Where Do Instructions Come From? Addressing the Problem of Knowledge Acquisition in the Context of Instructional Text*

Keith Vander Linden[1], Cecile Paris[2], and Shijian Lu[2]

[1] Department of Computer Science, Calvin College,
Grand Rapids, MI 49546, USA
kvlinden@calvin.edu
[2] CSIRO, Mathematical and Information Sciences,
Locked Bag 17, North Ryde, NSW 1670, AU
{Cecile.Paris, Shijian.Lu}@csiro.dit.syd.au

Abstract. Instructional text has been a popular target for research-oriented generation systems because it is a useful and relatively constrained sub-language. The success of these systems shows that existing technology is adequate for generating draft instructions; the problem, as is typical of generation work in general, has been with the acquisition of domain and lexicogrammatical knowledge. The Isolde project has addressed this acquisition problem by building tools for mining raw domain knowledge from existing formal models (e.g., UML models for software systems) and for inferring and tailoring lexico-grammatical knowledge from the domain elements.

1 Introduction

A number of research systems have been successful at generating drafts of procedural instructions (e.g., Mellish/Evans [7], COMET [6], TechDoc [14], Drafter [10], WIP [16], SPIN [3], WYSIWYM [11]). It is, therefore, clear that current generation technology is adequate for generating at least portions of the instructions found in simple user's documentation. These systems have not, however, become commercially successful. The basic reason for this is that the acquisition of the knowledge required as input by these systems is too difficult. It is simply easier to enter the text by hand.

Some of the systems mentioned above (e.g., Drafter, WYSIWYM) have developed user interfaces to facilitate the input of this knowledge. While these interfaces have proven to be usable and useful, it is not clear that they adequately solve the knowledge acquisition problem. They may ease the input process, but it is still the case that all the knowledge must be either hard-coded in the system or entered by hand. It might be worth going to this trouble if the resulting knowledgebase were useful for

* This work is supported by ONR grants N00014-99-1-0906 and N00014-97-0064, the Calvin College research fellowship program, and the CSIRO.

purposes other than generation and re-generation, but it is usually tailored specifically for this purpose. Generating in multiple languages could also increase reuse, but it would probably be easier to write the text by hand and then translate it automatically.

In contrast, successful generation systems for other sub-languages (e.g., Meteo [2]) operate in domains in which suitable input representations are available because they are already built for other reasons. These sub-languages also tend to use restricted lexicogrammatical resources, which can be predefined. Neither of these is the case for the instructional sub-language. Pre-built models do exist, but they are neither complete nor universal, and the domains for instructions are not restricted, so while the grammatical resources can be predefined, the lexical resources cannot.

The purpose of the Isolde project has been to identify those pre-built resources that do exist in the common practice of system and user interface development, and to extract or "mine" the necessary domain and lexical knowledge from them. We have had some success in mining knowledge from models built using the Unified Modeling Language (UML [13]). However, we have also had difficulty because in practice, UML models are not complete, and routinely contain information that is useful for the system designer, but either useless or erroneous for generation. To address this, we have built a set of tools that:

- collect the knowledge that can be mined from existing models,
- allow the user to add to it, modify it, and configure it into a form that is useful not only for text generation, but also for interface design,
- generate text drafts, allowing the user to modify the knowledgebase elements that underly any incorrect expressions.

This paper will describe each of these functions in turn, doing so in the context of an extended example. We will then draw general conclusions for language generation applications.

2 Mining the Knowledge Required for Instructions

This section will discuss the two basic types of knowledge required by an instruction generation system: domain knowledge and linguistic knowledge. Experience with instruction generation systems like Drafter gives us a very good idea of what this knowledge should include and how it should be structured. We focus here on identifying the commonly used system models from which this knowledge can be mined.

2.1 Domain Knowledge

The domain knowledge required for procedural instructions comprises: (1) the objects used in the application domain; (2) the graphic objects presented by the user

interface; and (3) the tasks that are performed on these objects. These elements are typically represented in a standard slot-filler knowledgebase format, in which the actions are arranged in a procedural, plan-goal hierarchy, and the objects are linked to the actions as case-role fillers. The structure of these representations is described in detail elsewhere (cf. Drafter).

The goal of Isolde is to mine this knowledge from existing sources. To do this, we follow the lead of Drafter, which prototyped a mechanism that mined objects of type (2) from an interface building tool [9]. We have found that object-oriented (OO) system models, as produced by software engineers [12], are a good source of all three types of knowledge. These models tend to provide:

- *use case diagrams* - A use case identifies a thread of potential use for the system to be constructed. Use case diagrams associate the various types of users with the use cases in which they take part.
- *class diagrams* - These contain an hierarchical arrangement of the classes used in the system. Each class specifies its own operations and attributes.
- *interaction diagrams* - These diagrams, though less common, describe the sequence of interactions between objects that take place in the execution of a use case. They specify which objects send which messages to which other objects, as well as the time dependencies between the message events.

We will now describe how each of these models can contribute important elements for the knowledgebase. We use the Unified Modeling Language (UML) notation as it has become a standard for OO system modeling, and will illustrate the models using BMS, a building maintenance system developed by Esprit Systems Consulting for which complete UML models are distributed freely by Rational Corporation.

UML Use Case Diagrams. Figure 1 shows three use cases for BMS. We can see that the object PhysicalEmployee (an instance of Actor or User) can perform 3 tasks: Requesting access to a room, Requesting a phone for a room, and Adjusting the temperature for a room. Clearly, this diagram is a source of user types and of high-level user goals with respect to the system. The primary problem here is that the names of the actors and the spellings of the verbs and objects are chosen by the software engineer. We have little control over this. As a default, our extraction mechanism (implemented in Rose's scripting language) produces a instance of the user type, using the exact spelling of the actor in UML as the name. It also produces high-level tasks by parsing the names of the use cases, assuming that the action is the first word, the actee is the second, and the remainder of the words are other adjuncts. If this is wrong, and it likely will be, the technical author will see it in the drafted text and correct it with facilities described in the generation section below.

UML Class Diagrams. Figure 2 shows a portion of the class diagram for the BMS system. In addition to being useful in building the BMS system, classes such as room and section will also be mentioned in the instructions. Our extraction mechanism, therefore, creates domain objects for each UML class, using the spelling of the class name as a default lexical root. Again, if the spelling is unacceptable for the

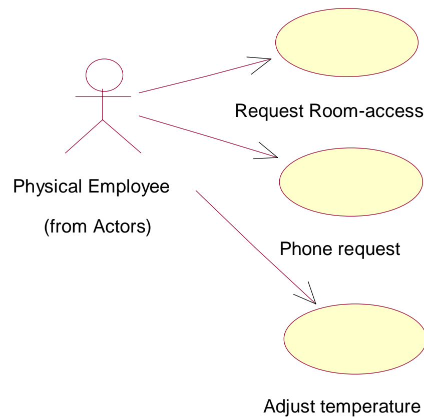

Figure 1. A portion of the UML Use Case diagrams for BMS

instructions, the author can modify it later. For the BMS example, we were forced to add classes for the GUI objects used in the BMS interface. These were not in the original BMS class diagram because the programmers that built the interface did not use Rose to model the GUI interface.

UML Interaction Diagrams. Figure 3 shows the interaction diagram corresponding to the use case of adjusting the room temperature with the BMS system. Here, the sequence of events that instantiate a use case are presented in temporal order (from top to bottom). Each event specifies the object that initiates the event (the source of the arrows) and the object that is acted upon (the destination of the arrows). In this example, the user selects the adjust temperature menu item and the system responds by displaying a dialog box. The user then selects the appropriate temperature and room, clicks the ok button, and the system responds by setting the temperature appropriately. This diagram specifies several actions that will be expressed in the instructions, but it also includes a number of internal programming details that are not relevant for the end-user documentation. Our extraction routine, therefore, attempts to filter out the inappropriate details. Briefly, this filtering is done by a set of content selection heuristics which extract all user initiated actions, and the last system action in any sequence of system actions (See [5] for more details).

Taken together, these three diagrams provide approximately 70% of the resulting task model (discussed in the next section)

2.2 Linguistic knowledge

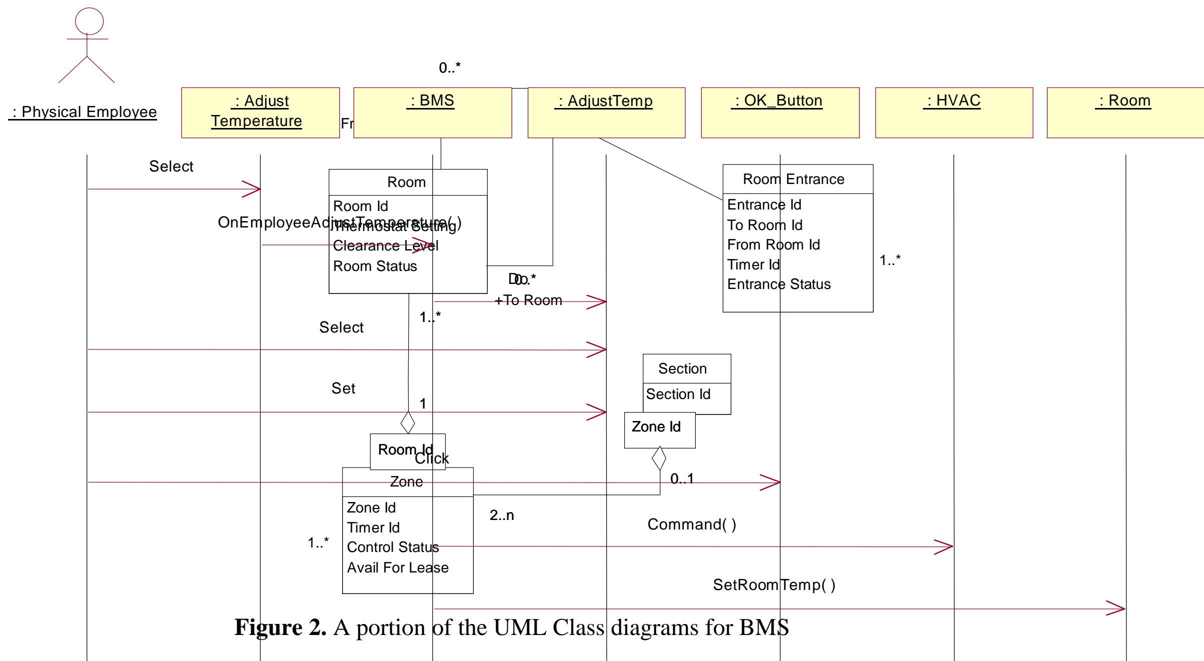

Figure 2. A portion of the UML Class diagrams for BMS

Figure 3. A portion of the UML Interaction Diagrams for BMS

The linguistic knowledge required by instruction generation includes the lexical resources used to express the relevant objects and tasks, and the grammatical and discourse knowledge used to fashion instructional sentences and texts. As was the case with the domain knowledge, instruction generation systems typically hard code this information as well. This approach has proven to work well for grammatical and discourse knowledge because they tend to be constant from one instructional domain to the next. Thus, the Isolde system follows the practice of previous systems by coding this sort of knowledge directly into the text and sentence planning resources.

The lexical knowledge, on the other hand, does change, which requires the users of other systems to build new lexical resources for each new domain. As we have discussed earlier in this section, our knowledge extraction mechanism infers lexical spellings for the domain knowledge elements that it extracts. These spellings are likely to be wrong, but they are usable for generating a first draft of the instructions, which the technical author can then fix up as discussed below.

3 A Task Model Editor

As we saw in the previous section, the knowledge that we were able to extract from the UML models is not likely to be correct in all cases. In addition, because engineers may not use all the UML diagrams, the knowledge we extract may not be com-

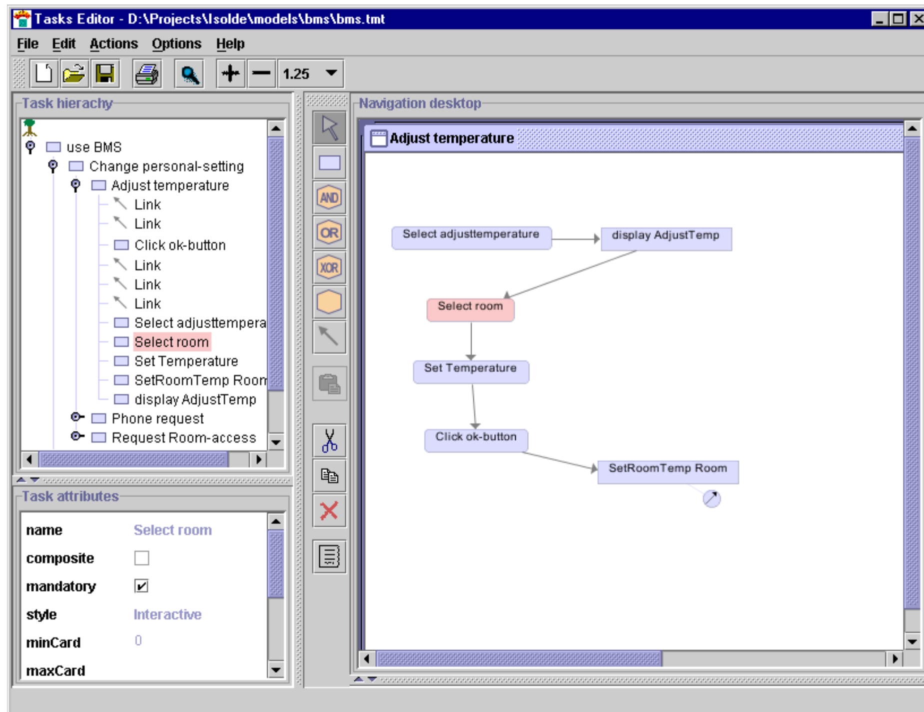

Figure 4. The adjust-temperature task decomposition in TAMOT

plete either. We have, therefore, built a tool that allows a human to configure a complete input for the generation system. This input is not unlike the inputs typical of instruction generation systems, except that it takes the form of a *task model.* Task models are procedural models of human tasks, goals and system responses; they are commonly used by user interface designers. Our tool, called TAMOT, is shown in Figure 4. It supports the Diane+ task modeling notation [15] and isimplemented in Java 1.2 with the SWING widget set.

On the right-hand side of this figure, we see a simple task decomposition for the user task of adjusting the temperature of a room. The oval-cornered boxes represent the user actions of selecting the adjust-temperature option, selecting the desired room, setting the desired temperature, and clicking the OK button. The rectangular boxes indicate the system actions of displaying the dialog box and setting the room temperature. The arrows indicate the proper sequence of the tasks. The left-hand side of the interface contains a task hierarchy, which allows the user to browse all of the tasks, and a Task Attributes window, which shows the current attribute settings for the selected task.

The tasks shown in the figure were all derived automatically from the UML model, though we did have to format the model in a more readable way and remove one system task that would not have been helpful to the user. Manipulations of this sort

are performed by dragging tasks around on the Navigation Desktop, or by selecting them and editing their properties in the Task attributes window (on the lower left). The representation of each of these tasks is linked in a knowledgebase representations of the domain knowledge required to express the task in the generated instructions.

One key feature of this tool is that it supports a modeling formalism taken from the HCI literature rather than from the internal representation of the text generator. Although similar to the plan-based procedural hierarchies typical of other systems, it includes features useful for task modeling as well. It can, therefore, function as a stand-alone task modeling tool. Indeed, it is currently being used as such by an interface design consultant in our group. Thus, the construction of the task model is useful not only to drive the generation process, but also to aid in the design of the inter-

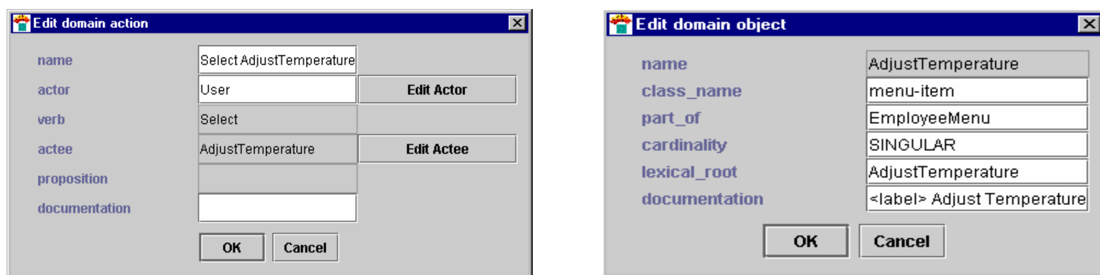

Figure 5. Domain Action and Domain Object Editing Dialog boxes

face. While the use of task models is not universal in applied interface design, there are more and more professionals turning to them to support design.

4 Instruction Generation

The generation system is implemented as a separate server process, implemented in Harlequin Common Lisp for the PC. This system, a descendent of the Drafter generation system, includes (1) the Moore and Paris text planner; (2) a new sentence planner implemented with extensions to the text planner; and (3) the KPML development environment. This system is not fundamentally different from the original Drafter generator, except that uses a new sentence planning mechanism, it is implemented as a server, and it plans hypertext links and canned text.

The primary problems the Isolde generator faces are due to the fact that much of the input was not hand-crafted for generation. Rather, it was extracted from a UML model built by a software engineer for the purpose of designing a software system, and from a task model built by an interface designer for the purpose of designing a graphical user interface. As an example, consider the following text which was generated for the first draft of the BMS instructions: “The system domodels the Adjust

Temperature dialog box." Here, we see that the verb "domodels" would be meaningless to the end user. This wording comes from the UML model in which the software engineer choose the method name "doModel()" to describe the action of presenting a dialog box to the user. The engineer then used this method in the interaction diagram. We are powerless to stop this, and indeed, don't want to interfere when the software engineers and interface designers use their tools to build models. What we do instead is take the output of the external models as it comes and allow the technical author to step in and fix those elements of the text that are incorrect.

We implement this "fix it when it's broken" approach using a mouse-sensitive, hypertext display buffer (based on a tool prototyped in Drafter) in which the author can click on the expressions that are wrong, and get a dialog box that allows them to edit the domain model entities from which the text was generated. Parts of this facility are shown in Figure 5. The domain action, shown on the left, is linked to its case roll fillers via the "Edit Actor" and "Edit Actee" buttons. The editing dialog box for the actee is shown on the right. The technical author is able to use these and other dialog boxes to iteratively modify the domain and lexical knowledge used to drive the generation process until they get the text they want. The final output of our example (with the corrected spelling for "domodel") is shown in Figure 6.

3 Conclusions

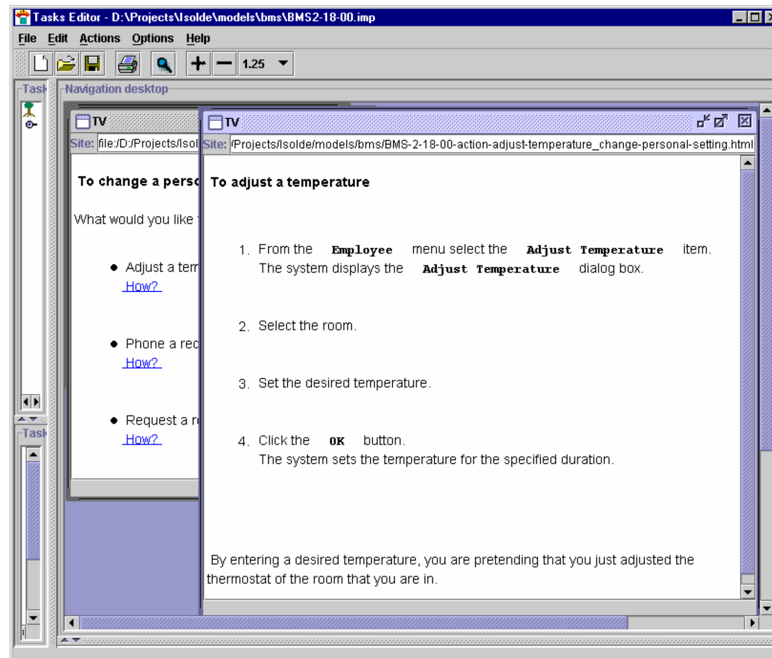

Figure 6. A portion of the hypertext output as displayed in TAMOT

This paper has discussed some practical issues involved in fielding instructional text generation systems. The current generation technology is adequate for generating instructions, but the knowledge resources required as input to the process must be mined, as much as possible, from existing sources. A PC-based implementation of the Isolde system was discussed as an example of an approach to this problem.

We identify two basic conclusions from this continuing work. First, to be practical, a generation project must take pains to find readily available sources of input knowledge. Assuming that a technical author will be willing to manually input all or most of the resources by hand is probably unacceptable. This paper presents one approach to doing this in the context of instructions. Second, given that external resources will probably not be enough to drive the generation process completely, one must also build a tool that allows an author to configure the resources that are mined. It would be best if this tool supported a generally useful modeling language rather than a generation-specific one. TAMOT is an example of such a tool.

Acknowledgements

The authors acknowledge a debt to the legacy of the original Drafter project undertaken at the ITRI, University of Brighton, UK from 1993-1996. They also acknowledge the involvement of colleagues at the CSIRO, including Sandrine Balbo, Thomas Lo, and Nadine Ozkan.

References

1. Bateman J. A.: Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, 3(1) (1997) 15-55

2. Kittredge, R., Polguere, A.: Generating extended bilingual texts from application knowledge bases, *Proceedings of the International Workshop on Fundamental Research for the Future Generation of Natural Language Processing,* Kyoto, Japan (1991) 147-160
3. Kosseim L. and Lapalme G. : Content and rhetorical status selection in instructional text. In *Proceedings of the 7th International Workshop on Natural Language Generation*, Kennebunkport, ME (1994)
4. Lavoie, B., Rambow, O., Reiter, E.: Customizable Descriptions of Object-Oriented Models. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, Washington, DC, (1997) pp. 265-268.
5. Lu, Shijian, Paris, C., Vander Linden, K.: Towards Automatic Construction of Task Models from Object Oriented Diagrams, in S. Chatty and P. Dewan (eds.), *Proceedings of the IFIP Working Conference on Engineering for Human-Computer Interaction* (1998)
6. McKeown, K., Elhadad, M., Fukumoto, Y., Lim, J., Lombardi, C., Rogin, J., Smadja, F.: Natural language generation in COMET, *Current Research in Natural Language Generation,* chapter 5, Academic Press (1990)
7. Mellish, C., Evans, R., Natural language generation from plans, *Computational Linguistics,* 15(4) (1989) 233-249
8. Moore J. and Paris C.: Planning text for advisory dialogues: Capturing intentional and rhetorical information. *Computational Linguistics*}, 19(4) (1993) 651--694
9. Paris C. and Vander Linden K.: Drafter: An interactive support tool for writing multilingual instructions. *IEEE Computer*, 29(7) (1996) 49-56, July
10. Paris, C., Vander Linden, K., Fischer, M., Hartley, A., Pemberton, L., Power, R., and Scott, D.: A support tool for writing multilingual instructions. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence,* August 20-25, Montreal, Canada, (1995) Pages 1398-1404.
11. Power, R., Scott, D., and Evans, R.: What You See is What You Meant: direct knowledge editing with natural language feedback, *Proceedings of the 13th European Conference on Artificial Intelligence*, ECAI'98, Brighton, UK, August (1998)
12. Pressman, R: *Software Engineering A Practitioner's Approach*, 4thed, McGraw Hill, 1997
13. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual,* Addison-Wesley, Reading, MA, (1999)
14. Rösner and M. Stede: TECHDOC: A system for the automatic production of multilingual technical documents.In *Proc of KONVENS-92*, Berlin. Springer (1992)
15. Tarby, J-C. and Barthet, M-F. The DIANE+ Method. In *Proceedings. of the 2nd International Workshop on Computer-Aided Design of User Interfaces* (1995)
16. Wahlster, W, Andre, E, Finkler, W, Profitlich, H, Rist, T: "Plan-based integration of natural language and graphics generation", *Artificial Intelligence,* 63(1-2) (1993) 387-428