Generic Propagation Algorithm for Goal Models

Hao Luo

CSI 6900 report submitted to Prof. Daniel Amyot
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

uOttawa

University of Ottawa
Ottawa, Ontario, Canada

May 2011

Acknowledgement

I would like to thank Dr Daniel Amyot for his support and guidance. Many fruitful discussions and flexibility to meet after work with him are greatly appreciated.

My wife Ling and my son Travis are the source of my power, I thank them for having always been there with me.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
CP	Constraint Programming
GRL	Goal-oriented Requirement Language
JaCoP	Java Constraint Programming solver
SAT	Satisfiability
UCM	Use Case Maps
UI	User Interface
UML	Unified Modeling Language
URN	User Requirements Notation

Chapter 1. Introduction

1.1. Motivation

jUCMNav [10] is an Eclipse [4] plug-in specialized for the visualization, design and analysis of User Requirements Notation (URN) [8] models. The Goal-oriented Requirement Language (GRL) [7], part of URN, is supported by jUCMNav for modeling goals and other intentional concepts. Currently, two major categories of evaluation approaches, qualitative and quantitative, exist and are supported by this tool. We will focus on the quantitative evaluation approach in this project, i.e., contribution, satisfaction, and importance levels will be represented with integer values. GRL model evaluation in jUCMNav is currently using a bottom-up approach that allows users to ask “What if?” questions [1]. The demand for a top-down or a more generic algorithm that supports ad-hoc queries in GRL models is growing, as these are necessary to answer “Is it possible?” and other optimization questions.

1.2. Contributions

A generic quantitative propagation algorithm is presented in this project. The implementation of the algorithm uses JaCoP [9], a constraint solver library written in Java, to compute a solution to a set of equations obtained from a given GRL model. The supported type of links between the intentional elements are AND decompositions, OR decompositions, dependencies, and contribution. This project even enables circular dependency in jUCMNav and is a much more generic algorithm compared to bottom-up and even top-down approaches as it allows for GRL intentional elements to be initialized anywhere in the model (not just roots and leaves).

1.3. Report Outline

Chapter 2 presents background information related to URN, jUCMNav and Java constraint solvers. In Chapter 3, implementation details and decisions made along the way are discussed. Chapter 4 shows the validation performed as well as the results, and finally in Chapter 5 we draw conclusions and talk about possible future work.

Chapter 2. Background

2.1. jUCMNav

“jUCMNav is a graphical editor and an analysis and transformation tool for the User Requirements Notation (URN). URN is intended for the elicitation, analysis, specification, and validation of requirements. URN combines two complementary views: one for goals provided by the Goal-oriented Requirement Language (GRL) and one for scenarios provided by the Use Case Map (UCM) notation” [10].

GRL models can be evaluated (i.e., the satisfaction levels of intentional elements are computed) based on a set of initial satisfaction values captured in a GRL strategy. jUCMNav currently supports qualitative evaluations (where satisfaction levels range from Denied to Satisfied) and quantitative evaluations (where satisfaction levels range from -100 to 100) . However, the current algorithms are limited to bottom-up propagation, where the satisfaction of higher-level intentional elements is computed from that of lower-level intentional elements. Top-down algorithms are considered much harder as they are akin to a search problem rather than a testing problem.

Our project is based on jUCMNav tool, and we will be providing an implementation of a generic propagation algorithm in the tool and visualize the solutions directly inside the tool. The results can be exported to other report formats as jUCMNav also contains report generators (to PDF, HTML, and Word’s RTF) and transformations to various formats, including Excel’s CVS and Message Sequence Charts [12].

2.2. Constraint Programming Tools

“Constraint programming (CP) is a programming paradigm wherein relations between variables are stated in the form of constraints. CP differs from the

common primitives of imperative programming languages in that CP does not specify a step or sequence of steps to execute, but rather the properties of a solution to be found" [3]. We see the possibility to transform a GRL model into a system of CP constraints. There are then plenty of existing tools that can solve the constraints system, thus giving us back, provided an initial GRL strategy, the GRL model fully populated with all satisfaction levels.

JaCoP is an open-source Java library offered under the GPL lisence, which provides Java users with CP technology [9]. It provides a large number of global constraint-solving mechanisms as well as a modular design of searches to support efficient modeling and tailored searches. We use JaCop to implement our algorithm in this project, and more details about how JaCop is used to model our problems at hand are given in Chapter 3.

There are other CP solvers written in Java that are available freely. Choco [2] is another open-source Java library for CP that builds on an event-based propagation mechanism with backtrack-able structures. Choco is offered under a BSD license. Both of the tools support various kinds of constraints, from simple binary satisfaction types (A or B is true) to advanced ones like global cardinality and knapsack, etc. Both JaCoP and Choco are equally as good for providing excellent performance and abundant documentations, and both offer academic friendly licenses.

We adopted JaCoP during the prototyping stage of this project, talked with the authors and obtained advice on how to model our problems efficiently in JaCoP. This process served as a mutually beneficial activity that gave us some insight of the tool that enabled performance optimizations and provided feedback to them which help them discovered a bug in JaCoP. At the end we decided to keep using it because of the above reasons and in order to comply with this project's tight time constraints.

2.3. Related work

Horkoff et al. proposed an interactive, iterative approach in [11] to find solutions in goal- and agent-oriented models. They encode the propagation rules obtained from qualitative analysis into conjunctive normal form, iteratively applying a SAT solver and human intervention to search for an acceptable solution. They provide a framework to support their evaluation procedure that axiomatizes propagation in the framework, including the role of human intervention to potentially resolve conflicting contribution or promote multiple sources of weak evidence.

In [13], Weiss et al. presented a security pattern selection approach that allows one to understand in depth of the trade-offs involved in the patterns and the implications of a pattern to various security requirement, as well as supports the search for a combination of security patterns that will meet given security requirements. They use GRL models to show the contributions a pattern makes on security-related properties, then map the models into Prolog to reason about the evaluation mechanism and to project the effect of combining security patterns.

Other goal languages that are similar to GRL do exist. KAOS [15,16] is a goal-oriented requirement engineering language and method used to capture requirements in terms of objects, goals, actions, constraints and agents. A quantitative algorithm can be used to evaluate the partial satisfaction of goals by computing the weighted average of the sub-goals' satisfaction. Letier et al. in [5] have extended KAOS models with a probabilistic layer for reasoning about partial satisfaction. TROPOS [14] is an agent-oriented language and software development methodology that includes the concepts of agents and goals. While it supports forward and backward propagations, conflicts cannot be resolved; instead, two values (one positive, one negative) are left separate.

While each of the above tools and algorithm addresses one or more specific problems, none of them can get a solution in a simple goal model, where

only Goal and one of its two Subgoals are initialized, as shown below, in an automatic fashion. Only a very general algorithm which allows for any node to be initialized can do this.

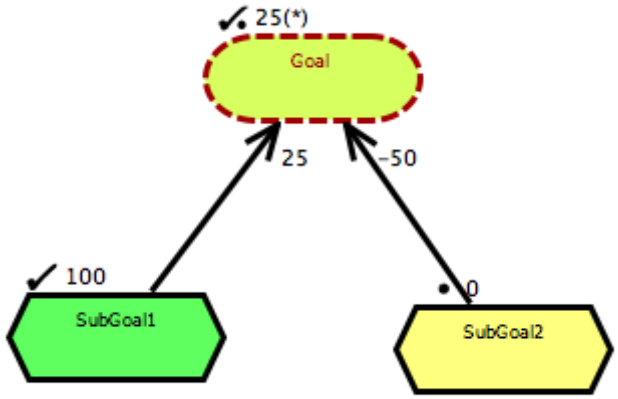

Figure 1. Simple Goal Model

Chapter 3. Generic Constraint-Based Algorithm

GRL uses a scale of [-100,100] in quantitative analysis. This gives us a basis for the domain of all variables that are going to participate in evaluations. We break down the mathematical interpretations of different types of links below. Although only base cases are shown here, one can easily decompose a more complicated, generic GRL graph to apply the formulas, by only looking at one node, as well as at all the links coming into that node at a time.

3.1. Quantitative Analysis - Base Cases

3.1.1 Two nodes with a contribution link

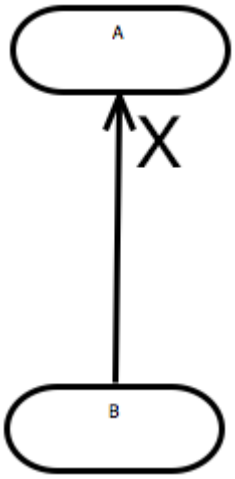

Figure 2 Node B contribute to Node A by X

Constraint-based representation: **A = MAX (-100, MIN(100,(B * X)/100))**

3.1.2 Two nodes with an AND decomposition link

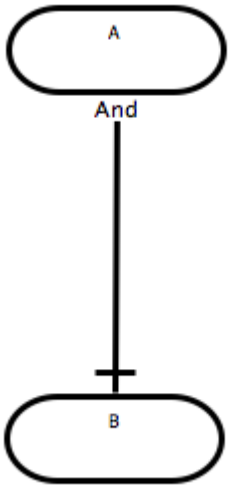

Figure 3 Node B AND-compose Node A

Constraint-based representation: **A = B**

3.1.3 Two nodes with an OR decomposition link

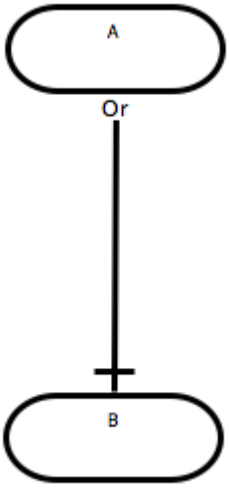

Figure 4 Node B OR-compose Node A

Constraint-based representation: **A = B**

3.1.4 Two nodes with a dependency link

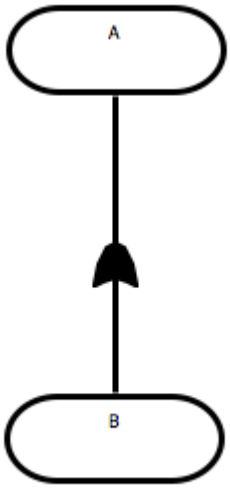

Figure 5 Node B depends on Node A

Constraint-based representation: **B <= A**

3.2. Quantitative Analysis - Generic Cases

3.2.1 N nodes with contribution links

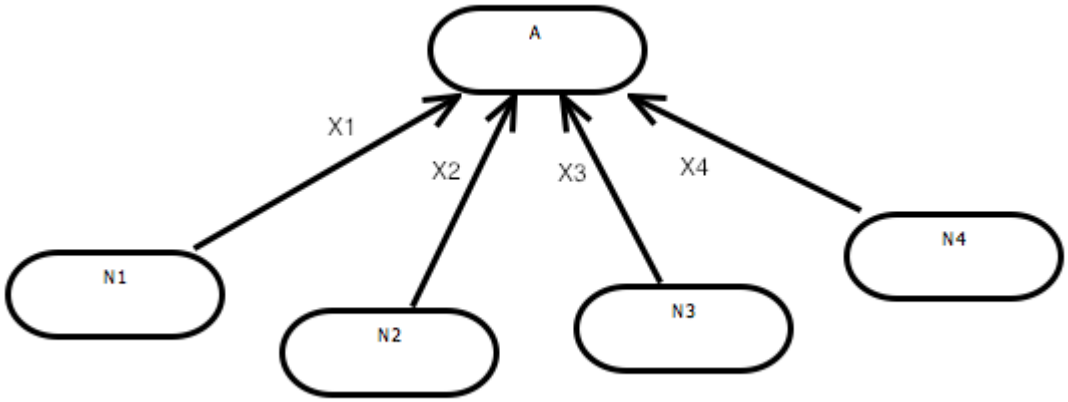

Figure 6 Node 1 to k contribute to Node A by X_1 to X_k

Constraint-based representation: **A = MAX (-100, MIN(100, (X1*N1 + X2 *N2 + … + Xk *Nk) / 100))**

3.2.2 N nodes with AND decomposition links

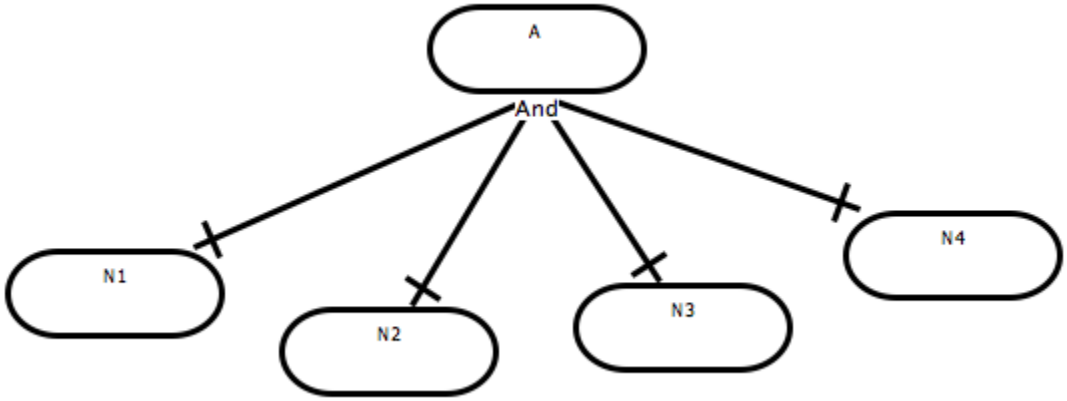

Figure 7 Node 1 to k AND compose Node A

Constraint-based representation: **A= MIN(N1, N2, ..., Nk)**

3.2.3 N nodes with dependency links

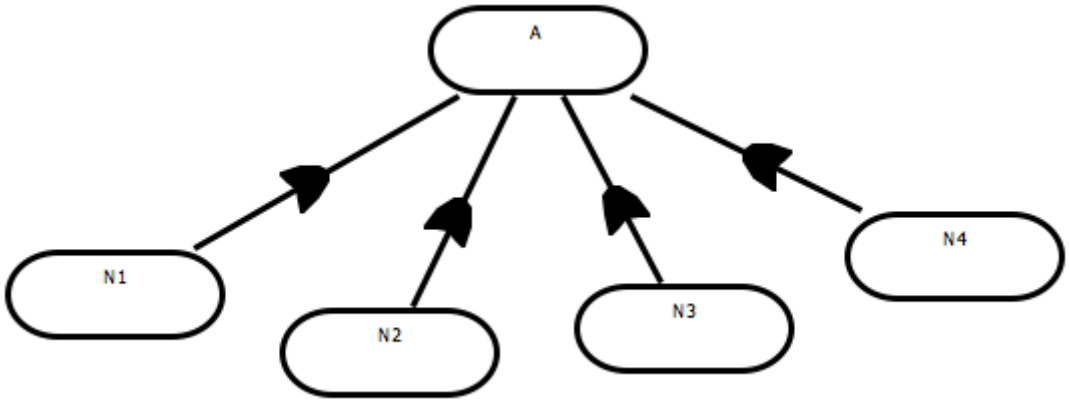

Figure 8 Node 1 to k depend on Node A

Constraint-based representation: **(N1 <= A) AND (N2 <= A).....AND (Nk <= A)**

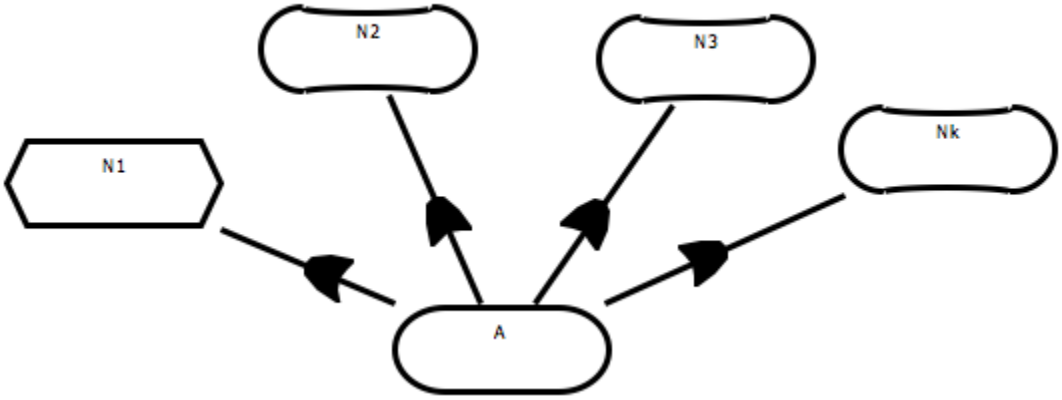

Figure 9 Node A depend on Node 1 to k

Constraint-based representation: **(N1 >= A) AND (N2 >= A).....AND (Nk >= A)**

3.2.4 N nodes with OR decomposition links

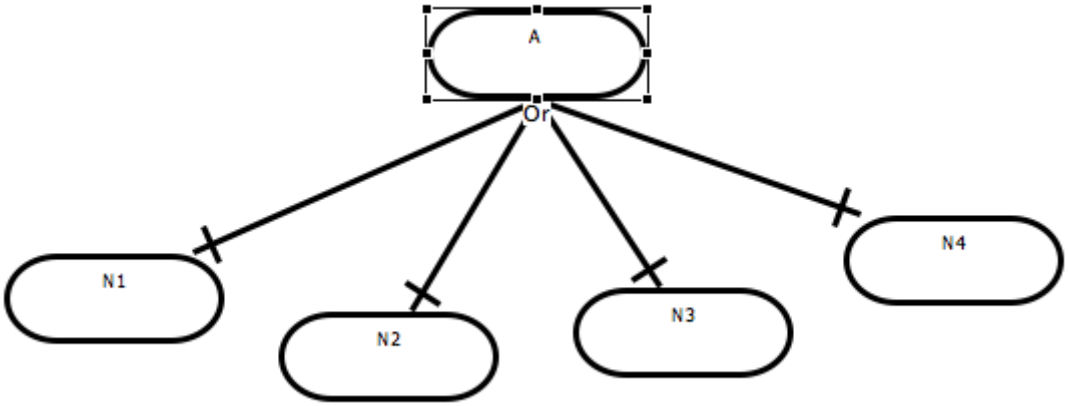

Figure 10 Node 1 to k OR compose Node A

Constraint-based representation: **A = MAX(N1, N2, ..., Nk)**

3.3. Quantitative Analysis - Hybrid Case

Consider a hybrid case where Node A has many incoming links from many other nodes as show below.

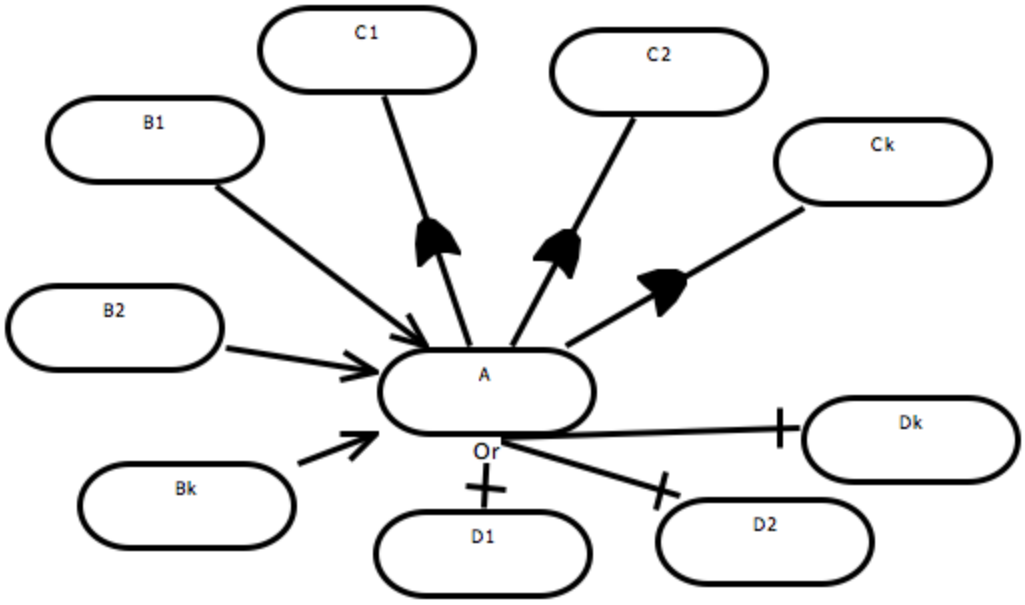

Figure 11 Many nodes with Node A

The constraint-based representation will be three components logically AND together.

(A <= C1) AND (A <= C2) AND ... AND (A <=Ck)
AND
A = MAX (-100, MIN(100, (MAX(D1, D2, ..., Dk) + X1 *B1 + X2 *B2 + … + Xk *Bk) / 100))

3.4. Usage of JaCoP

A GRL graph can be expressed with constraint-based mathematical equations and inequations, which naturally leads to our next step, i.e., finding a solution that satisfies all the conditions imposed by those formulas. As mentioned in section 2.2, JaCoP is our choice of constraint solver, specifically, JaCoP version 3 is used by including the jar file in our project lib directory. We treat each intentional element node in the GRL graph as a unique variable in JaCoP with as domain {-100, 100}. All the equations and in-equations derived from sub-graphs can be translated into JaCoP constraints, which later are used in finding a solution. Below is a list showing how that is being done:

- **A = B** corresponds to constraint **XeqY(A,B)** in JaCoP
- **B <= A** corresponds to constraint **XlteqY(B, A)** in JaCoP
- **C= Min(A,B)** corresponds to constraint **Min({A, B}, C)** in JaCoP
- **(A <= C) AND (B <= C)** corresponds to constraints **XlteqY(A, C)** & **XlteqY(B, C)** in JaCoP
- **C = Max(A,B)** corresponds to constraint **Max({A, B}, C)** in JaCoP
- **C = MAX (-100, MIN(100, (60 * A + B * 50) / 100))** needs to be modeled as multiple simple constraints like the above mentioned ones, as well as a couple of more advanced constraints like **SumWeight({A, B}, {60, 50}, AB}** and **XdivYeqZ(AB, 100, ABOver100)**. A few intermediate variables other than the A, B, C will also be needed in order to express the formula in terms of JaCoP constraints.

3.5. Algorithm

Given a GRL model with a set of intentional elements and their relations as defined by the links among them, we present here an algorithm to analyse the model and propagate values, which conforms with the model definitions & constraints, to each un-initialized node in the graph. At a high-level view, our algorithm can be outlined as the following pseudo code:

```
Algorithm GenericPropagation
Inputs GRLmodel:GRLspec, currentStrategy:EvaluationStrategy,
Evaluations:HashMap
Output void

Set allConstraints = ∅

for each element:IntentionalElement in GRLmodel.intElements {
    Set AndSet = ∅
    Set OrSet = ∅
    Set DependencySet = ∅
    Set ContributionSet = ∅

    // categorize children by type of links
    Analyze(element, AndSet, OrSet, DependencySet,
ContributionSet)
```

```
 // transform relations into CP constraints
    GenerateConstraints(element, AndSet, OrSet, DependencySet,
ContributionSet, allConstraints)

}
    //    JaCoP does its search here
    Calculate(GRLmodel, Evaluations, allConstraints);

    inject the result values back into evaluation map
    Populate(Evaluations);

return
```

AndSet, OrSet, DependencySet, ContributionSet contains the collection of intentional elements that has AND decomposition relation, Or decomposition relation, Dependent relation, Contribute relation, respectively, with **element** after the ***Analyze*** step.

```
Algorithm Analyze
Inputs element:IntentionalElement, AndSet:Set, OrSet:Set,
DependencySet:Set, ContributionSet:Set
Outputs void

for each link:ElementLink in element.linksDestination {
    IntentionalElement source = link.src
    if(link.type == AndDecomposition){
      AndSet.add(source)
    }else if(link.type == OrDecomposition){
      OrSet.add(source)
    }else if(link.type == Dependency){
      DependencySet.add(source)
    }else if(link.type == Contritution){
      ContributionSet.add(source)
    }
}
return
```

GenerateConstaints step will create the constraints involving **e** and the collections of the elements depending on the type of relations and the rules defined previously in section 3.1 and 3.2 .

```
Algorithm GenerateConstraints
Inputs element:IntentionalElement, AndSet:Set, OrSet:Set,
DependencySet:Set, ContributionSet:Set, allConstraints:Set
Outputs void

allConstraints.add(getConstraint(element, AndSet))
allConstraints.add(getConstraint(element, OrSet))
allConstraints.add(getConstraint(element, DependencySet))
allConstraints.add(getConstraint(element, ContributionSet))
return
```

During ***Calculate*** step the initialized intentional elements (if any) will be considered as constant constraints before doing the global search of the solution.

```
Algorithm Caculate
Inputs GRLmodel:GRLspec, Evaluations:HashMap,
allConstaints:Set
Outputs void

for each e:IntentionalElement in GRLmodel.intElements {
    int value = Evaluations.get(e)
    if(value != 0){
       allConstraints.add( getConstraint(e,value) )
    }
}
JaCoP.Search(allConstraints)
return
```

The solution (if any) will be updated to the **Evaluation** map that holds the (element,value) pairs in ***Populate*** step.

```
Algorithm Populate
Inputs Evaluations:HashMap
Outputs void

Set resultEvaluations = JaCoP.getSearchResult()

for each entry:<e:IntentionalElement, value:int> in
resultEvaluations {
    int originalValue = Evaluations.get(e)
    if(originalValue == 0){
       Evaluations.put(e,value)
    }
}
return
```

3.6. Rationale behind some decisions made

Intentional element nodes with initial values are included as “constant” variables in the CP search. It is not needed to have constants to be part of the search but it will not cause any additional performance issues as they will be the first set of variables to be grounded to a value in the domain, thus I included those constants for the completeness and easier to read purposes for later on development.

Leaf nodes are intentional element nodes which do not have any nodes depending on them or have any decompositions/contributions from any node. Leaf *Tasks* are special in GRL analysis in that they usually represent resources or alternatives that either choose to include, or not to include. As a result, it is favourable to limit leaf Task nodes to only have a value of zero (means not chosen) or one hundred (means chosen), unless they have initial values which means constants in the CP search. I implemented this special handling of leaf nodes in the project based on this observation.

We are only looking for one solution in the whole solution space, but the search will find all solutions that satisfy the constraints. In order to pick one

solution, I instructed the CP solver to return the minimum values for each variables. This heuristic can be easily changed pragmatically to other ones like in domain maximum values, median values, etc.

Last but not least, the GRL quantitative representation in jUCMNav uses zero as the default value for uninitialized nodes. To be able to tell whether there is no solution found or the solution is all nodes have zero values, I use a Boolean type variable called ***foundSolution*** to track it. Other developers can utilize this variable when they build other functionalities on top of this project.

Chapter 4. Validation

The accurateness of the algorithm solving unknowns in GRL models need to be validated and performance of the algorithm needs to be measured. Thus we made two test cases where the leaf nodes in the models have been initialized, i.e, other bottom-up algorithms in jUCMNav will be able to find solutions, which can be use to test the correctness of the algorithm proposed in this project. Also, two other test cases that cannot be solved by bottom-up algorithms will be ran against our algorithm but solutions will be verified by inspection. In both cases, we will measure performance related criteria.

4.1. Experiment Setup

Hardware & OS: Macbook Pro, OSX Snow Leopard

Figure 12 Hardware & OS

Software: Eclipse 3.5, jUCMNav 4.3, Java Version: 1.6

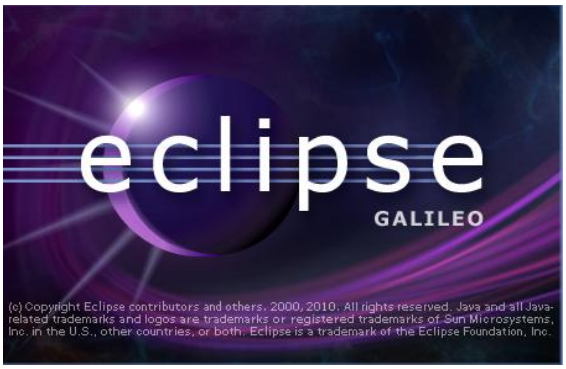

Figure 13 Eclipse Galileo

4.2. Experiment Approach

We used the following approach: Take a jUCMNav GRL model that is solvable by a bottom-up algorithm, run both types of algorithms on it (existing quantitative algorithm and new generic, constraint-based algorithm), and collect the measurements. Then, make another strategy where only top-level intentional element nodes are initialized, such that a bottom-up algorithm will not be able to find solutions, run both algorithms to verify that generic algorithm can solve the model and collecting the measurements.

4.3. Experiment Tasks

We choose two GRL models, each with two evaluation strategies where one strategy has all leaf nodes initialized and the other does not. For each model and each strategy we will try to evaluate twice, once with our new algorithm and once with a existing quantitative bottom-up algorithm.

1. A synthetic model with nodes from A to M

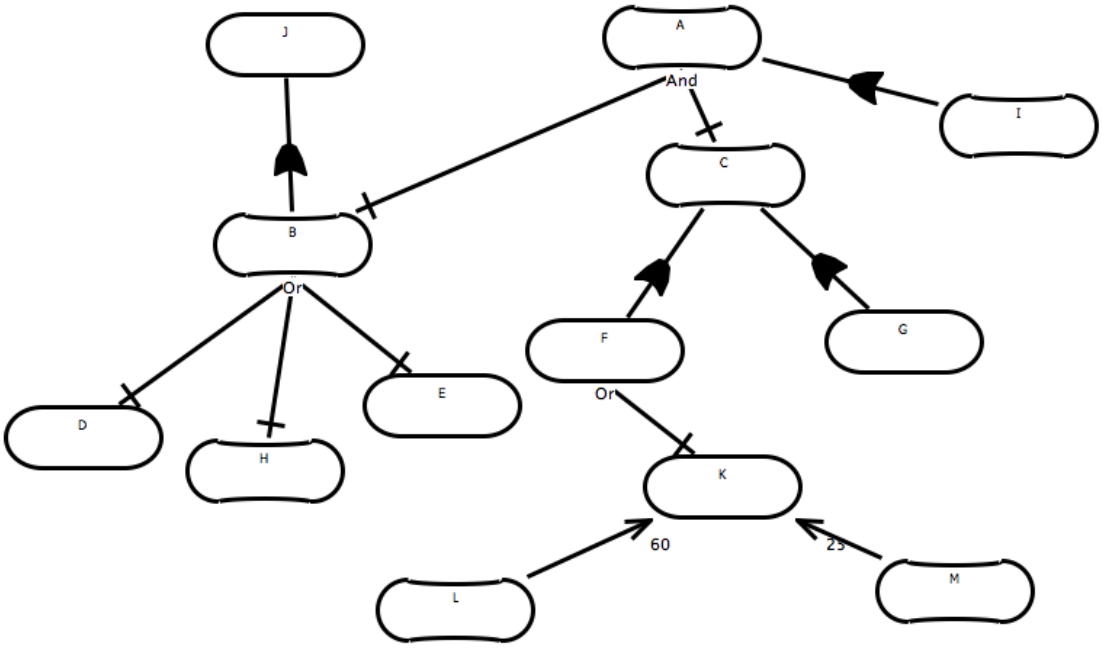

Figure 14 Synthetic GRL model

2. A conference paper management system model

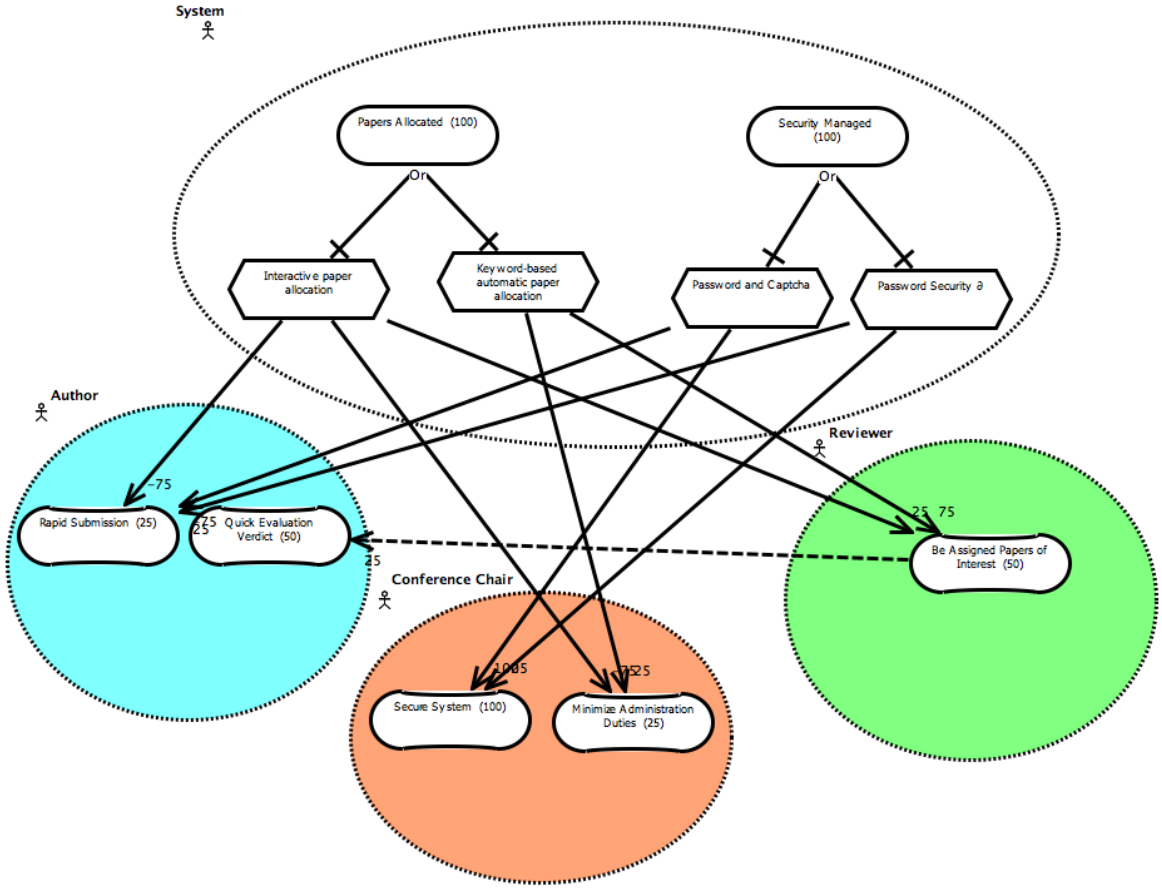

Figure 15 Conference Paper system model

4.4. Experiment Results

We list the comparisons of ability to solve the models and the performance measurements in the two tables below. We will use these notations:

S: Synthetic GRL model

C: Conference paper system model

L: Strategy where leaf nodes are initialized with evaluations values

R: Strategy where "root" nodes are initialized and leaf nodes are not initialized

Quantitative: Existing quantitative bottom-up algorithm

Generic: Algorithm proposed in this project

Table 1 Solvable by different algorithm

	S + L	S + R	C + L	C + R
Quantitative	Yes	No	Yes	No
Generic	Yes	Yes	Yes	Yes

Even when both type of algorithms solved the model, the solutions may not be the same if not all the leaf nodes are initialized. For e.g, in the conference paper system model, when we initialize two out of the four leaf nodes the solutions found by the two algorithm are slightly different.

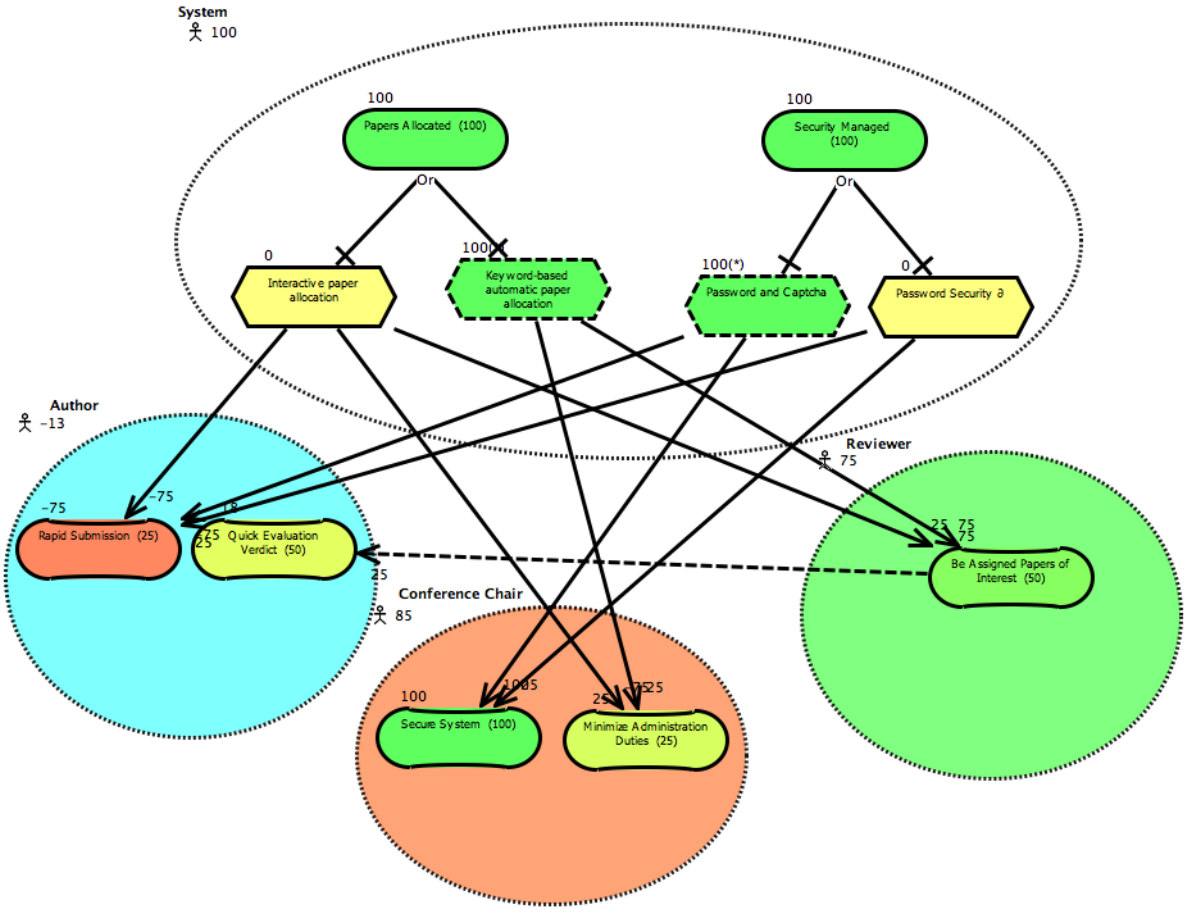

Figure 16 Solution found by quantitative algorithm

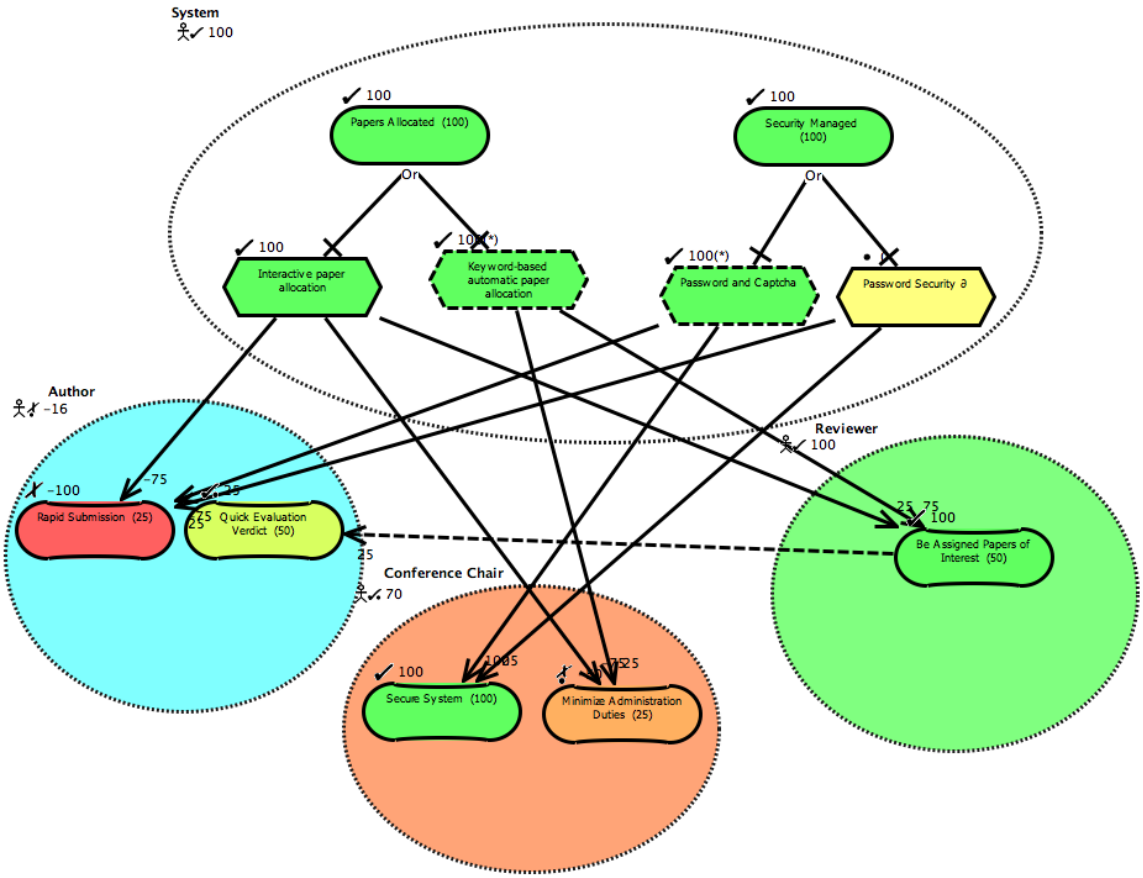

Figure 17 Solution found by generic algorithm

The time used is done by code instrumentation in EvaluationStrategyManager class before and after the invocation of algorithms.

Table 2 Time used by different algorithms (milliseconds)

	S + L	**S + R**	**C + L**	**C + R**
Quantitative	18	N/A	15	N/A
Generic	16	3	26	25

As shown, the generic algorithm will be able to solve some models and strategies that bottom-up algorithms cannot handle. The performance is usually on par with bottom-up algorithms in case both type algorithms are able to solve the problem.

In the generic algorithm, having more initialized nodes usually results in better performance. This is expected since we have more constant nodes in the CP search, thus the solutions space will be reduced drastically.

Not all strategies will have solutions for a model, which should not be surprising. First of all, we limit the solution domain to be [-100,100], more often, initialization of nodes could lead to conflicting constraints that will never be satisfiable at the same time. In these cases, no value of any nodes will be changed after the search is done, and an error message will be displayed on the console.

From time to time, one may run into a situation where search takes a long time. After all, the CP searching [6] we are performing here is attacking NP-hard problems with good heuristics.

Chapter 5. Conclusions

This report presented an implementation of generic propagation algorithm in GRL models.

5.1. Contributions

The report made the following contributions

- We have showed a quantitative analysis process to transform goal models into mathematical formulas, hence providing a generic and formal semantics for GRL models that subsumes existing bottom-up algorithms with operational semantics.
- We have introduced a constraint programming approach to solving the goal models as a system of predicates and unknowns.
- We have implemented our approach as an algorithm in the jUCMNav tool to help answer “is it possible” kinds of question against goal models.
- We have validated our algorithm by comparing the results to existing bottom-up algorithms.
- We have confirmed that the generic algorithm can solve goal models that can not be solved by bottom-up algorithm, if a solution exist.
- We have measured performance of generic algorithm and compared the statistics collected from bottom-up algorithms.

5.2. Future Work

This project serves as a prototyping work to assure that it is possible to have a generic algorithm that can propagate satisfaction values across the goal model.

- XOR type of links can be added by going through the same process of defining the mathematical meaning of the link then translated the relationship defined by the link into constraints and solve for solution.
- GRL actor evaluations only has limited support in the current algorithm. It is possible to provide a way to initialize actors in the future.
- Contribution tolerance support can be added (as in current propagation algorithms)

- Quantitative analysis is used in the generic algorithm but it is possible to extend it to support qualitative analysis and hybrid approaches in the future.
- We were looking for one solution in the whole solution space, the heuristic used is to select the minimal value in the domain when trying to determine the value for a node. This behavior can be changed to other value selection heuristics, and we can also build a UI component to allow the user to select one at run time. It might be possible to even have different heuristics locally so we can tell the CP solver to minimize one node and maximize another.
- While we do not see any type of constraints to directly support localized min or max for specific intentional elements in JaCoP, it might be possible to mimic this behaviour by providing a way to do as many as needed iterative searches that initialize those specific intentional elements at min/max values and increment/decrement during the iterations, if necessary.

References

1. Amyot, D., Ghanavati, S., Horkoff, J., Mussbacher, G., Peyton, L. and Yu, E. (2010) Evaluating Goal Models within the Goal-oriented Requirement Language. International Journal of Intelligent Systems (IJIS), Vol. 25, Issue 8, August 2010, 841-877.
2. Choco Solver. http://www.emn.fr/z-info/choco-solver. Accessed 02/23/2011.
3. Constraint Programming. http://en.wikipedia.org/wiki/Constraint_programming. Accessed 02/23/2011.
4. Eclipse platform, http://www.eclipse.org. Access 02/23/2011.
5. Emmanuel Letier, Axel van Lamsweerde: Reasoning about partial goal satisfaction for requirements and design engineering. SIGSOFT FSE 2004: 53-62
6. Francesca Rossi, Peter Van Beek, Toby Walsh: Handbook of constraint programming, volume 35. Elsevier; 2006
7. GRL, Goal-oriented Requirements Engineering, http://www.cs.toronto.edu/km/GRL. Accessed 02/23/2011.
8. ITU-T – International Telecommunications Union: Recommendation Z.151 (11/08), User Requirements Notation (URN). Geneva, Switzerland, November 2008.
9. JaCoP, Java Constraint Programming Solver, http://www.osolpro.com/jacop/. Access 02/23/2011.
10. jUCMNav 4.3. http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome. Accessed 02/23/2011.
11. Jennifer Horkoff, Eric Yu : Finding solutions in Goal Models: An Iterative Backward Reasoning Approach. ER'10, Proceedings of the 29th international conference on Conceptual modeling, Springer, 2010, 59-75.
12. Message Sequence Chart. http://www.sdl-forum.org/MSC. Accessed 02/23/2011.

13. M. Weiss, H. Mouratidis : Selecting Security Patterns that Fulfill Security Requirements.
14. Paolo Giorgini, John Mylopoulos, Roberto Sebastiani: Goal-oriented requirements analysis and reasoning in the Tropos methodology. Eng. Appl. of AI 18(2): 159-171 (2005)
15. van Lamsweerde, A. Requirements Engineering: From Craft to Discipline. In: Proc. 16th ACM SigSoft Int. Symp. on the Foundations of Software Engineering (FSE'2008), Atlanta, USA; 2008.
16. van Lamsweerde A. Requirements engineering: From System Goals to UML Models to Software Specifications. John Wiley & Sons; 2009. 712 p.