A Visual Application Generator for the Chronobot

Shi-Kuo Chang, Mehmud Abliz, Yaw Gyamfi and Chang Liu
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260 USA
{chang, anti, mehmud, changliu}@cs.pitt.edu

Abstract: We describe the design of a visual application generator, which creates customized interfaces that can be easily modified and deployed. With this visual application generator, the user can specify the requirements using an easy-to-understand iconic language. Unlike most other visual programming languages the icons are allowed to have multiple meanings. This unique feature makes our tool easy to use by the end user. The visual application generator for the chronobot is described in detail, including the customized syntactic/semantic rules to translate visual sentences into the application program.

1. Introduction

In this paper we describe a visual application generator designed based on the visual language theory. There are several advantages: First, using a visual language to describe the problem for the application generator is more concrete, direct, and convenient than using text. Second, users can express their ideas more easily and freely in the 2D space available. Third, our tool provides a reconfigurable user interface which makes full use of commercially available code generation toolkits to provide a general purpose visual application generator. Fourth, the tool enhances flexibility by supporting dynamic addition/removal of icons (the building blocks of visual sentences) while using the same parser in the core. Last but not least, the icons can have multiple meanings, which are automatically disambiguated by the parser so that the tool is easy to use by the end user.

We try to make this tool a general purpose visual application generator that can be customized and retargeted for different application domains. However, the target application of this tool is the chronobot project.

A chronobot is a device for storing and borrowing time [4]. Using the chronobot one can borrow time from someone and return time to the same person or someone else. It is a convenient device for managing time. The underlying premise of the chronobot is that there is a way to exchange time and knowledge. For example one spends time to acquire knowledge and later uses this knowledge to save time. A group of people can also find some means to exchange time and knowledge. Thus the chronobot is a device to facilitate the exchange and management of time and knowledge for the benefits of a community.

The chronobot was implemented as a web service and applied to e-learning (for learners to exchange tutoring time and course-related knowledge), job placement (for job seekers to find temporary/permanent jobs and exchange job seeking knowledge) and senior citizen care (for health care providers and family members to exchange health care service and knowledge). For the quick deployment of the chronobot service it is desirable to have a chronobot application program generator. An application generator is a software program that generates application programs from intuitive descriptions of the problems to be solved, rather than by traditional programming methods. It enables users to quickly create a program of their own with minimal effort and little programming knowledge. With an application generator, a user only needs to specify the steps required for his or her program, with very little or even no codes to be written. In what follows we describe the user interface (Section 2), the visual language (Section 3), the visual application
generator (Section 4) and an experimental system (Section 5) in detail.

2. The User Interface

The visual application generator’s user interface is shown in Figure 1. It provides the users with a panel and buttons with icons. Using this user can compose iconic sentences. The iconic sentence is composed in a 2D space using Hor, Ver, and Overlay operators on two icons or two sentences or on icon and sentence. Since there is no grid to restrict the programmer, he/she is free to place icons anywhere on the screen. The onus is on the programmer/user to specify the groupings. For instance, if the user wants to write icon1 hor icon2, the user must utilize the hor-group operator provided in the editor. The operators are color coded for convenience: the horizontal operator is colored in blue, the vertical operator in yellow and the overlay operator in red.

After the user has finished composing the iconic sentences, the parser parses the icons (tokens) to check for syntax and semantics. It returns the groups of icons that form distinct sentences or the icon that is responsible for an error if any.

The tokens are the output from the visual editor. Embedded in the visual sentence parser is a grammar definition for the iconic language. The visual sentence is parsed by the parser, which creates a simple parse tree and the code generator takes on from there.

3. The Visual Language

A visual language is a set of visual sentences. Each visual sentence is a spatial composition of object icons and/or operation icons. It represents a complex conceptual entity or a sequence of operations. An icon \( I \) is represented in the following form: \( I = (IM, PM) \), where \( IM \) and \( PM \) are the physical image and the set of semantic meanings or the attributes of icon \( I \) respectively. Each icon can have a single physical image and multiple attributes. If the corresponding attributes of two icons are matching based on some definite operations, a new meaning will be generated as the result of combination. Thus, depending on their attributes, several icons could be combined by operators to generate a visual sentence.

The general domain-independent spatial icon operators are Ver (for vertical composition), Hor (for horizontal composition), Ovl (for overlay composition) and Con (for conceptual composition). The first three are implicitly represented in the spatial relation between the two icons, while the last one is explicitly represented as a connected line and often used to connect two concepts.

In our work we replace the last conceptual composition operator with a group operator Grp to represent the group composition. The usage of this Grp operation is similar as the usage of bracket in mathematical expressions. If we specify that some icons are in one group, these icons will be parsed first. Thus, the group operation allows us to make several icons into a group with high priority. This group will be regarded as a new entity in the rest of the parsing process.

The icons may have multiple meanings. Such ambiguous icons are disambiguated when the icons are combined with other icons by the
operators. For example the icon “hammer” means either “bid” or “place a bid” depending on its position. If the icon “hammer” is used as the first operand of the vertical, overlay or horizontal operator its meaning is “bid”. If it is used as the second operand its meaning is to “place a bid”. In general icons can be disambiguated depending on what other icons are present, what operators are applied to them, and their positions as operands. The user can freely enter a visual sentence and the parser will generate the translated sentence and present the sentence, or the code, to the user for verification purpose. After a while the user will gain an intuitive understanding on how to use the icons to compose visual sentences.

3.1. Syntax

In order to design the visual interface of the application generator we define a context-free grammar, which could be customized for different applicant areas such as Chronobot, Chinese characters, etc. The grammar is defined as: $G = (N, X, OP, s, R)$

where $N$ is the set of non-terminals, $X$ is the set of the terminals (or icons), $OP$ is the set of spatial relation operators, $s$ is the start symbol, and $R$ is the set of production rules [A].

Based on the above definition, we further define a context-free grammar. To describe the grammar directly, we used the pair of bracket symbols $[...]$ to represent the Grp operation. We also defined the priorities for the four operations in decreasing order: $Grp > Ovl > Ver > Hor$.

$N$ (non-terminal set) := \{Sentence, Term, Factor, Group\}

$X$ (terminal set) := \{icon\}

$OP$ (operator set) := \{Ver, Hor, Ovl, [ ]\}

$s$ (start symbol) := Sentence

$R$ (production rules set) := \{
    Sentence := Sentence Hor Term | Term
    Term := Term Ver Factor | Factor
    Factor := icon Ovl Group | Group
    Group := icon | [Sentence]
\}

In this grammar, we could group the icons in sequence together by using the operation $[...]$ first, and then we could take this group as a new icon which can be arranged with other icons to build up valid sentences with any complexity.

3.2. Semantics

We introduce the semantic rules for the above grammar in two parts. Firstly, the general attributes of terminals and non-terminals are described. Secondly, the semantic rules for computation are described based on the definitions of attributes. To simplify the descriptions, the terminals and non-terminals will be both called as objects.

3.2.1. Attributes of terminals and non-terminals

There are two types of attributes in this grammar: unitary ones and binary ones.

A unitary attribute represents the meaning of current object when it is used alone without any syntactic operation. At most time, it represents the direct meaning, such as “what it looks like” and etc. Thus, we used .meaning as the unitary attribute.

The binary attribute is to represent the meaning of current object when it is in a syntactic operation. Since each syntactic operator has only two operands, the binary attribute of an object must be able to distinguish the different locations where the object is in. For each object, it has two binary attributes $op_1$ and $op_2$ related to one syntactic operation. $op_i$ means the object is the $i$-th operand in the current operation $op$.

3.2.2. Semantic rules

We are interested in three types of semantic operators: likeness, location and conceptual joint. Likeness means that what the icon looks like, which is related only to a single icon. Location means that the different locations of icons represent different meanings. Location operator is corresponding to the overlay operation of two icons. Conceptual joint operation means that linear combination of two icons represents the concept. This operation is represented by vertical or horizontal operations.
The general descriptions for the above three semantic operations are as listed in the Table 1:

<table>
<thead>
<tr>
<th>Semantic Operation</th>
<th>Likeness</th>
<th>Location</th>
<th>Concept</th>
<th>Joint</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Description</td>
<td>Get the direct meaning of current object</td>
<td>1. get location of I1, then can get the attribute .ovl1</td>
<td>1. get location of I1, then get the attribute .hor1/ver1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. get location of I2, then can get the attribute .ovl2</td>
<td>2. get location of I2, then get the attribute .hor2/ver2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. new attributes = I1.ovl1 + I2.ovl2</td>
<td>3. new attributes = I1.hor1/ver1 + I2.hor2/ver2</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. General Description of Semantic Operations.

Thus, we define the semantic rules as in Table 2, the symbol .attributes means the set of all attributes to the current object.

Table 2. Semantic Rules.

4. The Chronobot Application Generator

In this section we will take the Chronobot as an example to illustrate how the grammar and semantic rules work for a particular application. These rules are generated from the grammar described in Section 3, with some special characteristic for the Chronobot application. In other words, the grammar we defined above is general to many applications. But for each specific application, we can customize the icon definitions and semantic rules.

4.1. The Visual Language

4.1.1. Syntax

In the Chronobot application, there are two types of functionalities. The simple one describes only one user action, while the other one describes more than one user actions. In other words, the second functionality, we call it complex action, is composed of several simple functionalities. Thus, we will describe the grammar for complex functionalities based on the description of simple ones.

We are interested in three major functionalities of Chronobot application: add function (which adds something into others), list function (lists something required by users) and delete function (delete or remove some existing entity). We are also interested in the functionalities which could be represented by only one icon in the visual language.

Derived from the generic grammar shown in Section 3, we defined a specific grammar for the Chronobot application as follows. The valid
visual sentence should be simple or complex functionalities. We will use the symbol CSimple to represent the simple ones, and CComplex for the complex ones. Also, the terms as Add_X in the following grammar represent the add functionality, and the same for List_X, Delete_X and etc. The operations are the same as them in the generic grammar. In order to differentiate from the generic definition, we use the terms begin with the capital letter C for the Chronobot application. We divided the grammar into three parts:

(i) Description for valid visual sentence:

\[ CSentence := CSimple | CComplex \]

(ii) Description for Simple functionalities:

\[ CSimple := Add_X | List_X | Delete_X | X \]

\[ Add_X := Add_icon Hor X \]

\[ List_X := List_icon Hor X \]

\[ Delete_X := Delete_icon Hor X \]

\[ X := CIcon | CIcon Hor X \]

Here, the term Add_icon, List_icon and Delete_icon represent the icons corresponding to the actions as their names shown.

(iii) Description for Complex functionalities:

\[ CComplex := CSimple Ver CFactor \]

\[ CFactor := CSimple \]

\[ CFactor := CSimple Ovl CGroup \]

\[ CGroup := [CSentence] \]

\[ CGroup := CSimple \]

4.1.2. Semantics

Before we describe the semantic rules for the grammar defined in section 4.1.1, we will first introduce two types of predicates. There are two similar predicates in each type. They are used to detect if the current object in the grammar rule has the corresponding attributes. The predicates are:

Overlay Predicates: is to detect if the current object \( cobj \) has the attributes \( ovl_1 \) or \( ovl_2 \). If it has, then the predicate will return a true value, otherwise it will return false.

Vertical Predicates: is to detect if the current object \( cobj \) has the attributes \( ver_1 \) or \( ver_2 \). If it has, then the predicate will return a true value, otherwise it will return false.

Based on the above two types of predicates, we will define the semantic rules for the grammar in Section 4.1.1 as shown in Table 3. We also divide them into three parts as before: (i) description for valid visual sentence such as Rule 1 below, (ii) description for simple functionalities such as Rule 3 below, and (iii) description for complex functionalities such as Rule 12 below. The following table gives details for the three examples of semantic rules.

<table>
<thead>
<tr>
<th>Seq.</th>
<th>Syntactic rules</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( CSentence := CSimple )</td>
<td>Create a new object ( R ) for sentence; ( CSentence\text{.attributes} = CSimple\text{.attributes} ) Return ( R );</td>
</tr>
<tr>
<td>3</td>
<td>( CSimple := Add_X )</td>
<td>Create a new object ( R ) for ( CSimple ); ( CSimple\text{.attributes} = Add_X\text{.attributes} ) Return ( R );</td>
</tr>
<tr>
<td>12</td>
<td>( CComplex := CSimple Ver CFactor )</td>
<td>Create a new object ( R ) for ( CComplex ); If ( \left( (Ver_1(CSimple) == \text{True}) &amp;&amp; (Ver_2(CFactor) == \text{True}) \right) ) then ( CComplex\text{.attributes} = CSimple\text{.ver}_1 + CFactor\text{.ver}_2 ) Return ( R );</td>
</tr>
</tbody>
</table>

Table 3. Three examples of semantic rules.

4.2. An Example

To demonstrate the expressiveness of the grammar, we give a specific example, which is related to the Chronobot application [C]. There are three related icons in this example:

1. Icon_room
2. Icon_list
3. Icon_hammer

The semantic attributes for each icon is as following (the symbol “/” means that there is no corresponding semantic meaning for the attribute by far, but may be used for error detection in the future):
Table 4. Semantic attributes for the icons.

Thus, the syntactic structure for the generated sentence “List all the bid rooms I placed a bid” could be represented as:

\[( \text{Icon-list } \text{Hor}_1 \text{Icon-hammer } \text{Hor}_2 \text{Icon-room} ) \text{ Ovl } \text{Icon-hammer} \]

Figure 2 shows the parse tree for this syntactic structure, where each node represents a terminal or non-terminal and each arc represents an operation between nodes.

Figure 2. Syntactic Structure for the Example Sentence.

The above parse tree is what will be generated by the VLParser. A subsequent traversal of the parse tree will relay the results of the parser to the code generator. In the case of the previous example, the output from the parse tree traversal will be:

\[ \text{Icon-list.hor}_1 \text{ Icon-hammer.hor}_2 \text{ Icon-room.hor}_2 \text{Icon-hammer.ovl}_2. \]

Invoking the names of the corresponding attributes, the sentence will be read as list bid_room room if bid=true. This sentence can be read as list all bid rooms for which the condition place bid is true. The visual sentence is shown in Figure 3.

Figure 3 shows a sample screen dump. Icons within the same semantic unit are grouped under one color. Currently it is the user responsibility to specify the groups. This was found to be the easiest way of developing the interface since parsing is done in multiple directions. The only other alternative was to have the system try all acceptable ordering, which would obviously not be efficient if the number of icons in the sentence are beyond a certain number and the sentence employs multiple operators. For instance, the coding of the above example will be as follows: [1 HOR 2 HOR 3] OVL [2] where ‘1’ refers to the first icon “list”, ‘2’ the second icon “hammer” and ‘3’ the third icon “room”.

Figure 3. Visual sentence (upper box) and screen dump for code (lower left box).
5. An Experimental System

The experimental system comprises the visual language editor, the parser and the application generator. The experimental system serves as a proof-of-concept prototype. The visual editor takes in the user/programmer’s input, converts it to a string of tokens which is then passed to the parser and subsequently the code generator. In our example, the code generator is a web-page generator that builds dynamic pages with “jsp.”

In this section we describe the screen dumps of the examples shown in Section 4. The first example is list bid rooms. It consists of three icons: icon_list, icon_hammer and icon_room grouped by the ‘horizontal’ operator. The second example is bid and remove bid. This example uses two distinct icons; icon_hammer (for bid) and icon_delete (for delete). The sentence for this example is icon_hammer ver [icon_delete hor icon_hammer], where the ‘ver’ operator stands for conjunction. To form this sentence with the visual editor, the user/programmer selects the appropriate icons, arranges them in the correct order and envelops them in the required operator from the ‘Tools’ menu. In this case icon_delete and icon_hammer will be placed in a ‘Horizontal’ operator which for the following group: [icon_delete hor icon_hammer] (1). The next step is to enclose (1) and the stand-alone icon_hammer in the ‘Vertical’ operator, signifying a conjunction, resulting in [icon_hammer ver [icon_delete hor icon_hammer]].

The following figures show the Visual editor’s screen dumps. Figure 4(a) shows the screen dump for the process of creating the icon groups for [icon_list hor icon_hammer hor icon_room] while Figure 4(b) shows the screen dump for [[icon_hammer] ver [icon_delete hor icon_hammer]]. In the code window superimposed on the editor, the resulting token string is represented as a combination of numbers. These numbers serve as indices to the icon definition structure array for the parser. This was found to be the easiest implementation. The parser uses the index to retrieve the corresponding icon definition from the array.

The parser and the code generator work in the background. After the visual coding phase, the result (in the superimposed code window) is fed to the parser which produces an output of the form: bid and delete bid, for the sentence in Figure 4(b).

6. Discussion and Future Work

There are many visual programming approaches [2, 3, 8]. Our work is more closely related to the following three: Chimera, which is a visual programming system for the end user based upon the concept of program-by-examples [6]; Viper, which is a general-purpose visual...
programming environment based on the data-flow model with a fully programmable interpretive command language to define the flow graph operators [1]; and Hi-Visual, which is a iconic programming system based upon icons [5]. The major difference between these approaches and ours is that in our approach the same combination of icons can have several different meanings, and the meanings are determined by the parser. In other words, in our approach ambiguity in the visual expression is regarded as a virtue rather than a vice. It simplifies the user’s task of specifying the application program. In this regard our work is influenced by Minspeak, which is a visual language for augmentative communication using multiple-meaning icons [7]. The difference is that Minspeak is a linear visual language while our visual language is two-dimensional.

When new functions are to be added to an application, the user can enrich the icons by assigning them new meanings. For example the icon “hammer” together with the icon “user” can be assigned a new meaning “select a bid”. The user interface can be extended to allow user’s enrichment of icons. The user can assign multiple meanings to an icon in the enrichment process. However at the end such ambiguity will have to be resolved by the user and a well-defined application program is generated. This unique feature makes our tool easy to use by the end user.

An open research topic is whether the user can be assisted in the systematic introduction of new meanings to an icon, for example the first operand always signifying one thing, and the second operand signifying another. The explicit specification of such rules will lead to a “universal icon algebra”. However one can also argue against such an approach because it will ultimately restrict the creative use of ambiguity.

References:


