Modelling systems that integrate programming language and environment mechanisms

Keith J. Ransom and Chris D. Marlin

Department of Computer Science
The Flinders University of South Australia
Adelaide, South Australia

{keith,marlin}@cs.flinders.edu.au

Abstract

Once we relax the assumption that it must be possible to specify programs solely in terms of text and enter them in isolation from other tools, the range of possible program development mechanisms is significantly increased. Thus, in the light of advances in the field of integrated software development environments and in view of the wider availability of suitable workstations, we should reconsider the way we perceive (and, hence design) programming languages. This paper describes on-going work aimed at exploring the role of the programming language in the context of modern software development environments. The work is currently focused on two fronts: the development of a formalism for describing both a programming language and associated environment mechanisms, and the design of environment mechanisms that support software maintenance and reuse, complementing those traditionally provided by programming languages. This paper will focus on work in the first of these two areas.

1 Introduction

Advances in hardware technology inevitably lead to changes in software technology. The advent of VLSI technology has enabled the production of low-cost high-powered CPUs and dedicated graphics hardware, leading to the proliferation of personal computer workstations with high-resolution displays and advanced graphics capabilities. In many application areas, this has caused a shift away from batch-oriented software, designed to run with a minimum of user interaction (often relying solely on file-based input-output), towards highly visual, highly interactive applications which aim to allow the user to enter input and view output in a more appropriate manner, and provide valuable feedback in a timely fashion. One application area in which this shift has been particularly pronounced is that of engineering calculations related to electrical circuits, where calculations once carried out in a batch-oriented way are now part of interactive CAD systems. One application area in which the shift has been surprisingly slow to take hold is that of software development itself. Despite the fact that the technology for program entry has progressed through a series of stages from paper tapes and punch-card systems, through to screen-based editors with graphical user interfaces, the precept that program code is merely a sequence of text characters has remained largely unchallenged. Indeed, for the majority of programming languages used today, even those designed during the workstation era, programs could be specified equally well using punch-cards (speed of entry notwithstanding) as they could via an editor with a graphical user interface.

The basic tenet of our work is that programs should always be developed under the control of an integrated software development environment on a modern workstation; in such an environment, a coherent collection of software tools share representations of the artefacts they manipulate, and may operate in synchrony without explicit invocation by the user (where appropriate). Such a level of integration implies that the representation of a section of code under construction need not match the one used to enter it, nor the form in which it is displayed; this is contrary to traditional program development where programs are entered, changed and displayed in textual form. Stipulating a development environment with support for bitmapped graphics implies that we can abandon the

0-8186-7171-8/95 $04.00

linear sequence of characters as the canonical representation of programs under construction. Hence, more appropriate visual representations of existing language constructs can be employed, and new constructs can be designed which may have been overlooked previously for lack of a convenient textual means of specification. Although "two dimensional" program layouts with various graphical elements (lines, boxes, circles, etc.) can still be supported in non-itegrated, batch-oriented environments by employing parser technology, the range of notations that may be used is limited to those for which parsing strategies exist (such as those described in [1]); there is no such restriction in an integrated environment, since the structure of a program is not inferred from its visual appearance, but rather from the operations used to construct it.

Given that the above tenet directly impacts the kinds of programming language constructs that are feasible, a goal of our work is to re-examine the role of programming languages (and hence, their design) in the light of advances in the field of integrated software development environments. In addition, we seek to extend the benefits of certain "programming mechanisms" to documents from other stages of the development life-cycle (such as design documents and software process descriptions, for example). Questions that arise from such considerations include:

1. What should the definition of a programming language encompass?
2. Where is the boundary between the programming language and environment?
3. Which parts of a language and environment should be formally specified, and which of these parts should be regarded as "standard"?
4. Which mechanisms are best provided by the programming environment, and which by the language?
5. What are convenient forms for entering, and for displaying, common programming language constructs?

We are currently developing a formalism for describing both a programming language and associated environment mechanisms. Section 2 classifies various aspects of the extended notion of program construction that we desire to support, and discusses the first three of the questions stated above. In Section 3 we introduce the concept of a binding table which employs a complementary combination of language and environment mechanisms. The formalism we have developed in order to specify mechanisms which embody our extended notion of program construction is described in Section 4; the example developed in the Section 3 is used to illustrate the various aspects of the formalism. Some tentative conclusions, and on-going work related to the last two questions above, are discussed in Section 5.

2 Extending the notion of program construction

A description of a traditional text-based programming language typically involves the specification of the textual symbols which can be arranged to form a program, the valid combinations of such symbols, as well as a description of the meanings ascribed to the various combinations. In such descriptions, the form used to construct programs, the form used to display them, and the form with which the semantic rules are directly associated, are necessarily the same. As part of our investigation of how programming languages might differ if we assume the facilities of a workstation-based integrated software development environment, we wish to avoid such a restriction. Thus, we have adopted an alternative view of programming language descriptions which reflects our desire for a clear divide between the conceptual notion of a program and the way that one is built. We view the description of a programming language *per se*, to be that which describes the fundamental conceptual elements of which programs are comprised (such as statements, declarations, and expressions, for example), as well as rules which state which programs have valid interpretations (i.e., rules defining the static semantics) and rules which specify what it means to execute a valid program (i.e., those relating to the dynamic semantics). Such a description may be useful for reasoning about programs, writing compilers, etc., but contains no information about how to build or display a program. These activities, building and displaying programs, are dependent upon the environment in which programs are to be developed. Hence, for those interested in how to construct a program in a given language, or how to understand a visual representation of such a program, a programming environment description is also necessary. A programming environment description should define the various construction and display operations in terms of manipulations of the conceptual elements described in the language description, and in terms of the facilities offered by the environment (such as mouse, keyboard, text and

graphics, for example).

As stated previously, we wish to support the development of programs in the context of a workstation-based integrated software development environment, making appropriate use of all available input and display technologies, not merely keyboard and text. Thus, we seek a formalism suitable for providing both programming language and environment descriptions of the form discussed above, descriptions to be employed by users and designers of program development mechanisms. More specifically, we have used the following criteria to decide upon a suitable formalism:

- The formalism must be suitable for modelling the static semantics of "conventional" programming language mechanisms. Since we are concerned with the impact of software development environments upon program construction, and not program execution, we currently have no requirement that the formalism should support the specification of dynamic semantics.

- The formalism must be suitable for modelling interaction with the program representation (an arrangement of the constructs described in the language definition) via environment mechanisms. For a given environment mechanism, it must be possible to describe the actions required by the user to invoke the mechanism, as well as how such invocation affects the program.

- The formalism must also be able to specify how the various program constructs are displayed. In addition to plain text, graphical elements (such as lines, circles, boxes and windows) should be supported.

- Certain forms of "code" which would typically be displayed in a non-textual way, do not map well to hierarchical program representations; in particular, they correspond to a graph-like arrangement of program constructs. The formalism should allow the description of such notations, which are common in the pre-implementation stages of the software development life-cycle.

- Abandoning batch-oriented development tools in favour of an integrated programming environment admits the possibility of incremental semantic analysis, providing more rapid feedback for the programmer and giving meaning to "incomplete" programs; furthermore, the ability to ascribe meaning to partially specified programs is necessary for supporting the reuse mechanisms that we also wish to investigate. Thus, the formalism must be amenable to supporting the generation of a semantic analyser that acts in unison with the various program construction mechanisms.

- Documents ("code") from the various stages of the software life-cycle do not exist in isolation, rather they are often related in meaningful ways ("program A implements design B", for example). The formalism must be able to capture the semantics of such relationships, and support the specification of environment mechanisms which span multiple "languages".

3 A binding table mechanism

One way in which the creation of generic code components can be fostered is to supplement the text-based name binding mechanisms of traditional programming languages with a mouse-oriented binding interface (or one employing any other pointing device) and an accompanying graphical display, whilst retaining a textual specification of the algorithmic aspects of a component.

A tool which illustrates this concept is shown in Figure 1. The editor shown consists of two windows: the binding table window and the algorithm window. The binding window is used to display each of the names that have no binding defined within the section of code (algorithm) shown in the algorithm window. A binding table and associated algorithm constitute a particular usage of the algorithm; a single algorithm may be associated with many different binding tables depending upon the context in which it is being used (or reused). Each entry in the binding table indicates the meaning of the given name in the current usage of the algorithm; these name bindings may be established or modified by pointing to other binding tables or code representations.

The binding table mechanism is intended to support the separation of the algorithmic detail from that which relates to the binding of names, partitioning code creation into two distinct activities: algorithm development and binding. Thus, during algorithm development, the programmer is able to focus not on the particular values of the bindings for the application being created, but rather on the "generic" algorithm being constructed.

Providing support for name binding via a graphical interface, within the context of an integrated software development environment, offers a number of ad-

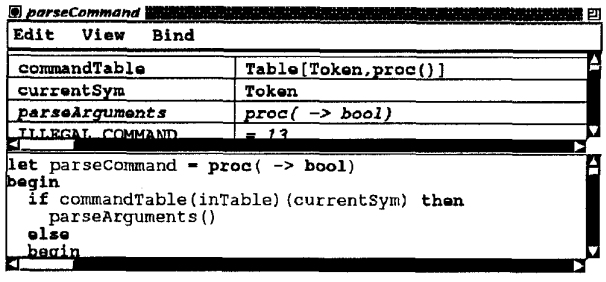

Figure 1: A source code editor with associated binding table.

vantages over text-based language mechanisms. The names appearing in a section of code which are intended to act as notional parameters to the code (i.e. those that would change across configurations) are clearly displayed, separate from the body of the code. Thus, locating and re-binding such names is simplified.

Adopting a "point and click" approach to name importation saves much laborious typing, when compared to mechanisms such as Ada generics [2], for example. It is unnecessary to modify sections of code that define entities to which identifiers are bound, simply to ensure matching names. Hence, an algorithm can be coded using names relevant to its domain, rather than to those relevant to the implementation domain.

The binding table mechanism is, to some extent, language independent, in that from a description of the binding table and the specification of a given language in our formalism, a tool could be produced that supports the use of the binding table for the given language. Thus, the binding table represents a useful complement to languages such as C and Pascal, which have comparatively unsophisticated data control facilities.

Rather than exploring further the advantages of the binding table *per se*, this paper will merely use it as an illustrative example of how the notion of a programming language may change if we assume that software development occurs within an integrated software development environment with interaction facilities typical of those available on current workstations. The binding table comprises a mixture of linear textual programming language notation (used in the algorithm specification part), two-dimensional representation (the table part) and an interface centred around a pointing device (used for making bindings).

The remainder of this paper will use this example as the basis of illustrations of our formalism for describing programming notations which are not restricted to solely textual forms.

4 Modelling program development mechanisms

We are currently developing a formalism that meets the requirements stated in Section 2, the intention of which is to enable the description of language and environment mechanisms in a clear and unambiguous manner, and to allow the implementation of such mechanisms to proceed automatically from their formal description. Using the formalism, the specifier builds a multi-layered model of a language and associated environment mechanisms, layered in the manner illustrated by Figure 2. In the figure, a horizontal line indicates that the layer above the line explicitly refers to information defined in the layer below the line.

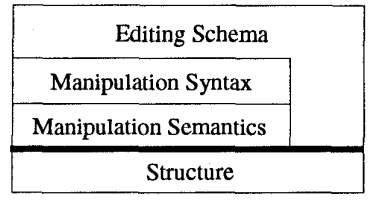

Figure 2: Model layers.

The lowest layer in Figure 2, the *structure* layer is composed of a declarative specification of the information structures that are used to represent sections of code for the language being modelled[1]; this includes a description of what it means for a piece of code to be in a consistent state.

Figure 3 illustrates the notation used in the structure layer. Lines 1, 30 and 36 of the figure show that constructs called `Procedure`, `Expression`, and `Identifier_Use` are being defined. Line 36 further indicates that `Identifier_Use` is a *sub-class* of `Expression`, and thereby extends its structure and semantic rules. Lines 3-6 define the parts of a `Procedure` that must be specified by the user. For example, Line 5 indicates that an `Procedure` has a part named `locals` which is itself a set of `Declaration` objects. A set is used in this instance since the order of local declarations has no semantic significance. Lines 9 and 10 specify that there are two semantic values (values that can be derived from a collection of objects represent-

[1]Recall that *code* need not imply the traditional notion of programs.

```
1  Procedure ::= {
2    syntactic parts
3      name           : string
4      parameter_list : list of Parameter
5      locals         : set of Declaration
6      code           : Block
7
8    semantic values
9      requires_binding : set of string
10     bindings         : set of Binding
11
12   functions
13     Is_Parameter(name : string) : boolean ::=
14       ∃ p in parameter_list  (p.name = name)
15
16     Declared_Locally(s : string) : boolean ::=
17       ∃ d in locals (d.name = s) or Is_Parameter(s)
18
19   semantic rules
20     ∀ s in code.statements (s.procedure <- this)
21
22     ∀ name in requires_binding (
23       ∃ b : Binding (
24         b.name <- name ;
25         bindings contains b ;
26       )
27     )
28 }
29
30 Expression ::= {
31   semantic values
32     type : Type
33     procedure : Procedure
34 }
35
36 Identifier_Use : Expression ::= {
37   syntactic parts
38     name : string
39
40   semantic values
41     binding : Binding
42
43   semantic rules
44     if not procedure.Declared_Locally(name) then
45       procedure.requires_binding contains name
46
47     binding <- select b in procedure.bindings ( b.name = name )
48
49     type <- binding.target.type
50 }
```

Figure 3: Defining structure.

ing code), a set of strings called `requires_binding` and a set of `Binding` objects called `bindings`.

The nature of each semantic value is defined in the semantic rules. Semantic rules are essentially a series of predicates which, if satisfied, imply that the code is in a consistent state. The calculus employed when specifying such predicates was developed in accordance with our requirement that the formalism must be amenable to supporting the generation of a semantic analyser that acts in unison with the various program construction mechanisms. Such predicates may employ various simple operators (the full details of which are not given here) as well as functions defined by the specifier. Lines 12-17 define two functions `Is_Parameter` and `Defined_Locally`. The function `Is_Parameter` is used within the definition of `Defined_Locally`, which is itself used in the semantic rule on line 44. This latter rule effectively states that if an identifier appears within the body of a procedure, then its name should appear in the `requires_binding` set for that procedure. When processed by the incremental semantic analyser, the rule on lines 22-27 ensures that for every name in the `requires_binding` set of a procedure, a `Binding` object with that name will exist in the set `bindings` associated with the procedure. The rule on line 47 defines the *Binding* associated with an `Identifier_Use`. The rule on line 49 defines the type of the `Identifier_Use` expression in terms of that binding.

The *manipulation semantics* layer in Figure 2 describes the operations that may be performed on the structures defined in the structure layer. These operations may include functions similar to those used in the structure layer, as well as procedures (functions with no return type). Whereas the functions defined in the structure layer are those that are used in the definition of semantic rules, the functions defined in the manipulation syntax layer represent an interface to the structure from the upper layers. The behaviour of each procedure, such as `Bind` and `Unbind` in Figure 4, may be derived from the given predicate which is effectively an immediate post-condition for that procedure. Such predicates are restricted to those which may be enforced by our incremental semantic analyser. For example, when the procedure `Unbind` associated with `Procedure` in Figure 4 is invoked, the target associated with the given binding (the parameter to the procedure `Bind`) will no longer be defined.

The *manipulation syntax* layer shown in Figure 2 consists of a declarative specification which maps sequences of abstract events to the operations described in the manipulation semantics layer. For example,

```
1  Identifier_Use {
2    Bind(export : Binding) ::=
3      binding.target <- export.target
4  }
5
6  Procedure {
7    Bind(b : Binding, export : Binding) ::=
8      b.target <- export.target
9
10   Unbind(b : Binding) ::=
11     undefined (b.target)
12 }
```

Figure 4: Defining the semantics of manipulations.

the first rule in Figure 5 specifies that when an identifier usage is selected, followed by a definition exported by some module, and the abstract event *Link* is invoked, then the `Bind` operation (defined in Figure 4) should be called appropriately. Similarly, the second rule of the figure specifies that a binding may be formed by first selecting an entry in the binding table of a procedure instead of an identifier usage in the body of the procedure. However, these descriptions are at an abstract level, since we do not yet know what it means to "select an identifier" or "select an entry". That is, abstract events (like `Select_Id` and `Select_Entry`) defined in the manipulation syntax layer need to be bound to actual user interface events via a series of declarations within the *editing schema*, the uppermost layer in Figure 2. For example, Figure 6, lines 7 and 8 specify that the abstract event `Select_Entry` occurs as a result of the actual event `Double_Click`[2] being generated over one of elements of the set `bindings` (refer to line 10 of Figure 3) associated with the `Procedure`.

The editing schema also describe how the various structures defined in the structure layer should be displayed on the screen. This is acheived by simply by declaring the graphical objects that are used to compose the display of the given construct, and their relationship to one another. For example, line 2 of Figure 6 specifies that a `Procedure` is displayed using a `paned window`, the first pane of which is a *vbox* (line 3) which vertically tiles the views of each of the bindings associated with the `Procedure` (lines 4 and 5). Graphical objects such as those found in most mod-

[2]Strictly speaking, `Double_Click` is itself an abstract term, bound at the level of the particular windowing system being used.

```
1  Select_Id(id) Select_Export(module,export) Link =>  id.Bind(export)
2  Select_Entry(proc,binding) Select_Export(module,export) Link => proc.Bind(binding,export)
3  Select_Entry(proc,binding) Delete => proc.Unbind(binding)
```

Figure 5: Defining the syntax of manipulations.

ern window systems, have corresponding primitives in the formalism, such as `paned window`, for example).

Other graphical objects including `vbox` and `hbox` (following the page-layout model of TEX), support the tiling of components in various pre-defined ways. A generic `box` is provided for modelling free-form layout. Each editing scheme is effectively parameterised by the editing shema of its constituent parts. For instance, line 5 of Figure 6 specifies merely that the "view" of each `Binding` is contained within the `vbox`. The exact details of how a `Binding` is drawn are specified in its own editing scheme, along with the description of any events that may relate to it. It is important to note that, in general, there need not be a one-to-one correspondence between visual artefacts and the structures used to represent a section of code.

```
1  Procedure ::= {
2    paned window name (
3      vbox (
4        ∀ binding in bindings (
5          view binding
6
7          when Double_Click on binding do
8            Select_Entry(m)
9        )
10     ),
11     vbox (
12       ...
13     )
14   )
15 }
```

Figure 6: An editing scheme.

The emboldened line in Figure 2 indicates the division between the description of the programming language and the description of the programming environment. Which of the layers shown in Figure 2 should be considered as standard across all implementations of a language and environment remains an open question at this stage. Regarding only the *structure* layer as standard implies that providers of language systems are free to implement their own construction and display mechanisms. While this might imply that users can choose the environment that best suits their needs (a novice programmer may choose an environment different than that chosen by a skilled programmer, for example), it might also create difficulties for programmers shifting from one environment to another. In the past, programming language definitions ([2], for example) have defined programming language syntax down to the level of the format for particular tokens; user interface events, such as `Double_Click` can be regarded as the analogs of tokens in our extended notion of program construction; thus, there is some precedent for suggesting a level of standardization close to the top layer in Figure 2.

There is a large body of work that is relevant to the development of the formalism outlined above. In relation to the structure layer, there have been many formalisms proposed for describing the semantics of programming languages, including attribute grammars [3], two level grammars (described in [4]), production systems [5], and many more. Such formalisms have typically been designed only with traditional text-based languages in mind, or are not well suited to the generation of incremental semantic analysers; thus, we have chosen to employ our own notation inspired by the use of attribute grammars by Reps [6], Horwitz [7], Hedin [8] and many others, but extending the notion to support non-hierarchical program structures. Related to the manipulation semantics layer are approaches such as those in [9] and [10]. Relating to the upper two layers of Figure 2 are unparsing schemes for text-based programming languages (such as that in [11]), and work such as that in [12, 13] on the construction of user-interface facilities suitable for programming environments.

5 Summary

On-going work on the design of complementary programming language and software development environment mechanisms to support software engineering

activities has been outlined. This work is based on the assumption that software components will always be developed and composed within an integrated software development environment of some kind.

An important part of this work has been the development of a formalism for describing programming language semantics which encompasses the extended notion of "language" implied by having combinations of traditional programming language features working in concert with software development environment mechanisms. This formalism was outlined in Section 3 of the paper, and is currently being used in the description of some examples of combined language/environment program construction paradigms. Two such examples are discussed below[3]:

- One way in which the creation of more generic code components can be fostered is to supplement the text-based name binding mechanisms of traditional programming languages with a pointing-oriented binding interface and accompanying graphical display, whilst retaining a textual specification of the algorithmic aspects of a component.
- In order to support some level of post hoc reuse, and to ease the task of maintaining multiple versions of related code, an environment mechanism is being developed which supports the creation of *derived components* by monitoring the way in which an existing component is modified when deriving a new component from it. By establishing the relationship between a derived component and the component from which it is derived, it is possible to automatically update the former as a result of modifications to the latter.

References

[1] S. S. Chok and K. Mariott. Parsing visual languages. *Australian Computer Science Communications*, 17(1):90–98, 1995.

[2] Reference manual for the Ada programming language. Technical Report ANSI/MIL-STD-1815A, United States Department of Defense, 1983.

[3] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[4] J. C. Cleaveland and R. C. Uzgalis. *Grammars for Programming Languages.* Elsevier, New Holland, Inc., New York, 1977.

[5] H. F. Ledgard. Production systems: Or can we do better than BNF? *Communications of the ACM*, 17(2):94–102, 1974.

[6] T. Reps. *Generating Language-Based Environments.* M.I.T. Press, Cambridge, Massachusetts, 1984.

[7] S. Horwitz. Adding relational query facilities to software development enviornments. In H. Ganzinger, editor, *ESOP88: 2nd European Symposium on Programming*, volume 300, pages 269–283. Springer-Verlag, New York-Heidelberg-Berlin, 1988.

[8] G. Hedin. An object-oriented notation for attribute grammars. Technical Report LU-CS-TR-89-42, Lund Institute of Technology, Lund, Sweden, 1989.

[9] L. R. Dykes and R. D. Cameron. Towards high-level editing in syntax-based editors. *Software Engineering Journal*, 5(4):237–244, 1990.

[10] F. Arefi, C. Hughes, and D. Workman. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3):349–360, 1990.

[11] N. Habermann, R. Ellison, R. Medina-Mora, P. Feiler, D. S. Notkin, G. E. Kaiser, D. B. Garlan, and S. Popovich. The second compendium of gandalf documentation. *Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1982.*

[12] M. Young, R. Taylor, and D. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, 1988.

[13] P. Dewan and M. Solomon. An approach to support automatic generation of user interfaces. *ACM TOPLAS*, 12(4):566–609, 1990.

[14] K. J. Ransom and C. D. Marlin. Supporting software reuse within an integrated software development environment. In *1995 ACM SIGSOFT Symposium on Software Reusability*, pages 233–237. ACM Press, New York, New York, 1995.

[3] Further details can be found in [14].