Athos - A Model Driven Approach to Describe and Solve Optimisation Problems*

An Application to the Vehicle Routing Problem with Time Windows†

Benjamin Hoffmann
KITE - Kompetenzzentrum für Informationstechnologie
Technische Hochschule Mittelhessen
Friedberg, Hesse, Germany
benjamin.hoffmann@mnd.thm.de

Kevin Chalmers
School of Computing
Edinburgh Napier University
Edinburgh, Scotland
k.chalmers@napier.ac.uk

Neil Urquhart
School of Computing
Edinburgh Napier University
Edinburgh, Scotland
n.urquhart@napier.ac.uk

Michael Guckert
KITE - Kompetenzzentrum für Informationstechnologie
Technische Hochschule Mittelhessen
Friedberg, Hesse, Germany
michael.guckert@mnd.thm.de

ABSTRACT

Implementing solutions for optimisation problems with general purpose high-level programming languages is a time consuming task that can only be carried out by professional software developers who typically are not domain experts. We address this problem by developing the Domain Specific Language Athos that allows declarative specification of Vehicle Routing Problems with Time Windows (VRPTW). The model is input to a generator that creates programs to solve the VRPTW in a multi-agent environment (NetLogo) which is further extended with a Genetic Algorithm optimiser.

We discuss the overall Athos architecture and compare the models with the generated code to demonstrate the benefit for developers by discussing general language related considerations. A case study with a published benchmark gives proof for the practical feasibility of our approach.

Beyond the quality criteria discussed in this paper future work will include extensive field experiments with domain experts applying the language to harden the language and improve usability.

Keywords – Genetic Algorithms, Model Driven Software Development, Multi Agent Systems, Vehicle Routing Problems

CCS CONCEPTS

• **Applied computing** → **Transportation**; Forecasting; • **Software and its engineering** → **Source code generation**; **System modeling languages**; *Software development techniques*;

KEYWORDS

Genetic Algorithm, Model Driven Software Development, Multi Agent Systems, Vehicle Routing Problems

ACM Reference Format:
Benjamin Hoffmann, Kevin Chalmers, Neil Urquhart, and Michael Guckert. 2018. Athos - A Model Driven Approach to Describe and Solve Optimisation Problems: An Application to the Vehicle Routing Problem with Time Windows. In *Proceedings of* . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper describes the application of a DSL, named Athos, to the domain of Vehicle Routing Problems (VRPs). VRPs are a range of real-world problems which require the planning of one or more vehicles to visit a number of customers. There exists many variants of the basic problem which can encompass constraints such as loading capacities, time windows for visits or requirements for breaks from driving. Traditionally, VRPs have been modelled in a general-purpose, high-level language, such as C or Java. The model is typically coded in the same language as the algorithm used to solve it. This approach frequently leads to each attempt to solve a VRP being coded from scratch, which has two major drawbacks:

- Development time is wasted producing many similar VRP models in a high level language.
- Domain experts (e.g. transport planners) normally lack software development experience and need to engage the services of a software engineer to develop the model.

The authors believe that the development of a DSL for VRPs will overcome the above two problems. The specification of a model using a DSL will be quicker than using a high-level language. The use of a DSL should allow a domain expert to be able to undertake some or all of the model development. This approach allows faster development of models and also increases the influence of the domain export on the model formulation. Our principle motivation in carrying out this work is to empower the domain expert allowing them to apply their expertise and experience directly to the problem via the DSL. We examine the question how a DSL can simplify the

*Produces the permission block, and copyright information
†The full version of the author's guide is available as `acmart.pdf` document

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

modelling of a VRP and allow the model to be solved and visualised. After discussing related work this paper presents the Athos DSL and its architecture i.e. the Athos syntax and the process of generating code for a target platform. General language quality issues are addressed before the practical feasibility of Athos is demonstrated with a case study inspired by a real world problem derived from a cooperating company business case. We relate the case study to a published benchmark and to compare results.

2 RELATED WORK

In [15] Steil et al. present a model for the the expression, execution, evaluation and engagement of routing plans to which they refer as the 4Es model. They use the model to map all relevant steps in the domain of patrol routing to appropriate software components. In the presented approach, routing algorithms are defined by means of a DSL named *Turn*. The DSL is used to describe algorithms that allow an agent to determine its next target node in a graph network. To this end, Turn enables the user to specify algorithms for target selection as a composition of so-called set reduce functions (SRFs). An SRF is given a set of possible target nodes and reduces this set according to certain criteria. Turn is used to chain and configure an arbitrary number of such SRFs and optionally assign an execution probability for each SRF. An SRF chain then is executed until only one node is left which becomes the new target node. If the subset comprises multiple nodes, one of these nodes is chosen randomly as the next target node. Algorithms written in Turn are executed by the PatrolSim environment which also evaluates routes produced by the algorithms according to a pre-defined set of metrics. A geographic information system (GIS) engages users through provision of patrol routes upon request.

The presented 4Es model is similar to our approach in that it is situated in the domain of VRPs. The creation of satisfactory patrol routes is a problem related to the Vehicle Routing Problem with Time Windows (VRPTW). What both approaches have in common is the application of a DSL for the specification of agents' behaviour. Though both DSLs can be used to direct agents through a network of nodes, they differ in the way target nodes are specified. While Turn is mainly a composition of rules that successively reduce a set of potential target nodes, Athos descriptions allow an explicit statement on which node or set of nodes to visit or to create an optimal tour from a given set of nodes. Another subtle difference is the way behaviour changes are defined in both languages. In Turn, events that lead to a change of behaviour are "hidden" inside the SRF. When an SRF returns an empty set of nodes it is skipped, which leads to the application of another SRF and thus to the change of behaviour. However, when exactly an empty set is returned is defined inside the called SRF. In Athos, it is possible to explicitly state when a change of behaviour has to occur by providing a boolean-expression following the keyword `when`.

Another aspect that completely distinguishes the 4Es model from our approach is that in the 4Es model the cost to travel between any two nodes is defined as the minimum number of vertices between these nodes. In other words, the cost is defined as the minimum hop-count. This approach is not only one-dimensional, but also extremely static. Once the distance between two nodes is defined, it remains the same throughout the entire simulation. Athos allows the definition of an arbitrary number of cost functions for travelling inside the network. Moreover, these cost functions can contain factors that dynamically change depending on the current traffic situation.

Literature on DSLs designed for the specification of complex vehicle routing problems in a multi-agent environment is scarce. By contrast, considerably more research has been conducted on general DSLs with a focus on multi-agent systems.

The GAMA framework [7] aims to provide support in the creation of spatialised, multi-scale agent-based models. It features a DSL named GAML that was created with the intention of facilitating the design of agent-based simulation experiments. Hence, it seeks to raise the abstraction level while at the same time allowing for easy parametrisation and flexible visualisation of the models. GAML is a feature-rich language that allows to describe arbitrary details of the model. A characteristic trait of the language is its capability to integrate multiple-levels of abstraction in one simulation. So it is capable of integrating a coarse-granular model for a road network with several fine granular models that represent the inside of buildings adjacent to the roads. GAML can be considered a language that increases the abstraction level of models compared to those created in other simulation platforms like Repast or Netlogo. However, since the GAMA framework is not tailored towards one specific application domain, models written in GAML seem complex. Domain experts with little to no experience in any programming might find concepts such as data types or the definition of aspects to visualise agents in the simulation hard to understand.

Theoretically, all of these approaches could provide support in the development of agent-based simulations for real-world VRPs. However, the main benefit of these approaches would be a simplified modelling of general agent concepts such as facts known by an agent. None of these approaches can be used to generate traffic simulations from a DSL specification without considerable additional effort. This effort would be required for the development of either an appropriate simulation-platform that processes the models created by the DSL or a generator which would transform the DSL programs to an executable traffic simulation.

3 ATHOS

3.1 The Problem Domain

The field of vehicle routing encompasses problems ranging from the Travelling Salesman Problem (TSP) [4] to the Vehicle Routing Problem (VRP) [6], environmental considerations, fleet mix (Fleet Size and Mix Problem) and customer time windows (Vehicle Routing Problem with Time Windows – VRPTW). For an overview of problems and problem-solving techniques the reader is directed to [11].

Within the VRPTW, a set of customers must be visited. Depending on the context, the visit may be a pick-up or a delivery. Each visit has to take place within a specified time window. If the vehicle arrives prior to the commencement of the window, then it must wait until the window commences. If the vehicle arrives after the end of the time window then the visit cannot be made. The problem may have a fixed number of vehicles or the number of vehicles may be optimised as part of the problem. Vehicles start and end

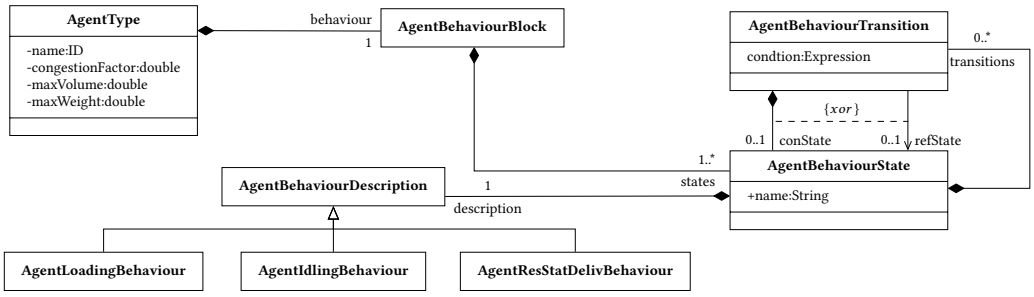

Figure 1: Athos' meta-model elements to represent agents' behaviour as state machines.

their journeys at a depot. Some formulations of the problem feature multiple depots.

The VRPTW is typically formulated as an optimisation problem, typical objective criterion include vehicles, distance travelled, financial costs or environmental impact. Initially many studies [5] formulated the VRPTW as a bi-objective problem, focusing on vehicles and distance travelled. More recent studies have formulated the VRPTW as a many-objective problem ,and in some cases producing a range of non-dominated solutions in order to allow the user to make the final choice of solution [14].

The VRPTW and variants have been applied to many real-world problems used in an operational context [17] and a planing context [16] . In both contexts, the user of the software will be a domain expert, e.g. a logistics analyst or transport planner. In many industrial contexts the problem to be solved will include constraints specific to the organisation who own the problem. Such specific constraints could include the working conditions of staff, the types of vehicle in use, environmental or financial considerations. The implementation of such constraints leads to an increased workload for those responsible for the technical implementation, which would typically utilise a high-level programming language such as Java or C++. The development of a DSL in this field has two possible advantages: firstly a reduction in development time, and secondly the possibility of domain experts producing systems rather then passing their requirements to a software engineer.

A requirement of any problem instance is travel time data between customers and depots. Depending on the optimisation criterion used it may also be necessary retain distance or emissions data as well. Some academic studies (such as [6]) have held such data in an origin-destination matrix, or simply used the Euclidean distance between customers. Real-world examples [16] [17] usually model the underlying street graph, in which case a path-finding algorithm is required to find the route between customers.

Taking into account the above, a DSL within vehicle routing must be capable of encompassing many different formulations of the VRP including differing vehicle types, time windows, capacity constraints, driving time constraints as well modelling the underlying street graph.

3.2 Usage of Implicit State Machines

In Athos, the behaviour of agents is defined by means of *behaviour blocks*. Each agent type features exactly one agent behaviour block in which an arbitrary number of agent *behaviours states* is defined. These behaviours represent the states agents can assume. An agent can only be in exactly one state at each point in time. The observable behaviour of an agent – i.e. the actions it displays for the outside world to perceive – is only one of two parts that form an agent's state. The second part is a set of behaviour transitions. These transitions define which stimuli trigger an agent to change its state as well as which state to assume when a given stimulus occurs. This means that an agent can change its state without changing its observable actions.

As an example, take an agent that is in a state to perform a delivery tour. If the agent runs low on the product delivered on its tour, it returns to a depot and replenishes its stock. At a pre-defined point in time, the agent may change its state but still continue to deliver products. The difference is in the case when the agent runs low on the product. Instead of returning to the depot, the agent now returns home and continues deliveries on the following day.

The meta-model elements Athos uses to represent agent states are illustrated in Figure 1. Each *AgentType* is associated with exactly one *AgentBehaviourBlock*. The AgentBehaviour block serves as a container for one or more *AgentBehaviourStates*. An agent's state in Athos is associated with exactly one observable behaviour that the agent exhibits when being in the respective state. The meta-model refers to this observable behaviour as an *AgentBehaviourDescription*. The DSL already offers several such AgentBehaviourDescriptions. Future versions of Athos will introduce further behaviour descriptions. Thus, language users already have control on the way agents behave in the course of the simulation while further additions of behaviour descriptions will further increase the possibilities of behaviour descriptions.

As was already mentioned, an agent's state also features definitions of stimuli that lead to a change of state. In order to model this, an *AgentBevahiourState* also comprises an arbitrary number of *AgentBehaviourTransitions*. Each transition is associated with a conditional expression. This expression represents the stimulus that triggers the transition. Additionally, each transition is associated with exactly one target state. This target state can either be a named state which is accessible to any transition of the respective *AgentBehaviourBlock* (in Figure 1 represented by the directed association relation) or it can be an anonymous state whose definition is embedded in the definition of the *AgentBehaviourTransition* (represented by the composition relation).

```
1  agentType customDeliveryAgent congestionFactor 60.0 maxWeight 100.0
2  behaviour loadVehicle loadCargo soap relQuantity 0.4, towels relQuantity 0.6
3      when finished do waitAWeeBit;
4  behaviour waitAWeeBit idle for 500.0 when finished do deliverGoods;
5  behaviour deliverGoods deliver (n1, n2, n4, n5, n6, n8) everything
6              when quantityOf soap < 20.0 || quantityOf towels < 10.0 do return1
7              when finished do return2;
8  behaviour return1 returnToDepot nearest
9              when finished do loadVehicle2;
10 behaviour return2 returnToDepot nearest when finished do end;
11 behaviour loadVehicle2 loadCargo soap relQuantity 0.3, towels relQuantity 0.7
12             when finished do idleSomeMore;
13 behaviour idleSomeMore idle for 200.0 when finished do res;
14 behaviour res resume deliverGoods at last;
15 behaviour end vanish;
```

Listing 1: Example program with behaviour specification

The listing above illustrates an example of an agent type that features multiple states with their associated behaviours and transitions. The first line defines some general properties of the agent type. The next line defines the first state an agent of the respective type can be in. The first state defined for an agent type is always considered the entry state, i.e. the agent starts in that state.

The Athos concrete syntax does not distinguish between the state and the behaviour of an agent. In fact, the keywords used in the concrete syntax deliberately avoid any vocabulary associated with state machines. The reason for this syntax-design decision is that the language is intended to mirror the way language users think about the problems they model with Athos. While state machines are a powerful concept, not every practitioner in the field of vehicle routing is familiar with it.

The entry state of the program features a loading behaviour. This is used to define how the capacity of the vehicle is to be used. In the concrete example, the agent will load 40 percent of its available cargo with soap and 60 percent with towels. The next line defines a transition to the next state. LoadingBehaviour is a state always associated with an instant behaviour, i. e. by default, it does not consume any time. In order to model the duration it takes for a driver to load a vehicle, the target state should be a state associated with a waiting behaviour. For this, Athos features IdlingBehaviours that can be used to let agents idle on the map.

The idling state has the agent waiting for 500 ticks and then transition to a state named 'deliverGoods' which is associated with an instance of AgentStrictDeliveryBehaviour. The term strict in this context refers to the fact that the agent does not do any optimisation but strictly visits the nodes in the exact order given in the behaviour definition. The AgentStrictDeliveryBehaviour implicitly has the agent deliver everything demanded by the agent. Later language versions will allow a fine-granular specification of which quantity of which product an agent delivers to a customer. When executing the delivery behaviour, it is possible that the agent runs low on a product and it is guaranteed that the agent will have visited every customer of the specified tour at some point.

In case that the agent runs low on a product (in the example specified in absolute units), it transitions into state 'return1'. In this state, the agent will return to the depot. In case that the agent finished the tour, it will transition to state 'return2'. Just like 'return1', 'return2' is a state in which the agent moves back to the depot. The difference between both states is, that in the former, the agent replenishes its stock before it resumes the delivery behaviour, while in the latter, the agent simply disappears from the simulation upon return at the depot.

Note that in the resume behaviour ('res' in the listing), the keyword `at last` is used. This specifies that the agent returns to the last customer visited on the delivery tour. Thus, it is ensured that the customer's demand is fulfilled completely before the agent turns to the next node in the network. Alternatively, the keyword `at next` can be used. In that case, the agent will visit the customer that it would have visited had it not returned to the depot. Further versions of the language will allow more options to define which node to visit after replenishment (i.e. only return to last customer if not all demands were fulfilled or rearrange the order of remaining customers).

Figure 2 depicts how the behaviour defined for agents of the type `customDeliveryAgent` is interpreted as a state machine. Each behaviour specification from the program is interpreted as a state in which the respective activities are performed. The conditions for state changes are translated to transitions that use the conditions in the program within change and time events. To actually model a state machine, Athos leverage's the meta-model element discussed earlier.

In a final step, the Athos generator translates each state machine instance to the target language. Though Athos aims to support multiple target languages, the translation to NetLogo will be used here as an example. For each state, the generator creates three anonymous functions (also known as lambdas). The first anonymous function, represents the entry behaviour, i.e. it prepares everything for a proper execution of the second anonymous function. This second anonymous functions corresponds to UML's do behaviour. Here, the observable actions exhibited by the agent when in the respective state are defined. This anonymous function also comprises the code that checks the necessity of a state change due to the occurrence of a change or time event.

The third anonymous function contains code that may clean up certain data structures. Most importantly, this anonymous function also features code that stores important information about the state that is about to be left. As an example, for a state associated with the AgentStrictDelivery behaviour, the last customer to be served as well as the next customer on the list is stored in a dictionary and later retrieved by the respective resuming state. It is important to note that the key used in the dictionary cannot simply be created from the state that the agent is about to leave. Since there may be several different agents in one simulation that may at some point perform and leave a delivery state, a unique identifier of the respective agent instance is added to the key. This ensures that each agent resumes the correct behaviour.

Listing 2 above illustrates what is generated for each state of a state machine: For each state three commands are generated, as explained in the last paragraph. In order to avoid naming conflicts inside the generated code, every method triple is also given two identifiers. 'M#' (where '#' represents a number) associates the commands with state machine number #. Since a given type of behaviour may be used more than once in a state machine, a second identifier 'B#' is needed, that features a number that increases each time a given behaviour type is used.

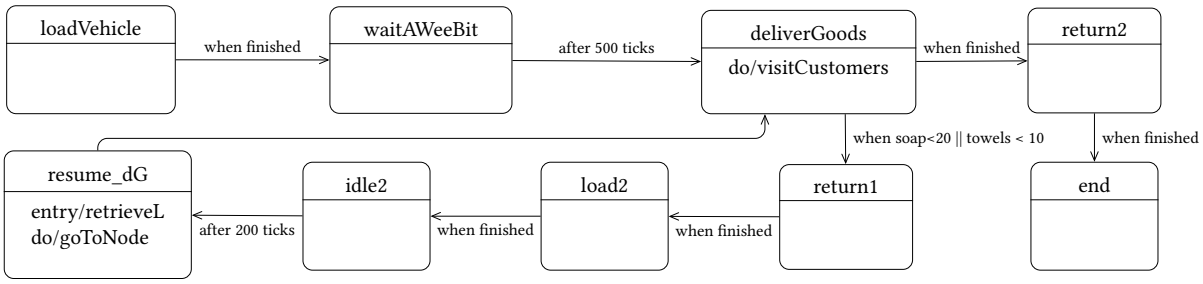

Figure 2: Behaviour of custom delivery agent interpreted as a state machine.

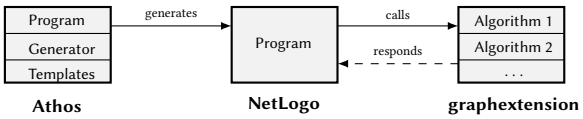

Figure 3: Graphextension architecture

```
1  to <BehvrName>-M<#>-B<#>-entry
2    set currentStateFinished false
3      <more initialisation code>
4      transStateAndRun
5        [-> <BehvrName>M<#>-B<#>-main]
6  end
7
8  to <BehvrName>-M<#>-B<#>-main
9    <Activities for behaviour>
10     if <everything done>[set curStateFin true]
11     if <condition> [transStateAndRun
12       [<BehvrName-M<#>-B<#>-exit [newStateName-M<#>-B<#>-entry]]]
13     <Activities for behaviour>
14     if curStateFin
15     [transStateAndRun[<BehvrName-M<#>-B<#>-exit[newStateName-M<#>-B<#>-entry]]]
16 end
17
18 to <BehvrName>-M<#>-B<#> [nextState]
19   <cleaning statements>
20     <write data to tables>
21     transStateAndRun nextState
22 end
```

Listing 2: Template for generated NetLogo code

The statements that define the observable behaviour of an agent are defined in the second of the three commands. This command thus features the code for state transitions. If a specified condition is true, the exit command is called together with a reference to the entry method of the next state. The fact that the current state is finished is represented by a flag 'curStateFin'. This flag allows to defer the transition to the exit-command in order to perform some specific actions in the main command for a last time.

3.3 Athos graphextension Package

Figure 3 illustrates the process by which simulations are generated in Athos. The program is given to a generator that uses templates to generate code in a target language which in the scope of this paper is NetLogo. Specifications made in the DSL are thus translated to NetLogo which solves optimisation tasks by handing the relevant data to a graphextension package that features several algorithms for the solution of traffic and transport related optimisation tasks. The solution is then returned to NetLogo where it is further processed. In order to solve VRPTWs, we implemented an evolutionary algorithm based on the one presented by Ombuki et al. [14].

In order to explain the exact implementation inside Athos graphextension package, a condensed explanation of the relevant steps will be given now[1]. The parameters that can be set by the user when defining a VRPTW problem are underlined. If a user does not explicitly define parameter values, sensible default values taken from the literature are applied.

Lines 1 – 3 The algorithm operates on a Graph $G(V, E)$ that comprises a set of nodes V and a set of edges E. The function d_f assigns a distance to each edge. The function t_f assigns a temporal value to each edge. This value represents the time it takes an agent to cross the respective edge. The customer nodes are a proper subset of the set of all nodes, $C \subset V$. Each customer node has to be visited by one agent (also referred to as vehicle) of a homogeneous set of agents with a maximum capacity of c_{max}. Agents start at a special node d called a depot. The depot may not be a customer node, thus $d \in V \setminus C$. The function h_f assigns a demand to each customer node. e_f defines an earliest time to each customer. This value represents the opening of the time window, within which each customer has to be visited. Along those lines, function l_f assigns the ending of a time window to each customer and the depot. The value assigned to the depot by l_f is the point in time until all vehicles must have returned to the depot from their respective tour. Via function s_f a service time is associated with each customer. The service time this is the time a visit at the respective customer takes.

Lines 5 – 6: First, the algorithm creates a *timeMatrix* and a *distanceMatrix* that store the value for the shortest time and the shortest distance between any two customers or any customer and the depot. While in complete graphs these matrices just store the respective values of the edge connecting two customers or a customer and a depot, in non-complete graph the shortest distance as well as the shortest time between those nodes first has to be computed using Dijkstra's algorithm (cf. [8]).

Lines 7 – 24: The next step is the creation of the initial population whose size is determined by the parameter *popSize*. For the generation of a chromosome, the algorithm uses two different strategies. With a probability of *simplePermuProb*, the algorithm simply generates a random permutation of all customers to visit. With the

[1]The Java source code of our implementation can be downloaded from https://athos.mnd.thm.de/ParetoCapacitatedSolutionVRPTW.java

```
1  Input: G(V, E), d_f : E → ℝ, t_f : E → ℝ, C ⊂ V, d ∈ V \ C, c_max ∈ ℝ
2         h_f : C → ℝ, e_f : C → ℝ, s_f : C → ℝ, l_f : C ∪ d → ℝ
3  Output: List of routes ℛ_Tours
4  Begin
5  Derive time matrix T_ij, i, j ∈ C ∪ d using Dijkstra's and t_f.
6  Derive distance matrix D_ij, i, j ∈ C ∪ d using Dijkstra's and d
7  //Create initial population P
8  set P := ∅;
9  while |P| < popSize
10   if rand(0,1) < simplePermuProb
11     create a random permutation p of elements in C and add p to P;
12   else
13     set C_copy ← C;
14     init p as empty chromosome;
15     while C_copy ≠ ∅ do
16       set p[p.length] ← getAndRemoveRandomElement(C_copy);
17       while ∃c ∈ C_copy : < maxDistance
18         set c_nearest ← nearestToFrom(p[p.length − 1], C_copy);
19         set p[p.length] ← c_nearest ;
20         set C_Copy ← C_copy \ c_nearest ;
21       od
22     od
23   fi
24 od
25 // Derive a List of routes for each chromosome
26 set Ω ← ∅
27 foreach p ∈ P do
28 set ℛ_tours ← ∅
29 set r[] ← newEmptyTour
30 ℛ_tours.add(l[])
31   foreach c ∈ p do
32     if customer c can be reached inside constraints add c to r[]
33     if costomer c can't be reached and r[] is empty ⇒ infeasible
34     else create new empty tour r[], add it to ℛ_tours and try to add c as first customer.
35   od
36 od
37 foreach(ℛ_tours ∈ Ω) do
38   Try to improve total distance by adding last customer of route r_i[] to route r_i+1[].
39 od
40 for(i = 0 to generations do
41   for(ℛ_tours ∈ Ω) do
42     calculate pareto rank for ℛ_tours;
43     set w_Σ(ℛ_tours) ← w_1 · cntTours(ℛ_tours) + w_2 · totalDistance(ℛ_tours);
44   od
45   set M_Mating ← ∅
46   while |M_Mating| < 2 · (popsize − 1) do
47     𝒯_Tournament ← randomly select tournamentSize elements from Ω
48     if rand(0,1) < takeBestProb
49       add best element to M_Mating( decide by rank in case of equality by w_Σ())
50     else
51           set M_Mating ← 𝒯_Tournament ∪ M_Mating
52     fi
53   od
54   o ← perform pairwise BCRC crossing of successive elements of M_Mating
55     if rand(0,1) < mutationProb do mutate(o) od
56     add o to next generation.
57   finally add individual with highest w_Σ to next generation
58 od
59 return bestIndiviudal (decide by rank in case of equality by w_Σ).
60 End
```

Listing 3: Pseudocode of the evolutionary algorithm

complementary probability a greedy strategy is applied. Here, the algorithm creates a copy of the set of customers to visit (C_{copy}). The customers in this set are then transformed into a chromosome by randomly picking a first customer that is added to the chromosome p at the top position 0 (which corresponds to the length of the chromosome at this point). Then, the algorithm checks whether there is a customer remaining in (C_{copy}) whose distance to the customer at the top of the chromosome (who was last added to the chromosome) is less a equal to the threshold parameter *maxDistance*. In case that at least one such customer is found, the one with the shortest distance to the customer at the top is added to the chromosome and becomes the new customer at the top. The procedure then is reiterated until no customer within the threshold can be found or the set of remaining customers is empty ($C_{copy} = \emptyset$). In the first case, a new customer from C_{copy} is randomly chosen and added at the top of the chromosome. In the latter case, the loop is left and depending on the current population size, a new chromosome is created or the initialisation phase is finished.

Lines 25 – 39 In the next step, the initial population P of chromosomes is transformed to a set Ω that contains all lists of tours. For this, each chromosome p is transformed to a list of tours $\mathcal{R}_{tours}$ where each tour $r_i[]$ is a list of customers. For each chromosome, customers are processed as they are ordered in the chromosome. If a customer can be visited within time and capacity constraints, it is added to the current tour. If a visit would violate any of the constraints, the old tour is closed and a new one is created. Thus, the timer and capacity counter (not explicitly given in the listing) are reset and the customer is placed as the first customer of the new tour. Should visiting a customer as the first customer of a tour still violate time or capacity constraints, then the problem is simply infeasible. This is because the arrival time at a customer who is first in a tour, is the earliest arrival time possible. Same goes for the capacity of the agent which is at its maximum possible value when it arrives at the first customer. In a second phase, for each list of tours it is checked whether the total distance of all tours can be improved. This is done by shifting the last customer of tour $r_i[]$ to tour $r_{i+1}[]$ and placing this customer as the first customer of tour r_{i+1}. If the total distance can be improved and the tours do not violate any constraints, the result is kept.

Lines 40 – 44 The main loop of the algorithm continues to operate on the tours derived in the previous step. The number of iterations is specified with the *generations* parameter. The first step inside the main loop (line 42) is the calculation of the pareto rank for every list of tours $\mathcal{R}_{tours} \in \Omega$ as described by Ombuki et al [14, p. 22]. In short, this ranking seeks for all non-dominated lists of tours $\mathcal{R}_{tours} \in \Omega$. Such a list is non-dominated, if there is no other list that comprises less (or the same amount of) tours and a shorter (or the same) total distance (but it must be less in one of the two dimensions). When all non-dominated lists of tours are found, they are assigned rank 1 and removed from the ranking space. Then, the procedure is repeated for the remaining lists of tours whose non-dominated lists will be assigned rank 2 and so on. Next (line 43), a weighted sum w_Σ is calculated for each list of tours. Here, parameter w_1 is the weight associated with the number of vehicles and parameter w_2 is the weight associated with the total distance of all tours. The weighted sum w_Σ then simply is the sum of w_1 multiplied by the number of tours in the list ($w_1 \cdot cntTours(\mathcal{R}_{tours})$) and w_2 multiplied by the total distance of all tours in the list ($w_2 \cdot totalDistance(\mathcal{R}_{tours})$.

Lines 45 – 53: The second part of the main loop creates the next generation. For this, a mating set M_{Mating} is created. Via tournament selection the mating set is filled until $popsize - 2$ parent elements are found. Parent elements are found by creation of a tournament set which is filled with randomly chosen lists of routes $\mathcal{R}_{tours}$ from Ω. The exact number of lists is determined by the *tournamentSize* parameter. Another parameter, *takeBestProb*, then represents a threshold for a randomly generated number in the interval $[0, 1)$. If the random number falls below the threshold, only the best list of tours is chosen for the mating population. The best list is the one with the lowest rank and in case that there are two

or more lists with the lowest rank, the one with the best weighted sum value w_Σ is chosen. If the random number exceeds or equals the threshold, all elements from the tournament set are added to the mating population.

Lines 54 – 59: In the final part of the main loop, two elements of the mating set M_{Mating} are crossed via a BestCostRouteCrossover (BCRC) operator to produce offspring *o*. With a probability *mutationProb*, the offspring is subjected to a mutation operator. Again, we refer to Ombuki et al. [14] for details on these operators. As a result, our current implementation only the one list with the best w_{Sigma} value of the final generation. It is worth noting, that it is also possible to specifically search for a solution that does not contain more than a given number of tours. This is relevant for cases when a dispatcher cannot deploy more than a given number of vehicles. However, it might also be possible that the minimum number of vehicles for a feasible solution exceeds the desired number of vehicles in which case the problem is to be deemed infeasible.

3.4 Discussion of Quality Criteria

An important but often neglected task in the development of DSLs is evaluation [2]. Though lines of code (LoC) may not be a conclusive indicator for any quality aspect of a DSL, they can provide a first impression on the amount of development and maintenance effort [10]. For DSLs whose programs are translated to another language, comparison of LoCs in the source programme and LoC in the generated output can provide a first notion of a DSL's efficiency [18].

For this reason, the two example programs presented in this article, as well as the corresponding generated NetLogo code, shall be discussed to get a first impression of Athos' potential to ease the development of agent-based traffic and transport simulations. At this point it is important to mention that the generator's target language, NetLogo, can be considered a DSL in its own right. It provides concepts and primitives for agent-based programming. Thus, it can be assumed that NetLogo already reduces development effort compared to its target language Scala or any other general purpose language (GPL).

Table 1 gives information on the LoC metric for the two example programs presented in this article. Example I is the program given in Listing 1. Example II is the example which will be discussed in Section 4. For both examples, two columns are defined. The one in the column labelled 'Athos' contains information on the amount of code required by our DSL. Accordingly, the 'NetLogo' column informs about the number of generated lines of code in the target language NetLogo. The table gives information on the total amount of code as well as code required for behaviour as well as network definition.

It is obvious, that Athosconsiderably reduces the amount of code required to model the respective problem. This is true for both code concerned to define the behaviour of an agent as well as code used to set up the network in which agents move. Especially in the second example the amount of network related code in Athos is only a fraction of the NetLogo code derived from it. This is because the network defined in the case study features a complete network of 50 nodes, i.e. 2500 edges. While Athos only requires a few lines of code for this, the generator derives 2500 edge definitions from this and another 2500 lines for each edge attribute to be set. Though the generated amount of code can be drastically reduced by adapting the generator templates for complete networks, the overall amount of saved LoC still suggests that Athos facilitates the specification of traffic and transport problems.

Another metric that can be used to gain insight on the effort required to test and maintain a programme is McCabe's cyclomatic number [12]. For strongly connected control graphs of a program it is defined as

$$v(G) = e - n + p \tag{1}$$

where e denotes the number of edges, n the number of nodes and p the p the connected components of the control graph.

Figure 5 illustrates two control graphs. The upper graph is the control graph of the NetLogo state machine code[2] generated for lines 5 – 7, (the specification of a delivery behaviour) of the Athos program given in Listing 1. As is described in Section 3.2, each behaviour description in Athos results in three NetLogo commands (entry, main, exit). The control graph represents the control flow of the generated state-entry and state-main commands. The state-exit command was omitted since it does not contain any control structures. The pseudo-code in Listing 4 is a simplified version of the NetLogo entry-command in which all relevant control structures have been preserved. From the control graph it can be seen that the cyclomatic complexity of the NetLogo program is $43 - 31 + 1 = 13$ which is a value that indicates high maintenance and testing effort for the NetLogo code.

The lower graph is the control graph for the original Athos code from Listing 1. For this graph, the cyclomatic complexity is 9-6+1=4. This indicates considerably reduced maintenance and testing effort compared to the generated NetLogo code. Of course, similiar to the LoC metric, the cyclomatic complexity applied to a single example is not a conclusive metric. But it can be considered another indicator for Athos' capability of facilitating the creation of traffic and transport simulations.

Both LoC and cyclomatic complexity are basic quantitative measures that can be used to coarsely estimate the effort a program requires to be understood, tested and maintained. In order to present a heuristic evaluation (cf. [13]), we conclude this section with an application of the cognitive dimensions of notation (CDN) framework [3].The CDN framework is first and foremost a tool that allows to discuss several aspects or dimensions of a notation. Thus, it is

[2]The generated NetLogo state machine code can be inspected at https://athos.mnd.thm.de/StaticDeliveryBehaviour.txt

Table 1: Lines of code comparison bewteen Athos and Netlogo for two example programmes.

Part	Example I		Example II	
Part	Athos	NetLogo	Athos	NetLogo
Behaviour	21	660	4	321
Network	31	579	82	43691
Rest	4	112	3	272
Total	56	1351	89	44284

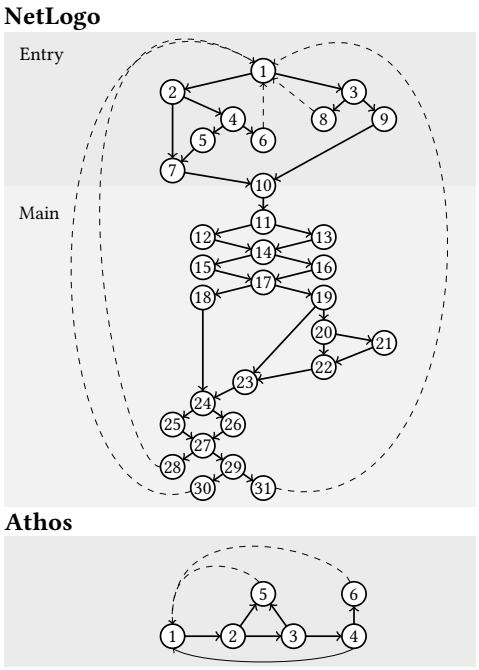

Figure 4: Control graph.

important that the following statements are be regarded as a critical self-assessment rather than analytically proven facts. A more detailed and quantified approach based on the CDN framework (cf. [1]) is among our future goals.

One of the first steps to be performed in the application of the CDN framework is the creation of an activity profile. In this profile, it is to be defined what activity is to be supported by the notation and which dimensions are especially important for this support. In order to keep this self-evaluation as short as possible, we will simply discuss four dimensions from the CDN framework that we deem of highest importance for a DSL that intends to facilitate the modelling of agent-based traffic and transport problems.

Visibility This dimension is concerned with how easy it is for language users to access and/or manipulate relevant aspects of the problem description. Athos is designed to allows users to create programs that feature high visibility. Though it could be argued that the language encapsulates many details inside its abstract and high-level language elements, it is important to look at what details are actually encapsulated. Athos seeks to hide those details that are normally of no relevance to domain experts. This removes unnecessary distractions and brings attention to elements of actual relevance. This increases visibility as it allows users to perceive and understand relevant elements of the program quicker. A good example for this is the definition of an optimisation task. For most users, it might be sufficient to define that the defined task be executed in a near-optimal manner. Users then just describe the task and use the `optimise` keyword. The system will find an appropriate algorithm with sensible parameter values and provide an acceptable solution. Some users, however, may want more control over the applied algorithm as well as the parameter values used during the execution of the algorithm. For those users, the language allows for the selection of a specific algorithm and specification of parameter values for the selected algorithm. The specified parameters are then passed to the algorithm in Athos' graphextension library where they are considered and processed by the respective algorithm (see Section 3.3).

Viscosity The number of user interactions certain goals require is the focus of this dimension. Due to its high abstraction level, Athosgenerally should be of low viscosity. This is especially true for The definition of agent-behaviour and optimisation tasks which can be described in a very concise manner. It is also rather easy to change the default function for the calculation of the amount of time agents need to travel a given road in dependence of various parameters like the current total congestion factor. As is shown in the case study of this article, Athoshighly facilitates the creation of complete graphs with homogeneous functions. However, if a non-default function is to be associated with a great number of roads, then this requires the user to add the symbolic name of the function for each road in the network, which. The association of non-default duration functions to roads can thus be considered ot feature an increased amount of repetition viscosity. The definition of demands and time windows for nodes is another language aspect with high viscosity. The problem in both cases is the fact that aspects which have to be individually defined for each entity of a system naturally increase the viscosity of the language.

Closeness of Mapping This dimension is concerned with the proximity of the language to the domain it is applied to. This dimension is especially important for a language like Athos that aims to be usable by domain experts without a background in programming. To achieve this, appropriate keywords have to be chosen and used in a logical order that closely resembles the way users would use them when giving an informal problem description. Though the DSL's concrete syntax is designed with this in mind, it is yet to be formally evaluated by domain experts. In the near future, we will formally evaluate the appropriateness of the selected keywords and their general structure. What may be difficult to some users is the ordering of keywords. There are many cases in which several different orders would appear to make sense that it seems impossible to define a keyword ordering / structure that satisfies the preferences of any user completely.

Hard mental operations This dimension focusses on the extent to which a notation draws on the mental capabilities of its users. This is another dimension we will investigate more formally in the near future in order to formally proof that Athos considerably reduces the amount of cognitive effort required for problem modelling compared to general-purpose languages like Java or languages like NetLogo. Athos facilitates several aspects of traffic and transport simulations which otherwise would require considerable mental efforts. One aspect where this becomes especially clear is an agent's movement through the modelled network. A large amount of the generated NetLogo code deals with ensuring that agents adhere to the movement rules of the simulations. It has to be ensured that agents follow the correct paths without leaving the roads to roam freely on the map. Agents must adhere to the speed limitations

determined by the network's functions. Agent's must also be informed when they have arrived at one of their customers so that they can perform the actual delivery before setting course for their next customer. All this is transparent to the user when using Athos. If a user wants an agent to travel from node A to node B, it suffices to specify a simple destination behaviour in which B is defined as the target node. The generated system will then take care that the agent travels from A to B on the shortest route possible. If a user insists that the agent has a brief stint at node C before moving to B, then all that needs to be done is to define two different destination behaviour, namely from A to C and from there to B. Here, it might be sensible to allow the definition of nodes that must be visited before arrival at a given destination within one single statement. It is likely that one of Athos' future versions will feature such syntactic elements. However, these are more a concern of viscosity or visibility rather than hard mental operations.

```
1  to perform-strict-delivery-M1-B1-entry
2      if(agent-resumes-this-state){ (1)
3        set tourList = tourDict.getTourListForThisStateAndAgent(self);
4          set tourIdx = indexDict.getIndexForThisStateAndAgent(self);
5          if (tourList[tourIdx] ≠ currentCity){ (2)
6            tourIdx = tourIdx - 1;
7              if (tourList[tourIdx] ≠ currentCity){ (4)
8                calculateDataToGetTo(tourList[tourIdx + 1]);
9                  goToSate(positionCorrection); (6)
10             }else{set nextCity tourList[tourIdx];} (5)
11         }
12         goToState(perform-strict-delivery-M1-B1-main);(7)
13     }else{
14         if (currentCity ≠ tourList[0]){ (3)
15           calculateDataToGetTo(tourList[0]);
16             goToState(positionCorrection); (8)
17         }else{
18           set nextCity = tourList[1];
19             goToState(perform-strict-delivery-M1-B1-main); (9)
20         }
21     }
22 end
```

Listing 4: Pseudocode for entry command

4 CASE STUDY

In this section we will discuss one of several case studies taken from the business model of a company that installs and maintains toilet facilities for others institutions e.g. companies, restaurants. An integral part of this model is periodically delivering consumed materials e.g. soap and towels according to an elaborately developed plan. While this is mostly done in stable, predetermined routine tours installations for new customers and repairs have to be planned ad-hoc. For both cases the company keeps depots with stocked materials and runs a fleet of travelling service technicians who do maintenance and refilling tours as well as initial installations for new customers and repairs. Tours have to be planned to minimize time and fuel consumption while respecting temporal constraints i.e. opening hours and general availability and the time needed for the requested service. Routine tours are planned based on experience and are rearranged on a long term scale only. Installation and repair tours have to planned whenever there is a demand for it.

The company keeps depots which all have a set of customers assigned each with a defined service interval to service. So basically this means that a VRPTW has to be solved for every depot.

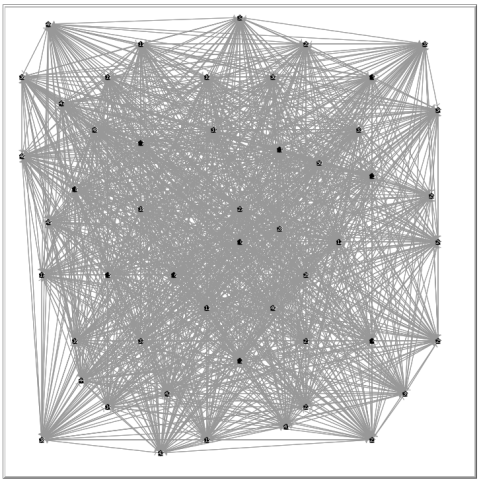

Figure 5: Visualized Solomon's R101.50 problem*

* see http://web.cba.neu.edu/~msolomon/r101.htm

We will now show how repair and installation tours can be modelled with Athos. For simplicity we only define tours on which new equipment gets installed.

Athos allows to define a representation of geographical locations in an abstract network of nodes and edges. Geographical information can potentially be imported from data sources like OpenStreetMap[3]. We have already experimented with that and see this as a feature that will be implemented later.

```
network
nodes
node n0 (1.0, 1.0) node n1 (1.0, 8.0)
node n2 (2.0, 11.0) node n3 (4.0, 6.0)
node n4 (5.0, 12.0) node n5 (8.0, 11.0)
...
edges
  edge undirected e0 from n0 to n1
    length 0.0 cfactor 2.0 function normal
  edge undirected e1 from n1 to n2
    length 0.0 cfactor 2.0 function normal
...
```

Products to be delivered must be defined together with weight, volume and profit per unit.

```
product soapDispenser
      weight 8 volume 5
      profit 15
```

As service agents can carry out new installations and repair existing ones we assume homogeneous fleets and therefore only have to define a single type of agent with required behaviours. For simplicity we model agents which only deliver products and therefore only have a single behaviour.

```
agenttype deliveryTour
          congestionFactor 1
          maxweight 1000
 behaviour deliver
        when finished returnToDepot
```

[3] www.openstreetmap.org

Agents start their tour in a depot which must be declared to be sources and keep lists of the clients i.e. demands that they are responsible for. With the option <textit>optimise maxAgents n</textit> Athos will create a tour with a number of vehicles less or equal to n. Without the option Athos will suggest a number of vehicles.

```
sources
    n1 sprouts (staticDelivery)
       isDepot soapDispenser
       agentsStart route (n2, n3, n4)
```

For each target in a route additional information i.e. demand, time window, etc. are defined.

```
demands
    n2 hasDemand soapDispenser
       absQuantity 10.0
       earliestTime 10 latestTime 30
       serviceTime 20
```

The problem sketched before matches well with the *R101.50* benchmark problem published by Solomon [4], which builds a service tour for a single depot serving 50 customers. We applied Athos to this setting. The generated NetLogo program with its integrated Genetic Algorithm produces a tour with a length of 1088 which is reasonably close to the optimal tour which has a length of 1044[5]. Athos programs have a simple structure that can easily be integrated with an ERP program that delivers information about demands.

5 CONCLUSION AND FUTURE WORK

The DSL Athos allows to define optimisation problems declaratively and shifts implementation details to the generator which creates code for suitable target platforms. It was applied to TSP-like routing problems [9] before and extended to be able to define and solve more complex vehicle routing problems. Argumentation along the line of quality parameters (e.g. cyclomatic number) showed that Athos programs are easier to understand and work with. An industry case study which was matched with a published benchmark demonstrated that the architecture can solve practical problems and is relevant as a tool for such problems. Future work will extend the language capabilities to express more problems from the domain. Beyond the more qualitative aspects of general language quality criteria discussed here more aspects will be examined and measured through field studies in which domain experts will compare working with Athos to using their current tools. While creating the case study, discussion with companies from the trade showed that practical applications will use Athos as interface to their transportation management system. The structure of the language make it suitable for such an integration which will be another aspect of future research.

REFERENCES

[1] Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahão, and António Ribeiro. 2015. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software* 101 (2015), 245–259. https://doi.org/10.1016/j.jss.2014.11.051

[2] Ankica Barisic, Vasco Amaral, and Miguel Goulao. 2012. Usability Evaluation of Domain-Specific Languages. In *2012 Eighth International Conference on the Quality of Information and Communications Technology (QUATIC)*, João Pascoal Faria (Ed.). IEEE, Piscataway, NJ, 342–347. https://doi.org/10.1109/QUATIC.2012.63

[3] Alan Blackwell and Thomas Green. 2003. Notational systems–the cognitive dimensions of notations framework. *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science. Morgan Kaufmann* (2003).

[4] William J Cook. 2014. *In Pursuit of the Traveling Salesman Mathematics at the Limits of Computation.* Princeton University Press.

[5] S. Dabia, E. Demir, and T. Woensel, van. 2014. *An exact approach for the pollution-routing problem.* Technische Universiteit Eindhoven.

[6] George Dantzig, Bernard Ramser, and John Hubert. 1959. The Truck Dispatching Problem. *Management Science* 6, 1 (Oct 1959), 80–91.

[7] Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, Duc An Vo, Nghi Quang Huynh, and Alexis Drogoul. 2013. GAMA 1.6: Advancing the art of complex agent-based modeling and simulation. In *International Conference on Principles and Practice of Multi-Agent Systems*. 117–131.

[8] Benjamin Hoffmann, Michael Guckert, Thomas Farrenkopf, Kevin Chalmers, and Neil Urquhart. 05252018. A Domain-Specific Language For Routing Problems. In *ECMS 2018 Proceedings edited by Lars Nolle, Alexandra Burger, Christoph Tholen, Jens Werner, Jens Wellhausen*. ECMS, 262–268. https://doi.org/10.7148/2018-0262

[9] B. Hoffmann, M. Guckert, T. Farrenkopf, K. Chalmers, and N. Urquhart. 2018. A Domain-Specific Language For Routing Problems, Lars Nolle, Alexandra Burger, Jens Werner Christoph Tholen, and Jens Wellhausen (Eds.). European Council for Modeling and Simulation. https://doi.org/10.7148/2018-0005 32nd European Conference on Modelling and Simulation, Wilhelmshaven, Germany, May 22nd – May 265h, 2018.

[10] J. Rosenberg. 1997. Some misconceptions about lines of code. In *Proceedings Fourth International Software Metrics Symposium*. 137–142. https://doi.org/10.1109/METRIC.1997.637174

[11] G Laporte and P Toth. 2013. Vehicle routing: historical perspective and recent contributions. *EURO Journal on Transportation and Logistics* (2013).

[12] Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[13] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 249–256.

[14] Beatrice Ombuki, Brian J. Ross, and Franklin Hanshar. 2006. Multi-Objective Genetic Algorithms for Vehicle Routing Problem with Time Windows. *Applied Intelligence* 24, 1 (2006), 17–30. https://doi.org/10.1007/s10489-006-6926-z

[15] Dana A. Steil, Jeremy R. Pate, Nicholas A. Kraft, Randy K. Smith, Brandon Dixon, Li Ding, and Allen Parrish. 2011. Patrol Routing Expression, Execution, Evaluation, and Engagement. *IEEE Transactions on Intelligent Transportation Systems* 12, 1 (2011), 58–72. https://doi.org/10.1109/TITS.2010.2065224

[16] Neil Urquhart and Achille Fonzone. 2017. Evolving Solution Choice and Decision Support for a Real-world Optimisation Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, 1264–1271.

[17] Neil B. Urquhart, Emma Hart, and Alistair Judson. 2015. Multi-Modal Employee Routing with Time Windows in an Urban Environment. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1503–1504.

[18] Timo Wegeler, Friederike Gutzeit, Aurèle Destailleur, and Bernhard Dock. 2013. Evaluating the benefits of using domain-specific modeling languages: an experience report. In *Proceedings of the 2013 ACM workshop on Domain-specific modeling*. 7–12.

[4] http://web.cba.neu.edu/ msolomon/r101.htm

[5] http://web.cba.neu.edu/ msolomon/r1r2solu.htm