): Inflow (Zero; Plus), Volume (Zero, Plus, Max), Outflow (Zero, Plus, Max)
- Relations: I^+(Inflow, Volume), I^-(Outflow, Volume), P^+(Volume, Outflow)

Algorithm 1 Construct State Graph

```
procedure MakeGraph(intialState,causalGraph)
    stateGraph ← graph.addHead(initialState)
    stateStack ← stack.push(initialState)
    while stack.length(stateStack)! = 0 do
        currentState ← stack.pop(stateStack)
        nextStates[] ← computeNextStates(currentState, causalGraph)
        for index ← 0 to length(nextStates) do
            childState ← nextStates[index]
            exists ← checkStateExists(state)
            if notexists then
                assignStateNumber(childState)
                graph.addChild(stateGraph,currentState,childState)
                stack.push(stateStack,childState)
            else
                graph.addChild(stateGraph,currentState,childState)
    return stateGraph
```

Algorithm 2 Compute next state transitions

```
procedure COMPUTENEXTSTATES(currentState,causalGraph)
    newState ← NULL
    newState ← applyPointChanges(currentState, causalGraph)
    if newState not NULL then
        newState ← applyStaticChanges(newState, causalGraph)
        return newState
    newStatesList[ ]
    counter ← 0
    newStatesList ← applyIntervalChanges(currentState, causalGraph)
    for index ← 0 to length(nextStatesList) do
        newState ← applyStaticChanges(newStatesList[i], causalGraph)
        isConsistent ← checkStateConsistency(newState)
        if isConsistent then
            newStatesList[counter] ← newState
            counter++
    return newStatesList
```

Algorithm 3 Applying Point Changes

```
procedure APPLYPOINTCHANGES(currentState,causalGraph)
    newState ← currentState
    for index1 ← 0 to length(causalGraph.entities) do
        for index2 ← 0 to length(causalGraph.entities[index1].quantaties)
do
            if newState[index1][index2].derivavtive == 0 then
                if newState[index1][index2].magnitude == 0 then
                    applyDerivative(newState[index1][index2])
                if currentState[index1][index2].magnitude == MAX then
                    applyDerivative(newState[index1][index2])
    for index1 ← 0 to length(causalGraph.relationships) do
        if causalGraph.relationships[index1] == "Influence" then
            index2, index3 ← causalGraph.relationship[index1].recievingPartyIndices
            if currentState[index2][index3].derivative == 0 then
                applyInfluenceRelationship(newState)
    return newState
```

Algorithm 4 Applying Static Changes

```
procedure applyStaticChanges(currentState,causalGraph)
    newState ← currentState
    condtion ← TRUE
    while condtion == TRUE do
        for index1 ← 0 to length(causalGraph.relationships) do
            beforeState ← newState
            if causalGraph.relationships[index1] == "Proportional" then
                applyProportionalRelationship(newState)
            if causalGraph.relationships[index1] == "Equivalence" then
                applyEquivalenceRelationship(newState)
            condition ← ifDifferent(beforeState, newState)
    return newState
```

Algorithm 5 Applying Interval Changes

```
procedure applyStaticChanges(currentState,causalGraph)
    newState ← currentState
    newStatesList[ ]
    counter = 0
    for index1 ← 0 to length(causalGraph.relationships) do
        if causalGraph.relationships[index1] == "Influence" then
            applyInfluenceRelationship(newState)
        if ifDifferent(nextState, currentState) then
            newStatesList[counter] ← newState
            counter ++
    newState ← currentState
    for index1 ← 0 to length(causalGraph.entities) do
        for index2 ← 0 to length(causalGraph.entities[index1].quantaties)
do
            applyDerivative(newState[index1][index2])
            if ifDifferent(nextState, currentState) then
                newStatesList[counter] ← newState
                counter ++
    newState ← currentState
    newState ← applyExogenous(causalgraph.typeOfExogenous, newState)
    if ifDifferent(nextState, currentState) then
        newStatesList[counter] ← newState
        counter ++
    return newStatesList
```