Planning Operators in User Interface Modeling

Krzysztof Gajos
kgajos@cs.washington.edu

May 27, 2003

1 Introduction

Following our last discussion, I went back to lit search in order to explore the use of planning operators in user interface modeling. Conveniently, Gieskens and Foley published an article titled “Controlling User Interface Objects through Pre- and Postconditions.” [1] This article covers a number of situations that we had discussed so I am going to summarize its main points in this writeup.

2 Gieskens and Foley Article

One of the ways to describe the dialogue of an application is with an event language. Pre- and postconditions are concidered to be a “particularly useful form” of an event model. They can be used to parially describe semantics of an application. For example, pre- and postconditions are sufficient to fully encode simple dialogue (thus parts of the interaction with the user are performed without any calls to the application). Furthermore, the pre- and postconditions can be used to automatically generate help for interface widgets.

Gieskens and Foley assume two kinds of preconditions: those that control whether a widget is visible, and those for deciding if it should be enabled. Given the level of automation that we envision, I assume that we SUPPLE should not make this distinction.

As far as communication is concerned (either with the application or among different widgets), Gieskens and Foley suggest using a blackboard – but that’s not too relevant for us at this point.

They allow pre- and postconditions to make use of the values of state variables and widget values. They also allow comparisons and arithmetic operations (though they encode them as specialized predicates rather than actual operations). The expressive power of what they proposed can be illustrated by the following two examples (with notation modified by me):

```
nextTrack
    pre:   currentTrack ¿ 0
           currentTrack ¡ totalTracks
    post:  currentTrack := currentTrack + 1
```

```
goToTrack
    pre:
    post:  currentTrack := getValue(this)
```

The first example illustrates the complexity of expressions allowed in pre- and postconditions. The second example shows a situation where the value of the widget is used as the new value of a state variable.

In the later parts of the paper, they note that some times changes to the UI widgets should not immediatelly result in the changes to the underlying state variables. Such situations take place, for example, in printing dialog boxes – the most recent settings should not be changed until we actually press the print button. To address this problem, the paper proposes using a "propose" keyword to mark proposed but not committed postconditions. These postconditions can be committed afterwards as a postcondition of the print button. I think that in our case we will not need the "propose" keyword. The way we build types and the instantiate them, we can keep changes only within the local scope until the print (or commit or go) button is pressed. This remark applies to compound command types (i.e. write-only types).

The most obvious problem with their system seems to be the lack of scoping – all state variables are global which makes it impossible to easily encode an interface that, for example, allows the user to edit several email messages at once. It also creates the problem addressed by the "propose" keyword.

References

[1] Daniel F. Gieskens and James D. Foley. Controlling user interface objects through pre- and postconditions. In *CHI*, pages 189–194, 1992.