GRACeFUL

Global systems Rapid Assessment tools
through Constraint FUnctional Languages

FETPROACT-1-2014 Grant Nº 640954

D4.3: Concept Maps to System Dynamics

Translation of GRACeFUL concept maps
to the Constraint Functional Programming layer

Lead Participant:	Chalmers (WP leader: P. Jansson)
Partners Contributing:	KU Leuven
Dissemination Level:	PU
Document Version:	Draft
Date of Submission:	2017-07-25
Due Date of Delivery:	2017-07-31

Translation of GRACeFUL concept maps to the Constraint Functional Programming layer

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

Abstract

This third deliverable (D4.3) of work package 4 presents the translation of GRACeFUL concept maps (expressed as GRACe programs) to the Constraint Functional Programming (CFP) layer. GRACe programs model the system dynamics of GRACeFUL Concept Maps and the CFP layer is used to compute model solutions. This report builds on the description of GRACe in "D4.2: A Domain Specific Language (DSL) for GRACeFUL Concept Maps" (delivered in project month 24) and the third release of the CFP layer "haskelzinc". (The first release was described in "D5.1: Domain-Specific Language for the Constraint Functional Programming Platform" and the latest version is available from the Haskell package repository Hackage.) The work leading up to this deliverable is within Task 4.4 "implement a middleware for connecting the DSL to the CFP layer" and the full source code of the implementation is available on GitHub.

Contents

1 Introduction

This report describes the third deliverable (D4.3) of work package 4 of the GRACeFUL project. The software we present in this document can be downloaded from 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. It is common practice to simulate a model described by a GCM, however, this process is unfortunately both time consuming and expensive. The GRACeFUL project tries to alleviate this problem by expressing a GCM as a *constraint program*, which should reduce the analysis time considerably.

A GCM is created by stakeholders in a so-called Group Model Building (GMB) session using a visual editor. Once the GCM is complete, the visual editor submits a representation[2] of the GCM to our DSL layer, described in deliverable D4.2 [1], which in turn passes on the GCM to the Constraint Program (CP) layer. The DSL can be regarded as an intermediate layer between the visual editor and the CP layer, which increases modularity, simplifies the translation, and reduces the dependency on a particular constraint solver.

The main challenge for work package 4 is to create a DSL that is expressive enough to model GCMs as envisaged by the stakeholders, while still being able to translate to a constraint program. A constraint program is a collection of (unknown) variables and constraints, for which a constraint solver tries to find a solution (values for the unknown variables) that satisfies as many of those constraints as possible.

In this document we explain how we translate a program in terms of our DSL to a constraint program using the CP interface provided by work package 5, see deliverable D4.2 [1]. In Section 2 we give a short summary of our DSL and show how the main concepts, such as ports and parameters, are expressed in terms of a constraint program. We continue with an explanation of the software architecture in Section 3. We end this report with some concluding remarks in Section 4.

We have also written a tutorial on how to create GRACe components as a separate document available here: https://github.com/GRACeFUL-project/DSL-WP/blob/master/tutorial/pdf/GRACeTutorial.pdf.

[1]https://github.com/GRACeFUL-project/ in the `GRACe` repository.
[2]We use JavaScript Object Notation (JSON) as an exchange format

2 A GRACe-ful translation

This section covers how a model written in our DSL, called GRACe, is translated to a constraint program. This translation is done in several stages. First we translate a GRACe program to haskelzinc [2], which is developed in work package 5. In the next stage the program is compiled from the haskelzinc intermediate representation to a particular constraint programming language, which in our case is MiniZinc[3]. The results from the constraint solver are parsed by our DSL and returned to the user.

2.1 A small DSL program

We start with a small example to explain the syntax and elements of a GRACe program. A central element of a GRACe program is a *component*. In fact, a GRACe program is a collection of components that can be connected to each other. The general structure of a GRACe component starts with the declaration of its ports and parameters, followed by constraints on those, and ends with exposing the ports and parameters to other components. The following example defines a component which represents a pump with a given capacity (`cap`):

```haskell
pump :: Int -> GCM (Port Int, Port Int, Param Int)
pump cap = do
  inPort   <- createPort
  outPort  <- createPort
  capParam <- createParam capacity

  constrain $ do
    inflow   <- value inPort
    outflow  <- value outPort
    capacity <- value cap

    assert $ inflow `inRange` (0, capacity)
    assert $ inflow === outflow

  return (inPort, outPort, capParam)
```

The pump component is implemented as a Haskell function that takes an integer parameter (`cap`) and returns a GRACe component with two ports, representing the inflow and outflow of the pump, and a parameter for limiting the pump's capacity. The type signature on the first line in the above code snippet reflects this.

The ports and parameter can be connected to other components, so that they can interact with other components in a larger model. A port is created with a call to the `createPort` function, and is of a particular type, such as the integer port `flow` defined above. A GRACe parameter is created with the `createParam` function and can be regarded as a port with an initial value. It is important to distinguish between normal (Haskell)

[3] http://www.minizinc.org/

parameters, such as `cap`, and parameters defined in the GRACe DSL, such as `capacity`. Haskell parameters (function arguments) are used to describe families of components at model building time. A model is ready for constraint solving when all these arguments (Haskell parameters) have been supplied. Parameters defined in the GRACe DSL can be connected to other components, which may influence their values at constraint solving time.

The ports and parameters are the interface of a component and we use them to interact with other components. An important feature of our DSL is that we can put *constraints* on those ports and parameters. We use these constraints to model the system dynamics of a GRACeFUL Concept Map. The pump component, for example, constrains the flow through the pump to be positive and smaller than its maximum capacity, and ensures that the inflow is equal to the outflow. Using the `constrain` function we can embed constraints in the component. In fact, the `constrain` function takes a representation of a constraint program as argument. A constraint program can query the value of ports and parameters using the `value` function. These values can then be used to express constraints using the `assert` function. The `assert` function support many different constraint expressions.

We can group components, such as the `pump` component, in a library. Using such a library we can create GRACe DSL programs, which we can analyse with a constraint solver. Before we continue and define an example GRACe program, let us first create a second component. The following simple component just exposes a port with a constant volume of rainfall:

```haskell
rainfall :: Int -> GCM (Port Int)
rainfall volume = do
  volumePort <- createPort
  set volumePort volume
  return volumePort
```

The `set` primitive is used to constrain the value of a port to a specific value, in this case to the `volume` parameter.

Using the `pump` and `rainfall` components we can create a, somewhat contrived, GCM program:

```haskell
main :: IO ()
main = putStr =<< runGCM prog
  where
    prog :: GCM ()
    prog = do
      (inflow, outflow, _) <- pump 100  -- create a pump
      rain <- rainfall 10               -- let it rain
      link rain inflow                  -- connect rain to pump
      output outflow "pump outflow"
```

The above example creates a DSL program called `prog`, which instantiates two components: a pump with a particular capacity, and some rainfall. The `link` DSL function on line 8 takes two arguments, which can be either ports or parameters, and connects them to each other. We explain the semantics of this connection later in this section. The `output` primitive can be used to include the value of a port (or parameter) to the output returned by the constraint solver. The solver is run by calling the `runGCM` function and returns a string as result, which we print on the command line.

2.2 Translating to MiniZinc

Before running the constraint solver, the DSL program is translated to a corresponding MiniZinc program. The MiniZinc representation is saved in a temporary file, which is given to the MiniZinc constraint solver. The example above is translated to the following MiniZinc code:

```
1   var -10000000..10000000: v3;
2   var -10000000..10000000: v2;
3   var -10000000..10000000: v1;
4   var -10000000..10000000: v0;
5
6   constraint ((v2) == (100));
7   constraint (((0) <= (v0)) /\ ((v0) <= (v2)));
8   constraint ((v0) == (v1));
9   constraint ((v3) == (10));
10  constraint ((v3) == (v0));
11
12  solve satisfy;
13
14  output ["{\"pump outflow\" : \(v1)}"];
```

The generated MiniZinc constraint program consists roughly of four parts: some variable declarations, constraints on those variables, a strategy how to solve the constraints, and the desired output.

MiniZinc variables are declared with the keyword `var` followed by its domain (in this case a range from `-10000000` to `10000000`) and then a unique name. These variables are generated during the translation from a GRACe program to a constraint program. Each port and parameter maps to a particular constraint variable. For example, the `capacity` parameter of the `pump` component is mapped to the constraint variable `v2`. In the example program we instantiated the `pump` component with a capacity of 100 units, which is used to initialise the `capacity` parameter. This initialisation gets translated to a constraint that equals the value of the constraint variable corresponding to the `capacity` parameter to the given argument (`100`), as we can see on line 6 in the generated MiniZinc program.

We can restrict the values of ports and parameters by embedding constraints in a component. For example, we constrained the value of the `inflow` port of the `pump` component to be in the range from 0 to the maximum capacity. These embedded constraints are translated to constraints in the generated MiniZinc program. On line 7 in the generated

MiniZinc program we have a constraint that limits the value of `v0`, which is the corresponding constraint variable to the `inPort` port of the `pump` component, to be in the given range.

When we link two ports (or parameters), such as on line 8 in the example `prog` DSL program, we introduce a constraint in the generated MiniZinc program that states that the values of the corresponding constraint values are equal to each other. For example, we linked the rainfall to the input of the `pump` component, which got translated to the constraint on line 10 in the MiniZinc program. Note that the generated constraints have some extra pairs of parentheses to make sure we generate valid programs with the correct precedence. Some of these are superfluous and we may leave them out in the future.

We instruct the constraint solver to search for a solution that fulfils all such constraints with the `solve satisfy` instruction. The last part of the generated MiniZinc program, on line 14, instructs the solver how to generate the output. The output instructions use corresponding constraint variables instead of port names. The output returned by the constraint solver is parsed by the DSL and returned by the `runGCM` function.

3 Software architecture

The software stack of the GRACeFUL project consists of a visual editor frontend, a network layer, a GCM component library, a DSL called GRACe, a middleware called haskelzinc, and a choice of external constraint solver. In this section we briefly describe the different layers to explain the context.

Visual Editor

The top layer of the software stack is the visual editor. It provides a graphical user interface where the user can build GCMs as a graphical map from available components. The visual editor is implemented in the untyped functional language JavaScript, using the Data Driven Documents library (D3.js).

The first running prototype (used for the year 2 review demo) was developed in the GRACeFUL-project GitHub repository VisualEditor and the most recent version in GRACeFULEditor.

Communication with visual editor

Communication between the visual editor and the GRACe layer takes place through a RESTful Web service written in Haskell. JSON objects are sent between the two layers via requests to this service and handled on both ends.

The GRACe web service is available as the `RestAPI` executable in the GRACe repository.

GCM component libraries

The visual interface allows the user to access a chosen library of GCM components. These components are written in GRACe, and each component has a corresponding JSON interface which is sent to the visual editor when the user requests the library in question.

GRACe

GRACe is a domain specific language embedded in Haskell. It is used to express GRACeFUL concept maps (GCMs) and GCM library components. GRACe programs representing GCMs are compiled to haskelzinc constraint programs, and the resulting solutions are passed back to GRACe.

Haskelzinc

Haskelzinc is a Haskell interface to the MiniZinc constraint programming language. It provides

- a Haskell abstract syntax tree for the MiniZinc language, with which one can represent MiniZinc models in Haskell
- a human-friendly DSL for building MiniZinc model representations
- a pretty printer to print the representation of a MiniZinc model in MiniZinc
- a parser that returns a representation of the solutions obtained by running the MiniZinc model
- a set of functions useful for building a custom FlatZinc solutions parser. An additional module gives the possibility to directly obtain the solutions of a MiniZinc finite domain model.

The latest version of haskelzinc (currently 0.3) is available from https://hackage.haskell.org/package/haskelzinc.

MiniZinc

MiniZinc is a constraint programming language in which constraint satisfaction and optimization problems can be modeled independently of constraint solvers.

MiniZinc models are compiled into FlatZinc, a low-level solver input language. There are a variety of solvers that can be used to solve problems stated in FlatZinc.

Information on how to install and use MiniZinc is available at www.minizinc.org.

4 Conclusion

We have designed and implemented a translation from the DSL called GRACe (describing GRACeFUL concept maps) to the underlying Constraint Functional Programming layer. We have presented code examples, how they are translated, and the software architecture. The source code is available on GitHub and installation instructions are available in the appendix.

The next actions in work package 4 is

- 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 GRACe.

In parallel, the translation, including the source and target DSLs will gradually evolve to handle more and more of the requirements extractable from GMB sessions with stakeholders.

A Installation and usage

In this section we outline the process of installing the GRACe software and its required dependencies. We start with an overview of the software dependencies required for developing GRACe programs (Appendix A.1). Following this, we provide installation instructions for a platform-independent package built on the Docker platform, intended for users who only wish to execute a pre-existing example (Appendix A.2).

A.1 Software dependencies

The GRACe language is an embedded domain-specific language implemented in the Haskell programming language, and uses the solver tools from the MiniZinc software distribution. Hence, development and execution of GRACe programs requires the following software dependencies to be met:

(i) The MiniZinc and Gecode solver software.

(ii) A Haskell toolchain able to download packages from Hackage, for instance the Haskell Platform.

Detailed instructions for installing the Haskell Platform is available at https://www.haskell.org/platform/. The preferred way of installing the MiniZinc and Gecode components is by way of the bundled binary packages available at http://www.minizinc.org/software.html.

Finally, instructions for setting up the GRACe library for building and executing GRACe programs is available at the GRACe GitHub repository.

A.2 Installation using Docker

In addition to the installation instructions for the GRACe development tools (Appendix A.1) we also provide a platform-independent Docker image containing an executable for the `OilCrops` example, written in GRACe.

The `OilCrops` example contains a small optimization problem in which the objective is to dedicate a set amount of farmland area to three different crops, with the goal of maximizing the yield of vegetable oil produced from these crops. A full description of the example can be found at GitHub[4].

Running the `OilCrops` example requires the Docker Community Edition (CE) to be installed. Docker CE, as well as installation instructions are available at https://www.docker.com/products/docker. Once Docker CE is installed, the example can be executed using the Docker application as follows. Open a terminal (the command prompt for Windows users) and execute the commands

```
docker pull eugraceful/grace-examples:latest
docker run --rm eugraceful/grace-examples:latest
```

This will run the `OilCrops` example and write the problem solution to standard output.

References

[1] Patrik Jansson. D4.2: A domain specific language for graceful concept maps. Deliverable of the Global systems Rapid Assessment tools through Constraint FUnctional Languages (GRACeFUL) project. FETPROACT-1-2014 Grant No 640954, 2017.

[2] Klara Marntirosian Tom Schrijvers, Paolo Torrini. D5.1: Domain-specific language for the constraint functional programming platform. Deliverable of the Global systems Rapid Assessment tools through Constraint FUnctional Languages (GRACeFUL) project. FETPROACT-1-2014 Grant No 640954, 2016.

[4] Source: https://github.com/GRACeFUL-project/GRACe/blob/master/examples/OilCrops.hs and https://github.com/GRACeFUL-project/DSL-WP/blob/master/tutorial/pdf/GRACeTutorial.pdf.