Guiding User Interfaces Equationally

T. B. Dinesh
CWI (dinesh@cwi.nl)

S. Üsküdarlı
Programming Research Group, University of Amsterdam, (susan@fwi.uva.nl)

Amsterdam, The Netherlands

October 31, 1996

1 Introduction

Algebraic or equational specifications are popular due to their simplicity. We advocate that such an approach could be of use for guiding user interfaces. If a user interface design process is to be accessible to a diverse user population that include novice computer users, it has to break out of the two stage user interface design— the first stage, where a user interface builder is used to develop the look of the interface and the second phase which requires programming the semantics in a *low level* language.

Equational specifications can be executed by orienting equations as left-to-right rewrite rules. Visual algebraic specifications are considered as a formalism for specifying visual languages [ÜD95], where among other tools, generic syntax directed editors for constructing visual terms can be generated [Üsk94]. Algebraic specifications, like functional languages, suffer from the lack of constructs that allow interaction with an user during execution. This is a drawback for specifying languages that are interactive by nature.Also of importance is the utility of seamlessly providing the user-interface specification and interaction, since on the average, user interface development accounts for approximately 50% of the cost of producing an application [Gra95].

We examine the utility of extending algebraic specifications (to handle interaction) in meeting the demands of interactive tools for visual languages and user interfaces. As a result of this extension, it is possible to provide a user definable specification of a language interface itself. We concentrate on the utility of a such an extension in the presence of the generated visual term editors by using only the generic features of these term editors. We argue that interaction with a user is nothing but constructing terms in editors. Thus, during execution, the user will be presented with terms whose construction must be completed.

The particular appearances of such editors could be delegated to a separate user-interface specification.

In the next section we briefly discuss an algebraic specification formalism to provide some context for our work. In Section 3, we again briefly indicate the nature of the extension we are investigating. In Section 4, we give a detailed example that demonstrates our approach.

2 Algebraic Specifications

Consider the following signature that describes a language defined by sort B:

sort B
functions

$true$	$\rightarrow$	B
$false$	$\rightarrow$	B
$B \vee B$	$\rightarrow$	B

One can think of these are BNF rules (reading right to left) where B denotes a non-terminal and $true, false$ and $\vee$ denote terminals. With the above signature we can construct *terms* of the form $true$, $false$, $(true \vee false) \vee true$, $\cdots$.

2.1 Conditional Rewrite Rules

Conditional equations are used to specify language semantics. *Conditional rewrite rules* [BK86] are used to execute conditional equations.

A conditional rewrite rule takes the form

$$\frac{s_1 = t_1, \cdots, s_n = t_n}{s_0 = t_0}$$

with $n \geq 0$, and s_i, t_i $(0 \leq i \leq n)$ terms. Usually, some well-definedness constraints are imposed on the variables of the conditions in order to ensure their definedness during the execution.

For example, the following oriented (unconditional) equations describe the semantics of the language B.

$$true \vee x = true$$

$$false \vee x = x$$

where x is a variable over the sort B, which we sometimes write as: $x \rightarrow B$.

2.2 Meta-variables

Meta-variables are place holders available while syntax-directed editing. They represent the holes in incomplete terms. A hole of sort *SortName* is represented with by $\langle SortName \rangle$ and can be replaced with any term of the sort

SortName. They allow interactively building the intended term by choosing among permissible substitutions of the language constructs. Multiple occurrences of place-holders of the same sort are independent of each other. E.g., In the term $\langle B \rangle \vee \langle B \rangle$ the two $\langle B \rangle$s represent separate (unrelated) place-holders.

3 Interaction

We describe how an algebraic specification formalism, interpreted as a term rewriting system, can be extended to accommodate interaction.

The situation and the extension is illustrated by a toy example. Consider the following set R of oriented (conditional) equations (i.e., rules):

$$true \vee false = true$$

$$\frac{x \neq false}{true \vee x = true \vee y}$$

where x and y are variables over the sort B (described in the previous section).

This is not a term rewriting system in the usual sense as the second rule introduces y, a new variable on the right-hand side. However, as we will show, this extension (in some form or other) of the notion of term rewriting systems is essential in our case. If we start reducing a term $true \vee true$, it matches the left-hand side of the second rule (binding x to *true*). The condition $x \neq false$ succeeds (as x is bound to *true*) and this term will be rewritten to say $true \vee y'$, where y' is a renaming of variable y — different from other existing variables. Now, for further reduction of this term, it has to match one of the left-hand sides again[1]. The unbound variable y' cannot match *false* (in the first rule) but it can match x in the second rule. In our case, the condition $x \neq false$ would fail since it cannot be determined that x is not *false*. Therefore, the term $true \vee true$ reduces to $true \vee y'$ and the reduction stops. It can be restarted by concretizing y' to any valid term (any term of sort B). This narrowing substitution that happens external to the rewriting essentially models input.

In this case, rewriting continues as long as *true* is entered interactively and stops as soon as *false* is entered, terminating the interactive reduction process. We can denote such a situation by:

$$true \vee true \quad \xrightarrow[true * false]{* \quad R} \quad true$$

where $true \vee true$ is the initial term, $\xrightarrow{* \; R}$ denotes the multi-step reduction relation over R, $true * false$ is a regular expression describing the input sequence, and *true* is the resulting normal form.

[1]Since y' is an unbound variable, the term $true \vee y'$ could unify with either *false* in the first rule or with the variable x in the second rule. Taking into account both these possibilities is the subject of narrowing based term rewriting systems. However, in a typical term rewriting system only matching (and no unification) is present.

3.1 Input

The reason the rewriting stops in the y' case is pathological since it was not because $true \vee y'$ could not match any of the left-hand sides, but because a condition failed. For interaction, we need an interpretation that prevents $true \vee y'$ from matching any of the left-hand sides independently of any conditions of the rules. This is important since a desire to interact is a commitment, as an interaction is observable (modifies the world). We write the above rules as follows:

$$true \vee false = true$$

$$\frac{x \neq false}{true \vee x = true \vee \chi(\langle B \rangle)}$$

Note the use of $\chi(\langle B \rangle)$ in place of y. Since y was declared to be of sort B, $\langle B \rangle$ indicates a place holder which needs to be filled in; and "χ" lifts the B term to a term of a special sort say, χ-sort[2] — thereby preventing it from matching any B term. We can also read this as an indication that only an external process can narrow its contents. The $\chi(\langle B \rangle)$ term becomes a B term by projecting the term that was used as a replacement for the place-holder. Thus

$$\chi(\langle B \rangle)[true] = true$$

where $t_\chi[t_B]$ indicates the replacement of the place-holder ($\langle B \rangle$ here) in t_χ by t_B and retracts the χ-term to a term of sort B so that it could match variables of sort B. Again, the place holders have the usual meaning that every occurrence is unique.

3.2 Output

Until now, we have considered input but what is interactive output in such a rewriting environment? We can put additional constraints on the nature of values expected from an external process. For instance, we can require that the replacements for the place-holder matches certain patterns. In the above, instead of allowing all B values, one could restrict the possible substitutions to $\langle B \rangle$ to values that are of the form $false \vee \cdots$. Consider the alternate set R_1 of rules:

$$true \vee false = true$$

$$\frac{x \neq false}{true \vee x = true \vee \chi(false \vee \langle B \rangle)}$$

[2]The non B-ness propagates upwards— parent is not a B term since a child is not a B term.

Here the constraint on the input is that it should not only be of the sort B, but should also have the form $false \vee \cdots$. Thus a user can provide a term that narrows the contents of χ and then the projection of the term that replaced the place-holder would be the value of the χ term. Thus

$$\chi(false \vee \langle B \rangle)[false \vee true \vee false] = true \vee false$$

for the case in the second rule of R_1. A term $true \vee true$ reduces to a term of the form $true \vee \chi(false \vee \langle B \rangle)$ which could further reduce to $true \vee e$ (for some B term e) when a user provides the term $false \vee e$. The χ terms can be thought of as retraction functions that retract to the value of its place-holder. Thus every χ term is allowed only one place holder, the sort of which is the sort of the term it retracts to. In an interactive sense, this means that a user is constrained to provide a term of the form $false \vee \cdots$ for the reduction to proceed further. The "$false\vee$" provides to the user, context information while inputing a value to $\langle B \rangle$. The context information can in turn be perceived as output. Note that, this results in requesting specific patterns. Thus:

$$true \vee true \qquad \xrightarrow[(false \vee true)* \ (false \vee false)]{* \quad R_1} \qquad true$$

Alternatively, $\chi(false \vee \langle B \rangle)$ in the second rule could be $\chi(enter\ value : \langle B \rangle)$ where "$enter\ value : B$" is a valid term over some sort.

Finally, as a special case of this situation, for illustration purposes, is R_2:

$$true \vee false = true$$

$$\frac{x \neq false}{true \vee x = true \vee \chi(false)}$$

Here, the interactive-rewriting terminates only after receiving an "input" of $false$. The sort and the value the χ-term retracts to when it has no place-holders would in general be the identity value. Here a user has defined it to retracts to value $false$ after communication. Using R_2, the situation is:

$$true \vee true \qquad \xrightarrow[false]{* \quad R_2} \qquad true$$

even though one can see that the normal rewriting can terminate without any need for interaction. We can read this as "term reduces to $true$ with an output $false$".

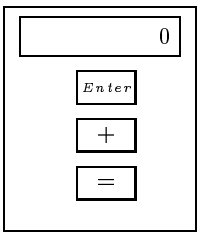
4 Calculator Example

In order to demonstrate how a user interface can be guided by equations we provide a very simple example of a calculator. Albeit small, this example describes a graphical language with semantics requiring human interaction. This example, while being very simple, is not fine tuned for any specific interaction style and thus only the default interaction is discussed.

Building a user interface is done in two phases. The first phase is to specify the look of the interface. This is similar to many common user interface builders available today. The additional flexibility we provide is that certain syntactic constructs can be grouped together by using a small constraint language for layout of the interface look. The details of the visual syntax specification of the calculator is not provided here. But rather, the focus is on aspects related to input and output and its specification. However, the following sort definitions are needed to follow the forthcoming specifications.

$$
\begin{array}{ll}
\boxed{\textit{Enter}} & \rightarrow \; \textit{OP} \\
\boxed{+} & \rightarrow \; \textit{OP} \\
\boxed{=} & \rightarrow \; \textit{OP} \\
\boxed{\textit{NUM}} & \rightarrow \; \textit{DISPLAY} \\
\textit{OP} & \\
\textit{OP} & \\
\textit{OP} & \rightarrow \; \textit{OPS} \\
\boxed{\begin{array}{c}\textit{DISPLAY}\\ \textit{OPS}\end{array}} & \rightarrow \; \textit{CALC}
\end{array}
$$

The *Calculator* language defines three operations: $\boxed{\textit{Enter}}$ for resetting the current value of the calculator; $\boxed{+}$ for adding a new value to the running total of the calculator; and $\boxed{=}$ for displaying the total. The sort *OPS* describes that three *OP*s are placed in vertical alignment and the sort *CALC* describes that a *DISPLAY* and *OPS* are layed out such that the *OPS* is centred and below the *DISPLAY*. This is specified by constraining these appropriately [ÜD95]. A visual term of the calculator could be, depending on nature of constraints specified:



4.1 Calculator Query Syntax

The calculator requires two input functions: one for retrieving an "*OP*" selection and one for retrieving values. The values in question are numbers, the definition of which is imported (predefined).

In order to interact we need to define syntax for queries, which would generally be an extension to the Calculator syntax itself. Queries can be as simple or complicated as desired. The simplest queries are just meta-variables appearing in a term window without indicating any context. In essence the goal of a query is to fetch some value and the manner in which the input is retrieved only bares in interface aspects. The point is that the language designer can simply define input prompts (which are in fact the outputs) in an uniform and convenient manner. The utility of these definitions can be seen in Section 4.2.

$$\begin{array}{llll} CALC & NUM & \rightarrow & CALC\text{-}Q \\ CALC & OP & \rightarrow & CALC\text{-}Q \end{array}$$

For example, *CALC-Q* will be used to inform the current state of the calculator to a user who can then respond by "building" the desired input term. In this case, either the *NUM* or the *OP*, depending on the context.

4.2 Evaluations Semantics

After the first stage which is specifying the desired user interface components and the layout, the second stage involves specifying the semantic component of the user interface. This is done using equations. Note that one might need to specify additional syntax during this stage that need not be part of the user interface itself. For the syntax of the calculator evaluation we use the additional syntax of the *eval* and *eval-op* and define their functionality.

$$\begin{array}{lll} eval(CALC,NUM) & \rightarrow & NUM \\ eval\text{-}op(CALC,OP,NUM) & \rightarrow & NUM \end{array}$$

The semantics is defined using conditional equations as explained in Section 2.1. These equations make use of the following variables:

$$\begin{array}{lll} Calc & \rightarrow & CALC \\ Ops & \rightarrow & OPS \\ TheOp & \rightarrow & OP \\ Store, Num, Num' & \rightarrow & NUM \end{array}$$

For example, *Calc* could be bound to any calculator (visual) term that can be composed from the above syntax specification for the sort *CALC*. Furthermore, in this example, we use the notation $\boxed{\text{term}}$ instead of the $\chi(term)$ used in Section 3. We provide a brief explanation for each equation below.

equations

Evaluating a CALC term with a given store, is to query for an operation and then evaluate the term using the result of this query. The variable *TheOp* represents the result of interaction that would be obtained after interactively binding the variable to an operation. Note that the current *Calc* contents are displayed to the user in order to provide the context for interaction. The right hand side of the equation gets the user desired operation which is bound to the variable *TheOp* which guides the interface to the next interaction caused by *eval-op*.

$$[1]\quad \frac{\text{TheOp} \;=\; \boxed{\blacksquare\, Calc\ \langle \text{OP} \rangle}}{eval(Calc, Store) \;=\; eval\text{-}op(Calc, TheOp, Store)}$$

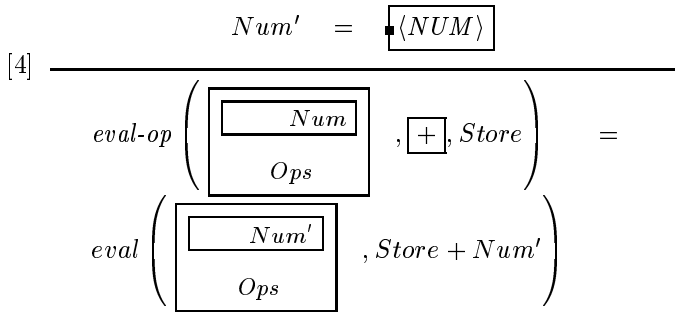
Evaluating a *Calc* when the operation is $\boxed{Enter}$ is to query for a new number which will be displayed in the calculator.

$$[2]\quad eval\text{-}op\left(\boxed{\begin{array}{c}\boxed{Num}\\ Ops\end{array}}\ ,\ \boxed{Enter},\ Store\right) \;=\; eval\left(\boxed{\begin{array}{c}\boxed{\blacksquare\,\boxed{\langle NUM \rangle}}\\ Ops\end{array}}\ ,\ 0\right)$$

Evaluating when the operation is $\boxed{=}$, is to display the value in the *Store*.

$$[3]\quad eval\text{-}op\left(\boxed{\begin{array}{c}\boxed{Num}\\ Ops\end{array}}\ ,\ \boxed{=},\ Store\right) \;=\; eval\left(\boxed{\begin{array}{c}\boxed{\text{Store}}\\ Ops\end{array}}\ ,\ Store\right)$$

Evaluating when the operation is $\boxed{+}$, is to query for a number and display the result of the query, as well as storing the sum of the new number and the old store as the new store.



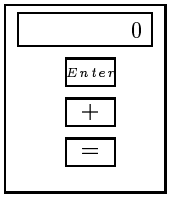
Note that Num' occurs in two places on the right-hand side. This does not mean that the interaction would be twice but is equivalent to having an auxiliary function that distributes the result of interaction to the two occurrences.

4.3 Interaction issues

Thus far, we have touched upon the syntax and semantic aspects of the calculator. In this section, we discuss how all these specifications can be brought together to yield a practically useful set of tools for an end user environment for this language.

4.3.1 The term editor

The term editor, which is generated from the syntax specification of a language, allows the creation of terms of that language. For the *Calculator* language, the *Calculator Term Editor* allows the creation, for example, of the following term:



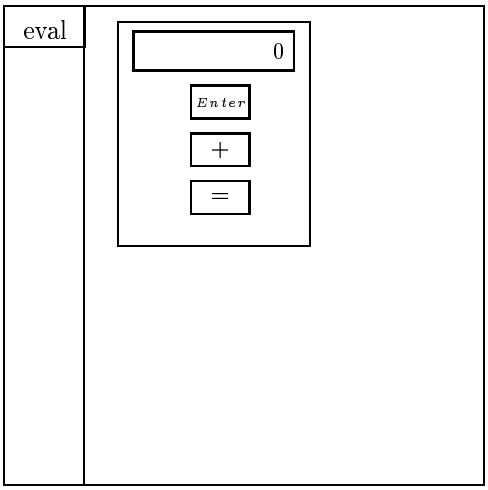
4.3.2 Input and output representation

When an input request is presented the user can replace the meta-variable with a permissible replacement as dictated by the language syntax, which is always type correct and represented just as the syntax is defined (graphical input). Thus, the variable which was unbound becomes bound after the user interaction. In the case of human interaction we may very well prefer to present the input request in a more user-friendly manner. For example, we may prefer to have: "Please enter an option: ⟨OP⟩".

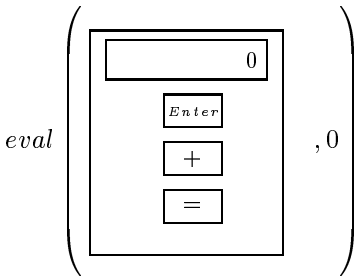
4.3.3 Term reduction

After a visual program (a term) is constructed we want to execute it using the semantics. To do so, we apply the *eval* function defined in Section 4.2.

To start the scenario, first a calculator term must be created. This is done in a term editor over CALC:



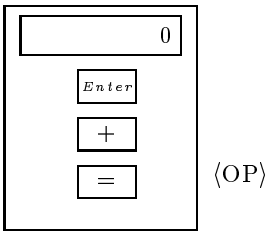
In this editor, the eval button is defined to apply the "eval" function to the *CALC* term constructed in it[3]:



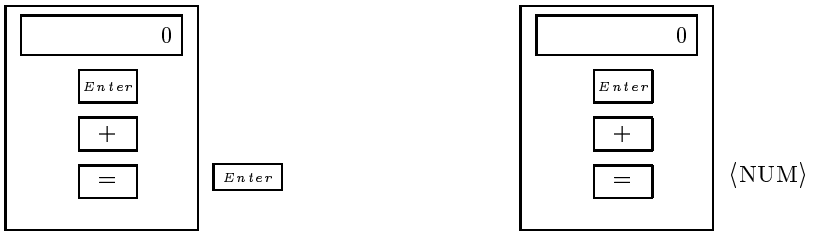
Now let us follow a scenario to see how the equations deal with input and output during evaluation. Note that, at the time of evaluation the exact appearance of the calculator is determined. The actual ordering of the operations is determined when the calculator term is constructed. The syntax, in fact, permits any ordering or even repeated occurrences of the operations as long as there are three operations. After the evaluation is requested, this calculator term is continually rewritten driven by the input received.

[3]The term editor supports the binding of a function from a language specification to a button.

The rest of the scenario shows the term in the editor as it is rewritten. The equation number references are from Section 4.2. Applying the *eval* function to the term, invokes an external-match due to the [ ] (i.e., χ term) present in equation [1], which presents the term to a user:



The meta-variable demands input from the user, who can syntactically choose from a menu which presents the permitted operation or select an appropriate subterm from the existing term. The latter choice means that the user can select any operation from the calculator term. If the user selects [Enter] then the term becomes the one on the left below, using which the *eval-op* function matches equation [2] which invokes yet another I/O (the right term):



Now, in order to continue, a number must be provided. Considering that the number 5 is entered, the rewrite of equation [2] can be completed, which is another *eval* function matching equation [1] again.

Notice that the terms driven by input and output are presented in a window. Clearly, additional research is required to address the various ways that intermediate terms as well as the input and output can be presented. In the report [DÜ96] we investigate how information could maintained (called Share-Where maintenance) so that the initial look of the calculator is preserved through the interaction. We have not discussed the issue of how certain window control information can be incorporated.

5 Conclusions

Direct manipulation user interfaces consist of interactive widgets of various kinds. Many of them are event based (assist in handling the various input events) but a variety of them are geometry based [OJK95]. We are interested in not only composing these geometry based widgets to build direct manipulation user interfaces but also provide meaning to these compositions using equations. To describe this we abstract away from event based user interfaces by assuming a syntax based editor that helps build the desired "widgets".

We present a simple model for guiding user interaction, that with the help of certain editor tools, and mechanisms for defining user short-cuts (some event based "widgets") would result in practical user interfaces that are more flexible than ones of today — that only allow connectors between components of the user interface while the semantics is specified in a language (like C++ or C) external to user interface specification language.

Acknowledgements

We would like to thank Jan Heering and Arie van Deursen for their comments and opinions on Section 3.

References

[BK86] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

[DÜ96] T. B. Dinesh and S Üsküdarlı. Specifying input and output of visual languages. Technical Report P9610, Programming Research Group, University of Amsterdam, August 1996. Extended version with Share-Where maintenance; ftp://ftp.fwi.uva.nl/pub/programming-research/reports/1996/P9610.ps.Z.

[Gra95] T.C. Nicholas Graham. *Declarative Development of Interactive Systems.* Number NR.243 in GMD-BERICHT. R. Oldenbourg Verlag, Munchen/Wien, 1995. PhD Thesis; Technical University of Berlin.

[OJK95] Dan R. Olsen, Brett Ahlstrom Jr., and Douglas Kohlert. Building geometry-based widgets by example. In *Proceedings of CHI'95*, 1995.

[ÜD95] S Üsküdarlı and T. B. Dinesh. Towards a visual programming environment generator for algebraic specifications. In *Proc. 1995 IEEE Symposium Visual Languages*, September 1995.

[Üsk94] Susan Üsküdarlı. Generating visual editors for formally specified languages. In *Proc. 1994 IEEE Symposium Visual Languages*, October 1994. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Usk94.ps.Z.