Semi-Automated Linking of User Interface Design Artifacts

Said S. Elnaffar and T.C. Nicholas Graham

Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6

Abstract. User centered design involves the creation of design artifacts such as task and architecture models. It is increasingly accepted that such artifacts cannot effectively be created separately, but instead coevolve incrementally, so that information obtained from the development of one artifact contributes to the development of the others. In user interface development, different people with different backgrounds typically participate in the development of these artifacts. Consequently, communication is important for coevolution. This research demonstrates how different design artifacts can be linked semi-automatically. We illustrate this technique using Adligo, a computer-based tool for generating links between the User Action Notation (UAN) task model and the Clock architectural model. Our results show that in two case studies, we were able to generate 90% of possible links with error rate of 0% to 12% with limited human assistance.

1 Introduction

Numerous design activities contribute to the usability of an interactive system [1]: the production of *task models* helps to record the human activities that a computer system is meant to support, *scenarios* and *task-oriented specifications* record how users perform these tasks via a particular user interface, while *software architectures* provide a high-level view of the system's implementation.

An ongoing trend in processes for interactive system design has been to use information from each of these design activities to support the others. For example, task-oriented specifications can be used to automatically generate user interface architectures [14] and user interface dialogue specifications [13], or even to completely generate user interfaces [11, 15].

Even when the relation between design activities is not this explicit, it is helpful for designers carrying out one activity to have access to information from others. For example, Cockton and Clarke have shown the importance of explicitly linking documents capturing context of use of interactive systems to design documents [3]. In our earlier work with the Vista environment [1], we showed how linking behavioural design artifacts (such as task models and task-oriented specifications) to constructional artifacts (such as code) can help relate the differing points of view of HCI designers and software engineers. In Vista,

users simultaneously see up to four styles of design docuemtent in a visual browsing environment. Clicking on part of one document highlights highlights related parts of other documents. This allows developers to answer questions such as, what tasks may be impacted by the modification of a particular architecture component, or whether two similar action sequences in a UAN [10] specification are implemented by the same mechanism. Brown and Marshall have since extended this work to link user interface scenario documents to implementation design documents [2]. Linking approaches therefore help in relating information that may be presented from different points of view, allowing information from different design documents to be more easily used by other designers and developers.

The difficulty with linking approaches is that they typically require large numbers of links to be specified manually. For example, Vista is capable of automatically linking artifacts grouped exclusively within the behavioural or constructional domains, but requires a user to provide links that bridge the gap between these two points of view. Cockton and Clark's approach requires links to be explicitly made using an LD Relationship Editor [3]. Hand-specifying links is tedious, time-consuming and error-prone. For example, in the two case studies presented in section 5, approximately 100 links in each needed to be coded by hand using the Vista system. Furthermore, as design artifacts evolve throughout the life cycle of the interactive system, links continually need to be changed, again by hand. In order for link-based approaches to be practical, therefore, some kind of automation of the process of finding and maintaining links is required. Automating the generation of links is challenging, however, as different design artifacts are typically developed by different people, perhaps using incompatible terminology, and perhaps involving informal components such as English language text.

This paper presents a mechanism for the semi-automated generation of links between task-oriented specifications in the User Action Notation (UAN) [10] and architecture models expressed in the Clock architecture style [8]. A user must provide a small set of rules to guide the link generation process. As shown in section 4, these rules are presented in a simple tabular format called a *dictionary*. This approach has been implemented in *Adligo*, a tool which inputs a UAN specification, a Clock architecture and a dictionary, and outputs a set of links suitable for browsing with the Vista environment. As shown in the case studies of section 5, when provided with fewer than ten user-specified rules, Adligo was capable of generating 90% of links found by a human, with an error rate of 0% to 12%. With 16 rules, 100% of links were found, with no errors.

This paper is organized as follows. Section 2 presents an example application, and motivates the utility of developing links. Section 3 introduces Adligo's rule-based solution to link generation, while section 4 discusses Adligo's implementation. Finally, section 5 reports on the results of two case studies used to evaluate the effectiveness of Adligo.

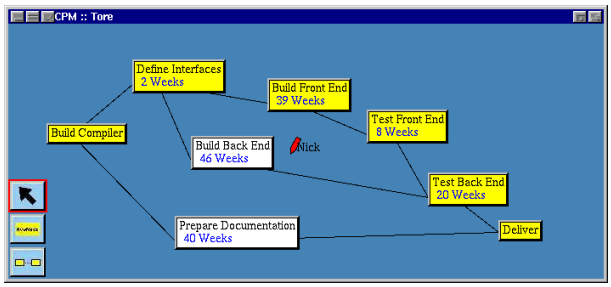

Fig. 1. User interface of a groupware critical path planning application [6].

2 Example: Critical Path Planning

In order to motivate the problem of link generation, we will use the example of a groupware critical path planning application [6]. Using this application, we will informally discuss some of the difficulties of automatically generating links. We will then use the critical path planner to illustrate Adligo's dictionary, allowing semi-automatic generation of these (and other) links.

Several design artifacts (collected in [7]) contributed to the development of the critical path planner. The system is based on a task model adapted from Dilworth [4]. Planners carry out two basic tasks – breaking up the project into a network of job steps ordered by their dependencies, and allocating resources to these job steps. Figure 1 shows the user interface of a system supporting these tasks.

A UAN task-oriented specification was developed to show how each of the planning tasks can be carried out using the interface of figure 1. Figure 2 shows one of the tables from this UAN specification, describing the task of repositioning one of the job step nodes in the critical path network. In order to reposition a node, the user first moves the mouse pointer over the node, clicks on it, drags the mouse, and releases. As the mouse is moved, the node follows the mouse pointer. As the node is moved, it is *locked*, so that none of the other users can move the node at the same time. Locked nodes are shown with red text on other users' displays, so that the other users can see that they are not permitted to move that node. The full UAN description contains 18 such tables [7].

Figure 2 also shows a view of the software architecture of the critical path planner. The architecture is based on the Clock architecture style [8], a layered extension of the Model-View-Controller (MVC) architecture [12]. In this architecture, the system is decomposed into *components*, responsible for I/O at some point of the display, and *ADT's* responsible for maintaining system state. The links shown in the figure reveal the relation between the architecture and the system it implements.

For example, the mouse click actions (`Mv/M^`) that start and end the movement of a node are handled by the `mouseButton` method of the `browseNode` component. Similarly, the movement of the node itself (`~[x,y]`) is handled by

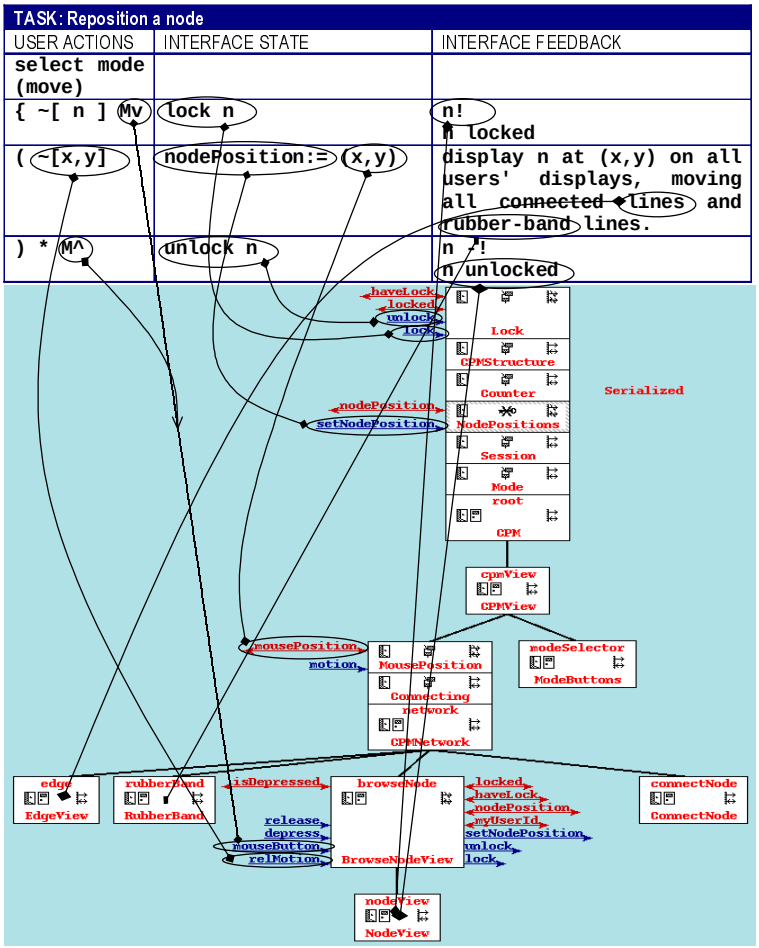

Fig. 2. Example links derived by Adligo.

the `relMotion` method. Locking of the node is handled by the methods of the `Lock` ADT. Modifying the position of the node (`nodePosition:=(x,y)`) is handled via the `setNodePosition` method of the `NodePosition` ADT. The current position of the mouse pointer (`(x,y)`) is obtained via the `MousePosition` ADT.

These links can be visually browsed using Vista [1], helping developers to associate task-oriented specifications with user interface implementations. However, it is clear that such links are highly tedious to generate by hand. It is not feasible to derive such links completely automatically – the developers of the UAN specification and of the software architecture have used different terms to describe the same parts of their artifacts, and the UAN specification contains informal English prose that is not amenable to automatic processing.

	DICTIONARY: Standard		
Rule #	RULE SCOPE	UAN PATTERN	CLOCK ARCHITECTURE PATTERN
S2		Mv	mouseButton
S3		M^	mouseButton
S6		~[$(x), $(y)]	motion | relMotion
S21		$(obj)+=1	increment$(obj) \| inc$(obj) | increment
S26		currentPos	mousePosition
S27		(x,y)	nodePosition | userPosition
S28		$(anyVar) :=	set$(anyVar)

	DICTIONARY: The CPM Planner		
Rule #	RULE SCOPE	UAN PATTERN	CLOCK ARCHITECTURE PATTERN
C1	Create a new node	select mode	Network
C2		mode	button
C6	Reposition a node	n	nodeView->browseNode
C9		currentMode:=	setMode
C10		mode	Mode
C11		connect $(obj1) to $(obj2)	setConnectionTarget
C12		lines	edge.view
C13		rubber-band	rubberBand.view
C14		solid line from n1 to n3	edge.lineFrom

Fig. 3. Selected rules extracted from the standard and custom dictionaries used to link design artifacts in the critical path planning application.

The next section shows how Adligo provides a simple rule-based mechanism that allows links to be semi-automatically derived. As will be shown in section 5, 90% of the 108 links that a human found in the CPM example were derived using only 8 rules in the Adligo dictionary.

3 Semi-Automated Generation of Links

The key to being able to find links between the behavioural task-oriented specification and the constructional user interface architecture is to bridge the difference in point of view of the two forms of specification. A single component might contribute to the implementation of several tasks; similarly, a single task may be carried out through the use of multiple components.

The basic approach to generating links is to provide a set of *linking rules* in a dictionary. Figure 3 shows a subset of the rules used with the critical path planning application. The rules are divided into a *standard* dictionary which is included for every application, and a *custom* dictionary, which provides rules specific to a particular application.

Rules establish a correspondence between parts of a UAN specification and parts of a Clock architecture. For example, rule S2 in figure 3 establishes that the pattern `Mv` in a UAN specification corresponds to the method `mouseButton`

in a Clock architecture. This rule states that whenever the `Mv` user action is encountered in a UAN specification, there should be some `mouseButton` method in the architecture that implements this user action. This is a *rule* since the correspondence applies whenever the `Mv` symbol is encountered. As shown in figure 2, rules S2 and S3 generate links showing that in the *Reposition a Node* task, the `Mv` and `M^` user actions are handled by the `mouseButton` method of the `browseNode` architecture component.

3.1 Multiple Targets

Sometimes, a pattern in a UAN specification may correspond to different locations in a Clock architecture, depending on implementation choices made by the developer. For example, motion events in Clock may be treated as absolute or relative to the current position, handled either by a `motion` or `relMotion` method. Therefore, rules must permit a UAN pattern to match a set of possible architecture patterns.

As shown in rule S6, multiple architecture patterns may be combined using a disjunction ("`|`") symbol. The UAN pattern specifies that the general form in UAN for moving to a new screen location is `~[`x`,`y`]`, where x and y are some identifiers. In this rule, the symbols `$(x)` and `$(y)` represent variables that may be matched to arbitrary identifiers in the UAN specification. The architecture pattern establishes that mouse motion user actions may be handled by `motion` or `relMotion` methods.

In figure 2, rule S6 generated the link between `~[x,y]` to the `relMotion` method implemented by the `browseNode` architecture component.

3.2 Rules with variables

Very powerful rules can be written using variables. For example, a common pattern of correspondence between UAN specifications and Clock architectures is that assignment to some value in UAN is implemented via a *set* method in the architecture. For example, in figure 2, the UAN `nodePosition := (x,y)` is implemented via the method `setNodePosition` in the `NodePositions` ADT.

Rule S28 searches for correspondences of this general form: the UAN pattern `$(anyVar):=` is matched to a method in the architecture of the form `set$(anyVar)`. Variables in the UAN and architecture patterns must be unified in finding a match.

Similarly, rule S21 identifies a common correspondence case, showing how incrementing a value may be matched to any of a set of increment methods.

This ability to use variables, unification, and multiple architecture patterns allows simple rules in the standard library to identify common usage patterns, generating many links without requiring any custom rules at all.

3.3 Custom Rules

The standard rules described above help provide links for built in user actions of UAN (such as user actions corresponding to mouse and keyboard input), and

for commonly observed usage patterns in specifications. However, real specifications normally require additional rules, primarily to resolve inconsistencies in terminology between the UAN designer and the user interface implementer.

For example, rule C11 in figure 3 shows how the English text `connect` x `to` y corresponds to the method `setConnectionTarget`. Such correlations are not difficult for a human to provide, but would be difficult to find automatically.

Sometimes, such correspondences must be scoped to a particular task. For example, in the UAN task *Reposition a Node* (figure 2), the symbol n refers to some node in the critical path network. It would be dangerous to establish a global rule stating that n always refers to a node, since this may not be true for every task specification, and might lead to the generation of incorrect links. Rule C6 of figure 3 shows how a rule establishing this correspondence can be scoped to apply only to a specific task.

Similarly, rules may be scoped to apply to a particular method or component in the architecture. For example, rule C14 explicitly specifies that `solid line from n1 to n3` corresponds to the `lineFrom` method of the `edge` component. Similarly, rule C6 explicitly specifies the location of the target component in the architecture (i.e., the component `nodeView` is specified to be a child of the component `browseNode`.)

Restriction of the scope of rules can have the effect of creating rules that are so specific that they generate only one link. Such rules can be useful in specifying explicit links when the generality of the rule-based approach is not appropriate.

4 Implementation

We now briefly describe how the Adligo tool uses the dictionary described in the last section to generate links. For an in-depth description of Adligo's algorithms, see [5]. The generation of links includes the following steps:

1. Find a *match* for one of the rules in the UAN specification;
2. Find the *context* for the match;
3. Starting from the context component, find the best match for the architecture pattern.

The notion of a context component is key to the generation of links. A context is a part of the user interface to which user actions are applied. The standard UAN mechanism for changing a context is the user action `~[c]`, which specifies that the user moves his/her pointer into the context of screen region `c`. Following a context change, Adligo assumes that subsequent user actions are applied to the new context.

Consider for example the sequence `~[n] Mv` of figure 2, as interpreted using the rules shown in figure 3. The following sequence of rule applications leads to the generation of a link to the `mouseButton` method of the `nodeView` component:

1. The user action `~[n]` changes the context. `n` matches the rule C6, setting the context to the component `nodeView->browseNode`.

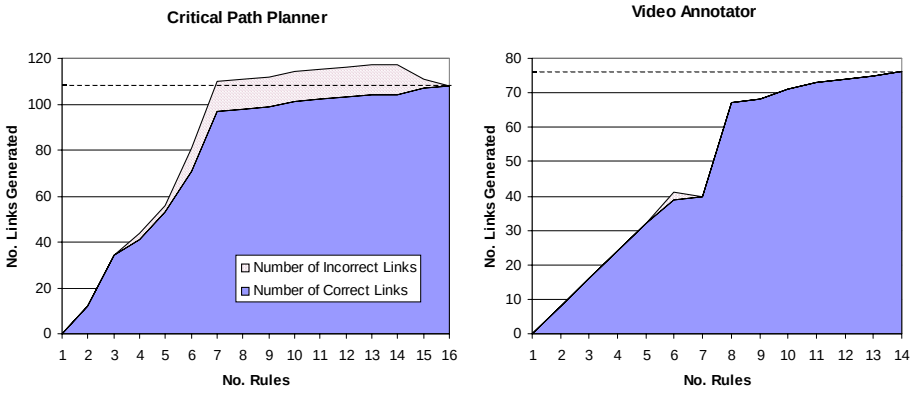

Fig. 4. Results of CPM and Video Annotator case studies. The dashed horizontal line shows the number of links found by hand.

2. The user action `Mv` matches rule S2. Adligo attempts to match a method `mouseButton` within the context of the `nodeView->browseNode` component.
3. The parent of the `nodeView` component implements the `mouseButton` method; the link is therefore generated to the method `browseNode.mouseButton`.

5 Evaluation

In order to evaluate the effectiveness of our semi-automated technique for linking user interface artifacts, we performed two case studies on existing applications. These applications are the critical path planner [6] described in section 2, and a groupware video annotator tool [9], both designed and developed prior to this work with Adligo. Both case studies involved over 100 lines of UAN, divided into approximately 20 tables. The implementation architectures of the examples consist of 13 and 30 components, containing over 100 methods. Therefore, the examples are small enough for us to derive links by hand for purposes of comparison, but large enough to expose how well Adligo functions.

In order to create a correct set of links against which to compare Adligo's performance, we derived by hand a set of links between the UAN task-oriented specification and the Clock architecture. We then ran Adligo to mechanically derive a set of links. Through this process, Adligo's dictionary was tuned to successively improve the links generated. Finally, we compared the hand-derived links to Adligo's links. For each case where the links differed, we either decided that the hand-derived links were incorrect and updated them, or decided that the generated links were incorrect, and recorded an error.

Figure 4 shows the number of correct and incorrect links generated by Adligo as rules were successively added to the dictionary. In both examples, adding eight rules was sufficient to generate approximately 90% of available links, while 16 rules was sufficient to generate 100% of the links found by hand.

The percentage of generated links that were incorrect ranged between 0% and 12% in the two applications, finally dropping to 0%. As the number of rules increases, the error rate initially increases (as the number of generated links increases.) As rules are added, the error rate then decreases as the rule set becomes more precise.

5.1 Analysis

These results show that, at least for these two examples, Adligo is highly successful at generating links. With a small number of rules, in excess of 90% of available links can be automatically generated, with an error rate within approximately 10%. Creating rules to this level of accuracy appears to be relatively little work. Rules are easy to write, as they are syntactically presented as a dictionary, in which correspondences are written directly using the UAN notation. Further studies with external users will be required to determine how willing developers will be to create linking rules.

The simplicity of the dictionary format carries a cost of expressiveness – only simple control over scoping of rules is provided, and only simple patterns based on variables are allowed. The examples we have performed to date allow us to tentatively conclude that these restrictions are not problematic in practice. In cases where the dictionary language is not sufficiently expressive, rules can be added to the dictionary that either explicitly add links or explicitly rule out error cases. In our examples, when such rules are added, coverage rates climb to 100% while error rates drop to 0%. This shows that if Adligo users wish to invest the time to refine their rule sets, very accurate performance can be obtained.

The Adligo approach is most successful if the UAN specifier is methodical in the use of UAN. For example, if the "`:=`" symbol is used consistently to indicate change of interface state, standard rules related to assignment will be invoked. If the same names for tasks and contexts are used consistently, then custom rules will be more likely to generate correct results. For example, many of the custom rules in the critical path planner dictionary specifically deal with the use of poor names in the UAN specification such as `n`, `n1`, `n2`, etc. to refer to nodes in the network. While the requirement of consistency and use of clear naming conventions adds an extra burden on UAN designers, such conventions also lead to specifications that are easier for humans to read.

The rule-based approach has the potential advantage of being robust to changes in the underlying design documents. As new tasks are added or reworded, or as architecture components are modified and repositioned, it is likely that existing rules will continue to apply. Further experimentation will be required to demonstrate the extent to which rules are robust to evolution in design documents.

6 Conclusions

This paper has shown that it is practical to partially automate the generation of links between user interface design artifacts, even when these artifacts are

developed by different people and when the artifacts involve informal English prose. Users must provide a small set of rules in a simple dictionary format. From the rules, the Adligo tool derives links between task-oriented specifications and user interface architectures.

We plan to continue investigating the effectiveness of the rule-based approach, and plan to carry out experiments tracking the robustness of rules as design artifacts evolve.

Acknowledgements

This work was partially supported by the Natural Science and Engineering Research Council (NSERC) and by Communications and Information Technology Ontario (CITO). Greg Phillips provided helpful feedback on earlier drafts of the paper. Tim Wright was very helpful in connecting Adligo to Vista.

References

1. J. Brown, T. Graham, and T. Wright. The Vista environment for the coevolutionary design of user interfaces. In *Proc. CHI '98*, pages 376–383, 1998.
2. J. Brown and S. Marshall. Sharing human-computer interaction and software engineering design artifacts. In *Proceedings of OZCHI'98*, 1998.
3. G. Cockton and S. Clarke. Using contextual information effectively in design. In A. Sasse, M. Tauber, and C. Johnson, editors, *Proc. INTERACT99*. Kluwer, 1999.
4. J. Dilworth. *Production and Operations Management: Manufacturing and Services, Fifth Edition*. McGraw Hill, 1993.
5. S. Elnaffar. Semi-automated linking of user interface design artifacts. Master's thesis, Queens University at Kingston, Canada, May 1999.
6. T. Graham. GroupScape: Integrating synchronous groupware and the world wide web. In *Proc. INTERACT '97*, pages 547–554. Chapman and Hall, 1997.
7. T. Graham, H. Damker, C. Morton, E. Telford, and T. Urnes. The Clock Methodology: Bridging the Gap Between User Interface Design and Implementation. Technical Report CS-96-04, York University, Canada, August 1996.
8. T. Graham and T. Urnes. Linguistic support for the evolutionary design of software architectures. In *Proc. ICSE 18*, pages 418–427. IEEE Press, 1996.
9. T. Graham and T. Urnes. Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces. In *Proc. ICSE 19*. IEEE Press, 1997.
10. H. Hartson, A. Siochi, and D. Hix. The UAN: A user-oriented representation for direct manipulation interface design. *ACM TOIS*, 8(3):181–203, July 1990.
11. P. Johnson, H. Johnson, and S. Wilson. *Rapid Prototyping of User Interfaces Driven by Task Models*. John Wiley & Sons, 1995.
12. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, 1(3):26–49, August/September 1988.
13. P. Palanque, R. Bastide, and V. Senges. Task model – system model: Towards a unifying formalism. In *Proc. HCI International*, pages 489–494. Elsevier, 1995.
14. F. Paterno, C. Mancini, and S. Meniconi. Engineering task models. In *Proc. IEEE Conference on Engineering Complex Systems*, pages 69–76. IEEE Press, 1997.
15. P. Szekely, P. Sukaviriya, P. Castells, and J. Muthukamarasamy. Declarative interface models for user interface construction tools: the MASTERMIND approach. In *Proc. EHCI '95*, pages 120–150, 1995.