GRACeFUL

Global systems Rapid Assessment tools
through Constraint FUnctional Languages

FETPROACT-1-2014 Grant Nº 640954

Formal Concept Maps Elements Descriptions

D4.1

Lead Participant:	Chalmers (Cezar Ionescu, Patrik Jansson)
Partners Contributing:	Deltares, UPC
Dissemination Level:	PU
Document Version:	Final (version 2015-07-24)
Date of Submission:	2015-07-30
Due Date of Delivery:	2015-08-01

Formal Concept Maps Elements Descriptions

Contents

Abstract

This first deliverable of WP4 includes the initial domain modelling: concept map examples relevant for the CRUD case study, examination and decomposition of these concept maps into the key underlying elements, and a formal description of these concept map elements in terms of types and relations, expressed in a Haskell-like formalism.

1. Introduction

This document represents the first deliverable of Work Package 4 and is organised as follows. The first two sections establish the larger computing science context and the specific notation of the work, providing brief pointers to functional programming and domain-specific languages (DSLs).

Section 4 provides a broad picture of the GRACeFUL project and its relation to policy analysis. An important element of the GRACeFUL system is the kind of concept map that is used to define the policy problem to be solved. This is presented in Section 5, together with two examples resulting from case studies conducted within WP 2 under the direction of Sadie McEvoy (Deltares).

GRACeFUL concept maps are then formalised using a notation close to the functional programming language Haskell; the result is presented in the last section of the document.

2. Functional Programming

This section and the next are not self-contained. They serve as a reminder of the introduction to functional programming and DSLs presented during the GRACeFUL Kick-Off meeting in Delft, and to establish the notation used in the remaining sections. There are many introductions to functional programming, we prefer Bird (2014), Bird (1998), Bird and Wadler (1988), and Hutton (2007) (although sometimes billed as successive editions, there is surprisingly little overlap in the first three books, and they are all worth reading). For a canonical presentation of DSLs from a functional programming perspective, see Gibbons (2013).

2.1 The substitution model

In an imperative programming language, there are two kinds of syntactical phrases, conventionally called "expressions" and "statements". Typical expressions are arithmetical expressions (`x + 2, n^m`), boolean expressions (`a && b, not b`), etc. Typical statements are the assignment statement (`=` or `:=`), conditional statements (`if ... then ... else ...`) or loops (`for, while, ...`). Expressions are *evaluated*, and statements are *executed*. The results of evaluating expressions are values, the results of executing statements are changes in the state of the computation (and of the computer).

Functional programming does away with statements. Instead, we now have definitions, such as

```
fac n  =  if n == 0 then 1 else n * fac (n - 1)
```

The computational mechanism is *substitution* (see Abelson, Sussman, and Sussman (1996), Section 1.1.5): expressions are replaced by their definitions until no such substitution is possible. For example:

```
    fac 2
=
    if 2 == 0 then 1 else 2 * fac (2 - 1)
=
    2 * fac (2 - 1)
=
    2 * fac 1
=
    2 * if 1 == 0 then 1 else 1 * fac (1 - 1)
=
    2 * 1 * fac (1 - 1)
=
    2 * 1 * fac 0
=
    2 * 1 * if 0 == 0 then 1 else 0 * fac (0 - 1)
=
    2 * 1 * 1
=
    2
```

It is easy to understand the substitution model intuitively, but it is hard to give a rigorous description of it. For details, see Peyton Jones (1987), Section 2.2.

To tackle a programming problem in the imperative setting, we set up an explicit sequence of states of the computer; the solution is read off the last state in the sequence. In the functional programming context, we set up a (potentially very complex) expression with the help of definitions; the answer is given by evaluating the expression using the substitution model.

2.2 Types

Modern functional programming languages are strongly typed, and each valid expression must have a definite type which can be checked (or even inferred) at compile time. A new type is introduced by the keyword `data` and by rules that tell us how its elements are constructed. For example:

```
data Day  =  Monday | Tuesday | Wednesday |
             Thursday | Friday | Saturday | Sunday
```

introduces the set `Day` whose values are the labels `Monday`, `Tuesday`, ...

Infinite types are constructed by recursion:

```
data Nat  =  Z | S Nat
```

This definition tells us that a natural number is either just the label `Z`, or the label `S` preceding a natural number. In other words, it is the set `{Z, S Z, S (S Z), S (S (S Z)), ...}`. Mathematicians might prefer the representation `{Z, (S, Z), (S, (S, Z)), ...}`, which makes it clear that we are only dealing with tuples of labels. This representation is also close to the way the computer represents inductive types (Peyton Jones 1987, chap. 10).

New types can be constructed starting from existing ones, for example:

```
data Maybe a  =  Nothing  |  Just a
```

introduces, for any set `a`, the set `Maybe a = {Nothing} `join` {(Just, x) | x elem a}`. This is a standard datatype, allowing us to model partiality, and will be used below.

Types are one of the most important tools for writing specifications. In particular, most programs will have a *function type*

```
program :: Input -> Output
```

An expressive type system allows us to conclude many facts about `program` from its type.

An important information about a type is its *name*. The sets of values used to model credit and debit are the same, but whether a given value is to be interpreted in one way or in the other can make a great deal of difference. So we would like to say

```
Credit  =  Real
Debit   =  Real
```

but the symbol `=` can only be used with expressions. To avoid confusion, Haskell uses the keyword `type`:

```
type Credit  =  Real
type Debit   =  Real
```

We will make use of this ability to introduce new names for the same type below.

2.3 Type classes and polymorphism

Types allow us to organise the universe of values. It is often useful to organise the universe of types as well. For example, the only types that can be used to represent qualitative values are types that can be *linearly ordered*, i.e., for which we can tell, for any two values, whether they are equal, and if not, which is smaller and which is bigger. Natural numbers are like that, functions are not.

Different functional programming languages offer different mechanisms for organising types. Haskell has *type classes*. The type class `Ord`, which we'll use below, is introduced by

```
class Eq a => Ord a where
   (<), (<=), (>=), (>)  :: a -> a -> Bool
```

This tells us that a set `a` can only be ordered if its values can be compared for equality (`Eq a`). Further, in order for `a` to qualify as an *instance* of `Ord`, we must have defined the standard comparison functions for it.

Some functional expressions can be evaluated no matter what the type of the argument to which they are applied. For example, the identity function is defined by

```
id x  =  x
```

This equation defines, for every type `A`, a function `idA :: A -> A`. Thus, `id` itself is a template for creating such functions, one for each type. Such template-like functions are called *polymorphic*. In Haskell typings, lower-case variables denote universally quantified variables. For example:

```
id  ::  a -> a
```

is read: "for every type `a`, there exists a function `id :: a -> a`".

Type classes allow a more refined form of polymorphism. For example, the equation

```
max x y  =  if x < y then y else x
```

defines a function `max` for all the types whose values we can compare with `<`. Thus, a typing such as `a -> a -> a` would be too general, one such as `Int -> Int -> Int` too specific. The type class `Ord` allows us to *constrain* the polymorphism of `max`:

```
max  ::  Ord a => a -> a -> a
```

This is read: "for every type `a` that is a member of the type class `Ord`, `max` is a function of type `a -> a -> a`".

3. Domain-Specific Languages

Suppose we want to count the number of floating-point multiplications performed by a program. Since the built-in floating-point operations do not keep track of this information, we have to replace them with ones that do. A possible way of doing that is:

```
type CDouble = (Double, Int)

mkCDouble  ::  Double   ->  CDouble
add        ::  CDouble  ->  CDouble  ->  CDouble
mul        ::  CDouble  ->  CDouble  ->  CDouble

mkCDouble x            =  (x, 0)
add (x1, n1) (x2, n2)  =  (x1+x2, n1+n2)
mul (x1, n1) (x2, n2)  =  (x1*x2, n1+n2+1)
```

For simplicity, we have only shown the addition and multiplication operations. These new operations will act on "counted doubles", members of the type `CDouble`, represented by pairs that consist of the actual value and the number of multiplications required to produce it. Thus, we need a way of turning ordinary doubles into counted doubles, which is achieved by `mkCDouble`. We have separated the interface, consisting of the signatures of the operations, from the implementation, and we hope the example is straightforward.

Consider now a different problem, that of controlling the round-off errors in arithmetical operations. One way of achieving that is to work with *intervals*, rather than point values. Again, we will need to replace the standard operations with new ones, acting on "controlled values", i.e., intervals represented as pairs of numbers. Again, the following implementation should be straightforward:

```
type CDouble = (Double, Double)

mkCDouble  ::  Double   ->  CDouble
add        ::  CDouble  ->  CDouble  ->  CDouble
mul        ::  CDouble  ->  CDouble  ->  CDouble

mkCDouble x            =  (x, x)
add (xl, xu) (yl, yu)  =  (xl+yl, xu+yu)
mul (xl, xu) (yl, yu)  =  (lb, ub)
  where
  lb  =  minimum [xl * yl, xl * yu, xu * yl, xu * yu]
  ub  =  maximum [xl * yl, xl * yu, xu * yl, xu * yu]
```

You will notice that the interface is exactly the same as in the previous case. This points to a code duplication problem: we are doing the same work twice. Most software engineering methodologies, such as structural programming, object-oriented programming, or DSLs, can be understood as attempts at solving such problems.

In our case, the object-oriented solution would factor out the duplicated part by creating an abstract class or an interface. Then, new classes would be defined for "counted arithmetic" and "controlled arithmetic", which would inherit from the abstract class. Code which does not depend on the kind of arithmetic implemented "under the hood" would be written in terms of the abstract class, and thus be polymorphic with respect to the implementation. This suggests a possible refactoring in Haskell, by introducing a type class for the abstract arithmetic.

The DSL solution is somewhat different. It starts by recognising that the common part represents the *syntax* of arithmetical expressions, and makes this explicit by introducing a datatype for these expressions:

```
data CDouble  =  MkCDouble Double
              |  Add CDouble CDouble
              |  Mul CDouble CDouble
```

An element of `CDouble` is just a syntactical piece of data, such as

```
Add (MkCDouble 2.0) (Mul (MkCDouble 3.5) (MkCDouble (-1.2)))
```

or, in more familiar notation, `2.0 + 3.5 * (-1.2)`.

The various kinds of arithmetic give these syntactical elements different *semantics* or interpretations. There exists a *semantic domain* (`(Double, Int)` in the case of counted values, `(Double, Double)` in that of controlled values); the expressions will be evaluated to elements of this domain. In addition to the semantic domain, every interpretation will functions that evaluate the basic forms of arithmetical expressions: `MkCDouble x`, `Add x y`, and `Mul x y`. Thus, an interpretation is defined by a type and three functions:

```
type CAlg a  =  (Double -> a,
                 a -> a -> a,
                 a -> a -> a)

-- "Helper" functions:
mk (f, g, h)   =  f
a  (f, g, h)   =  g
m  (f, g, h)   =  h
```

In other words, an interpretation is given by an *algebra* (in the sense of *universal algebra*).

The *evaluation* of syntactical expressions in a specific algebra can now be implemented *generically*, as a *homomorphism*:

```
eval :: CDouble ->  CAlg a  -> a
eval (MkCDouble x)  alg     =  mk alg x
eval (Add x1 x2)    alg     =  a alg (eval x1 alg) (eval x2 alg)
eval (Mul x1 x2)    alg     =  m alg (eval x1 alg) (eval x2 alg)
```

The various kinds of arithmetic are then represented by different algebras:

```
countAlg  ::  CAlg (Double, Int)
countAlg   =  (mk, a, m) where
  mk x                  =  (x, 0)
  a (x1, n1) (x2, n2)   =  (x1+x2, n1+n2)
  m (x1, n1) (x2, n2)   =  (x1*x2, n1+n2+1)

controlAlg  ::  CAlg (Double, Double)
controlAlg   =  (mk, a, m) where
  mk x                  =  (x, x)
  a (xl, xu) (yl, yu)   =  (xl+yl, xu+yu)
  m (xl, xu) (yl, yu)   =  (lb, ub)
    where
    lb  =  minimum [xl * yl, xl * yu, xu * yl, xu * yu]
    ub  =  maximum [xl * yl, xl * yu, xu * yl, xu * yu]
```

In brief, the DSL approach starts by making explicit the syntax of the problem domain and the various semantical interpretations of this syntax. Almost always[1], the syntax is implemented as an inductive data type parametrised over the semantical domains, the semantics is given by an algebra, evaluation is a homomorphism of algebras, and the various applications are represented by particular algebras.

These algebras are equivalent to the classes of the object-oriented solution. Therefore, the DSL approach is somewhat "heavier", it needs a larger up-front investment. However, it is also more flexible: it makes a distinction between the applications that need only work with the *syntax* of expressions (such as parsers or pretty-printers), those that need to work only with the *semantics* (as in the case of our original tasks, counting multiplications or controlling round-off errors), and applications that need both (such as compilers and other forms of translation, or applications that need to mix various interpretations). Moreover, it offers new ways of understanding a problem domain, of formalising it, of dividing it into sub-problems, and of combining the resulting solutions.

4. Policy Analysis in the context of GRACeFUL

Policy analysis is a method for assisting policymakers in solving real-world policy problems, involving many stakeholders with conflicting interests, essential uncertainties, and many overlapping potential actions. Since its beginnings in the 50s in the world of operations research, policy analysis has been refined and given several different formulations. In one influential article, Warren Walker (2000) summarises the steps of policy analysis in the following way:

1. Identify the problem. Involves identifying the questions or issues involved, fixing the context within which the issues are to be analysed and the policies will have to function, clarifying constraints on possible courses of action, identifying the people who will be affected by the policy decision, discovering the major operative factors and deciding on the initial approach.

2. Identify the objectives of the new policy. Involves formulating the objectives that, if met, would solve the policy problem. Usually, there will be multiple conflicting objectives.

3. Decide on criteria (measures of performance and cost) with which to evaluate alternative policies. The criteria must be measurable (but note that qualitative values such as "big", "likely", etc. are acceptable). This step also identifies the ways in which the costs of the policies are to be estimated.

4. Select the alternative policies to be evaluated. A *policy* is defined "loosely" as "a set of actions taken to solve a problem", implying that alternative policies are defined in terms of more elementary *actions*. As many alternatives should be taken into account as possible, and the current policy must be included as "base case".

5. Analyse each alternative. Each alternative must be evaluated in every possible future context (scenario). Notice that the choice of scenarios which express uncertainty has

[1] For an analysis of the reach of this scheme, and the reasons for it, see Gibbons and Wu (2014).

to be part of the previous steps, even though this is not explicitly stated in Walker's article. This step involves commissioning external experts, collecting relevant data, setting up simulations, etc.

6. Compare the alternatives in terms of projected costs and effects. If none of the alternatives is good enough, then some or all of the steps 1-5 have to be re-examined.

7. Implement the chosen alternative.

8. Monitor and evaluate the results.

There are three major inter-related problems with the kind of procedure outlined here:

a. Perhaps the most important difficulty is the feedback explicitly introduced in step 6, after the execution of step 5. Step 5 is one of the longest and most expensive pre-implementation steps, since it involves external contractors, experts in the fields under consideration, and the setting up of potentially long-running computer simulations. And yet, at the end of this step, all the work done so far can be called into question and need to be redone. Depending on the amount of time that step 5 has taken (usually measured in months), this might not even be possible. Even if the problem, goals, criteria, or alternatives can be redefined, a new step 5 will still be quite expensive, both in terms of actual money and of time. And there is no guarantee that a second iteration will not result in a third, etc.

b. The second problem is that step 4 is error prone. As Walker underlines, "as many alternatives as stand any chance at all of being worthwhile" should be included. An alternative should not be excluded "merely because it seems impractical or runs contrary to past practice", and "personal judgements on such issues should be withheld". But this advice seems very hard to follow. Many people might not even be aware of excluding an alternative because it seems impractical and runs against their personal judgement: they will simply not consider it at all.

c. The third problem is that formulating many alternatives, being forced to consider even apparently "impractical" ones which "run contrary to past practice", increases considerably the costs of step 5 (recall that every alternative has to be considered in the context of *all* future scenarios). Moreover, the more results the stakeholders have to consider at the end of step 5, the more difficult it will be to present them in a perspicuous, useful manner.

The GRACeFUL system will offer feedback at every point of the problem definition in steps 1-3, for example by offering a library of criteria that could be used to assess chosen goals. But its main role will be to eliminate or alleviate these three problems, which all appear in steps 4-6. The key ideas are:

- Instead of relying on the stakeholders to provide alternative policies (sets of atomic actions), they will only be required to provide the actions that could go into building the alternatives. This is a much easier process, if only for cardinality reasons. The number of possible alternatives is always greater than the number of actions, since

each action can be considered an alternative policy in itself. In the worst case, however, in which the actions can be combined arbitrarily, there are, for `n` actions, on the order of `2^n` alternatives! More importantly, enumerating the actions is a more objective process than selecting alternatives against "personal judgements" or "past practices". The GRACeFUL system will maintain a database of atomic actions relevant to the policy problems at hand, in order to make this enumeration even easier.

- The question of finding the combinations of actions that fulfil the criteria and thus satisfy the desired goals, which was previously a matter of "test-and-rank", now becomes a matter of problem solving, in particular, a matter of *constraint programming* [2].

- In order to apply constraint programming, however, a suitable formulation of the problem is needed. This is achieved by building a qualitative or semi-qualitative model on the basis of the problem definition obtained from the stakeholders in steps 1-3 and of the list of actions. An essential feature of the model is that it will incorporate not just the constraints defined by the criteria, but also those introduced by relations between actions and factors. For example, an action requiring investment is constrained by available capital (an example of a factor).

- The qualitative model can then be used by the constraint programming layer to

  i. find alternative policies that can satisfy the goals,

  ii. identify inconsistencies in the problem definition,

  iii. present visualisations of possible future trajectories.

A key feature of the GRACeFUL system is that these outputs are accompanied by *explanations*. For example, the inconsistencies can be traced down to conflicting goals of stakeholders, resulting in conflicting criteria. As another example, the users can understand why certain actions are (or are not) selected.

Since the GRACeFUL system examines the space of *all* possible alternative policies, the problem of not including relevant alternatives is eliminated. Since the problem solving process of constraint programming is using the information supplied by the constraints, it can be much more effective then a simple "test-and-rank" process; thus, the problem of computational complexity is alleviated.

Additionally, constraint programming makes it possible to solve another problem that has plagued qualitative simulation systems: the explosion of possible future trajectories. By using the information provided by stakeholder goals and preferences, the constraint programming layer will be able to prune the graph of future trajectories, enabling a useful presentation of the simulation results.

The constraint programming layer will, however, not interact with the stakeholders directly. The information from and for the stakeholders will go through at least two

[2] Similarly to functional and logic programming, *constraint programming* is a declarative programming paradigm. The program is formulated in terms of a number of constraints, and the *constraint solver* identifies the solutions satisfying these constraints (there might be none, one, or many such solutions). For more information, see, e.g., Constraint.org.

layers of the GRACeFUL system: first, through the graphical visual interface, which will assist the stakeholders in building the problem definition, and which will present the results of the qualitative simulations, and second, the DSLs which will translate between these interfaces and the constraint programming layer.

5. GRACeFUL Concept Maps

We envisage the GRACeFUL process to follow broadly the de Haan and de Heer framework for solving complex problems (de Haan and Heer 2015). In this process, the result corresponding to steps 1-4 of policy analysis are presented in the form of a *GRACeFUL concept map*. This concept map contains all the elements of a policy problem definition discussed above: goals, criteria for assessing the achievement of those goals, a description of the system as a causal loop diagram involving factors and criteria, and the competing alternatives.

The steps leading to a GRACeFUL concept map are to be performed by the stakeholders, assisted by an expert controlling the GRACeFUL system. For example, the goals elicited from a stakeholder or groups of stakeholder, are refined to measurable criteria of achievement of those goals. This is achieved by discussions, brain storming sessions, etc., but also by interacting with the GRACeFUL system, which provides a growing library of criteria that are relevant for a given goal.

Similarly, the description of the system is elicited from stakeholders with help from the GRACeFUL system, which contains a library of relevant factors, relating them to the concepts under discussion. Here, 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*). In mathematical system theory, a collection of factors correspond to the *state* of the system. Similarly, *external factors* correspond to *inputs* to the system. As such, factors are assumed to be associated with measurable (if only in qualitative terms) 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*, generally assumed to be deterministic but with uncertain consequences.

The interaction between the factors, and between the factors and the criteria, represents a first description of the system (no alternatives are present, or only the baseline policy is assumed to be active). This description is in the form of a *causal loop diagram*. A causal loop diagram is a form of *directed, simple, labelled graph*, i.e.:

a. a link between two nodes has a well-defined source and a target (it is *directed*); intuitively, the link represents a causal relation between the source (cause) and the target (effect);

b. there can be at most one connection from one node to another (the graph is *simple*): either a causal relation exists, or it does not. Note, however, that since the graph is directed, there may be an inverse connection from the “effect” to the “cause”, as in feedback loops;

c. the connections are either *positive* (an increase in the value of the source causes an increase in the value of the target), or *negative* (an increase causes a decrease), hence the graph is *labelled*.

It can be seen from the last point that nodes *must* represent values that can be partially ordered. This rules out nodes that represent, say, stakeholders.

As an example, two such causal loop diagrams, produced in the student sessions organised as part of the CRUD case study, can be found in the Appendix.

The GRACeFUL system can assist in building the causal loop diagram, for example by suggesting connections, or by pointing out inconsistencies.

At this stage, if we can attribute qualitative values to the various nodes, the causal loop diagram describes a system that can make the object of a qualitative simulation. The criteria play the roles of constraints: the qualitative trajectories that lead outside the criteria can be pruned by the constraint programming system.

The causal loop diagram is enhanced to a GRACeFUL concept map by the addition of extra nodes and non-causal links. The extra nodes represent actions, goals, alternatives, and stakeholders.

Like all previous elements, the actions are also obtained from the stakeholders. As explained in the previous section, actions are the components of alternatives, among which will be found the policy solutions we are seeking. Alternatives are, according to the GRACeFUL Glossary of Terms, "action or combination of actions that influence the values of criteria". They correspond to Walker's "alternative policies", where a *policy* is defined "loosely" as "a set of actions taken to solve a problem". As in the case of goal definition, the GRACeFUL system can suggest such "atomic actions" that are useful to the goals in the context of the problem at hand.

The role of actions in the GRACeFUL concept maps is to give initial values to the factors and criteria they influence. They do not link via causal loops to these factors, since actions are not of the requisite type (they do not represent values that can increase or decrease). However, they will usually have non-causal links, representing constraining relationships to the factors.

In the other cases of extra nodes (goals, alternatives, and stakeholders), the links represent primarily "belongs to" relationships: there are links from stakeholders to goals, indicating which stakeholders have declared which goals; there are links from goals to the criteria by which they are assessed; finally, there are links between alternatives and the factors and criteria that they (i.e., their constituent atomic actions) influence.

Figure 1 (drawn by Sadie McEvoy (Deltares)) shows a graphical representation of the structure of a GRACeFUL concept map:

6. Formal Description of GRACeFUL Concept Maps Elements

We now formalise the structural elements of the GRACeFUL concept maps presented in the previous section. These elements are:

- goals

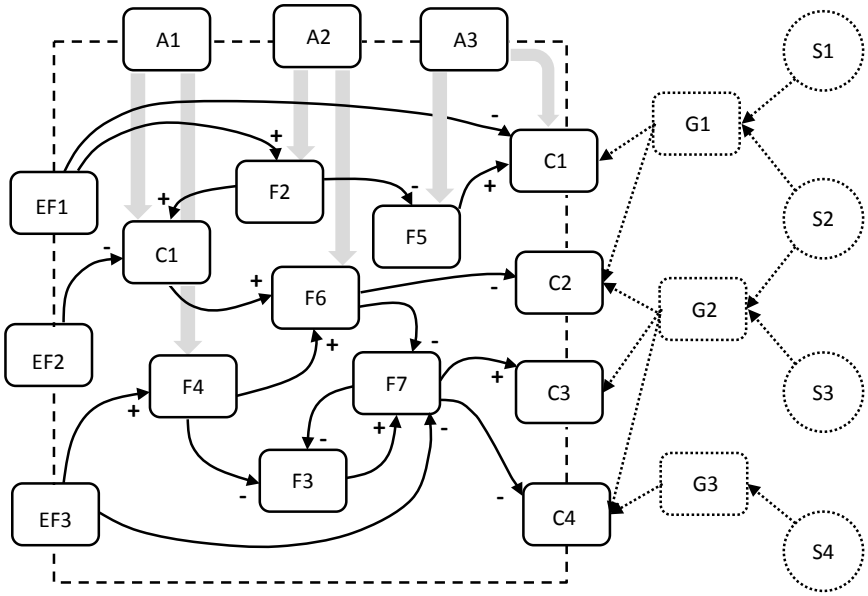

Figure 1: The structure of a GRACeFUL concept map

- criteria
- concepts
- factors
- connections
- stakeholders
- actions
- alternatives
- constraints

Let us start with concepts and factors, which are, in a sense, the simplest elements of the GRACeFUL concept maps. Concepts are high-level descriptions of the components of a system, and they are refined into factors, which, as we have seen, determine the state of the system. The most important feature of a factor is that it is *measurable*: it can take one value from a set of values. These values must be ordered, otherwise we do not have a notion of increasing or decreasing, as required by the use of factors in causal loop diagrams.

Different factors can be associated with the same type of value. For example, there might be factors that represent temperatures in different parts of the system, such as "water temperature" and "air temperature". Even if the values of water temperature and air temperature happen to be equal, the two factors are still different. That means that a defining component of a factor is its identifier, its name. This suggests formalising the

notion of a factor as a pair consisting of a name and a value. Such a formalisation makes explicit the part that stays the same, the name, in addition to the part that changes, the value. This is a common way of modelling states in functional programming.

On the other hand, we can certainly create a causal loop diagram without knowing the value associated to the factors. None of the factors in the causal loops in Figures 2 and 3 have values associated with them. Therefore, we must formalise factors as pairs consisting of a name and *potentially* of a value of an ordered type:

```
data (Ord value) => Factor value  =  MkFactor Name (Maybe value)
```

For the moment, we can take the names of factors to be just strings:

```
type Name  =  String
```

Since we aim for a system based on qualitative reasoning, most of the types of value we shall use are going to be finite types, consisting of an enumeration of the symbols for possible landmark values and for the intervals delimited by the landmark values. For example, a very common type of qualitative values is:

```
data QualitativeValue        =  Negative | Zero | Positive
                                deriving (Eq, Ord)
```

Here we have one landmark value, `0`, denoted by `Zero`, and two intervals: `(-inf, 0)`, denoted `Negative`, and `(0, inf)`, denoted `Positive`.

Continuing the example of temperature factors above, we can have:

```
t1, t2                 ::  Factor QualitativeValue
t1                      =  MkFactor "Water Temperature" Nothing
t2                      =  MkFactor "Air Temperature" (Just Positive)
```

Here, `t1` represents the factor "Water Temperature", which does not currently have a defined value, and `t2` represents the factor "Air Temperature", whose current value is `Positive`.

Factors are introduced via the refinement of *concepts*. However, once the factors have been defined, concepts play no further role: in fact, they do not even appear in the current versions of the GRACeFUL concept maps. This is not surprising: concepts are high-level descriptions of parts of systems. In mathematical systems theory, (parts of) systems are *defined* by their states, and so concepts are defined by the factors that they are "refined" into in the course of stakeholder interactions. Moreover, it is not clear that it is useful to distinguish between two concepts that are refined in identical sets of factors. Still, at this stage we think it useful to keep a record of the concept name, for example for the purpose of looking it up in a database of pre-defined (or pre-refined) concepts (thus creating re-usable building blocks). Formally, we have:

```
data Concept value        =  MkConcept Name [Factor value]
```

For simplicity, we have assumed that all the factors defining a concept have the same sets of possible values, in order to overcome some limitations of our Haskell notation.

We now move on to the formalisation of goals and criteria. Criteria are similar to factors: they are part of the causal loop diagrams, hence have values in an ordered set, names, and maybe a definite value. Additionally, however, criteria are associated to *predicates*, which tell us whether a given value satisfies a concrete criterion or not:

```
type Predicate a    =  (a -> Bool)
```

Therefore, we can formalise criteria as

```
data  (Ord value) => Criterion value  =
      MkCriterion Name (Maybe value) (Predicate value)
```

(Alternatively, we could have extended the type `Factor value`.)

For example:

```
profit                  ::  Criterion QualitativeValue
profit                   =  MkCriterion "Profit" (Just Zero) isPositive
                            where
                            isPositive x = x > Zero
```

introduces a criterion formalising "profits should be (strictly) positive", where the current value, `Zero`, does not satisfy the criterion.

Criteria result from the refinement of goals, just as factors result from the refinement of concepts. It is no surprise, therefore, that the formalisation of goals will be just like that of concepts:

```
data Goal value        =  MkGoal Name [Criterion value]
```

At this point, it is useful to briefly discuss an essential difference between *formalisation* and *implementation*. The implementation of goals in the GRACeFUL system will very likely be much more complex than the formalisation presented here. We mentioned the possibility of looking up goals in a library of pre-defined goals, in order to assist the stakeholders in their search for adequate criteria. We might have various kinds of pre-defined goals, which might themselves be related in complicated ways, for example in order to assess inconsistencies between them. We could use *ontologies* for the representation of goals, (and, for that matter, also concepts, factors, criteria, etc.), linking to Haskell RDF libraries, etc. Therefore, the implementation of `Goal` will likely result in a more complex datatype, such as:

```
data Goal            =  GoalRDF Triple
                     |  G1 PreDefinedG1
                     |  G2 PreDefinedG2
                     |  ...
                     |  FreeFormGoal String
```

Here, we are only concerned with *understanding* as clearly as possible the notion of goal, factor, etc. as pertaining to their use in the GRACeFUL concept maps. We do not search for the most efficient representation in computational terms. We are also stripping away anything that does not belong to the essence of these notions *as used in the concept maps*. One reason for this is that the formalisation undertaken here will be part of the *specifications* to the implementations carried out in the course of the project. The simpler the specification, the likelier it is that it will be correctly understood and implemented. Moreover, since the specification is that against which we measure the correctness of the implementation, it is important that we do not overly constrain this implementation, for example by deciding that goals are going to be represented via RDF triples. At this stage, such a decision would be largely arbitrary.

Now that we have formalised factors and criteria, the nodes of the causal loop diagrams, we can formalise these diagrams themselves. As mentioned above, these are instances of simple, directed graphs with labelled connections. The following datatype represents a simple directed graph, parametrised on the types of nodes and connections:

```
data Graph node conn  =  MkGraph ((node, node) -> Maybe conn)
```

In (other) words, a graph is given by a function which, to every ordered pair of nodes (a source and a target), associates either no connection, or just one connection.

The connections in a causal loop diagram are labelled either with a plus, or a minus:

```
data Connection       =  Plus | Minus
```

The nodes of our causal loop diagrams are either factors or criteria:

```
type Node value       =  Either (Criterion value) (Factor value)
```

Therefore, we can formalise a causal loop as

```
type CausalLoop value  =  Graph (Node value) Connection
```

GRACeFUL concept maps represent an enrichment of the causal loops with nodes representing the possible (atomic) actions and with nodes and links representing constraints between actions and factors.

Actions can influence the nodes of a causal loop diagram, by changing the value associated with that node. We can formalise such an influence as a function that can return a new value for a given node (or `Nothing` in the case of "no influence").

```
type Influence value  =  Node value -> Maybe value
```

An action is then defined as a named influence:

```
data Action value     =  MkAction Name (Influence value)
```

An alternative policy is, similar to a goal or a concept, a named set of actions:

```
data Alternative  value    =  MkAlternative Name [Action value]
```

One of the major outputs of the GRACeFUL system will be a list of alternatives which meet the goals and the constraints; we could define

```
type GRACeFUL_Solution value  =  [Alternative value]
```

As we have seen in the previous section, actions can be constrained by factors (e.g., investments are constrained by the available capital). This is common in systems theory, where the set of controls that can be used when the system is in a given state depend on that state.

Similarly, there can be constraint relations between the factors themselves. These constraints can either be associated with concepts, as in the case in which the constraints relate factors stemming from the refinement of the same concept, or with interactions between concepts.

The constraint relations are, in general, non-causal, and are thus genuine enhancements over a causal loop diagram. They can sometimes be represented by directed links, for example, "source value may not exceed target value", but in general they will require the introduction of a new kind of node. It is easy to see that a constraint such as "only one of these three factors may be negative" is not expressible as a link between two factors.

Constraints that can be expressed as links can be formalised as

```
data ConstraintLink value  =  MkConstraintLink (Predicate (value, value))
```

This corresponds to the intuition that we can model a constraint as a link only if it is a binary relation.

For example:

```
doesNotExceed              ::  Ord value => ConstraintLink value
doesNotExceed               =  MkConstraintLink gt
                               where
                               gt (v1, v2) = v1 <= v2
```

Constraints between more than two factors cannot be modelled as links. For these, we need to introduce nodes which collect the values of several factors or actions, and check if these values satisfy the constraint, operating in a similar way to the criteria. As opposed to the criteria, the value associated to a constraint is either true, or false.

In a binary relation such as `doesNotExceed`, we can distinguish between the two elements ("what does not exceed what?") based on the source and target order of a directed link. This is not possible in the case of a constraint represented by a node. Therefore, the various roles of the related values must be indicated by the links. As an example, consider the constraint given by

```
inBetweenPred              ::  Ord value => Predicate (value, value, value)
inBetweenPred (v1, v2, v3) =   v1 <= v2 && v2 <= v3
```

This constraint will be represented by a node. There will be three links into this node, representing the three arguments. These links will need to be labelled, in order for the constraint node to be able to tell which one of the arguments is associated to each link. We can introduce a type for these labels:

```
data Role                  =  V1 | V2 | V3
                              deriving Eq
```

A role link will connect nodes to a constraint node, specifying their role:

```
data RoleLink role         =  MkRoleLink role
```

Finally, a constraint node will be represented by a predicate on a list of role-value pairs:

```
data ConstraintNode role val  =  MkConstraintNode (Predicate [(role, val)])
```

For example, we can define the node associated to the `inBetweenPred` above by

```
inBetweenC                 ::  Ord value => ConstraintNode Role value
inBetweenC                 =   MkConstraintNode p
                               where
                               p as = inBetweenPred (toTriple as)
```

(we omit the implementation of `toTriple`).

There is one remaining element of the GRACeFUL concept map as presented above, namely the nodes representing the stakeholders. These are meant to relate the stakeholders to their goals, thus we can formalise them as:

```
data Stakeholder value  =  MkStakeholder Name [Goal value]
```

The nodes of the GRACeFUL concept maps are therefore:

```
data ConceptMapNode role value  =  CN (Node value)
                                |  AN (Action value)
                                |  GN (Goal value)
                                |  CO (ConstraintNode role value)
                                |  SN (Stakeholder value)
```

Besides the causal connections in the causal loop diagram, we also have links from actions to the factors and criteria they influence, from the criteria to the goals they assess, from stakeholders to the goals they "own", constraint links, and role links.

```
data ConceptMapConnection role value  =  CC Connection
                                       |  ActionToNode
                                       |  CriterionToGoal
                                       |  CL (ConstraintLink value)
                                       |  RL (RoleLink role)
                                       |  StakeholderToGoal
```

We can now define the GRACeFUL concept map as a simple, directed graph:

```
type ConceptMap role value  =  Graph (ConceptMapNode role value)
                                     (ConceptMapConnection role value)
```

A concept map `cm = MkGraph f` is *valid* if:

- `f (cn1, cn2) = ActionToNode` iff `cn1` is of the form `AN (MkAction name infl)`, `cn2` is of the form `CN node`, and `infl node` is not `Nothing`.
- `f (cn1, cn2) = CriterionToGoal` iff `cn1` is of the form `CN (Left crit)`, `c2` is of the form `GN (MkGoal name crits)`, and `crit` is an element of `crits`.
- `f (cn1, cn2) = StakeholderToGoal` iff `cn1` is of the form `SN (MkStakeholder sname goals)`, `cn2` is of the form `GN goal`, and `goal` is an element of `goals`.
- `f (cn1, cn2) = CC plusOrMinus` only if `cn1` is of the form `CN node1` and `cn2` is of the form `CN node2`.
- `f (cn1, cn2) = RL r` only if `cn1` is a factor node or an action node, and `cn2` is a constraint node.
- `f (cn1, cn2) = CL cl` only if `cn1` and `cn2` are factors or actions.

Acknowledgements

The authors thank Nicola Botta and Timm Zwickel of the Potsdam Institute for Climate Impact Research for the many discussions which have contributed to the production and improvement of this document.

References

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press.

Bird, Richard. 1998. *Introduction to Functional Programming Using Haskell*. Vol. 2. Prentice Hall Europe London.

———. 2014. *Thinking Functionally with Haskell*. Cambridge University Press.

Bird, Richard, and Philip Wadler. 1988. *Introduction to Functional Programming, 1988*. Prentice-Hall, Englewood Cliffs, NJ.

de Haan, A., and P. de Heer. 2015. *Solving Complex Problems*. Eleven International Publishing.

Gibbons, Jeremy. 2013. "Functional Programming for Domain-Specific Languages." In *Central European Functional Programming - Summer School on Domain-Specific Languages*, edited by Viktória Zsók, Zoltán Horváth, and Lehel Csató, 8606:1–28. LNCS. Springer. doi:10.1007/978-3-319-15940-9_1. http://link.springer.com/chapter/10.1007/978-3-319-15940-9_1.

Gibbons, Jeremy, and Nicolas Wu. 2014. "Folding Domain-Specific Languages: Deep and Shallow Embeddings." *International Conference on Functional Programming* (September). http://www.cs.ox.ac.uk/jeremy.gibbons/publications/embedding.pdf.

Hutton, Graham. 2007. *Programming in Haskell*. Cambridge University Press.

Peyton Jones, Simon L. 1987. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc.

Walker, Warren E. 2000. "Policy Analysis: a Systematic Approach to Supporting Policymaking in the Public Sector." *Journal of Multi-Criteria Decision Analysis* 9 (1-3): 11–27.

Appendix: Examples of causal loop diagrams

The following diagrams were produced in student group model building sessions organised as part of the CRUD case study. The objective of the sessions was to define policies for a climate resilient Stadspolders (a district of Dordrecht, Netherlands).

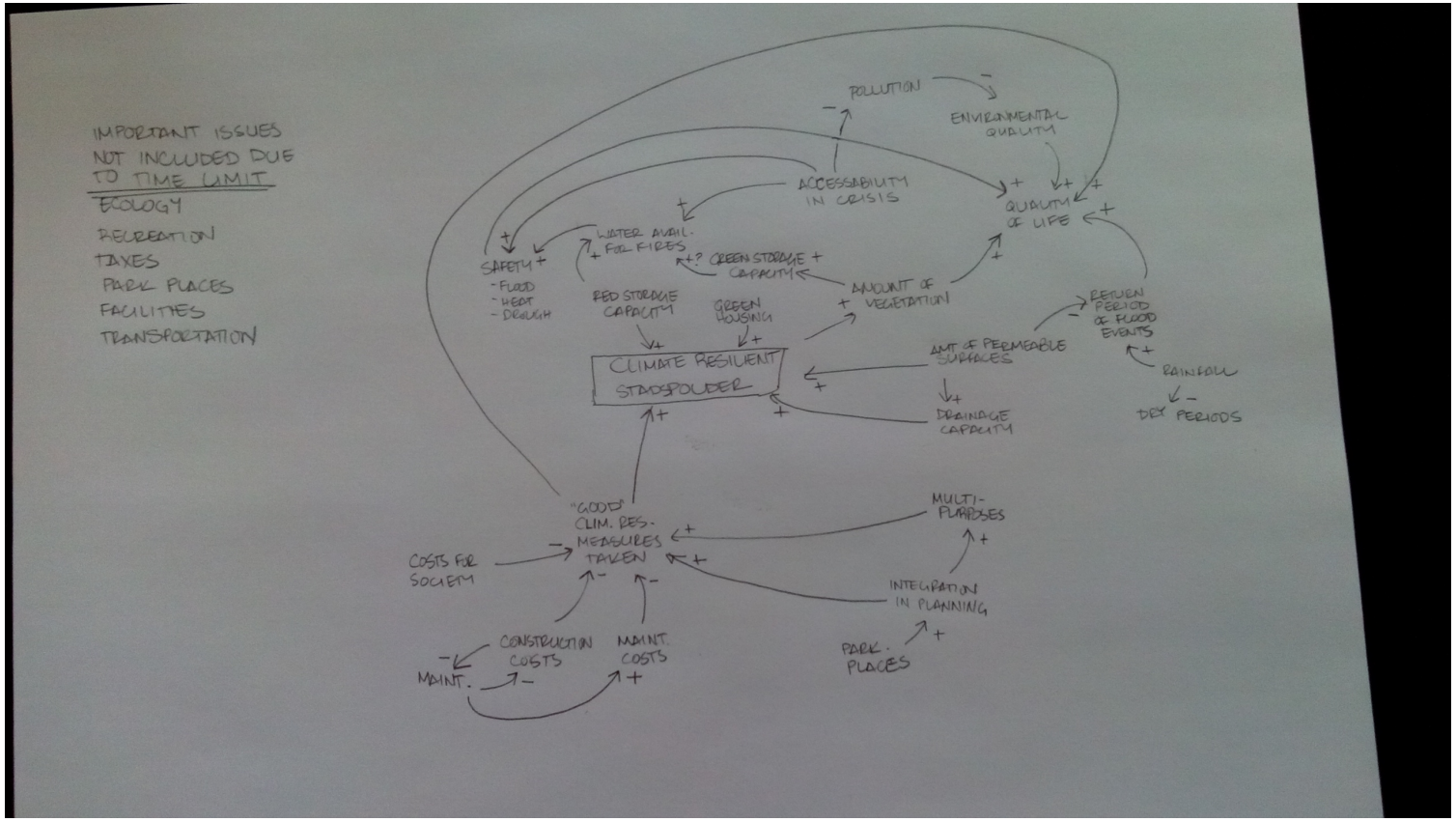

Figure 2: Causal loop diagram 1

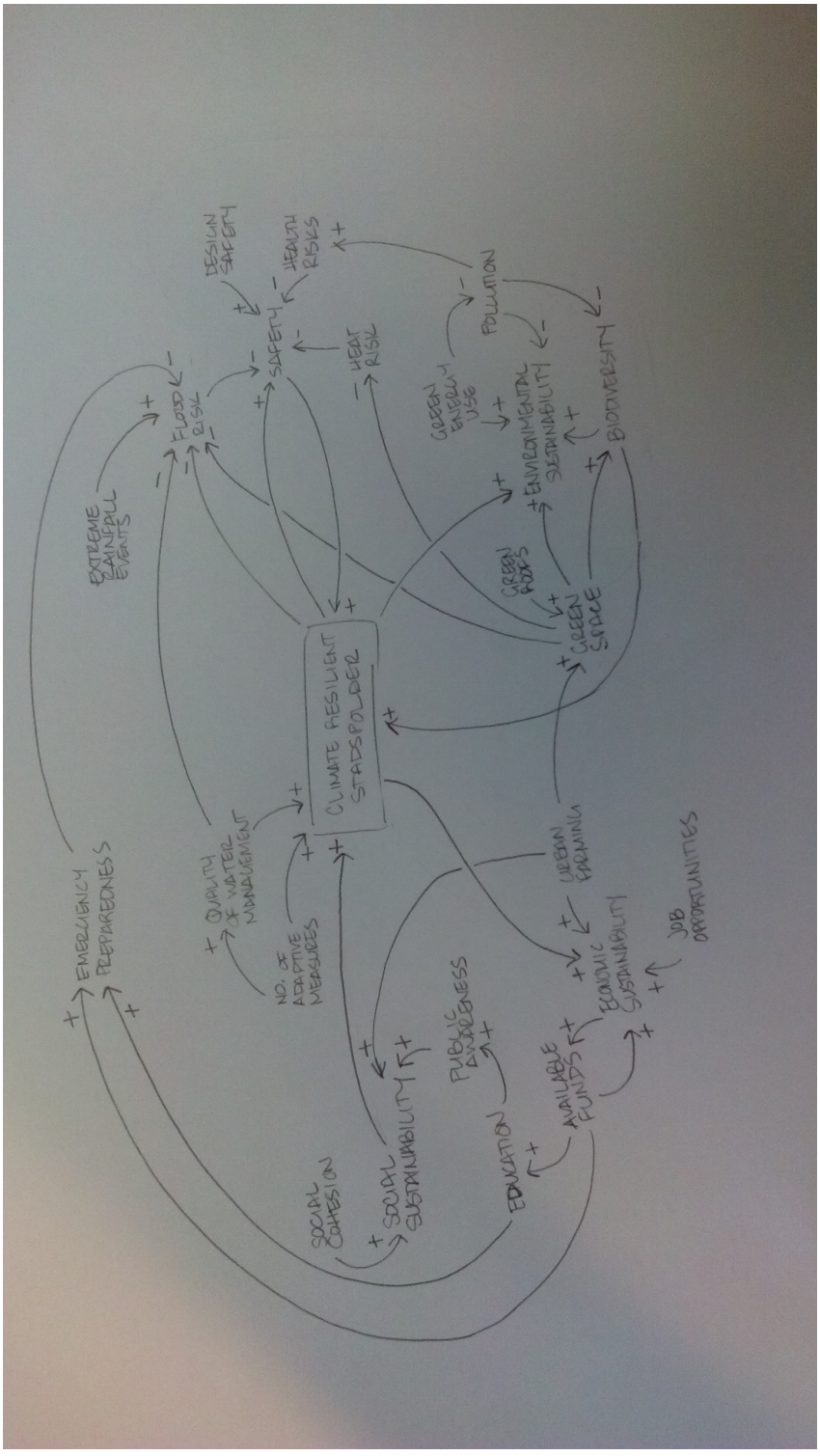

Figure 3: Causal loop diagram 2