Seph Newman

11/14/11

iEx6 - Programming environment survey

Software Engineering I - CS 4263

1. Three services a good programming environment should provide:
    a. **Informative/Helpful Errors:** A good programming environment should provide helpful information when it encounters an error. In the case of a compiler error, the environment should point to the exact line where it ran into the error and inform the user of what it saw and what it expected. Similarly, run time errors should strive to convey as much information as possible. While I understand that it's not always possible to provide in-depth information (such as line number of the error) due to the nature of the compiler, any information is always helpful. Furthermore, pleasant organization of this information and use of clear concise English goes a long way in helping the programmer find the error in his or her code. It is also helpful for the programming environment to show would-be compiler errors in real-time. That way the programmer can fix trivial issues (a forgotten semicolon or parenthesis) as he goes without having to stop and compile.

    b. **Powerful Debugging Tools:** A good programming environment should provide programmers with the necessary tools for debugging their code. At minimum, this should include the ability to run a program and show/highlight the line where the program encounters an error along with the current values of all local variables. Furthermore, any decent programming environment should allow programmers to set up break points for individual lines. When a program hits a breakpoint it should pause allowing the user to view the current state of the program and provide the necessary tools for stepping though the code. A truly good programming environment should allow for breaking at more dynamic conditions. Examples of this include setting conditional breakpoints (ex: break when a variable equals a certain value) or breaking when certain "watched" variables are either changed or read by a line of code. A good programming environment should also strive to make the controls for placing breaks and stepping through the code as intuitive as possible.

    c. **Effective Code Navigation/Organization Tools:** Large programming projects inevitably become a complex sea of dependencies. As they build up, more and more individual files become involved and/or individual files become increasingly long. To keep things manageable it's important to have sufficient navigation and organization tools. The most primitive navigational aid is a find tool and represents the bare minimum which a good programming environment should provide. An extremely helpful tool that is becoming increasingly common (Eclipse, Visual Studio, etc) is the ability to go straight to the definition of a function or find all lines of code which reference the given function. This is especially useful when cleaning up code since before removing a function, you can easily make sure nothing else is using it. Organizational tools to keep code files organized make large projects more manageable. Such tools include a good system for managing files/folders and dependencies. The ability to collapse and expand individual functions is also helpful. Such a feature allows a programmer to organize his or her code and better see the overall structure (as opposed to getting lost in the noise).

2. Three thinks I like about the Dracula programming environment:
    a. **Visual Cues:** The Dracula/DrRacket programming environment does a great job providing helpful visual cues when looking at a code file. The most obvious way this is done is within the color coding of different data types (ex: numbers and characters are green, strings are red). Especially notable in the color design is the decision to give parenthesis their own color (brown) since ACL2 (as well as any lisp variant) code uses them quite liberally. The color coding was helpful for me when trying to find the correct form for special characters. For example, I needed a carriage return character, so I typed #\carriagereturn, but the letters remained blue. I then tried #\return and it turned green indicating a valid character. An even more helpful visual cue is the parenthesis highlighting. When the blinking text cursor is positioned to the right of a closing parenthesis or left of an opening parenthesis, all the code between it and the other corresponding opening/closing parenthesis is highlighted. This is helpful when going back to check that a function has correct syntax. A programmer can check each statement to make sure things line up correctly.

    b. **GUIs:** I feel that Dracula does a good job presenting Graphical User Interfaces (and allowing design of them) in an effective manner. The prime example of can be seen when running property tests. The structure that shows each test with each test case nested within makes intuitive sense. Even better is the way you can see the test cases running in realtime. This allows you to clearly see when a test has an error or takes too long to run. This same basic information could have been conveyed in a plain text console, but it is far more effective in this GUI (although I do have some issues with what is actually displayed- see 3c). Similarly, the GUI teach pack ("world") is easy to use yet sufficiently powerful. It makes the typical object-oriented system of input listeners applicable to a functional paradigm.

    c. **Formatting Assistance:** One particularly nice aspect of DrRacket/Dracula is how it helps keep code in an organized structure as you type. When you finish a line and hit enter, it automatically indents the next line of code to the proper spacing. Of course, there is more to it than just indenting every line a bit more. Instead the environment automatically indents based on the function. For example, once you finish typing the first case in an if statement and hit enter, the else case is set up to be automatically aligned with the first case. If things somehow ever get off hitting tab seems to fix the indentation to what should be correct. This process can be applied to entire functions/files with the command "Reindent" and "Reindent All" which are available in the Dracula menu. This automatic intelligent formatting helps keep code looking manageable and organized as you type things in.

3. Three things I don't like about the Dracula programming environment:
    a. **Debugging Issues:** Probably my main issue with Dracula is the lack of proper debugging support. DrRacket natively supports debugging and when using most languages has no issues letting you set up breakpoints at the function level. I found this feature to be incredibly useful in Principles of Programming Languages class. When using ACL2, however, DrRacket doesn't allow placing of breakpoints. What's even more frustrating is that you can put your desired function calls at the bottom of the file, hit debug, and then step through the function as you would expect. Since this works fine surely proper breakpoints could be added without too much difficulty. Due to this lack of breakpoint support I find myself using all sorts of ridiculous workarounds to debug my code. For example, if I have an infinite loop issue in a function numerous layers deep that depends heavily on other functions it would take a long time to step through my code from the beginning to find out where the error is occurring. Instead I will place bad function call (ones that I know will produce an error) at various points in my code. When the debugger hits theses errors I make sure all the current values are what should be expected. Proper debug support (with breakpoints!) would have gone a long way in making my life easier.

    b. **Lack of Real-time Errors:** Today most IDEs provide real-time error checking. If you forget a parenthesis or miss a semicolon in Eclipse or Visual studio, the IDE will politely point you towards the error and suggest a fix. I frequently miss a parenthesis or two when typing out a function (an unfortunate fact that no amount of practice will change. Additionally, even after almost a full semester of ACL2, I still find myself typing functions in an object oriented format; like this: functionA(paramA , paramB). It would be rather helpful if DrRacket/Dracula would kindly pipe up and say things like, "Expected a parenthesis on line X" or "Cannot call function Y before it is defined."

c. **Issues with Random Testing and Theorem Prover:** While Dracula's general interface design for random testing is nice, I feel that in many ways it is too limited. I like being able to see a nested structure that shows each of my property tests and then, nested within, each of the test cases that was run. I take issue, however, with the fact that you cannot click on an individual test case and view the randomly generated values that were used. Why is it that for test cases that fail, you can look at these values, but not for test cases that succeed? Sometimes, I will have a property test that fails for around half of the test cases, but works for the other half. I can look at the failed cases and see if I can notice a pattern, but sometimes one is not obvious. In such cases it would be great to have access to the values of succeeded cases, to help me pinpoint the difference and thus fix the issue. Additionally, the theorem prover can be frustrating to use in that the error messages it spits out are difficult to parse. It is difficult to search through this wall of text and find something to point you in the right direction.

4. Memo describing programming environment (and rationale) for nuclear reactor control software.

TO:	Nuclear Tech INC
FROM:	Seph Newman
DATE:	November 14, 2011
SUBJECT:	Programing Environment for Design of Reactor Control Software

Choosing a programming environment for designing a Nuclear Reactor's control software is not a task to be taken lightly. Decision of a programming environment not only affects the programmers who are creating the software, but also has implications as to the known correctness, security, and maintainability of the system. We can eliminate Java as a possibility right away since the license agreement states: "You acknowledge that Licensed Software is not designed or intended for use in the design, construction, operation or maintenance of any nuclear facility." This is probably for the best as, due to the mission critical nature of this software, we are going to want a language that can be verified for correctness. Optimally, we want something that we can mathematically/logically prove won't fail. Additionally, we want a language that can work on real-time systems. That is, we need to be able to guarantee results with strict time constraints. There are several languages which meet these criteria, but I suggest the use of an Ada based language called SPARK.

In a mission critical situation such as this one, it is not only our own code that is under scrutiny, but also that of the compilers and ancillary software used to verify our code. As such, we will want to use a language and environment that meets with certain formal standards. Ada was originally designed in the late 1970s through the early 1980s under a contract with the United States Department of Defense. Since then Ada has become an international standard and is widely used in military applications as well as commercial projects in which correctness is vital (software bugs can result in severe consequences). This is due to its integrated safety considerations (run-time checks for unallocated memory, overflows, etc) as well as its ability to perform program verification.

SPARK is highly based on ADA, but with some limitations that eliminate potential ambiguities and insecurities. It specifically designed to support the development of mission critical applications such as this. Annotations are specified before functions, and can specify some general constraints on the memory flow. For example, an annotation might say that the function doesn't read any global parameters; it sets a value only using its input parameters. Annotations can get much more in depth to effectively act as formal tests. The "Examiner" can then analyze the code and annotations to make sure all conditions are fully satisfied. Furthermore, SPARK can also (optionally) perform different verifications (such as that the program is exception free). Checks and tests such as these will be imperative in the high-stakes design of nuclear reactor controls.

The actual IDE and tool suite used are less important (as Ada/Spark compilers all meet certain regulated standards), but I suggest the SPARK Pro Language and Toolsuite. SPARK Pro uses the GNAT Programming Studio IDE which provides integration with other third party tools that will be necessary (such as version control). The toolsuite itself contains theorem provers, testing tools, and other pieces of software that will aid our goal of correctness in design. Lastly, it should also be pointed out that, like existing Nuclear Reactors, we will want fail safe systems designed outside the scope of this control software.