

Data-driven Requirements Modeling: Some Initial Results with i*

Aditya Ghose¹, Metta Santiputri², Ayu Saraswati³, and Hoa Khanh Dam⁴

Decision Systems Laboratory
 School of Computer Science and Software Engineering
 University of Wollongong, NSW 2522 Australia
 Email: {aditya¹, ms804², sa783³, hoa⁴}@uow.edu.au

Abstract

Requirements acquisition is widely recognized as a hard problem, requiring significant investments in time and effort. Given the availability of large volumes of data and of relatively cheap instrumentation for data acquisition, this paper explores the prospect of data-driven model extraction in the context of i* models. The paper presents techniques for extracting dependencies from message logs, and for extracting task-dependency correlations from process logs. The preliminary empirical results are encouraging.

Keywords: Requirements acquisition, data-driven model extraction, i* model

1 Introduction

The connection between requirements and the data that might be used to generate, understand and analyze requirements has been largely ignored in the literature. Yet the growing ubiquity of data, the ability to access large-scale sensor instrumentation and the availability of “big data” tools has thrown up significant opportunities for developing a new generation of data-driven requirements engineering (RE) tools. These opportunities come in many forms.

First, data can alleviate the well-known challenges associated with requirements acquisition/elicitation [15]. Organizations are often unable to leverage the benefits of conceptual modeling and the principled use of enterprise architecture because of the (often steep) investment required. The phenomenon is an instance of the *knowledge acquisition bottleneck* - a problem with an even longer pedigree [1]. Conceptual modeling is a time consuming human task of considerable complexity. We will argue in this paper that developing a capability to “mine” requirements from data can pay rich dividends. Earlier work [4] suggests that tools that extract “snippets” of models (or *proto-models*) by mining legacy text and model artefacts (these latter being in different notations) can significantly improve modeler productivity (with some empirical results pointing to about a two-thirds reduction in modeler effort).

Second, data-driven requirements monitoring provides the ability to improve the quality of requirements specifications, which in turn lead to improvements in the quality of the systems delivered. Exe-

cution data provides the basis for extracting requirements (which may be viewed as abstract descriptions of the data that these are mined from). Deviations between the mined requirements and those originally specified by stakeholders can flag problems. Similarly, we might cluster data associated with the particularly “desirable” (as determined by stakeholders) parts of execution histories, and extract requirements from these. The requirements thus obtained would represent more accurate encodings of stakeholder intent.

Third, clustering data associated with “undesirable” instances of execution histories (again, determined by stakeholders) can help us mine requirements anti-patterns. Within the context of a given RE exercise, these anti-patterns would identify “no-go” areas (i.e., requirements that lead to undesirable consequences).

Finally, the ability to establish an online, real-time correlation between requirements and data can help us use requirements models as dashboards.

What we have outlined above are effectively four distinct hypotheses about how data (specifically, behaviour histories) can deliver value in the requirements engineering exercise. In this paper, we put the first two of these hypotheses to the test. We focus on a well-regarded early-phase requirements modeling language of long standing - the i* notation [20]. The use of i* makes the case for data-driven requirements engineering more compelling, for several reasons. i* is particularly effective in modeling high-level strategic requirements, and also supports distributed goal modeling. Consequently i* serves as a natural representation of complex organizational contexts. This paper presents some initial steps toward an evaluation, by devising and evaluating techniques that permit us to (partially) “mine” i* models from execution data. We restrict our attention to mining dependencies and the tasks within the depender and dependee actors that each dependency is associated with. We present two techniques: the *Dependency Extraction (DE)* technique which mines dependencies from message logs and the *Task-Dependency Correlation Extraction (TDCE)* technique which mines the tasks/goals in an i* SR model that are associated with each dependency from process logs.

Our focus on message logs and process logs is realistic. Message logs are routinely maintained within the enterprise context. Sometimes, these manifest as email repositories, but our current work does not address the deployment of sophisticated NLP techniques that would be required to mine these. Instead, we use an abstract, generalized messaging format that resembles a number of industry-standard electronic messaging formats such as RosettaNet [17], ebXML [5] and a host of EDI formats. These are clearly easier to mine

Copyright ©2014, Australian Computer Society, Inc. This paper appeared at the 10th Asia-Pacific Conference on Conceptual Modelling (APCCM 2014), Auckland, New Zealand. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 154, G.Grossmann and M. Saeki, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

than natural language message logs, but nonetheless provide a useful basis for a proof-of-concept tool. Process logs are also routinely maintained by firms. A variety of business process management tools as well as bespoke process logging tools can be leveraged to obtain these. Unlike process mining tools, however, we do not seek to extract process designs from process logs, but instead mine for patterns of task activations that point to the existence of a dependency.

We also simplify matters by assuming that the only i^* models of interest are those that involve only goal dependencies. We keep softgoals entirely outside the purview of our current discussion. Techniques for mining other types of dependencies represent an important direction for future work.

These techniques only support the mining of *partial* i^* models, specifically inter-actor dependencies, and tasks/goals associated with each dependency. We present two different evaluations of these techniques. First, we evaluate their effectiveness (in terms of precision and recall) in mining partial i^* models from behaviour histories that simulate the execution of an initial complete i^* model. Second, we validate the hypothesis that it is possible to generate better quality (i.e., more accurate) models by mining behaviour histories of imperfect “as-is” contexts that have been filtered by stakeholders (to remove behaviour traces that are undesirable). Much more detailed evaluation is possible, and is the focus of future work, but our preliminary results are encouraging.

The rest of this paper is organized as follows. The next section provides background on the i^* notation. The following two sections describe the DE and the TDCE techniques respectively, in detail. We then provide an evaluation of this approach, in which we separately evaluate the DE and TDCE techniques and then bring them together in evaluating how user input into filtering behaviour histories can lead to more accurate models. We then provide a brief discussion of related work before presenting concluding comments and directions for future work.

2 Background

i^* [20] is a well-known requirements modelling language which describes the organizational context of an information system based on the notion of intentional actors. In modeling an actor in i^* , we specify its goals, the means available to achieve these goals and how other actors depend on it to achieve their goals. Actors depend on each other for goals to be achieved, tasks to be performed, resources to be furnished and performance measures to be optimized. Such dependencies are described in an i^* strategic dependency (SD) model. There are four types of strategic dependencies that may be specified in an i^* model. A goal dependency models situations where an actor depends on another actor to achieve a goal. A resource dependency exists when an actor relies on another actor to provide a resource. A task dependency suggests that an actor needs another actor to carry out a task. Finally, softgoal dependencies capture the non-functional properties of a model. As noted above, we will focus only on goal dependencies in this paper (and argue that all other dependencies, except softgoal dependencies, can be reduced goal dependencies in form or another).

An i^* strategic rationale (SR) model describes internal interactions between goals and tasks within each actor. Specifically, it shows how a task can be decomposed into subtasks, subgoals, resources and softgoals (i.e. they need to be performed or satis-

fied in order for the task to succeed). In addition, it describes means-ends links, which describe alternative ways to achieve a goal. It may also describe how tasks contribute to achieving softgoals (positively or negatively). Figure 1 shows an example of an i^* SR model for a meeting scheduler system which we adapted from [20]. There are three actors here: *Meeting Initiator*, *Meeting Participant* and *Meeting Scheduler*. There are a number of dependencies between the actors. For example, the Meeting Initiator depends on the Meeting Participant to achieve the goal of *Attends Meeting*. The SR model also shows the exact tasks that are involved in a dependency (these are of particular interest in this paper). For example, the task Obtain AvailDate of actor Meeting Scheduler depends on the task Find Agreeable Date Using Scheduler to attain goal EnterAvailDate. In the next sections, we will describe how we mine existing message logs and process logs to extract those dependencies between actors and their tasks.

3 The Dependency Extraction (DE) Technique

The Dependency Extraction (DE) technique is intended to mine message logs for i^* dependencies, and is based on the following intuitive observations. All dependencies manifest themselves in messages, such as a request from the depender to the dependee at the creation of a dependency, and a message in the reverse direction when the dependency is fulfilled. Hence, a message log that maintains a record of all messages (over a certain period) between the actors of interest represents a rich repository of clues about these dependencies. Message logs are ubiquitous. A corporate email repository can be viewed as a message log, although the messages are entirely unstructured. In many cases, messages are structured such as in a variety of Electronic Data Exchange (EDI) languages, or in more recent standards such as RosettaNet and ebXML. Our current approach assumes a structured message log. We use a generalized message format in our evaluation, inspired by (and representing the common core of) the messaging standards discussed above. For our purposes, a message log is a sequence of messages consisting, at a minimum, the following components:

- An *interaction ID*, which is used to identify a conversation or *interaction*, but not an individual message.
- *Sender ID*
- *Receiver ID*
- A *timestamp* which describes the time when a message is sent or received (we assume message transmission to be instantaneous).
- A *message type*, which would involve types such as requests, responses etc.
- A *message payload*, consisting of the semantic content of the message (which might be imperative or descriptive or a variety of other speech acts).

In the spirit of RosettaNet, we assume that all messages that involve responses to an initial message (that starts a conversation, such as a service request) refer to a unique ID generated by the initial message. We shall refer to the set of all messages pertaining to such a unique ID as an *interaction*, the unique ID as the *interaction ID*.

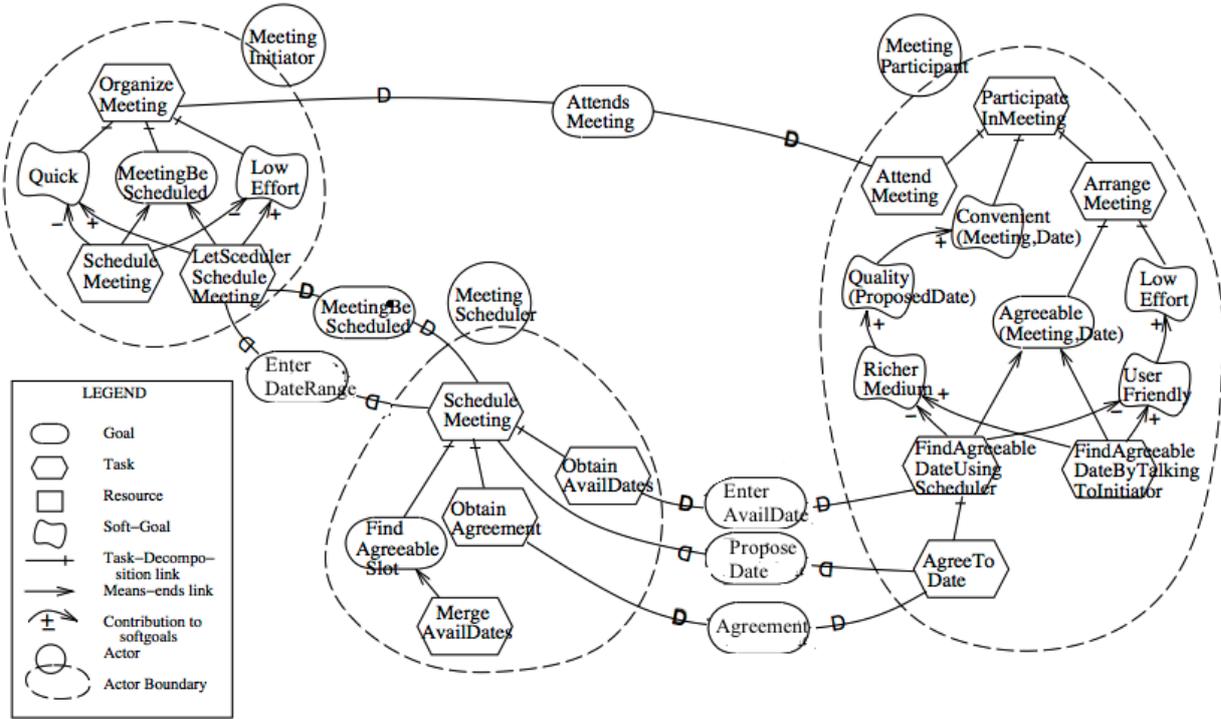


Figure 1: An i* SR model for a meeting scheduler system (adapted from [20])

Given the availability of unique interaction IDs, it is easy to extract complete interactions from a noisy message log where multiple interactions might be interleaved. Our next task is to extract the goal (e.g., the service request or product order) that is the object of the conversation. Goals are often represented in natural language using verb phrases. The information extraction techniques used for extracting verb phrases admit considerable complexity. For the purposes of our proof-of-concept evaluation, we assume an even simpler textual format, consisting of $\langle \text{verb}, \text{noun} \rangle$ pairs (such as buy book, supply product, assess claim etc.). Our technique for extracting these is as follows:

- We extract the set of all $\langle \text{verb}, \text{noun} \rangle$ pairs that appear in a given interaction.
- We annotate each element of this set with the number of messages that it appears in.
- We identify the element with the highest frequency and if it passes the threshold $k_{message}$, it referred to as the *goal designator* associated with the dependency.

We use the following procedure to identify dependencies from message logs:

- We partition the set of all interactions extracted from a message log into sets that share the same goal designator.
- We assume that a *significance threshold* $k_{interaction}$ is provided by the user. If a cluster of interactions (with the same goal designator) represents $k_{interaction}\%$ or higher of the set of all interactions, we treat that cluster as *significant* and indicative of a dependency.

4 The Task-Dependency Correlation Extraction (TDCE) Technique

In this section, we will present the the Task-Dependency Correlation Extraction (TDCE) technique that identifies the task in the depender actor and the task in the dependee actor that are associated with a given dependency. This information will be mined from the process logs.

The mining of task dependency correlations starts with process logs from different actors where the execution of each actor generates a distinct log. A process log consists of a list of tasks executed by an actor over time. Multiple process logs from different actors could be combined into one process log, as shown in the example in Table 1. This process log lists all tasks executed by all actors (either as the depender or as the dependee). By examining this log, we can observe that when actor i activates task a at time t_x , within some n units of time in the future, at time $t_x + n$, actor j activates task b . When this particular pattern of task activation become frequent (or satisfies a certain threshold), then there is an indication of a dependency between task a in actor i as the depender and task b in actor j as the dependee.

Each entry in the process log consists of:

- a *taskID*, which is used to identify certain task in an actor.
- a *timestamp* which describes the time when a task is activated by the actor. The timestamp of the first entry is t_0 which indicates the initial time - the timestamps of subsequent entries is increased each by one unit time.

The list of these entries will comprise a process log.

In this example, from the first row, we can observe that at the initial time t_0 , there are three different actors, each of which activates a task, i.e. actor A activates task a_0 , actor B activates task b_1 , and actor

C activates task c_0 . In the second row, the time is increased by one time unit to become $t_0 + 1$. At time $t_0 + 1$, actor A does nothing, actor B activates task b_4 , and actor C activates task c_1 , and so on.

In the process log shown in the example above, we can generate a good guess of which tasks participate in a dependency by examining the task sequence patterns that occurs in the process log. We adapt the GSP (Generalised Sequential Patterns) algorithm in order to mine this sequence pattern. It generates pairs of tasks sequences where the first task is from the dependor actor and the second task is from the dependee actor. For example a pair $\langle(a_0)(c_2)\rangle$ means that there is a pattern between $\langle(a_0)\rangle$ and $\langle(c_2)\rangle$ which indicates that there is a dependency between those two tasks with task $\langle(a_0)\rangle$ as the dependor and task $\langle(c_2)\rangle$ as the dependee. Then it will determine how frequent this pattern is in the log by counting the number of occurrence of each pair. This number of occurrence of each pair is called its support and the predetermined threshold is the minimum support.

The GSP (Generalised Sequential Patterns) algorithm was proposed by Srikant and Agrawal (1996). The algorithm takes as input a process log which consists of set of tasks ordered by time. It finds all sequences of tasks whose support is greater than the minimum support threshold specified by the user. In addition to the minimum support threshold, there are timing constraints that must be satisfied, namely the maximum time difference between the earliest and latest task activation and the minimum and maximum gaps between adjacent task. We leverage this algorithm, providing as input a process log ordered by time and obtaining as output all sequential patterns in the log.

The algorithm makes multiple passes over the log. The initial constraint for this part is that we are only interested in results that consist of two tasks, because our aim is to discover dependencies that relates pairs of tasks. Therefore we limit the pass to $k = 2$. The first pass of the algorithm finds all sequences with a single task in it along with their occurrence count (support). The output is 1-task long sequences or L_1 . On the second pass, the algorithm generates 2-tasks-long candidate sequences C_2 with L_1 as its seed. This is motivated by the fact that for a sequence to be frequent, all of its subsequences must also be frequent. As the support counts are determined, the sequences with support greater than the determined threshold are included in L_2 .

There are two main phases that are explained in detail below, in terms of how candidates are generated and how their support are counted.

1. *Join phase.* In this phase, we generate all candidates starting from candidates of length 1. The process is straightforward as all the tasks that were in the process log are placed in this set of candidates L_1 . To generate candidates

Actor	<i>actor</i>	<i>actor</i>	<i>actor</i>
Time	A	B	C
t_0	a_0	b_1	c_0
$t_0 + 1$	—	b_3	c_1
$t_0 + 2$	a_0	b_2	c_2
$t_0 + 3$	a_2	b_0	—
$t_0 + 4$	a_1	b_1	c_3
$t_0 + 5$	a_0	b_3	c_2

Table 1: Example of process log

of length 2, L_2 , a task from L_1 is joined with another that is also in the L_1 . If i and j are tasks belonging to L_1 , then task j is added to i . But for all candidates of length 2, there is one more constraint i.e. any two tasks in a dependency must not happen at the same time. Therefore any candidate dependency relating two tasks that activate at the same time must be excluded from the result. Initially task j should be added as a task-set $\langle(i j)\rangle$ and as a separate task $\langle(i)(j)\rangle$, but because of this constraint, we only add j as a separate task.

In our example from Table 1, we start with all candidates of length 1, L_1 . It would consist of $\langle(a_0)\rangle$, $\langle(a_1)\rangle$, $\langle(a_2)\rangle$, $\langle(b_0)\rangle$, $\langle(b_1)\rangle$, $\langle(b_2)\rangle$, $\langle(b_3)\rangle$, $\langle(c_0)\rangle$, $\langle(c_1)\rangle$, $\langle(c_2)\rangle$, and $\langle(c_3)\rangle$.

Next we need to eliminate these candidates according to the value of minimum support in the prune phase.

2. *Prune phase.* We eliminate candidates according to our constraints:

- (a) Since dependencies relate pairs of tasks from two different actors, any pattern that contains tasks from the same actor must be excluded from the result.
- (b) All the candidates with a support value lower than minimum support is excluded from the result.

Note that constraint (a) is only applied to 2-task-long candidates and is not applied to 1-task-long candidates. On the other hand, constraint (b) is applied in both cases.

Returning to our example, we have generated L_1 , and now all tasks in the L_1 must be examined against constraint (b). For this example, we set the minimum support as 2, which means that for any task or set of tasks to be classified as frequent, it must occur at least two times. In the log in Table 1, for actor A there are three different tasks (a_0 , a_1 , and a_2). The support for these tasks are 3, 1, and 1 respectively. Because the support for a_1 and a_2 are less than the minimum support, they do not included in the sequence of 1-tasks. We repeat this for all tasks and the result is shown in the first column of Table 2.

We continue to search for all candidates of length 2 by repeating the join and prune phases. This step is illustrated in Table 2. In the join phase, we start with the first candidate $\langle(a_0)\rangle$ in the first column. Thus, all sequences of form $\langle(a_0)(X)\rangle$, where X is any task, are searched. Recall that we do not search for $\langle(a_0, X)\rangle$ because it implies that the two tasks occurs at the same time. By combining $\langle(a_0)\rangle$ with the second candidate $\langle(b_1)\rangle$, we find 2-task-candidate $\langle(a_0)(b_1)\rangle$. A similar procedure is repeated for all sequences of the first column. All 2-task-long candidate sequences generated in the join phase are shown in the second column.

Next, according to constraint (a) in the pruning phase, we begin by determining whether the two events are executed by the same actor (if they do, then the sequence is eliminated). For example, in the second row, in sequence $\langle(b_1)(b_3)\rangle$, both tasks are executed by the same actor, namely actor B . Therefore the sequence is eliminated by applying constraint (a). Thus the remaining candidates for the second row are

Frequent sequences of length 1	Candidates of length 2	
	after join	after pruning
$\langle a_0 \rangle$	$\langle a_0(b_1) \rangle, \langle a_0(b_3) \rangle, \langle a_0(c_2) \rangle$	$\langle a_0(c_2) \rangle$
$\langle b_1 \rangle$	$\langle b_1(a_0) \rangle, \langle b_1(b_3) \rangle, \langle b_1(c_2) \rangle$	$\langle b_1(a_0) \rangle, \langle b_1(c_2) \rangle$
$\langle b_3 \rangle$	$\langle b_3(a_0) \rangle, \langle b_3(b_1) \rangle, \langle b_3(c_2) \rangle$	—
$\langle c_2 \rangle$	$\langle c_2(a_0) \rangle, \langle c_2(b_1) \rangle, \langle c_2(b_3) \rangle$	—

Table 2: Candidate generation

$\langle b_1(a_0) \rangle$ and $\langle b_1(c_2) \rangle$. The same also applies to sequence $\langle b_3(b_1) \rangle$ in the third row. We then determine the support count for each of the remaining candidates. The first candidate $\langle a_0(b_1) \rangle$ has support count 1, which is less than the minimum support - it is therefore eliminated. Next candidate, $\langle a_0(b_3) \rangle$, also has support count of 1, and is also eliminated. Candidate $\langle a_0(c_2) \rangle$ has support count of 2 - it is therefore included in the result. In the second row, we have two remaining candidates, $\langle b_1(a_0) \rangle$, and $\langle b_1(c_2) \rangle$. Both have a support count of 2 and are thus included in the result. We repeat this procedure for the rest of the candidates. The result is shown in the third column of Table 2.

The result patterns of this algorithm are all sequential patterns that occur between two tasks in the process log. For our process log example in Table 1, these are $\langle a_0(c_2) \rangle$, $\langle b_1(a_0) \rangle$, and $\langle b_1(c_2) \rangle$.

5 Evaluation

The purpose of the evaluation exercise is to establish the following:

- The Dependency Extraction (DE) technique and the Task-Dependency Correlation Extraction (TDCE) technique generate reasonably reliable results.
- Both these techniques can be leveraged to improve the quality of i^* models (assessed in terms of how closely a model corresponds to the “ideal” model, and hence to the reality being modeled) by leveraging user tagging (or filtering) of the logs recording the behaviour of an “as-is” system or process that the target system is intended to replace. Since i^* is also particularly effective as a domain modeling tool, an improvement in model quality might also entail obtaining a better representation of the context in which the target system is to be situated (or even more generally, a better model of the organizational context).

Our evaluation involved the generation of simulated behaviour histories (message logs plus process logs) given an i^* model. To achieve this, we *randomly executed* these models, the sense described below. For each distinct dependency in an i^* model, we generated a large number of *interactions* (specific numbers in the following subsections), with configurable levels of noise (thus we had noisy messages within interactions, and we had entirely noisy interactions that would not point to any reasonable goal dependency). The non-noise components involves messages and interactions that were deliberately constructed to conform to the i^* model at hand. The sum total of these interactions provided the message log that we mined. We similarly generated process logs by randomly selecting tasks/goals from the i^* model and allocating random timestamps to them. We, however, ensured that each dependency in the model was reflected at least once in the process log. In other words, if there was a dependency relating task a_i in actor A to talk

b_j in actor B in the model, we would ensure that the process log contained at least one entry for task b_j at a time point after that for task a_i .

5.1 Evaluation of the DE technique

We started with the i^* model shown in Figure 1, originally used in [20]. The model consists of 3 actors and 6 dependencies.

To simplify the process of obtaining *goal designers*, we assume that they consist of $\langle \text{verb}, \text{noun} \rangle$ pairs. We permit *goal designers* to also consist of $\langle \text{verb}, \text{verb} \rangle$ since some verbs in the past tense resemble nouns (participle adjectives or normalizing an adjective). When processing the payload, we extracted the pair by getting the first $\langle \text{verb} \rangle$ in the payload and the first $\langle \text{noun} \rangle$ following said verb.

We used the Stanford Log-linear Part-Of-Speech Tagger v3.2.0 [8] for tagging message payloads. The tagger takes a sentence such as *This is a sample sentence* and assign parts of speech, e.g., noun, verb, adjective, etc, like so *This_DT is_V B Zap_T sample_NN sentence_NN*. These tags conform to the Penn Treebank Tagset [14] where tag starting with $_V$ is verb and tag $_N$ is noun. So we can parse our payload with these tags to find the $\langle \text{verb}, \text{noun} \rangle$ pattern.

We performed experiments with 4 parameters. $n_{message}$ describes the proportion of noise messages in the complete message log (all non-noise messages permitted the extraction of the correct goal designer). $n_{interaction}$ describes the proportion of noisy interaction in the set of all interactions in the message log, where a noisy interaction is one which does not lead to any single identifiable goal designer. $k_{message}$ and $k_{interaction}$ are as defined in Section 4. We initially created a message log with no noise (i.e., neither noisy messages nor noisy interactions) consisting of 7381 interactions. Setting $k_{interaction} = 10\%$ and $k_{message} = 10$, we were able to extract all of the dependencies, as shown in Table 3.

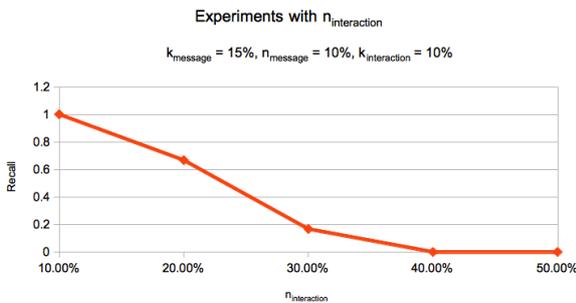
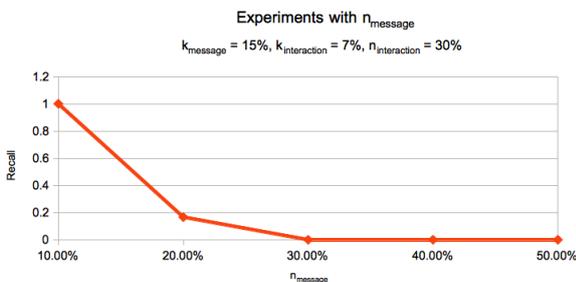
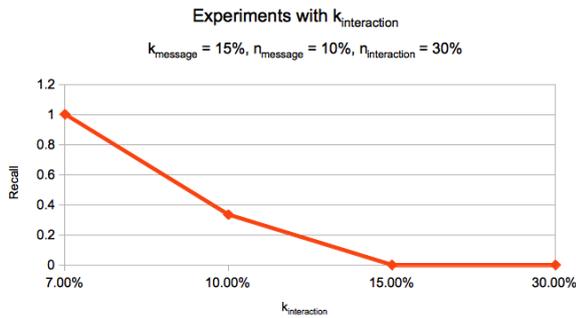
We plotted the performance of the DE technique (in terms of recall - there were no false positives and hence precision was always 1) against each of these parameters. We show 3 of the 4 results below (the final graph was omitted due to space considerations). As expected, recall decreases as the amount of noise increases (i.e., as $n_{message}$ and $n_{interaction}$ increase). Similarly, recall decreases as we get more selective in identifying dependencies (i.e., as $k_{interaction}$ increases). These results were generated using message logs that were between 25000 to 30000 messages long, with about 2000 interactions. The results were consistently the same.

5.2 Evaluation of the TDCE technique

Given a set of tasks as the input of our tool, we use two sets: the set of expected dependencies which were in the input model and the set of dependencies actually discovered in the result. Recall is defined by the number of correct dependencies discovered by our

Depender	Dependee	Goal Designator	Interaction Percentage
Meeting Scheduler	Meeting ParticipantA	Enter AvailDates	15.13%
Meeting Scheduler	Meeting ParticipantA	Agreement	18.02%
Meeting Initiator	Meeting ParticipantA	Attends Meeting	17.18%
Meeting Scheduler	Meeting ParticipantA	Propose Date	15.76%
Meeting Initiator	Meeting Scheduler	Meeting BeScheduled	17.55%
Meeting Initiator	Meeting Scheduler	Enter DateRange	16.37%
		total interaction	7381

Table 3: Results using non-noisy logs with $k_{interaction} = 10\%$ and $k_{message} = 10\%$



the minimum support. We performed separate experiments for each of those variable by varying the value of one variable and keeping the other fixed.

From the result, the recall is 1.0 indicating that every dependency in the expected result set is discovered. This can be explained by the fact that if there is a dependency between two tasks, there are frequent patterns between those two tasks in the log and it will be detected.

On the other hand, in addition to the expected dependencies, there are other dependencies that were discovered in the actual result but were not in the input model. This might happen because one factor that might affects the result is the interleaved of the entries in the task activation log. For example lets assume that we have a log consisting of two actors (let say actor A and actor B) with one dependency between these two actors (between task a_0 of actor A and task b_0 of actor B). Since the algorithm will find all the sequence in the log, there can be two different dependency discovered depends on the order of the entry in the log. This is illustrated below.

Actor	<i>actor A</i>	<i>actor B</i>
Time		
t_0	a_0	
$t_0 + 1$		b_0
$t_0 + 2$	a_0	
$t_0 + 3$		b_0

Table 4: Example 1

Actor	<i>actor A</i>	<i>actor B</i>
Time		
t_0	a_0	
$t_0 + 1$	a_0	
$t_0 + 2$		b_0
$t_0 + 3$		b_0

Table 5: Example 2

technique divided by the total number of expected dependencies in the result.

In this evaluation, we execute the input model including all of the dependencies. The task activation log will be created with the depender, dependee, the tasks and the dependencies from the input i^* model as the expected result along with other tasks from the input model as noise. The task activation log for each actor was created randomly but we deliberately input the execution of every dependency. Then we ran this log against our tool.

There are two inputs that we will use as variable in this evaluation of the TDCE technique: (a) the number of entries in the task activation log; and (b)

In example 1, the result would be two dependencies $\langle(a_0, b_0)\rangle$ and $\langle(b_0, a_0)\rangle$, while the second example the dependency would be $\langle(a_0, b_0)\rangle$. Therefore there are possibilities that any dependency might show up in the log although it was not in the model

The minimum support count for our evaluation in the first scenario is fixed at 1.0 which means that support count = $1.0 * \text{the number of execution}$. For example if we execute 10 times, then the support count is 10, so that any pattern that occurs 10 times or more is included in the result. For the second scenario, we use a task activation log with 2000 entries. The two graphs in Figure 3 show the precision of our technique in two different scenarios. As can be seen in Figure

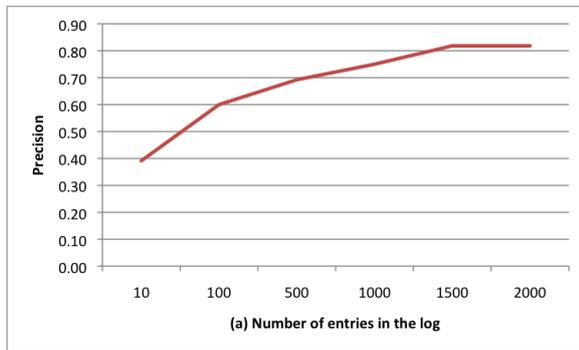


Figure 2: Precision relative to log size

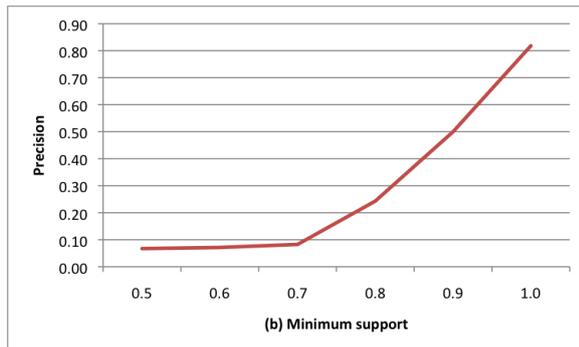


Figure 3: Precision relative to min support

3(a), with small log, the precision is not very accurate (given just ten entries, precision is only 0.39). But with a larger log, the precision is increasing up to 0.82. Precision remains relatively constant at 0.82 when the log reach 1500 entries or more. While in Figure 3(b), the higher minimum support will give more accurate result. For example, given the support of 1.0, precision reach up to 0.82 but precision drop to 0.5 when support is set to 0.9 and keep decreasing until under 0.1 with support of 0.7 or less. Hence the result suggest that both the number of the entries in the task activation log and the minimum support are very influential to the accuracy of the result. Therefore if we want to get more accurate result, we can do two things, either increase the volume of the data or increase the minimum support count.

Since in this evaluation we artificially created the task activation log and deliberately executed all dependencies in the input model, we acknowledge that they would not be representative for all possibilities of task activation log in practice. For instance, there might be a case where a dependency in the input model was not executed at all or was executed but in a number of times which was lower than the minimal support. In such cases, our technique might not be able to detect it.

Another limitation of this technique involves settings where multiple dependencies exist between the same pair of actors. Our current approach works well if we have a guarantee that only one dependency would exist between a given pair of actors. Thus, when we determine that a pair of tasks are related via a dependency, we are able to leverage the DE technique to identify what the goal designator for that dependency is. In the case of multiple dependencies between the same two actors, the DE technique would suggest multiple goal designators, while the TDCE technique would suggest multiple task pairs, but we would not have the wherewithal to associate the task

pairs with the goal dependencies identified by the DE technique.

5.3 Improving requirements quality: Evaluation

A key contribution of this work is the ability to achieve data-driven improvements in the quality of requirements models. There are two approaches to this that we explore. In the first, we explore settings where the user is able to describe the ideal *behaviour* (for our purposes, a *behaviour* will be described via a combination of a *message log* and a *process log*) of the system in question. We extract models from these idealized behaviours, as opposed to the noisy behaviours that have been the focus of the previous parts of the evaluation. In the second approach, we explore settings where the requirements engineering exercise is conducted in the context of an existing, “as-is” system or process(es), the behaviour of which we are able to log. The user filters this (potentially imperfect) behaviour generated by the existing system/process (by removing entries from the message and process logs that (in the perception of the user) represent manifestations of imperfect behaviour, and our machinery extracts models from these filtered logs. The evaluation involved a trained *i** modeler, who was asked to generate a model (Figure 4) which was not revealed to the research team. This model played the role of the “ideal” model against which the quality of the extracted models was evaluated. The quality of an extracted model was evaluated by either: (1) assessing how closely it conformed to the user’s “ideal” model (which was revealed to the research team after the model extraction phase was completed) or (2) obtaining input from the user suggesting that some dependencies that existed in the user’s intuitive understanding of the domain (and had been manifested in the idealized behaviours supplied by the user, but not in the “ideal” model) has been discovered by our machinery.

User-generated idealized behaviours: With the model in Figure 4 in mind, the *i** modeler gave us a message log with 12 entries (each of which was a request message) and a process log with 35 entries. Our machinery then extracted a partial *i** model with the following characteristics. Of the 23 dependencies in the original user model, we discovered 19 dependencies (relating the correct pairs of actors and tasks). We also extracted 9 new dependencies that were not part of the user’s original model. Figure 5 shows the model that was extracted. The bold lines denote false positives and the dashed lines denote false negatives.

User-filtered “as-is” behaviours: In this part of the evaluation, the *i** modeler revealed the idealized model to the research team. This was used to generate 5 behaviours (i.e., 5 sets of \langle message-log, process-log \rangle pairs). The message logs varied in length from 2 to 10 messages per log. The process logs varied in length from 9 to 13 entries. We additionally generated 5 incorrect behaviours, by randomly selecting 1 dependency in each case and randomly changing either the depender or dependee actor, or the source or target task. We then interleaved the 5 correct and 5 incorrect behaviours and presented these to the *i** modeler. Our intent was to simulate the execution of an imperfect system/process, whose behaviour would be represented by the interleaved logs. The *i** modeler was then asked to remove from the message and process logs entries that did not correspond to the intuitions that were represented in the idealized model. We then applied our machinery to extract a partial *i** model from the filtered logs. We discovered 19 of

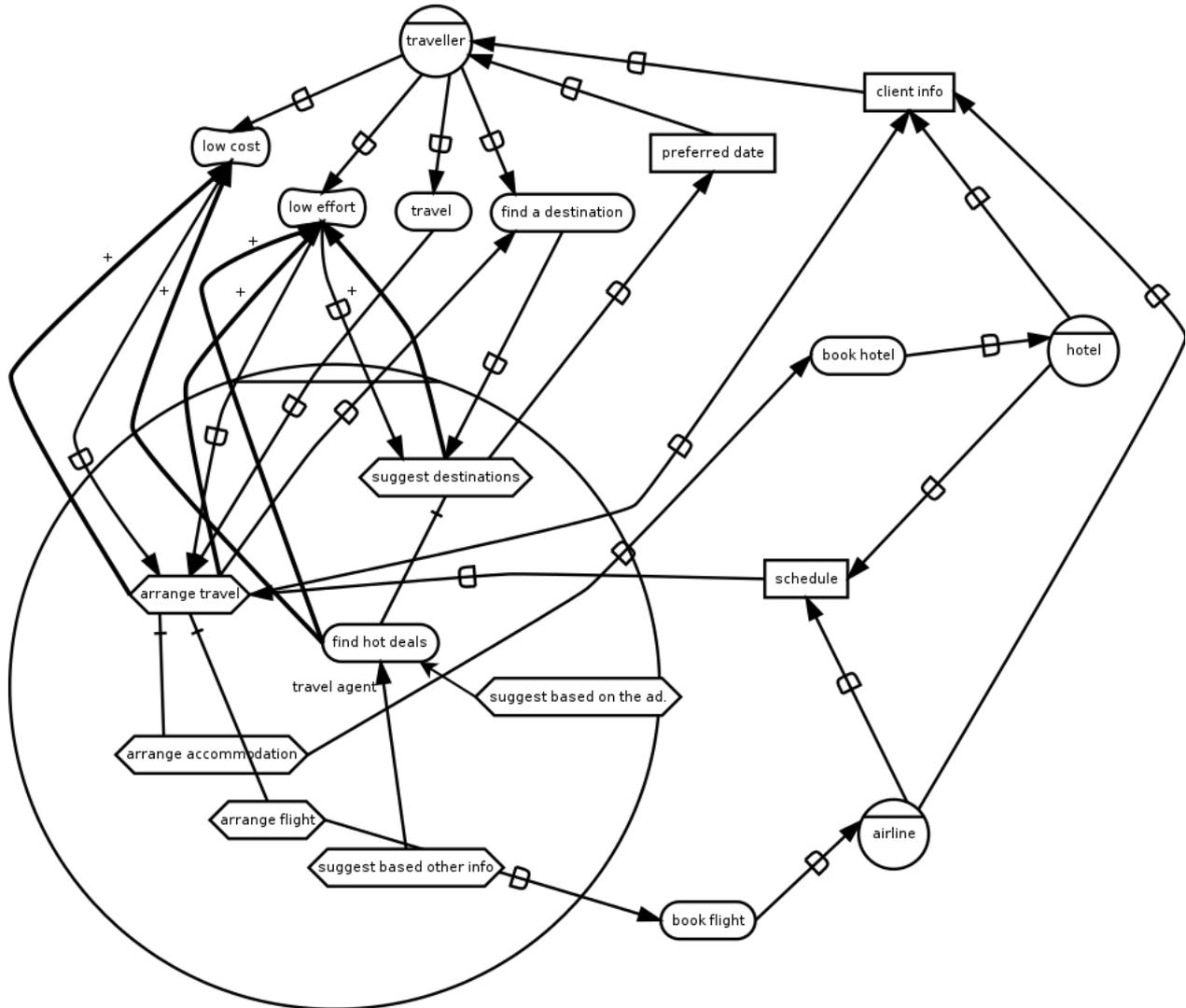


Figure 4: Model from user

the original 23 dependencies in the idealized model (this was an identical result to the evaluation using user-generated idealized behaviours) and 10 new dependencies. Of these 10 new dependencies, 6 were distinct to the dependencies extracted in the evaluation using user-generated idealized behaviours. The *i** modeler also suggested that 9 of the new dependencies discovered were largely in accord with his intuitions about the domain being modeled, but had not been reflected in the idealized model that he had initially generated. This suggests that this approach can help surface implicit requirements via the filtering of noisy behaviours. The extracted model is shown in Figure 6 below. As before, the bold lines denote false positives and the dashed lines denote false negatives.

As discussed in the previous section, the existence of multiple dependencies between the same pair of actors in this model prevented us from correlating the task pairs generated by the TDCE technique with the dependencies generated by the DE technique. The net upshot was that we had several “unnamed” dependencies. Nonetheless, discovering the existence of dependencies, even in the absence of goal designators, provides valuable insights.

Overall, this part of the evaluation suggests that there is merit in the general idea of using this machinery to improve the quality of requirements extracted, although the machinery missed some depen-

dencies and generated some false positives.

6 Related Work

The research reported in this paper is related in some ways to the existing body of work on *process mining* [18] [19], in that we also use process logs as one of several data sources. However, what we do with process logs is entirely different. Unlike process mining, we generate task-dependency correlations from this data. A proposal to leverage web logs to support non-functional requirements elicitation [16] shares some intuitions with our work (but uses different techniques as well as different inputs and outputs). A body of existing work on extracting requirements from natural language can inform the extraction of goal designators (see, for instance, [2] [3]), although our current evaluation uses a simpler proof-of-concept implementation. The Business Intelligence Model (BIM) [11] bears some relation to our proposal in its ability to serve as a data-driven dashboard for the enterprise. Approaches to run-time adaptation, such as in [6], concept discovery [12], ontology extraction [9] and model-based diagnosis [7] are relevant. The literature on requirements change (e.g., [10]) is also relevant. We expect that the literature on social network analysis [13] might provide techniques of relevance to this

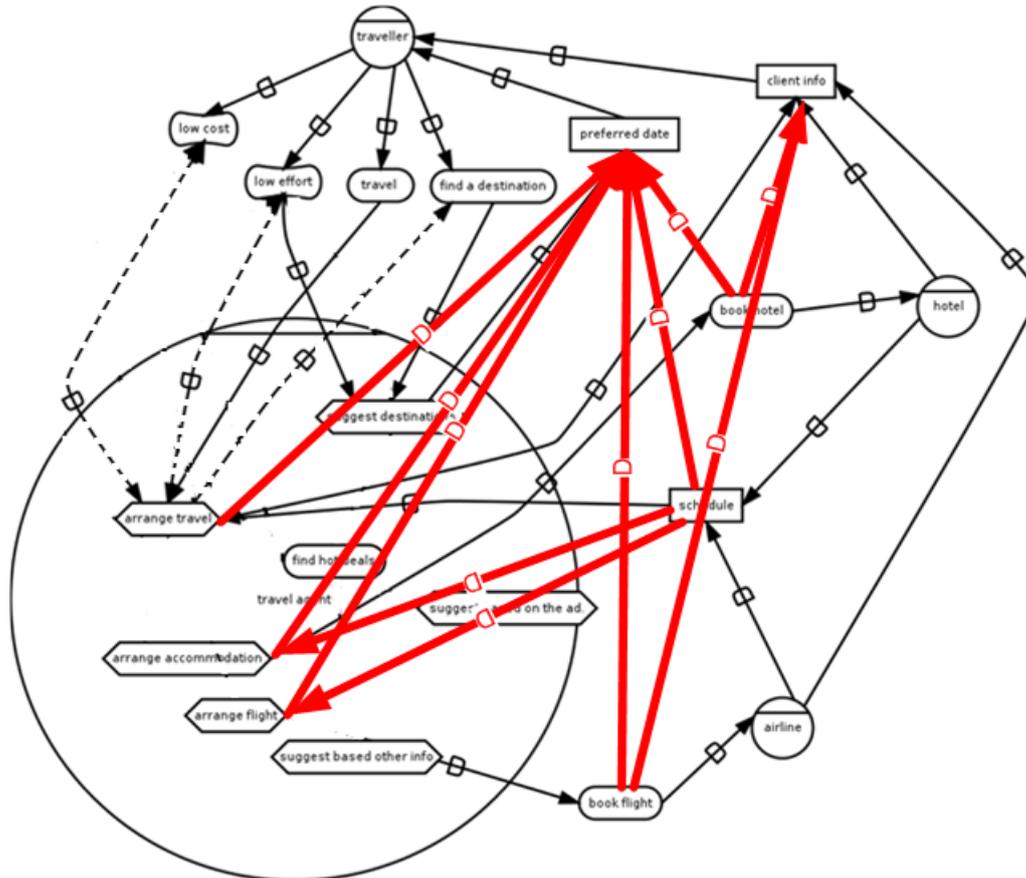


Figure 6: Extracted model-2

- [10] Shinpei Hayashi, Daisuke Tanabe, Haruhiko Kaiya, and Motoshi Saeki. Impact analysis on an attributed goal graph. *IEICE TRANSACTIONS on Information and Systems*, 95(4):1012–1020, 2012.
- [11] Jennifer Horkoff, Alexander Borgida, John Mylopoulos, Daniele Barone, Lei Jiang, Eric S. K. Yu, and Daniel Amyot. Making data meaningful: The business intelligence model and its formal semantics in description logics. In Robert Meersman, Hervé Panetto, Tharam S. Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi, and Isabel F. Cruz, editors, *OTM Conferences (2)*, volume 7566 of *Lecture Notes in Computer Science*, pages 700–717. Springer, 2012.
- [12] Markus Kirchberg, Erwin Leonardi, YuShyang Tan, Sebastian Link, RyanK.L. Ko, and BuSung Lee. Formal concept discovery in semantic web data. In Florent Domenach, DmitryI. Ignatov, and Jonas Poelmans, editors, *Formal Concept Analysis*, volume 7278 of *Lecture Notes in Computer Science*, pages 164–179. Springer Berlin Heidelberg, 2012.
- [13] David Lazer, Alex Sandy Pentland, Lada Adamic, Sinan Aral, Albert Laszlo Barabasi, Devon Brewer, Nicholas Christakis, Noshir Contractor, James Fowler, Myron Gutmann, et al. Life in the network: the coming age of computational social science. *Science (New York, NY)*, 323(5915):721, 2009.
- [14] Mitchell P. Marcus, Beatrice Santorini, and Mary A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1994.
- [15] John McDermid. Requirements analysis: Problems and the starts approach. In *Requirements Capture and Specification for Critical Systems*, *IEE Colloquium on*, pages 4–1. IET, 1989.
- [16] Odorico Machado Mendizabal, Martin Spier, and Rodrigo Saad. Log-based approach for performance requirements elicitation and prioritization. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 297–302. IEEE, 2012.
- [17] Laura Gibbone Paul. Rosettanet: Teaching business to work together. *Oct*, 1:4, 1999.
- [18] Wil MP Van der Aalst and AJMM Weijters. Process mining: a research agenda. *Computers in industry*, 53(3):231–244, 2004.
- [19] Boudewijn F van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP van der Aalst. The prom framework: A new era in process mining tool support. In *Applications and Theory of Petri Nets 2005*, pages 444–454. Springer, 2005.
- [20] Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, RE '97*, pages 226–, Washington, DC, USA, 1997. IEEE Computer Society.