In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17 Pages 227 - 241

Introducing #Dasher, A continuous gesture IDE, A work in progress paper

Luke Church

PolyMorphiX Networks
luke@church.name

Abstract. A work in progress Integrated Development Environment, #Dasher, based on continuous gestures is introduced. The challenges of such a user interface are considered with concept exploration studies and concept demonstrators. The language modelling requirements are considered and some other applications of the technology are discussed. Finally some questions currently being investigated are mentioned.

1 Introduction

While there have been substantial recent improvements in computer accessibility for disabled people, these improvements have generally not carried through into software development environments. This presents difficulties for some disabled users, including the increasing numbers who suffer from repetitive strain injuries.

This paper introduces #Dasher, a research software development environment operated by continuous gesture and limited button input. This paper also discusses other potential applications for derived technologies.

1.1 Speech Recognition and Software Development

Despite recent improvements there are severe problems in using speech recognition for development. The language models are highly unsuitable and the difficult navigation methodologies results in an interface with high viscosity [1]. This is problematic as it hinders the design explorations and structural modifications that are common in software development. The combination of these factors tends to make the environment inefficient, stressful to use and places a high strain on the developer's voice, potentially risking voice strain.

1.2 Dasher

Dasher [2] is a text entry system that operates by continuous gesture. The original version is steered by two-dimensional gestures conveyed by a mouse, touchscreen or gaze tracker [3]. It has been shown to be useful for users with impaired mobility, with

an expert user performance of up to 170 characters per minute using a mouse. Figure 1 shows the main panel of Dasher in operation. The usage mode is driving towards the desired item.

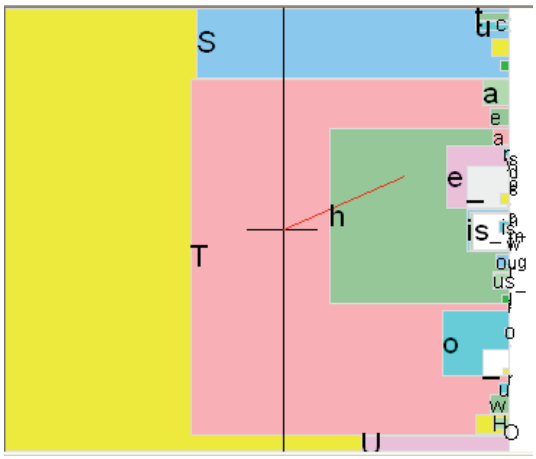

Fig. 1. Dasher's interface in operation. The user is in the process of writing 'The'

However Dasher has some difficulties for software development, principally:

- Prediction by partial match language models have specific problems for programming languages, these are addressed in detail in section 3.1
- Character by character acceptance is tedious for software development
- Dasher, being a general text entry system, lacks much of the functionality expected in a modern development environment
- Navigation within the Dasher interface has usability difficulties

1.3 #Dasher

This paper introduces #Dasher [4] (pronounced Sharp-Dasher). #Dasher is a software development environment relying only on continuous approximate gesture and a minimal usage of additional buttons, it is intended to solve some of the problems discussed previously.

#Dasher uses a Dasher like input mechanism, with a substantially different language model and a new interface for navigation. Work is also being considered on a debugging environment. #Dasher is intended to offer a close coupling between the editor and associated tools such as a compiler.

The #Dasher system can be roughly divided into two components, the user interface, and the language model. This paper discusses both, and briefly considers some other applications of the technologies that have been developed during the research.

2 #Dasher's User Interface

2.1 Design Goals and Concept Exploration

#Dasher's user interface is intended to offer all the functionality necessary for the development of small and medium projects. Where possible it is intended to align with the developers' way of thinking about the software they are developing.

Note that the investigations described below are pilot studies based on convenient local data. Please do not consider them to be definitive.

To discover the typical distribution of file lengths and method/constructor counts that would be handled in a typical C# project, a snapshot was taken of all of the C# files on a developer's hard drive. These comprised of some 22,000 C# files. The result is shown in Figure 2.

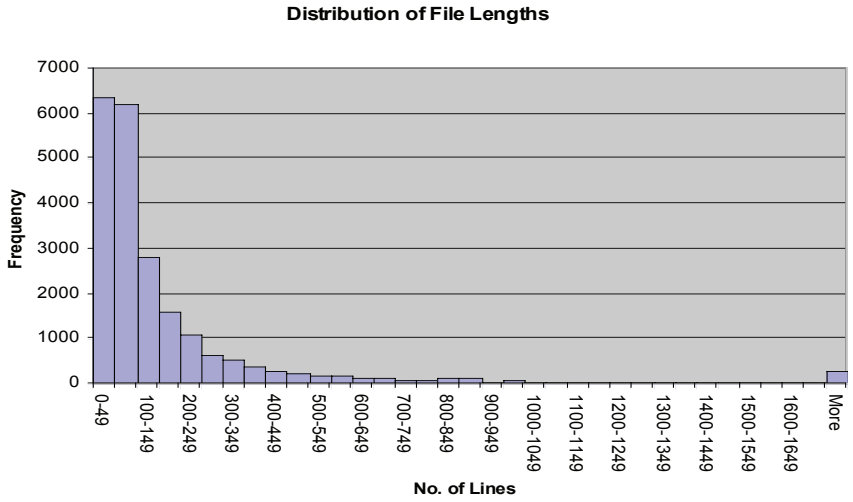

Fig. 2. Distribution of line lengths of C# programs. The high tail value is largely due to machine generated files, such as the cryptographic test functions in Mono

A regular expression was used to approximately count the number of methods in each file. Their distribution is shaped similarly to Figure 2. The results imply that the system should be optimised to operate on files with at least 1000 lines of code and 25 method blocks.

To gain a greater insight into a typical development process a screen-recording of a developer was taken over a period of several hours. The key observation from the recording is the number of navigations that take place during software development. In ~140 minutes, 238 navigations were recorded, not including return navigations. 66% of navigations were highly localised, of only a few lines. A catalogue of specific navigation behaviours is under preparation.

#Dasher has been designed such that navigation and entry are done using different interfaces to maximise the screen space available to each. This somewhat unconventional approach is mandated by the unusual method of information entry, which has a high screen real estate cost.

Due to the importance of navigation it is important that the user be able to switch rapidly between the creation and navigation modes. It is proposed to do this with buttons. Typically these would be the mouse buttons but potentially other buttons could be mapped to such input, for example foot buttons.

Work on the inclusion of debugging is still in concept exploration phase, for more information please see sections 4.2 and 5.1.3

2.2 Code Entry Mode

The code entry mode in #Dasher determines what symbols are associated with the rectangles. This is important as it will determine how closely the interface can align with the developer's 'stream of thought' and will bound the performance of the interface. The interface is proposed to use either tokens, characters or commands as symbols.

2.2.1 Tokens and dynamic alphabets

It is envisaged that the primary usage of #Dasher's input interface will be token based entry. In this mode each of the rectangles will be associated with a currently legal token. The constriction to legal tokens only is used to constrain the infinite set of possible tokens without introducing special usage modes for identifier declarations.

The sizes of the rectangles will be proportional to the probability of the token associated with that rectangle. This results in a dynamic element of the alphabet as the currently legal options change.

Tokens are generally preferred as they seem closer to a developer's stream of thought than characters. However in some circumstances it is unlikely that the token based entry will be viable.

2.2.2 Characters, internationalisation and constraining alphabets

In circumstances were token based entry is not possible it is anticipated that the system would fall back to character by character based entry until token based entry can be resumed. For example, during the declaration of an identifier its name must be entered in character mode; however once it has been completed token based entry can be resumed with an expanded alphabet including the newly declared token. Other cases where character based entry is required in C# include:

- Identifier declaration
- Comment entry/XML DocTag entry
- Arbitrary string entry (such as dialog box texts, URIs, file names etc.)

Under such circumstances #Dasher's input interface will be similar to that of 'classic Dasher' with an alphabet including the special symbols used in software development.

To manage the difficulties caused by the UTF-16 character base of C# [5] it is proposed that #Dasher hold a static alphabet of characters in use, as well as its dynamic alphabet of tokens. This alphabet could be largely inferred from existing source code and/or edited manually.

2.2.3 Command symbols

Command symbols allow for 'in interface' interactions with the system. They are proposed to be used to perform tasks, such as creating, saving, opening and compiling projects and their elements. They could also be used to provide the equivalents of the configuration dialogs required for management of projects, such as setting compile options for resources in .NET.

2.2.4 Additional decorations

The formatting of the rectangles is important in #Dasher. It can be used to provide visual clues with extra information to assist the developer. To provide an extensible system, it is envisaged that the rectangles have additional, size dependant, decorations applied to them in the graphics pipeline.

Currently proposed decorations are:

- Icons to assist with identification of classes, fields, methods etc. and their properties, scoping constraints etc.
- XML derived information for assistance in selecting overloads and passing the appropriate information to method calls etc.
- Rectangle and token colouring to assist in selection in a similar manner to conventional source code syntax highlighting
- Thumbnail sketching of resources where appropriate (e.g. icons)

2.3 Navigation Mode

The navigation system in #Dasher is intended to facilitate both moving the insertion point to another location and the exploration of the source code which are important in reducing the viscosity of the interface.

2.3.1 Concepts

The navigation system is proposed to use a multi-level view, each operating at a different level of detail. The highest level, referred to as the overview, operates on code units, typically source code files. The intermediate level, referred to as the context is used for selecting a region within a code unit. The final level, referred to as the detail level, is used for character level management of the insertion point.

The general principle of the navigation system is to use the probability of navigation to a given point to allow a decrease in the required gesture precision.

2.3.2 Concept Demonstrator

A concept demonstrator for one possible system has been developed. Note that this is not representative of the final navigation system. For example, it is shown without a

proper probabilistic navigation model and assigns navigation hotspots randomly. Figure 3 shows a screen shot of the concept demonstrator.

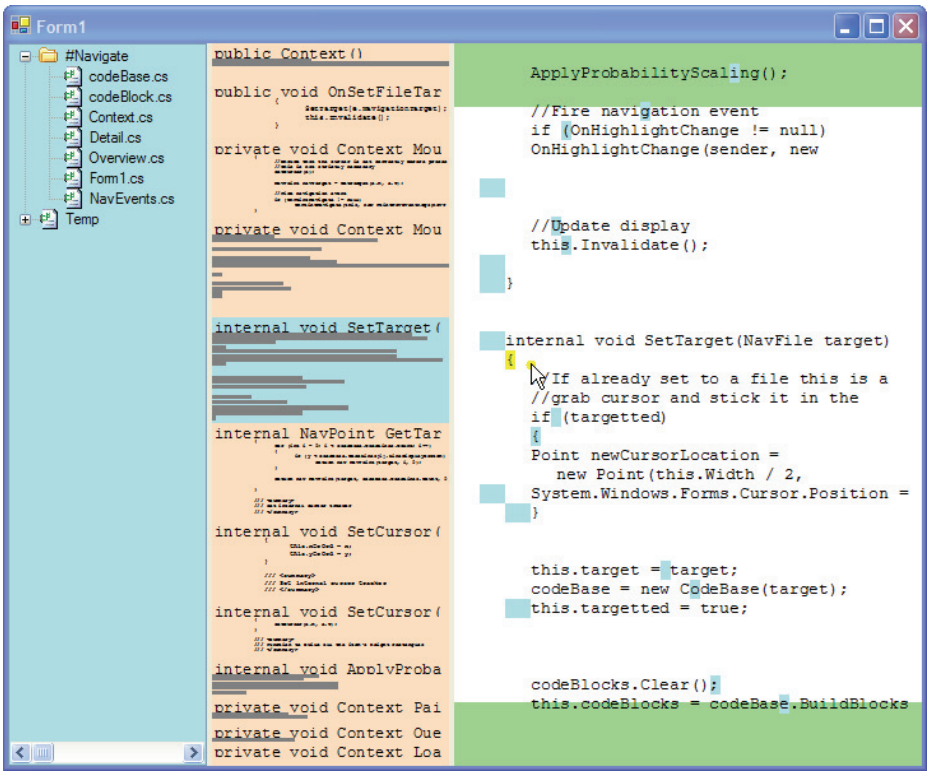

Fig. 3. Navigation system concept demonstrator. The leftmost panel is used to select a code unit, the central panel is used to select a code region and the rightmost panel is used to select a specific point in the source code. Normally the panels would be larger than shown

The user selects a file by clicking on it, the user then selects a region by moving the cursor vertically in the central panel, finally they select a specific point by moving the cursor until the desired location is highlighted in yellow. Clicking selects that insertion point. Dissatisfaction with the currently available points is indicated by hovering, this will result in an increased number of available points close to the cursor.

2.3.3 Proposed enhancements to the concept demonstrator

There are a number of pending enhancements to the concept demonstrator which are intended to make the navigation system more efficient to use.

2.3.3.1 Probabilistic enhancement

The main change will be the use of a navigation language model. Currently randomly generated probabilities are used for the detail view, and no probability model for the

context and overview modes. In the detail view mode the probabilities will be used to allocate the navigation targets more appropriately. This would allow a more realistic height field modification function to be developed to allow easier localised navigation.

In the context view the probabilities will be used in conjunction with the Gaussian functions to scale the blocks of source code between the folds and the individual lines.

In the overview mode the most effective way of applying the probabilities is a question still under consideration. A simple approach would be to use the probability to provide a scaling on the size of the name of the unit and its associated symbol.

2.3.3.2 Navigation History

It is frequently the case that a complex navigation is made, often involving navigation to several different points, followed by code editing, followed by a return to the original point. The navigation system should exploit this and other navigation history related behaviours to make it easier to return to previous points.

The currently proposed method to achieve this is through the allocation of history markers, in the form of arrows, whose opacity is proportional to the likelihood of returning to that location next. For example in a cascade editing operation caused by a variable rename, the probability of the user returning to the immediately previously renamed site is generally relatively low. Hence that site would be marked by a mostly transparent arrow. However the probability of the user returning to the location from whence they originally came, prior to the initial navigation, is quite high. Hence the original site would be demarked by a mostly opaque arrow.

How to make most effective use of the navigation history is still a question under open consideration. Please see section 5.1.2

2.3.3.3 Context dependent navigation

A further extension under consideration is the integration of a context dependent navigation function. The context of the source code presents many hints about where the user might wish to navigate to. This is frequently used by modern IDEs with functionality such as ‘go to definition’.

It is envisaged that such functionality could allow easier navigation between points of interest by overlaying an additional options display on the detail navigation mode when it is in use. The manner in which these options should be positioned, such that they don’t obscure the source code, requires further consideration. It may be possible to use code wrapping to avoid this issue.

2.3.4 Other navigation selection options

The system considered above is one of a number of possible navigation systems under consideration, this section briefly discusses some alternatives.

2.3.4.1 Probability ‘blob’ fields

It may be preferable to replace the detail view with a field based selection system. This may improve navigation performance as all points would be available at all times, it may also increase the visibility of the selection criteria used by the system.

A field based system would view the code base as a scalar field where each navigation location is a ‘point charge’. Associated with each point charge are a strength and a continuous monotonically nonincreasing strength function. At each location a field strength associated with each of the point charges can be computed. Clicking at a point results in the navigation point with the highest field strength at that point being selected. Figure 4 is a screenshot from a concept demonstrator.

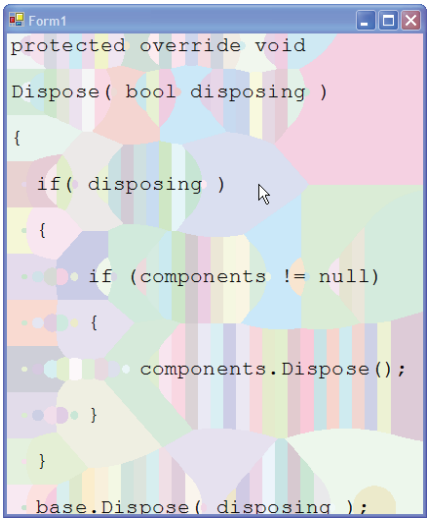

Fig. 4. Probability field concept demonstrator

In the screenshot the colour fields are used to display which pixels are associated with which navigation point. Whilst making the fields clear this results in usability issues and, even using pastelised colours, it is not suitable as a production system.

One possibility is to replace the colour fields with a highlighting system that emboldens the item that would be selected if the mouse is clicked, and possibly highlighting the space immediately surrounding it.

However it might be unclear to the user why a particular item was pre-selected if they were spatially closer to another item. It may be preferable to use a simple inverse distance calculation, even though this does not take probabilities into account.

Another possibility would be to use a modified form of the area shading system to make it clear why an item is pre-selected. However a way of doing so without distracting the user would need to be developed.

Hovering over an area indicates dissatisfaction with the currently available choices and would mutate the probabilistic height field, increasing the probabilities of items close to the cursor which would result in larger pre-selection areas. This makes it easier to select low probability items.

2.3.4.2 Diagram based navigation

It may be helpful to offer a diagram based system for navigation at the overview level. This may assist the developer in understanding the structure of their project and navigating more rapidly through complex projects.

In a diagram based navigation system elements of a diagram are associated with points in the source code. Selecting that element navigates to the associated location in the source code. Many similar systems maintain a virtual link between the source code and the diagram such that changes in one are automatically reflected in the other. UML is frequently used as a notation language for such systems.

Some systems use an exhaustive display, where each element of the source code is reflected by an item in the diagram. Whilst, in such systems, it is always possible to navigate to any location in the source code, the diagrams rapidly become complicated, such that their purpose as communications mechanisms is impaired. Some systems use the diagram to automatically generate a skeleton of code for the developer. For example BlueJ [6] takes this approach.

Other systems, for example Visual Studio 2005 [7], take a code centred view, where the user specifically requests the inclusion of a code element in their diagram. This may result in diagrams that are easier to use for communication, however they are no longer exhaustive.

It is envisaged that #Dasher would take the latter approach, meaning that diagram based navigation would be available as an option. It is envisaged that #Dasher would use a general approach for the code bindings with strategies to draw the diagrams. This would allow different diagram styles to be integrated.

To facilitate a lower precision of clicking it is proposed to use a probabilistic function associated with each item and, in a similar manner to the probability blob fields system discussed previously, to use a pre-selection based on the component of the diagram with the highest strength at the cursor.

2.3.4.4 Navigation model and structural assistance

All of the above functionality depends on a statistical model for navigation probabilities. This model is one of the current areas of research.

However statistical models of where the user is most likely to navigate to are not sufficient. They must be coupled with models of a language that assert which elements of the code are required for the user's perception of the structure of the code, even if they do not wish to navigate there. These items should always be displayed.

#Dasher's navigation concept demonstrator uses these to establish the folding points for the blocks. Currently it uses a regular expression to locate the first lines of method declarations. This will be replaced by proper syntax search.

This requirement for human centred definitions creates an extra requirement for building new language models, limiting the degree to which it can be automated.

3 Language Modelling

The language model of a Dasher style system is very important to the overall system usability. The decoration with appropriate probabilities is important for user

performance. The language model is also significant in the constraints it places on the system, for example, if the language model required the code to be left in a legal state during editing, the viscosity of the associated interface would increase considerably.

This section looks at Dasher's current language model, PPM, and considers the design of a replacement for #Dasher that mitigates some of PPM's issues when used with source code.

3.1 Prediction by Partial Match

Prediction by Partial Match (PPM) uses previous characters to predict the next character based on past usage frequencies. The maximum number of characters used to predict the next is referred to as the 'maximum order' of the compression model. Dasher's language model uses PPMD5, a PPM model with a maximum order of five. PPMD5 has been shown to be a powerful compressor for natural language [8]

The use of a PPM system results in a number of significant issues when applied to source code, two of which are outlined in the following sections.

3.1.1 A non-infinite context causes illegal suggestions

Using a context of five characters, in connection with the absence of a type system means that illegal suggestions will be made. Consider for example two classes, AlphaFactory which has a method A and BetaFactory which has a method B. If in a method a call 'AlphaFactory' is entered, as the context only has a depth of five it will not distinguish between the types and will suggest both A and B as possible methods.

In the general case this could not be solved by finite depth PPMs. Practically orders much higher than five would be necessary as identifiers in C# are frequently in excess of 20 characters, such a PPM would add a serious performance overhead.

3.1.2 No understanding of scope or type

A lack of understanding of scope means that the language model will not discriminate between different identifiers if they have the same name, despite potentially very different usage patterns, and different types, and hence different legal options. The system will suggest identifiers that aren't currently in scope as possibilities in a potentially very disruptive manner.

3.2 Language aware models

The problems associated with PPM models can be overcome by using language specific models. The language model proposed for #Dasher is a model of the C# language. The exact requirements placed on the language model are complicated by the requirements for error tolerance, performance and a stable user interface, however generally:

- The language model must, at any given point in the source code list all options that could legally follow
- The language model must decorate these suggestions and characters with probabilities

3.2.1 Possibility enumeration

The possibility enumeration system must return all legal possibilities at a given point in code, so that they can be displayed on the user interface. This is proposed to be achieved by a modified form of an LL(k) recursive descent parser, the detail of which is beyond the scope of this paper. However generally, matching is applied recursively until the insertion point is found. The recursive stack is then unwound and all forward predictions are considered. The possible legal items are the union of the starter sets of the forward predictions. Using an LL(k) parasable language results in no ambiguity given a lookahead of up to k tokens.

The requirement for k token lookahead is proposed to be handled by exhaustively expanding all possibilities up to the depth required to reach non-ambiguity. This will guarantee that all possibilities that result in a legal completion are suggested. It will result in the expansion of many parallel parse trees. Some of these trees will be invalidated when a token is accepted. Hence for each acceptance a scan should be made pruning all the invalidated trees before evaluating possible new expansions.

3.2.2 Additional functionality

3.2.2.1 Realtime type analysis

The system must also support realtime type analysis with no lexical pre-declaration of identifiers, as required by the C# specification [5].

3.2.2.2 Error tolerance/non-stream entry

The development process is not a linear one. Source code is entered in small chunks interspersed with frequent navigation. Part of the source code will frequently be left in an illegal state during the navigation and entry. The system must support this usage.

The details of this behaviour are currently under investigation.

3.2.2.3 Probability decoration

A detailed discussion of the probabilistic decoration system is beyond the scope of this paper. It is currently an active area of investigation. The difficulties are largely the development of inference on abstract syntax trees and their relationship to the code. Please see section 5.2 for further information on current research interests.

3.2.3 Performance and Concurrency

The performance of a Dasher style system is important for usability. The interface needs to operate smoothly to minimise distraction and visual strain. Ideally it should be capable of operating faster than the user, to prevent it being a limiting factor in the system's overall performance.

Both the possibility analysis and probability decoration systems must run in realtime, sharing system resources with the user interface, which is likely to be CPU intensive. Hence the performance of these elements is important. It is proposed to employ various performance acceleration techniques.

- Incremental analysis
- Snapshot support
- Stream insertion and multiple versioning of analysers
- Probabilistic lookahead search
- Parallelisation

3.2.4 Interface stability

It is important in a Dasher style interface to prevent the interface changing unexpectedly. Once a rectangle has been displayed on the screen it must have its size and relative location frozen. Hence the interface must be able to communicate with the language model to instruct it to cease probability refinement on a given item.

Furthermore interface stability is important for the operation of some of the performance optimisations in the language model. It also affects the balance of how far the probabilistic lookahead can extend vs. how many items are displayed on the user interface. Hence it requires careful tuning.

4 Applications

4.1 Development environments

While the obvious application of #Dasher is in assisting disabled people with software development, it potentially has other applications to more traditional development environments.

Many modern development environments offer a form of localised code completion which offer context dependent suggestions. This has typically taken the form of a type derived option enumeration, offering methods and properties of the associated object. Some environments are now extending this with a keyword suggestion capability; however this is done by inserting the keywords into the completion list, not taking legality in the context into account.

The ability to determine all legal possibilities at a given point offers a more robust code completion system where only legal possibilities are offered at any given point.

Some development systems are beginning to utilise probabilistic information to enhance their code completion systems. Visual Studio 2005 offers a ‘most used’ counter, and preselects the most frequent item from the code completion lists. Such functionality could be enhanced by a more powerful probabilistic language model such as the model that is required for the implementation of #Dasher.

The technology that is being investigated for state tracking could be used to enhance a standard debugger by giving a clearer perception of the program state without requiring tedious navigation through the class hierarchy.

4.2 Probabilistic error recovery

Compiler's error reporting systems do not tend to be very reliable in reporting either the appropriate number of errors or the errors themselves. A detailed example would be too lengthy for this paper, however examples can be found on the resources section of the project's website. [4]

Most compliers employ a form of error recovery to attempt to allow for continuation of the compilation of a project despite an error, so that further errors can be discovered in the same operation. However compilers often over-report or under-report errors. This can cause more edit-compile cycles than necessary. The incorrect reporting of errors can be misleading, especially for novice developers.

The techniques used for error recovery tend to be rather cautious. Some shift reduce parsers delete tokens from their stack until a synchronisation symbol is found. Some systems use exhaustive character replacement search to fix some trivial typographical errors.

A probabilistic error correction system might be help in several ways:

- Provide error reports which more closely match the cause of the problem
- Provide better error absorption and hence report more errors in the same program
- Provide suggestions as to how the errors might be corrected
- More efficient and accurate than an exhaustive character based search
- Provide more detailed error detection and warning in real time as the code is entered without requiring an edit-compile cycle

Such a system might operate on the basis of scanning the code, when a token is recognised that cannot belong to a legal continuation, evaluate what is the most probable legal continuation. Then using this to either replace, insert or delete the token. Care has to be taken when engineering such systems to ensure that they do not suffer from the possibility of an infinite loop, or allow changes to cascade into legal blocks of code. It is proposed to use a system of change termination boundaries to prevent this.

The probabilities of what the action and associated token should be, could be derived from a number of sources, the standard language model, change minimisation schemes and user error modelling.

Such a system would also allow suggestions to be made to the user as to what the appropriate corrective action might be, with an appropriate interface this could be used to speed the debug-edit-compile cycle.

By combining such technology with a probabilistic pattern recognition engine, it might be possible to build a probabilistic real-time code analysis tool.

4.3 Speech recognition

The language modelling system in #Dasher could provide the basis of a language model to allow speech recognition of source code by providing the recognition system with the information it needs to disambiguate utterances.

Due to the high navigation and editing rate in software development it seems unlikely that a purely voiced based system would be practical, however in

combination with a pointing based navigation system it might be possible to dramatically reduce the amount of keyboard usage necessary.

4.4 Navigation in Dasher

If the navigation system proves to be successful it may be possible to integrate a similar system into conventional Dasher. Perhaps in the case of natural language entry using the first line in paragraphs as folding points would be appropriate. If successful this might be useful in decreasing the viscosity of the Dasher interface.

5 Current Research topics

This section lists some of the questions currently under consideration. Research into usability of the system will be carried out once the software has reached an appropriate stage of development.

5.1 User Interface

5.1.1 Creation

At what stage is it appropriate to introduce new decorations onto the token shapes? How is it best to tune the lookahead display system?

5.1.2 Navigation

Is it better to use machine learning or pre-programming of activities such as usage patterns for renaming variables?

Are there smarter ways of using the navigation history? Would it be possible to have a 'scrollback' navigation system in the local navigation mode to facilitate rapid revisiting?

How should the local selection be done? How can code areas be displayed in a non-distracting manner? How should the probability fields be mutated over time? Could local magnification be used?

Could code wrapping be used to manage line length excesses? Should tab shrinking be applied? Should horizontal scrolling be used? Are Gaussians the most appropriate scaling functions?

What are the key elements for navigating around non-OOP code?

How should selection operations be integrated?

5.1.3 Debugging

How should state be displayed? If a directed graph is used, what should locals be rooted to? Initial research indicate people don't think about them like the VES does. How should multi-threading/distributed debugging be handled?

How should usage probabilities affect state visualisers? How should user induced mutations persist?

How should the forward projection of the control flow be displayed? How much can be done without side effects?

5.2 Language Modelling

What is the best way to perform Bayesian inference on trees? Context of declaration seems important, what other indicators are there? What is the best way to use variable roles?

To assist in decreasing viscosity it might be possible to use polymorphic inference to allow otherwise illegal items by inserting 'shadow' using statements, declarations etc. Is this a good idea? Will this just result in mistakes? How do you stop over specification?

5.3 Applications

How much will be gained from legality constriction when applied to evolutionary programming?

If the language model proved to be a better compressor than PPM for source code could it be used to improve source code compression for transfer and archiving? Could it be applied to provide a bounding on the entropy of source code?

Can the system be used with a runtime pattern recogniser to help build a probabilistic link between runtime behaviour and source code?

6 Acknowledgements

I would like to thank Alan Blackwell, Thomas Green and David Mackay for their time and many helpful discussions.

References

1. Green, T. R. G.: The cognitive dimension of viscosity: a sticky problem for HCI. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.) Human-Computer Interaction - INTERACT '90
2. Ward, D., Blackwell, A., MacKay D.: Dasher - a Data Entry Interface Using Continuous Gestures and Language Models. UIST 2000
3. Ward, D., MacKay D.: Fast Hands-free Writing by Gaze Direction. Nature 2002
4. Church, L., the #Dasher Project: http://www.sharpdasher.net
5. Standard ECMA-334: C# Language Specification. ECMA 2002
6. BlueJ - The interactive Java environment - http://www.blueJ.org
7. Visual Studio 2005, Microsoft. http://msdn.microsoft.com
8. Canterbury Corpus benchmark, http://corpus.canterbury.ac.nz/