GrowingLeaf: Supporting Requirements Evolution over Time

Alicia M. Grubb, Gary Song, Marsha Chechik

University of Toronto, Toronto, Canada
{amgrubb, gary, chechik}@cs.toronto.edu

Abstract. Goal modeling and analysis techniques help stakeholders consider possible tradeoff alternatives in their requirements and answer "what if" questions about those alternatives. Software projects today exist in an ephemeral state, and current goal modeling approaches do not answer questions about model evolution and changes in actors' intentionality. We have developed a technique [8] that allows stakeholders to explicate the dynamic nature of their intentional elements, and ask questions about how this dynamicity affects project outcomes.
In this paper, we present *GrowingLeaf*, a new web-based goal modeling and analysis tool that implements this technique for iStar models. We discuss its support for modeling, simulation, and static analysis and illustrate it for a small example of making lunch.

1 Introduction

Goal modeling and analysis techniques [3, 6, 9] help stakeholders consider possible tradeoff alternatives in their requirements and answer "what if" questions about those alternatives. Software projects exist in an ephemeral state [7], and current goal modeling approaches do not answer questions about model evolution and changes in actors' intentionality, such as when a resource becomes available at a specific point in time, or a goal's satisfaction is cyclic in nature.

Running Example: Jake is interested in having a quick and inexpensive lunch. Generally, his lunch options consist of either making a sandwich or buying *fancy* pizza. If he makes a sandwich, he has to prepare one with bread and meat. Currently, Jake has bread that will expire soon, but does not have any beef or chicken, his favourite sandwich meats. With this information, Jake wants to answer the questions (1) "given that he currently has bread and can buy other items, what possible scenarios exist?", and (2) "what possible scenarios exist that eventually result in him having lunch?".

We have developed a technique [8] that allows stakeholders to explicate the dynamic nature of their intentional elements, and ask questions about how this dynamicity affects project outcomes[1]. We use static analysis and simulation to answer questions such as those proposed by Jake. In this paper, we present *GrowingLeaf*[2], a new web-based goal modeling and analysis tool that implements this technique for i* (iStar) models.

In the remainder of this paper we give an overview of the tool and its features (in Sect. 2), present the architecture (in Sect. 3), and discuss the tool's usability and development decisions/challenges (in Sect. 4).

[1] See [8] for a discussion of how this technique compares with other iStar reasoning techniques.
[2] http://www.cs.toronto.edu/~amgrubb/growing-leaf

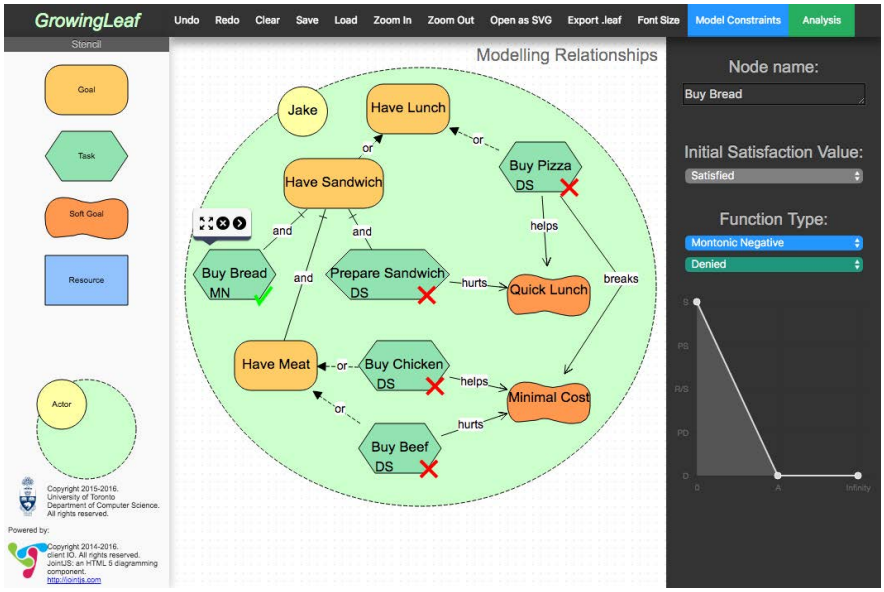

Fig. 1: Screenshot of GrowingLeaf with an iStar model of Jake's lunch plans.

2 Tool Overview and Features

Using our running example, we describe how GrowingLeaf is used to both draw iStar goal models and probe questions about model evolution and future evaluations of a specified goal. See [11] for an overview of iStar.

Modeling: We extend the static notion of an iStar model by allowing the qualitative satisfaction evaluation label (i.e., ✓, ✓●, ✗●, and ✗ in decreasing satisfaction order) of each intentional element (i.e., goals, tasks, resources, and qualities/soft-goals) to change over time. We capture this information by assigning each intentional element (intention) a dynamic function type. Basic dynamic functions include increasing, decreasing, constant, and stochastic. Compound functions consist of one or more basic functions that occur sequentially over multiple time periods (called *Epochs*). For example, the *Denied-Satisfied* (*DS*) function consists of two Epochs, a period of constant ✗ followed by a period of constant ✓, separated by an *Epoch Boundary* (EB). *Monotonic Negative* (*MN*), another compound function, is a period of decreasing satisfaction followed by a period of constant ✗, again separated by an EB. We create a unique symbolic constant for each combination of intention and EB. Users can also model more complex dynamics using the *User-Defined* function type. This allows the user to define arbitrary patterns of satisfaction, as well as model cyclic behaviour by creating functions with a repeating segment. See [8] for a complete list of dynamic functions.

GrowingLeaf allows users to build iStar *strategic rationale* models with actors, their intentions, dependencies, and intention links. The modeling view of GrowingLeaf is shown in Fig. 1 with Jake's lunch model in the centre canvas. The left panel contains the stencil of model elements, while the right shows attributes for the clicked element. Jake's top goal Have Lunch is decomposed into requisite tasks. He selects ✓ as the initial evaluation label for Buy Bread. All other leaf nodes in the model are set to ✗ (as shown). Using our dynamic function labels (appearing on the bottom left corner

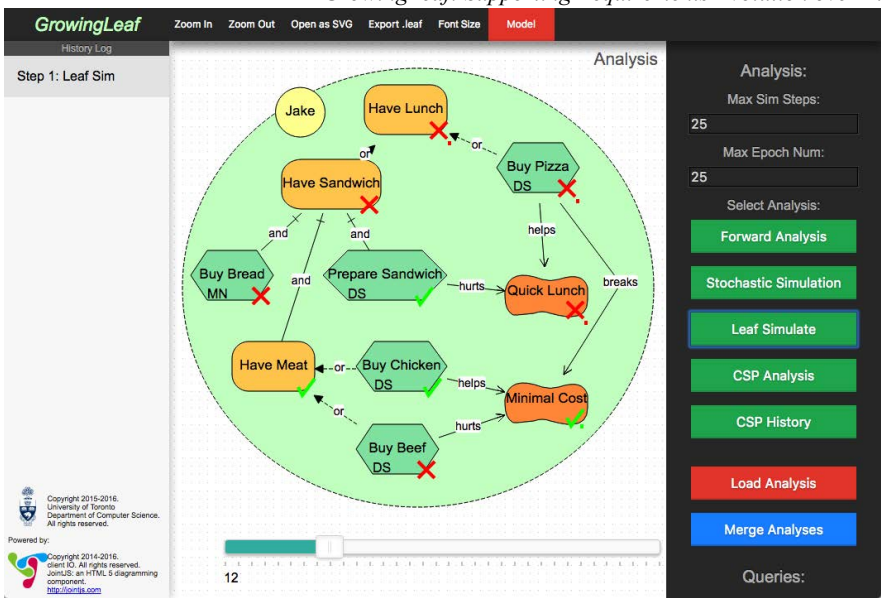

Fig. 2: Screenshot of GrowingLeaf Analysis view.

of each intention),he models Buy Bread as a *MN* function, while Buy Chicken, Buy Beef, and Prepare Sandwich are all modelled using the *DS* function. Since Buy Bread is currently clicked on the model, its attributes are shown in the right panel including a chart demonstrating the behaviour of the *MN* function.

Generally, users begin modeling by dragging actors and intentions (from the stencil on the left) onto the model canvas. When an intention is clicked, the right panel is populated and users can change the element's name, qualitative satisfaction evaluation label, and dynamic function type (with a chart demonstrating the behaviour to make dynamic functions more intuitive). These updates are then shown on the canvas. Dependency and intention links are created by dragging the $>$ icon from the source intention to the target intention. After selecting a link, its type can be changed in the right panel.

Analysis: With these dynamic functions, we generate simulation paths for goal models. A path is a series of models where between each step in the series intentions are updated based on their dynamics and new intention values are propagated throughout the model. We generate paths based on known initial satisfaction values, and desired intermediate/final satisfaction values (by encoding the model and dynamics as a Constraint Solving Problem). We use constraints to dictate relationships between symbolic constants (generated from intention EBs) for the purpose of improving analysis results.

GrowingLeaf's analysis view allows users to generate and view simulation results. To answer his first question about lunch, Jake selects Leaf Simulate in GrowingLeaf, with the initial model in Fig. 1. Fig. 2[3] shows the analysis view of GrowingLeaf with Jake's simulation results in the centre canvas. The slider, which appears at the bottom of the canvas, indicates that the model is showing time step 12 of the simulation. The simulation is now listed in the History Log in the left panel. At time step 12, Jake can

[3] Forward Analysis implements the analysis in [9], and Stochastic Simulation generates random values ignoring dynamic function types; both are outside the scope of this work.

see that Prepare Sandwich became ✓ despite Buy Bread having already transitioned to ✗. This simulation path does not make sense because the sandwich must be prepared before Jake's bread expires. Jake uses the constraint view to add the constraint that the EB of Prepare Sandwich must happen before the EB of Buy Bread (as modeled in Fig. 3). Jake generates a new simulation path with this added constraint, by returning to the analysis view and again selecting Leaf Simulate. With this added constraint Buy Bread becomes ✓ and Prepare Sandwich remains ✗, resulting in Jake's original goal to Have Lunch remaining ✗. To answer his second question, Jake selects CSP Analysis, and now the centre canvas shows him a path where Buy Pizza and Have Lunch become ✓. He then repeats this analysis by clicking CSP History and obtains a path that shows Buy Chicken and then Prepare Sandwich becoming ✓, again resulting in Have Lunch becoming ✓. CSP History allows users to generate a new path taking into account the previously generated path.

Generally, in the analysis view the user clicks an analysis type from the right panel. The analysis type is then shown in the History Log and the resulting analysis is shown on the centre canvas as a series of models with a slider at the bottom allowing the user to step through each time point. In the right panel, the user can also change the maximum number of simulation steps. We encourage users to iteratively ask questions over their models and permit stacking analysis results in the History Log. When an additional simulation is performed from the selected point on the slider, all satisfaction values up to the current time point are saved, and new satisfaction values are appended. Users can choose to merge previous analysis results forming a new sequence, by selecting the Merge Analysis button in the right panel. At any point, the user can return to the modeling view from either the initial or currently viewed state of the model. The constraint view (not shown) looks similar to the modeling view, but with dependency and intention links removed to enable users to add constraints visually with directed edges.

3 Tool Architecture

GrowingLeaf uses a simple client-server architecture. The front-end (client-side) is built using Javascript and HTML; the back-end (server-side) is implemented in Java. For analysis, we connect the front-end with the back-end through a Python CGI script, and an Apache HTTP Server. A high level overview of the architecture is shown in Fig. 4.

The front-end is built on top of *Rappid*, a modeling framework by *client IO* [4]. Rappid allows users to generate graphs using a drag and drop interface. *Backbone* is used to render customized side panels. In these side panels, GrowingLeaf allows users to specify constraints, relationships, and dynamics, as well as the type of analysis to be performed. Attributes specified in these side panels are stored in their corresponding objects in the model graph. The front-end handles all aspects of model creation and manipulation in order to eliminate the need for model storage on the server and reduce server calls. Our front-end extension is $\sim$13900 lines.

The back-end analysis, consisting of $\sim$5000 lines of Java code, uses a modified version of jUCMNav's systems architecture [3], and includes Java modules for producing both random simulation paths and creating constraint-based paths by interfacing with the JaCoP constraint solver [10]. When an analysis call is made by the client, all model objects are checked for appropriate syntax and then formatted into a text string. A server call is made, passing this string with the pre-specified type of analysis, and storing it a

text file on our server. After the computation, the tool generates a results file with each model intention corresponding to a row of satisfaction values, one for each time point. The results file is returned to the client side for rendering – see Fig. 2.

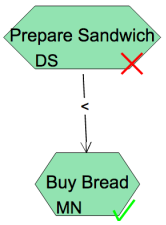

Fig. 3: Segment of lunch model showing the constraint between Prepare Sandwich and Buy Bread.

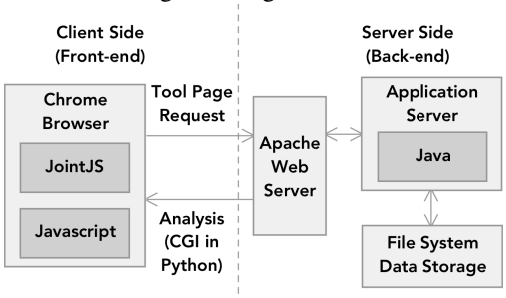

Fig. 4: Architecture of GrowingLeaf.

4 Discussion

Tool Usability: The tool is in its second version and has been used to model several non-trivial examples including the ones presented in [8], with further case studies underway. We have yet to conduct a usability study of the tool, but we have completed two rounds of user testing with SE graduate students, with and without exposure to goal modeling. None of the students had a formal exposure to our techniques. In the first round, we found that students had difficulty resizing intentional elements and had issues with their intuitive expectations of the 'enter' and 'backspace'/'delete' keyboard keys. They also had trouble understanding icons that appear when they clicked on a canvas element. We have since addressed many of these issues by disabling/simplifying elements. We did not have these same issues in the second round (with the exception of the 'delete' key problem); instead we found that students were not sure which analysis technique to use for the different questions they wanted to ask. We hope to mitigate this by adding descriptions of each analysis technique. Overall, everyone liked that the tool was web-based and found model creation and editing easy. This data is consistent with studies associated with the *Creative Leaf* tool[4] (which was built on top of the beta version of our tool[5]).We have completed the ethics approval process at our institution for an in-depth study of our tool and technique with graduate students who have training in iStar, and expect to complete the usability study shortly.

Design Decisions: Given the full array of iStar tools on the iStar wiki [1], we were hoping to add our analysis to an existing tool. We surveyed the available tools and reviewed a systematic comparison done in the literature [2]. None of the tools in active development/support appeared to have an easy way to extend their iStar meta-model and the ability to add icons/labels on top of intentions. Since we chose to make our own tool, we decided to create a web-based tool. We first built a beta version of our tool[5] to draw and evaluate goal models. We also introduced the concept of dynamic functions to represent temporal changes in the satisfaction value of intentions. Our original design

[4] http://creativeleaf.city.ac.uk

[5] http://www.cs.toronto.edu/~amgrubb/leaf

used a separate window for analysis results, but we observed that it was too taxing for users to switch between windows. When we added the constraints feature and investigated questions that took into account previous paths, we experimented with several layouts, ultimately settling on the one shown in Fig. 2.

Design Challenges: In developing our tool we encountered two major challenges. Since we chose to build a web-based tool, the first challenge was dealing with the many browsers and their versions. Thus, we limited our development to ensure that our tool works in Google Chrome first. Although we have experimented with our tool on mobile platforms, we do not intend to support them at this time.

When we wanted to add constraints between EBs, we were not able to show them on the canvas because the data model only contained a single type of link which we were using for dependency, decomposition, and contribution links. We had to revamp the architecture to add a second type of link. Given our substantial development effort prior to this discovery, we had to significantly refactor our codebase.

Future Work: Further work is necessary to evaluate the usability of our interface and the effectiveness of our tool with user studies. We want to make the tool compliant with the new iStar 2.0 Language Guide [5]. If our analysis achieves wider adoption, we will need to revamp the tool's architecture to make the communication between the front-end and back-end more efficient and allow for simultaneous server calls. Finally, we want to allow multiple users to simultaneously edit the same model from different browser sessions.

Acknowledgements: We would like to thank Jake Fear and the Software Engineering Group at the University of Toronto for their contributions to this work.

References

1. i* Wiki: i* Tools. `http://istar.rwth-aachen.de/tiki-index.php?page=i%2A+Tools`, 2015. Accessed: 2015-02-20.
2. C. Almeida, M. Goulão, and J. Araújo. A systematic comparison of i* modelling tools based on syntactic and well-formedness rules. In *Proc. of iStar'13*, pages 43 – 48, 2013.
3. D. Amyot, S. Ghanavati, J. Horkoff, G. Mussbacher, L. Peyton, and E. Yu. Evaluating Goal Models Within the Goal-Oriented Requirement Language. *Int. J. of Intell. Syst.*, 25(8):841–877, 2010.
4. Client IO. JointJS/Rappid - HTML 5 diagramming toolkit. `http://jointjs.com/rappid/tour`, 2016. Accessed: 2016-02-21.
5. F. Dalpiaz, X. Franch, and J. Horkoff. iStar 2.0 Language Guide. *arXiv:1605.07767*, 2016.
6. P. Giorgini, J. Mylopoulos, and R. Sebastiani. Goal-oriented Requirements Analysis and Reasoning in the Tropos Methodology. *Eng. Appl. Artif. Intell.*, 18(2):159–171, 2005.
7. M. W. Godfrey and D. M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 129–138, Sept 2008.
8. A. M. Grubb and M. Chechik. Looking into the Crystal Ball: Requirements Evolution over Time. In *Proc. of RE'16*, 2016.
9. J. Horkoff and E. Yu. Interactive Goal Model Analysis For Early Requirements Engineering. *Requirements Engineering*, 21(1):29–61, 2016.
10. K. Kuchcinski and R. Szymanek. JaCoP - Java Constraint Programming solver. `http://jacop.osolpro.com`, 2016. Accessed: 2016-02-21.
11. E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *Proc. of RE'97*, pages 226–235, 1997.