Bryan Eustice

112157108

15 November 2011

C S 4263

iEx6: Programming Survey

Describe three services that a good programming environment should provide

A good programming environment should be intuitive and user-friendly towards the programmer. A programming environment that is both user-friendly and intuitive to use should include an easy-to-use step-by-step debugger, a descriptive helpful error communication protocol, and a well-organized representation of a project's hierarchy. I will describe each of these features and discuss how they make a programming environment both user-friendly and intuitive for the programmer.

First, an easy-to-use debugging feature that allows the programmer to walk through each step of a program's execution is essential to a good programming environment. The debugger should allow the programmer to easily see the flow of execution through the code. This allows the programmer to track the order in which expressions are executed so that he or she can easily locate where an error is occurring in the code. When the programmer can quickly pinpoint where in that code that an error occurs, he or she can more quickly determine the cause of that error. This decreases the average bug fix time for a project and also causes less stress for the programmer.

Second, the environment should provide easy-to-read and well-organized error-reporting. An error report should be generated for each error. Each report should include a concise, yet easy to understand, summary of the error and a description of the location in the code at which the error was generated. First, the summary of an error should be short yet descriptive enough to convey the nature of that error to the user. It should not be overly long and convoluted with compiler-generated error codes that are useless to the average user. The error summary should be a well-organized description that is easy to read while only providing the information that the programmer needs to deduce the cause of that error in a timely manner. In addition, the environment should provide a sufficiently in-depth description of the location in the code from which the error originated. At minimum, an error location description should include the name of the source file from which the error originated and a number labeling the line in that file on which the error occurred. By including an error reporting system with these features, a well-made programming environment makes it easy for the programmer to recognize the type of error he or she is dealing with and where that error is occurring in the source code. As with an easy-to-use debugging feature, a user-friendly error reporting system decreases average bug fix time and reduces the user's stress level.

Lastly, a good programming environment should provide the user with a well-organized and intuitive visual representation of the hierarchy of a project. The hierarchy should include the functions in each source file and the relationships between all files included in the project. For example, NetBeans provides a nice graphical representation of a project's hierarchy that resembles the folder hierarchy in Windows Explorer. In addition, NetBeans also allows the user to hide function definitions when viewing a source file. When a function's definition is hidden in NetBeans, only its declaration is present at that function's location in the source code. A well-organized, intuitive representation of a project's hierarchy, such as the one in NetBeans, allows the user to easily visualize all parts of a project and how they interact with each other without having to scroll through the code over and over again. As a result, fewer errors are generated by the user and the few errors that are generated take less time to fix.

Every good programming environment should include an easy to use step-by-step debugger, an easy to read and well-organized system of error reporting, and a well-organized and intuitive visual representation of a project's hierarchy. All three features combine to decrease production time by reducing both the number of errors generated and the average error fix time. The grand result is less time spent in the coding phase of a project and a less stressful experience for the programmer.

Describe three things you like about the Dracula programming environment

The three features that I like the most about the DrAcula software development environment are the built in proof-checker for theorems, the check-expect function for testing, and the automatic highlighting of source code based on the number of parentheses present. Each feature can be used by the user to reduce the number of errors and the average error fix time for projects, which results in less time spent on each project. All three features are must-haves when programming practical applications in the ACL2 language.

First, the built in theorem proving feature of DrAcula makes it easy for the user to verify that the desired properties of the project will always hold with a given source code. The theorem-prover is simple to use and gives easy-to-understand graphical feedback. Theorems that are proven are highlighted in green while those that cannot be proven are highlighted in pink. In addition, the theorem-prover communicates to the user each step in the theorem proving process as it attempts to prove each theorem. The user can gain an understanding of the process DrAcula goes through to prove each theorem because he or she can see which functions and preceding theorems the environment uses to prove each function. If the user believes that a theorem that DrAcula failed to prove is provable, he or she can track through the proof and use the information in each step to come up with an alternative way of proving it. Likewise, DrAcula allows the user to include commands with each theorem that tell DrAcula how to go about proving it. For example, the user can tell DrAcula to induct on a previously proved theorem when proving a new theorem. The theorem-prover allows the user to easily verify that his or her program fulfills the properties specified in the project description, which will make it less likely that errors are found during testing or after the product is released.

Second, the check-expect function makes the testing of function definitions for sanity checks and corner cases easier and more stream-lined. The user only has to include a function call with a specified set of parameters and an expected return value in the call to the check-expect function. When the check-expects are executed, DrAcula responds by proving that the expected values are returned using the given function call on the specified parameters and notifies the user in plain English (or whatever native language the user speaks) whether or not the expected value would be returned. This drastically cuts down on the lines of code required for testing. In many cases, a check-expect only occupies one line of code where a user-defined test would take multiple lines of code.

Finally, the highlighting of source code based around the numbers of opening and closing parentheses present makes it easy for the user to see the hierarchy of statements in a function, theorem, or property definition. It makes it easy for the user to verify that he or she has the correct number of opening and closing parentheses in the project code. It also allows the user to easily check that they have provided the correct number of statements for each type of expression. For example, in the case of a nested if-statement, the user can make sure that each if-statement has its corresponding true and false branches. In the case of a let-statement, the highlighting feature is an indispensable tool for checking that the let-statement covers all desired expressions in the code. Tracking which if-statements are covered by each let-statement in a large nested if-statement can be confusing even with the parentheses-based highlighting. Without that highlighting, such a task would be a nightmare! The parentheses-based highlighting helps the user to more easily track their code, which results in fewer errors generated and a faster average fix time for the project.

The built-in theorem prover, the check-expect testing function, and the parentheses based code highlighting feature are three great features of the DrAcula programming environment that most users could not program without. They help to both decrease the number of errors generated and reduce the fix time for errors. For commercial applications, those features allow a project development team to produce higher quality software more efficiently. The ultimate result is a better product ready for market in a shorter amount of time.

Describe three things you dislike about the Dracula programming environment

The three features, or more precisely lack thereof, in the DrAcula programming environment that I dislike the most are the lack of a graphical representation of a project's hierarchy, the absence of tabbed browsing of the source files in a project, and the confusing, unintuitive error reporting system that it uses to generate reports for each error found. Corrections to these problems would make the average user's experience with the DrAcula environment a more positive one as it would make it easier for the user to visualize the relationships between the parts of each project and correct errors found in a timely manner.

First, the lack of a graphical representation of the hierarchy of functions and source files in DrAcula makes it difficult for the user to visualize how the parts of a program fit together. The user is forced to draw out the project hierarchy in a design document and refer to that document repeatedly during the implementation of the project design to keep track of it. While this is good software engineering practice anyway, it would still be convenient for the user if he or she could view the basic representation of the project hierarchy in the programming window so that those notes do not need to be dug out and consulted as often. I know I spent a great deal of my project time scrolling through the code and reading over my design notes just to remind myself what functions depended on other functions and what functions were called from what libraries. A graphical hierarchy in the programming window would have eliminated much of that time spent checking my code thereby decreasing my time spent working on projects.

The second feature that I feel could be added to improve the DrAcula programming environment is the tabbed browsing of the source files included in a project. Most mainstream programming environments such as NetBeans, Visual Studio, and Eclipse allow for the tabbed browsing of all source files in a project. It is disappointing that DrAcula forces the user to open each source file in a different window. It makes the simultaneous editing of source files more tedious and inconvenient. In addition, it makes it more difficult for the user to visualize how each source file fits into the project. Tabbed browsing would be a natural addition to a graphical hierarchy and similarly would reduce the amount of time the user spent navigating the code, which would decrease the time spent coding for projects.

The last, and for me the most irritating, feature of the DrAcula programming environment that I dislike is its confusing error reporting system. While working on the projects this semester, most of the error reports that DrAcula gave me were long, convoluted, and littered with undecipherable compiler labels to the point of being too complicated and disorganized for me to comprehend. In most cases, the reported location from which the error was generated was in some prebuilt library that I had never heard of. So I often could not determine the nature, cause, or even location of an error from the error report that DrAcula returned. Most error reports were useless to me. I would almost always have to scroll through the code and abstractly track the execution of the program over and over again to find the problem segment of my code that caused the error and understand why it was causing that error. On several occasions, I spent hours just trying to determine the nature of an error and what segment of the code I needed to change in order to fix it. Better organized and easier to understand error reports would have drastically reduced my error fix time and my stress levels while working on those projects. A simple, intuitive error reporting system seems like a must for a good programming environment to help the user locate and fix errors more quickly. Unfortunately, such a feature must be difficult to implement as I have never worked with a development environment that used an error reporting system that I liked. Though DrAcula is not alone in this respect, I would love for it to be the first programming environment to offer a truly user-friendly error reporting feature.

Like every programming environment DrAcula is not perfect. Likewise its developers are always looking for new features to add into future versions to make the user's experience working with DrAcula a more positive and constructive one. The three features that would provide the biggest help to me personally while using DrAcula would be a graphical representation of a project's hierarchy, the tabbed browsing of multiple source files in a single programming window, and a more user-friendly error reporting system. These changes would drastically boost the average user's productivity and decrease the time spent on projects.

Suppose you are a software engineer in an organization that has received a contract to design and implement the control software for a new kind of nuclear reactor to be constructed in a densely populated area. You have been assigned the task of choosing the programming environment to be used in the project. Draft a one-page memo that specifies the programming environment and provides a convincing rationale for that choice.

MEMORANDUM

TO:	20XX Energy Corporation Management Staff
FROM:	Bryan Eustice
CC:	Mr. Sean Landers, Project Manager
DATE:	November 14, 2011
SUBJECT:	Programming Environment for Kansas City Reactor Control Project

Last week, I was assigned the responsibility of choosing a programming environment for the Kansas City Reactor Control Project. The DrAcula software development environment would be best suited for the important, risky nature of the project.

The DrAcula software development environment is a basic programming environment that devotes a programming window to each source file in a project. Each window includes a basic text editor with a useful search and replacement function that makes the modification of multiple instances of keywords and expressions in the source code quick and easy. In addition, a separate editor resembling a command line is included for the purposes of testing the code and error reporting. Like most environments, it includes a step-by-step debugging feature for tracking the execution of the source code and locating errors. All this is included in a simple, easy to use graphical user interface with drop-down menus and buttons. However, the features of the DrAcula programming environment most relevant to the nature of our work are the simple and concise check-expect testing function and the easy to use theorem proving feature.

The control software in a nuclear reactor must be flawless especially if that reactor is located in a densely populated area. As such the extensive testing of our control software for the Kansas City nuclear reactor is a must. The check-expect testing function in DrAcula will simplify our testing methods and make them more concise. For each check-expect function call, all that is needed is a call to the tested function with specified testing parameters and an expected return value. Upon execution of the check-expect expressions, DrAcula tells the user whether or not each function returned its expected return value. As a result, less time will need to be spent on designing tests of the source code, which will decrease the time spent in the testing phase of the project.

However, the most vital feature of DrAcula to the Kansas City Reactor Control Project is the built-in theorem proving feature. Due to the nature of the project, the correctness and dependability of our reactor control software needs to be proven. The theorem proving feature of DrAcula will allow us to easily verify the workability of our control software in the targeted environment. DrAcula allows for the creation of specific theorems to test the universal workability of the functions in our project and the theorem proving feature can easily be used to verify that those functions will always work as intended. The user can clearly see which theorems hold and which do not. Theorems that DrAcula succeeds in proving are highlighted in green while those that DrAcula fails to prove are highlighted in pink. In addition, DrAcula displays each step in the induction process in a separate frame so that the user can understand how it goes about proving each theorem. Finally, DrAcula allows the user to specify how it should go about proving theorems. Commands telling the environment to induct on previously proved theorems can be added to the code as an attempt to get DrAcula to prove a failed theorem using an alternative approach. The theorem proving feature is easy to use and will allow us to better demonstrate the correctness, completeness, and dependability of our software to our employer, which will have a substantial positive impact on the company's decision to approve our product.