GRACeFUL

Global systems Rapid Assessment tools
through Constraint FUnctional Languages

FETPROACT-1-2014 Grant N° 640954

A Domain Specific Language (DSL) for GRACeFUL Concept Maps

D4.2

Lead Participant:	Chalmers (WP leader: P. Jansson)
Partners Contributing:	KU Leuven, UPC
Dissemination Level:	PU
Document Version:	Draft (version 2017-01-26)
Date of Submission:	2017-01-31
Due Date of Delivery:	2017-01-31

A Domain Specific Language (DSL) for GRACeFUL Concept Maps

Contributions by: Oskar Abrahamsson, Maximilian Algehed, Sólrún Einarsdóttir, Alex Gerdes, and Patrik Jansson.

Contents

Abstract

This second deliverable (D4.2) of work package 4 presents a Domain Specific Language (DSL) for describing GRACeFUL concept maps. This is a continuation of the initial work described in "D4.1 Formal Concept Maps Elements Descriptions" delivered in project month 6. The full source code of the language implementation is available on github and installation instructions are included in this deliverable. The implementation in Haskell can be seen as a formal semantics in terms of types and functions and this means that GRACeFUL has reached milestone M4.1 "DSL with formal semantics v1.0 ready". In addition we include a section comparing different approaches to modelling some of the formal semantics concepts relevant for GRACeFUL concept maps: causal loop diagrams, qualitative probabilistic networks, and difference equations.

1 Introduction

This document is the text part of the second deliverable (D4.2) of work package 4 of the GRACeFUL project. The software part is freely available in the project repository on GitHub[1].

The main task of work package 4 is to build a *Domain Specific Language (DSL)* for GRACeFUL Concept Maps (GCMs). A GCM is a representation of policy analysis that contains the main elements of a policy problem definition, such as goals, criteria, and a description of the system. A GCM is developed and manipulated by stakeholders during Group Model Building (GMB) sessions. The information from and for the stakeholders will go through at least two layers of the GRACeFUL system before it reaches the constraint solver. First, through the graphical visual interface, which will assist the stakeholders in building the problem definition, and which will present the results of the simulations, and second, the DSL that will translate between the visual interface and the constraint programming layer.

The DSL can be regarded as an intermediate layer between the visual representation of a GRACeFUL Concept Map (GCM) and a corresponding constraint program. Translating the visual representation directly to a constraint program would be difficult, and it would be hard to check if the generated program was correct. A DSL alleviates this problem and allows us to validate the correctness of a model. In addition, a DSL improves the scalability by abstracting away from the constraint solver. In the longer term this will lead to a DSL aimed at building scalable Rapid Assessment Tools (RATs) for collective policy making in Global Systems.

In the previous deliverable [1] we formalised the various elements of GCMs, mainly focused on Causal Loop Diagrams (CLDs). Progressive insight showed us that CLDs are not enough to model the systems the project is envisioning, such as for the Climate Resilient Urban Design (CRUD) case study. Instead of just using Causal Loop Diagrams we now also use *stock-and-flow* diagrams to model the system dynamics. Stock-and-flow diagrams allow for a more detailed (semi-)quantitative analysis. We have generalised

[1] https://github.com/GRACeFUL-project/

our DSL from the initial design in D4.1 such that it can handle Causal Loop Diagram as well as these stock-and-flow diagrams.

We have implemented the Domain Specific Language in the functional programming language Haskell [2]. Haskell is the so-called host language in which the DSL is embedded. Language features such as algebraic datatypes, higher-order functions, lazy evaluation, and a rich type system makes Haskell particularly suitable for defining DSLs. Our DSL is briefly explained in Section 3 and the full source code is freely available in the project repository on GitHub.

We continue this document by giving the necessary background information, in Section 2, for making the remaining sections more accessible. We present the details of the DSL for GCMs including some examples in Section 3. Section 4 provides installation- and usage instructions for the DSL implementation, as well as an overview of the software dependencies. Finally, we explore the formal semantics of the DSL in Section 5.

2 Background

We provide some background information about several aspects we use in later sections, such that they become more accessible. Whenever possible we give links for further reading as well.

Domain Specific Language Fowler [3] defines a DSL as follows: *a computer programming language of limited expressiveness focused on a particular domain.* A DSL is targeted at a specific class of programming tasks. By restricting scope to a particular domain, one can tailor the language specifically for that domain. There are two main approaches to implementing DSLs:

standalone A language with their own specialised syntax and parser to translate programs written in the DSL into the host language. An advantage is that the syntax can be tailor made for the target audience, and does not have to resemble the host language. However, creating such a DSL is labour intensive and it cannot easily reuse features, such as variables and conditionals, from the host language.

embedded An embedded language tries to offer a convenient syntax and abstraction mechanisms, but is offered as a library written in the host language. This means that all the existing facilities, such as abstractions standard libraries, are directly available. A disadvantage of an embedded DSL is that the users may be unfamiliar with the host language.

Gibbons [4] gives a good overview of implementing DSLs using functional programming.

Constraint programming Similar to functional and logic programming, constraint programming is a declarative programming paradigm, which means that the order of processing is not fixed. A constraint program is formulated in terms of a number of constraints. Such constraints are different from the common primitives of imperative programming languages in that they do not specify a step or sequence of steps to execute, but rather the properties of a solution to be found. The program that identifies the solutions satisfying these constraints is called a constraint solver. In general there might be none, one, or many solutions to a particular constraint problem.

The constraint programming approach is to search for a state in which a large number of constraints are satisfied at the same time. A problem is typically expressed as a state containing a number of unknown variables. The constraint solver searches for values for all the variables. For more information, see, e.g., http://constraint.org.

GRACeFUL Concept Maps (GCMs) In a Group Model Building session stakeholders perform a policy analysis that results in the form of a GRACeFUL Concept Map. This concept map contains important elements of a policy problem definition: goals, criteria for assessing the achievement of those goals, a description of the system involving factors and criteria, and the competing alternatives.

The term *factors* refers to characteristics of a system that can take a value, either quantitative or qualitative, that can change over time (source: the *GRACeFUL Glossary of Terms*). Similarly, *external factors* correspond to *inputs* to the system. As such, factors are assumed to be associated with measurable values. As in systems theory, a distinction is made between inputs that are beyond the control of the system, and which are potentially uncertain (or even unknown), and *actions* or combinations of actions, which represent the system theoretical *controls*.

The interaction between the factors, and between the factors and the criteria, represents a first description of the system. This description is in the form of a stock-and-flow diagram (which is a generalisation of Causal Loop Diagrams [5]). A stocks-and-flow diagram represents the structural understanding of a system — the causal structures that produces the observed behaviour. It reveals information about the rates of change of system elements and the measures of the variables of the system. A diagram consists of the following elements:

Stocks A stock represent a part of a system whose value at any given instant in time depends on the systems past behaviour.

Flows Flows represent the rate at which the stock is changing at any given instant, they either flow into a stock (causing it to increase) or flow out of a stock (causing it to decrease).

Converters Converters either represent parts at the boundary of the system (i.e. parts whose value is not determined by the behaviour of the system itself) or they represent parts of a system whose value can be derived from other parts of the system at any time through some computational procedure.

Connectors Much like in causal loop diagrams the connectors of a system show how the parts of a system influence each other. Stocks can only be influenced by flows (i.e. there can be no connector that connects into a stock), flows can be influenced by stocks, other flows, and by converters. Converters either are not influenced at all (i.e. they are at the systems boundary) or are influenced by stocks, flows and other converters.

Source/Sink Sources and sinks are stocks that lie outside of the models boundary — they are used to show that a stock is flowing from a source or into a sink that lies outside of the models boundary.

The elements of a policy problem are described in terms of stock-and-flow diagram elements, for example *factors* are represented by *stocks*.

TODO: explain this in much more detail.

(PaJa: no need to have "much more detail" for this deliverable. But we need to connect the sections a bit better.)

3 GenericLibrary: a DSL for GRACeFUL Concept Maps

GenericLibrary, henceforth abbreviated to GL, is a DSL for descriptions of GRACeFUL concept map components embedded in the Haskell programming language. The DSL addresses the issue of bridging the gap between constraint programming and the visualisation layer by providing abstractions for modular constraint programming. These abstractions are targeted at simplifying the description of GRACeFUL concept maps.

The DSL is divided into two parts. The first part, `GCM`, allows the user to describe the interactions of GRACeFUL concept map components and has facilities for constructing new components from existing ones. The second part, `CP`, features primitives for constructing constraint programs which describe the behaviour of an individual component.

3.1 The language

The GCM language describes components and how they are connected. The core abstraction in GL is that of the `Port`. A port is an entity which represents the way two components interact, it generalises the way factors from [1] can interact with each other. Ports can also be viewed as an abstraction of the concept of constraint variables from CFP. Each component exposes some information about the system through ports. As an example, a pump component may present one port for the current flow of water being pumped and another port for the maximum flow of the pump.

We first show the code for a somewhat simpler example: a `GCM` component modelling a fixed amount of rain falling from the sky.

```
rain :: Float -> GCM (Port Float)
rain amount = do
  port <- createPort
  set port amount
  return port
```

The CP language supports reasoning about integer and floating-point arithmetic, boolean expressions, and arrays. It has constructions like `value`, which reads the value from a port, and `assert` for expression constraints on the behaviour of a component. Computations in `CP` can be embedded in `GCM` using the `component` primitive.

We can now return to our pump, which is a `GCM` component parametrised over the maximum flow through the pump:

```
pump :: Float -> GCM (Port Float, Port Float)
pump maxCap = do
  inPort  <- createPort
  outPort <- createPort
  component $ do             -- This is in CP
    inflow  <- value inPort
    outflow <- value outPort
    assert $  inflow === outflow
    assert $  inflow ‘inRange‘ (0, lit maxCap)
  return (inPort, outPort)
```

Note that we need to use `lit` to lift `maxCap`, which is a value in the host language Haskell, into the embedded language GL.

Finally we show a more complicated component, a water storage with an `inflow`, an `outlet` to which we may connect e.g. a pump, and an `overflow`.

```
storage :: Float -> GCM (Port Float, Port Float, Port Float)
storage cap = do
  inflow   <- createPort
  outlet   <- createPort
  overflow <- createPort
  component $ do
    currentStored <- createVariable
    inf <- value inflow
    out <- value outlet
    ovf <- value overflow
    sto <- value currentStored
    assert $ sto === inf - out - ovf
    assert $ sto ‘inRange‘ (0, lit cap)
    assert $ (ovf .> 0) ==> (sto === lit cap)
    assert $ ovf .>= 0
  return (inflow, outlet, overflow)
```

In conclusion, we have seen that there is a clear separation of concerns in GL between the high level primitives for constructing complicated components from simpler ones, expressed in `GCM`, and the low level implementation details with which the `CP` language is concerned.

3.2 Expressing GRACeFUL concept map elements in GL

Many of the elements of GRACeFUL concept maps identified in [1] can be modelled in GL using language primitives such as `linkBy` (for connections), `createAction` (for actions), `assert` (for constraints) etc. GL being an *embedded* domain specific language featuring a rich set of primitive operations for reasoning in the target domain allows the programmer to construct abstractions which capture behaviour which generalises many of the concepts described in [1].

3.3 Example: Runoff flow

We show a small GL program which models a rain runoff area, like a town square, which has been provided with a pump to alleviate possible flooding issues (this is a common procedure in countries like the Netherlands). This example is a small part of a larger model used in the CRUD case study meant to show how GL can be employed to model concrete problems in CRUD. The program has three components and Fig. 1 shows a graphical view of how the components are connected.

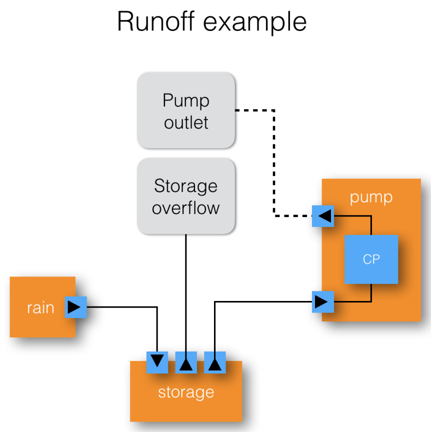

Figure 1: Runoff example structure

Given the components defined earlier, `pump`, `rain`, and `storage`, we may express Fig. 1 as:

```
example :: GCM ()
example = do
  (inflowP, outflowP) <- pump 5
  (inflowS, outletS, overflowS) <- storage 4
  rainflow <- rain 10

  link inflowP outletS
  link inflowS rainflow

  output "Overflow" overflowS
```

The `output` command lets us inspect the resulting value at a `Port` after all constraints have been solved.

Our example is compiled and run using the `runGCM` command. The constraint programming runtime, `MiniZinc`, informs us that with the numbers in our example the overflow will be zero.

```
ghci> runGCM example
{"Overflow" : 0}
```

While this example is easily calculated by hand GL is capable of expressing more complicated, and less self-contained, GRACeFUL concept maps including actions and optimization problems. As these examples are somewhat larger we omit them in this document and refer to the online resources[2].

4 Installation and software requirements

The purpose of this section is to outline the process of installing the GenericLibrary software and its require dependencies. Section 4.1 provides an overview of the software dependencies required for development using GenericLibrary. Section 4.2 provides installation instructions for a platform-independent package utilising the Docker platform. Section 4.3 provides instructions for installing and running the GenericLibrary software on UNIX-like platforms, for users with an existing Haskell toolchain.

4.1 Software dependencies

The GenericLibrary language is implemented in Haskell and uses the solver tools from the MiniZinc software distribution. Specifically, development and execution of GenericLibrary programs requires the following software dependencies to be met:

- The MiniZinc and Gecode solver software. These can be found in the MiniZinc software bundle. Section 4.3 outlines the process of installing these tools on UNIX-like systems.
- A complete Haskell toolchain able to download packages from Hackage. Two alternative tools (Stack and Cabal) are suitable for this purpose, and both are provided by the Haskell Platform.

Alternatively, for those who only wish to run the GenericLibrary examples, we provide an alternative solution in the following section.

4.2 Installation using Docker

Download and install the Docker app from https://www.docker.com/products/docker. Open a terminal (or the *command prompt* under Windows) and execute

```
docker pull eugraceful/generic-library
docker run --rm eugraceful/generic-library
```

This will download and execute the example located at examples/Examples.hs.

4.3 Installation with an existing toolchain

This section provides instructions for installing the MiniZinc/Gecode software dependencies on macOS and Linux systems. Moreover, we provide instructions for building and running GenericLibrary software using one of the existing toolchains mentioned in Section 4.1.

[2]See the examples directory of the GitHub repository for GenericLibrary.

4.3.1 Installing MiniZinc/Gecode: macOS

1. Download the complete MiniZinc distribution[3], and follow its installation instructions.
2. The MiniZinc/Gecode binaries will need to be added to your environment `PATH`. From a terminal, run

```
export MZ=/Applications/MiniZincIDE.app/Contents/Resources
echo export PATH=\"$MZ:\$PATH\" >> ~/.bashrc
source ~/.bashrc
```

4.3.2 Installing MiniZinc/Gecode: Linux

1. Download and install the complete MiniZinc distribution[4]. Place the contents of this file in a suitable directory, i.e. `~/MY_MZ_DIR/`.
2. Copy the solver tools from the extracted archive to a location on your path using

```
cd ~/MY_MZ_DIR/
cp solns2out mzn2fzn fzn-gecode mzn-gecode minizinc \
   ~/.local/bin
```

3. Set the `MZN_STDLIB_DIR` environment variable:

```
echo export MZN_STDLIB_DIR=~/MY_MZ_DIR/share/minizinc \
    >> ~/.bashrc
source ~/.bashrc
```

4.3.3 Running a constraint program

With the MiniZinc/Gecode software dependencies met, we are now able to build and execute GenericLibrary programs. First, clone the GenericLibrary source repository. From a terminal, run

```
git clone https://github.com/GRACeFUL-project/GenericLibrary/
```

- **Using cabal sandboxes**. Create a new sandbox and install the required dependencies with

```
cabal sandbox init
cabal install --dependencies-only
```

[3] https://github.com/MiniZinc/MiniZincIDE/releases/download/2.1.2/MiniZincIDE-2.1.2-bundled.dmg

[4] https://github.com/MiniZinc/MiniZincIDE/releases/download/2.1.0/MiniZincIDE-2.1.0-bundle-linux-x86_64.tgz

Build and execute the examples with

```
cabal build
cabal run examples
```

- **Using stack**. Stack will automatically take care of dependencies and sandboxing. Build and execute the examples with

```
stack build
stack exec examples
```

This will build and execute the examples located in the examples/Examples.hs source file of the repository.

5 Formal semantics

We would like to define formal semantics for our DSL in order to be able to reason formally about it. By formal reasoning we can confirm the DSL's robustness and gain further insight into it.

We started by considering causal loop diagrams, a specialization of the GCMs our DSL describes. This was done to simplify the initial scope of the work, with the expectation that the semantics defined for CLDs could then be extended to the more general GCMs. Work to extend and generalize these semantics to describe stock-and-flow diagrams, and thereby make them more consistent with the current implementation of the DSL, is ongoing.

We have also modelled CLDs *within* the `GenericLibrary`: the code for this model can be found in the file `QualitativeExample.hs`.

5.1 Causal loop diagrams

A causal loop diagram (CLD) is a directed graph used to display causal relationships between variables. The vertices represent the variables and the edges represent qualitative causal relationships, which can be positive or negative.

Different approaches can be taken in defining formal semantics to aid us in reasoning about CLDs. We have considered two such approaches: one based on qualitative probabilistic networks and the other on difference equations. We describe and compare these two approaches in the following sections.

5.1.1 Notation

We denote a positive causal relationship between A and B by $A \xrightarrow{+} B$ and a negative one by $A \xrightarrow{-} B$. Then $A \xrightarrow{+} B$ informally means that an increase in A causes an increase in B (and a decrease in A causes a decrease in B). On the other hand, $A \xrightarrow{-} B$ means that an increase in A causes a decrease in B (and conversely a decrease in A causes an increase in B). We denote the sign of the edge from A to B by s_{AB}, so $s_{AB} = +$ if $A \xrightarrow{+} B$ and $s_{AB} = -$ if $A \xrightarrow{-} B$.

A vertex A also has a sign s_A that denotes the total influence on A, so $s_A = +$ if there is an increase in A, $s_A = -$ if there is a decrease, $s_A = 0$ if there is no change and $s_A =?$ if we cannot determine the change in A.

5.2 Qualitative Probabilistic Networks

One approach to modelling and reasoning about CLDs is by using qualitative probabilistic networks (QPNs).

A QPN [6] is defined as a directed acyclic graph $G = (V, E)$ where the vertices, V, correspond to variables and the edges, E to qualitative probabilistic influences. These influences can be positive (+) or negative (-). The signs (?), for ambiguous influence, and (0), for probabilistic independence, can also be used to describe probabilistic relationships.

The meaning of signs on edges is defined according to first order stochastic dominance, as follows:

Let $F_B(\cdot|a_i, x)$ be the cumulative distribution function (CDF) for B given $A = a_i$. Then $s_{AB} = +$ means that for all possible values a_1, a_2 of A where $a_1 \geq a_2$, we must have:

$$F_B(b_0|a_1, x) \leq F_B(b_0|a_2, x),$$

that is,

$$P(B \leq b_0|A = a_1, x) \leq P(B \leq b_0|A = a_2, x)$$

for all possible values b_0 of B and any consistent context x. The context x ranges over all possible assignments to the variables other than A that influence B, that are consistent with both $A = a_1$ and $A = a_2$. The definition of $s_{AB} = -$ is the same but with $a_1 \leq a_2$.

In simpler terms, $s_{AB} = +$ means that greater values of A mean greater values of B are more likely, and $s_{AB} = -$ means that greater values of A mean smaller values of B are more likely.

These influences are symmetric, that is, if the edge from X to Y is reversed we must have $s_{XY} = s_{YX}$. Due to this symmetry it is possible to propagate an observed increase or decrease of one variable around the graph and find if other variables are likely to have increased or decreased.

This definition is broad enough to apply to many different systems and to be applicable to various real world situations.

5.2.1 Issues

We found some issues with QPNs that lead us to explore other approaches.

First of all, since QPNs were originally defined for acyclic graphs and the theory on them relies on acyclicity, they may not be the best fit to describe CLDs, in which cycles (feedback loops) are an important feature. Inference on QPNs containing loops is difficult to implement, and can lead to ambiguous results.

Second, the formal semantics of inference on QPNs is difficult to formalize since it relies heavily on not-so-simple probability theory. Additionally, QPNs are defined solely based on qualitative relationships and there is no obvious way to expand them to also describe quantitative relationships unless we have information about the probability distributions and conditional probabilities involved. GMB sessions will not produce data on probability distributions and estimating such probabilities in a reliable manner requires a sizeable dataset (that may not be available for a given GCM) as well as statistical expertise.

Lastly, since all inference in QPNs is probabilistic it leads to results that may not be as meaningful or concrete as we would like, such as "there is a heightened probability that x has increased", rather than "x has increased". For instance, a variable may decrease even though the cumulative probabilistic influence on it is positive.

5.3 Difference equation approach

Inspired by a system of tanks with water flowing from one to another, and in search of semantics that might also be extended to quantitative reasoning, we came up with the following approach.

We consider the values of the graph's vertices to be functions of the same variable, such as a time variable t.

If we have a graph with two vertices, X and Y, and one edge from X to Y, then $s_{XY} = +$ implies that

$$\frac{\partial Y}{\partial t} = G(X(t)),$$

where G is a monotonically increasing function (monotonically decreasing for negative causality, $s_{XY} = -$).

If the vertex Y has multiple parent vertices $X_1, \ldots, X_n$, then $\frac{\partial Y}{\partial t}$ depends on all the parent vertices. We can isolate the effect of a single parent vertex X on Y by differentiation.

In general we can then describe the causal relationship from X to Y as

$$\frac{\partial \left(\frac{\partial Y(t)}{\partial t} \right)}{\partial Y(t)} = g(X(t)),$$

where g has a primitive function G such that G is monotonically increasing if $s_{XY} = +$ and monotonically decreasing if $s_{XY} = -$.

This is somewhat more nuanced than $CLDs$ as they are described above, where $s_{XY} = +$ implies that an increase in X leads to an increase in Y, and a decrease in X to a decrease in Y. Here we may have some threshold value x_0 for X, where $G(x_0) = 0$, above which X always causes an increase in Y, but an increase in X causes a faster rate of increase in Y and a decrease in X causes a slowed rate of increase in Y, and vice versa.

Note that though $G(X)$ is monotonically increasing, it may not be strictly increasing, so we could for instance have $G(X) = 0$ for all $X < C$ for some threshold value C.

If the vertex Y has parent vertices $X_1, \ldots, X_n$, then we have

$$\frac{\partial Y}{\partial t} = \sum_{i=1} G_i(X_i),$$

where G_i is monotonically increasing if $s_{X_iY} = +$ but monotonically decreasing if $s_{X_iY} = -$.

In a discrete time system we consider $\Delta(X_t) = X_t - X_{t-1}$ instead of $\frac{\partial X}{\partial t}$, and write $\Delta(X_t) = G(Y_{t-1})$ instead of $\frac{\partial X}{\partial t} = G(Y(t))$. In simple cases we may only consider one time step with two values of t: t_{start} and t_{end}.

Here we explore how this approach relates to qualitative reasoning, but it could be extended to quantitative reasoning by solving the appropriate differential equations.

5.3.1 Simple qualitative model

We consider a qualitative discrete time system where all values of vertex variables are either +, -, 0, or ? (where ? is an ambiguous value assigned to a variable whose value cannot be deduced). These values have the partial ordering $- < 0 < +$, but ? cannot be compared to the other values. In place of addition and multiplication we have the operations $\oplus$ and $\otimes$, whose behaviour can be seen in the following tables:

$\oplus$	+	-	0	?
+	+	?	+	?
-	?	-	-	?
0	+	-	0	?
?	?	?	?	?

$\otimes$	+	-	0	?
+	+	-	0	?
-	-	+	0	?
0	0	0	0	0
?	?	?	0	?

The only strictly increasing function in this system is the identity function $id(x) = x$, and the only strictly decreasing function is the negation function $neg(x) = - \otimes x$.

For simplicity we consider the case where all initial values are set to zero and $G_e(0) = 0$, for all edges e. We only consider edge functions G_e where $G_e(?) =?$, since we shouldn't be able to make unambiguous deductions based on ambiguous values.

This is convenient for qualitative reasoning since then we are only concerned with increases and decreases rather than numerical values. The value of variable X at time t, which we denote by X_t, then tells us whether there has been a net increase or decrease in X.

Consider a graph with three vertices, Z and its two parents X and Y, $X \xrightarrow{s_{XZ}} Z$ and $Y \xrightarrow{s_{YZ}} Z$. Then we have

$$\Delta(Z_t) = G_{XZ}(X_{t-1}) \oplus G_{YZ}(Y_{t-1}),$$

where G_{XZ} and G_{YZ} are monotonically increasing or decreasing in accordance with s_{XZ} and s_{YZ}. If we only allow strictly increasing/decreasing functions we then have

$$\Delta(Z_t) = (s_{XZ} \otimes X_{t-1}) \oplus (s_{YZ} \otimes Y_{t-1}).$$

Consider a graph with three vertices A, B and C and two edges, $A \xrightarrow{s_{AB}} B$ and $B \xrightarrow{s_{BC}} C$. Then we have

$$\begin{aligned}\Delta(C_t) &= G_{BC}(B_{t-1}) \\ &= G_{BC}(B_{t-2} \oplus \Delta(B_{t-1})) \\ &= G_{BC}(B_{t-2} \oplus G_{AB}(A_{t-2}))\end{aligned}$$

If G_{BC} is linear, as is the case when we restrict the available functions to the strictly increasing/decreasing *id* and *neg*, we then have

$$\Delta(C_t) = G_{BC}(B_{t-2}) \oplus G_{BC} \circ G_{AB}(A_{t-2}).$$

If we only allow strictly increasing/decreasing functions we then have

$$\Delta(C_t) = (s_{BC} \otimes B_{t-2}) \oplus (s_{BC} \otimes s_{AB} \otimes A_{t-2}),$$

5.4 Comparison of approaches

We achieve the same results when inferring on CLDs no matter whether we use the QPN approach or the difference equation approach to describe the underlying semantics. Which method is simpler to understand and reason about is a matter of opinion, but we encountered some difficulties when working with the QPN approach that are outlined in section 5.2.1, which make us sceptical of the extensibility of that approach to quantitative analysis.

We believe the difference equation method is well suited to describing stock-and-flow diagrams as the approach was originally inspired by considering stock-and-flow systems, and we are working towards extending from CLDs to the more general GCMs.

6 Conclusion

We have designed version 1.0 of a DSL for describing GRACeFUL concept maps. We have provided a Haskell semantics implementing the DSL and the full source code is available on github. We have also explored alternative mathematical semantics using qualitative probabilistic networks and difference equations.

The next actions in work package 4 is

- implement a middleware for connecting the DSL to the CFP layer,
- build a testing and verification framework for RATs,
- assist WP3 in implementing the graphical user interface, and
- assist WP2 in building up a library of GRACeFUL concept map components expressed in the DSL.

In parallel, the DSL and its implementation will gradually evolve to express more and more of the requirements extractable from GMB sessions with stakeholders.

References

[1] Cezar Ionescu and Patrik Jansson. D4.1 formal concept maps elements descriptions. Deliverable of the Global systems Rapid Assessment tools through Constraint FUnctional Languages (GRACeFUL) project. FETPROACT-1-2014 Grant No 640954, 2015.

[2] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[3] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[4] Jeremy Gibbons. *Functional Programming for Domain-Specific Languages*, pages 1–28. Springer International Publishing, Cham, 2015. ISBN 978-3-319-15940-9. doi: 10.1007/978-3-319-15940-9_1. URL http://dx.doi.org/10.1007/978-3-319-15940-9_1.

[5] J.R. Burns. Simplified translation of CLDs into SFDs. In *Proceedings of the 19. International Conference of the System Dynamics Society*, 2001.

[6] M. P. Wellman. Fundamental concepts of qualitative probabilistic networks. *Artificial Intelligence*, 44(3):257–303, July 1990. ISSN 0004-3702.