A Java-based environment for teaching programming language concepts[*]

Manfred Hauswirth, Mehdi Jazayeri, and Alexander Winzer
Distributed Systems Group
Technical University of Vienna
Argentinierstraße 8/184-1, A-1040 Wien, Austria
`{M.Hauswirth,M.Jazayeri,A.Winzer}@infosys.tuwien.ac.at`
`http://www.infosys.tuwien.ac.at/`

Abstract - *We describe the SDE visual environment for the study of the semantics of programming languages. SDE supports the abstract machine called SIMPLESEM which is used to define the operational semantics of programming languages in a widely-used textbook on programming languages. By basing the environment on the World-wide Web infrastructure, the environment is immediately available to students and instructors around the globe. SDE enhances the student's learning by simulating and visualizing abstract concepts of semantics. Such visualization environments are invaluable in the study of abstract subjects. SDE is currently being used in several university courses in different countries. We describe the environment's architecture and capabilities and review some of our experiences in developing the environment.*

Introduction

Courses on concepts or principles of programming languages are a standard part of university computer science curricula. As opposed to courses on programming in particular languages, these courses emphasize commonalities and differences among many different programming languages. These courses consider the syntax and semantic components of programming languages. The study of syntax deals with static aspects of programming languages for which notations based on Backus-Naur Form (BNF) have proven adequate and have become standard tools for such study. For the study of semantics, however, there are several approaches and there is no consensus as to the best method. Existing approaches to semantic description may be categorized as axiomatic, denotational, and operational.

The operational approach defines the semantics of a programming language with reference to program execution on an abstract machine. In particular, the semantics of a language construct is defined by giving its translation in terms of the instructions of the abstract machine. Thus, to fully understand the semantics of a programming language, the student must learn the instructions of the abstract machine, how to translate programs of the programming language into programs of the abstract machine, and how the abstract machine executes its programs. In conventional courses, these are done by pencil and paper. Having learned the semantics of a language, the student should be able to translate a language construct into the instructions of the abstract machine. Given such a translation, how does the student know that the translation is correct? No matter how simple the abstract machine is, tracing the execution of constructs such as recursive procedure calls is tedious and error-prone because the student has to manually step through this code keeping track of all data items, the instruction pointer, stack, and heap.

A computing environment that supports program development, execution, and visualization for the abstract machine relieves the student from having to deal with the tedium of machine execution and instead to concentrate on the semantics of the language constructs. A visual execution of the program brings to life the semantics of the language constructs. Vetter [7] argues for "specialized tools" to improve course comprehension. SIMPLESEM Development Environment (SDE) is such a tool specialized for programming language concepts. SIMPLESEM is a semantic abstract machine introduced and used in the book Programming Language Concepts [4]. SDE is a graphical Java-based environment that allows the student to edit, run, and debug SIMPLESEM programs. SDE animates SIMPLESEM executions by visualizing the code and data memories of the SIMPLESEM processor. Users have interactive control over the memory contents and program flow at any time. Both code and data breakpoints are supported. These facilities allow easy debugging and appreciation of a program's control and data flow. This leads to a better understanding of the program's dynamic behavior [6]. The use of new technology to improve teaching and learning is currently an active area of discussion. A special issue of the Communications of the ACM [1] is devoted to new educational uses of computers in the campus environment. The virtual round table [3] discusses many interesting issues and implications of the

new IT educational infrastructure. Krämer [5] describes the general issues of distance learning, the paradigm shift from classroom teaching to interactive distance teaching, and a specific on-line course for distributed software engineering.

SDE is portable across many platforms and can be used either via the World-wide Web in a Java-enabled browser (applet version) or as a stand-alone application. To ensure identical functionality in both versions, despite the fact that Java applets and applications have different security requirements, the applet version emulates functionalities that are only allowed for applications, e.g. the applet version includes a simple file server for storing and loading code. SDE has a single program source for both versions. SDE is accessible over the WWW and requires no installation procedure on client computers. The only requirement for its use is a Java-enabled WWW browser.

The paper is structured as follows. First we describe the semantic abstract processor SIMPLESEM which is followed by an overview of SDE and its functionalities. Then we discuss the impact of SDE on teaching of language semantics. The final section summarizes the main points made, presents some lessons learned, and concludes the paper.

SIMPLESEM Concepts

SIMPLESEM is an abstract semantic processor to support an operational approach to semantics of programming languages. It enables understanding of a concrete programming language's concepts by mapping them onto sequences of SIMPLESEM instructions that can then be "executed."

SIMPLESEM consists of an *instruction pointer*, a *memory*, which is divided in two separate sections—code and data memory—and a *processor*. The *code memory* (C) holds the code to be executed and the *data memory* (D) holds the data which the code manipulates. Both C's and D's addresses start at 0 (zero) and both programs and data are assumed to be stored starting at this address. The instruction pointer (ip) is used to point at the next command in C to be executed. It is initialized to 0 and automatically incremented as an instruction is executed. Each instruction takes one location in C. Execution stops when the special instruction halt is encountered.

The notations C[X] and D[X] are used to denote the values stored in the Xth location of C and D, respectively. Modification of the value stored in a location is performed by the instruction set *target*, *source* where *target* is the address of the location whose content is to be set, and *source* is the expression evaluating the new value. For example, set 10, D[20] stores the value of location 20 into location 10.

Input/output in SIMPLESEM is achieved by using the set instruction and referring to the special registers read (input) and write (output). For example, set 15, read would read a value from the input device and store it in location 15, and set write, D[50] would print location 50's contents on the output device.

Values can be combined into expressions in a liberal and natural way; for example, set 99, D[15]+D[33]*D[41] would be an acceptable instruction to modify the contents of location 99.

To modify the sequential control flow, SIMPLESEM has the jump and jumpt instructions. The former is an unconditional jump to a certain instruction, e.g. jump 47 makes the instruction at C[47] the next instruction to be executed. The latter is a conditional jump which occurs if an expression evaluates to true. For example, for jumpt 47, D[3] > D[8] execution would continue at location 47 if the value in location 3 is greater than the value in location 8 otherwise execution continues with the next instruction.

SIMPLESEM allows *indirect addressing*. For example, set D[10], D[20] assigns the value stored at location 20 to the cell whose address is the value stored at location 10. Thus, if the value 30 is stored at location 10, the instruction modifies the contents of location 30. Indirection is also possible for jumps. For example, jump D[13] jumps to the instruction stored at location 88 of C, if 88 is the value stored at location 13.

As can be seen from SIMPLESEM's description so far, it is a simple machine and it is easy to understand how it works and what the effects of executing its instructions are. In other terms, we can assume that its semantics are intuitively known by students with a basic computer education. The semantics of programming languages can therefore be described by rules that specify how each construct of the language is translated into a sequence of equivalent SIMPLESEM instructions. Since SIMPLESEM instructions are already known, the semantics of newly defined constructs also become known.

All constructs and concepts of modern programming languages, such as loops, routines, recursion, stack and heap memory, block structure, scoping, parameter passing semantics, can be mapped onto the SIMPLESEM instructions described above. These mappings, however, are beyond the scope of this paper. The interested reader is referred to [4] for a detailed description.

Figure 1 gives a simple and intuitive example of mapping programming language concepts onto SIMPLESEM instructions.

```
main()                    0   set 0, read
{                         1   set 1, read
    int i, j;             2   jumpt 8, D[0] = D[1]
    get(i, j);            3   jumpt 6, D[0] <= D[1]
    while (i != j)        4   set 0, D[0] - D[1]
        if ( i > j)       5   jump 7
            i = i - j;    6   set 1, D[1] - D[0]
        else              7   jump 2
            j = j - i;    8   set write, D[0]
    print(i);             9   halt
}
```

Figure 1. A mapping to SIMPLESEM

The program on the left, given in a C-like language, computes the greatest common divisor of two integer numbers using a simple loop. The code on the right shows the same program in terms of SIMPLESEM instructions.

SIMPLESEM Development Environment

Students learn to step through procedural code early in their studies. Without software support, stepping through the code is a tedious and error-prone task. The SIMPLESEM Development Environment (SDE) is a graphical, Java-based environment that supports development and visualization of SIMPLESEM programs. We chose Java [2] for the implementation to make SDE accessible via the Internet as an applet. Java facilitates portability across a wide range of platforms. Thus it can be used immediately and run "off-the-shelf."

An applet is a piece of Java code that is downloaded from a WWW server and is executed on the client computer (see Figure 2). The WWW browser used for downloading provides the applet with a feasible runtime environment and takes care of security concerns, etc.
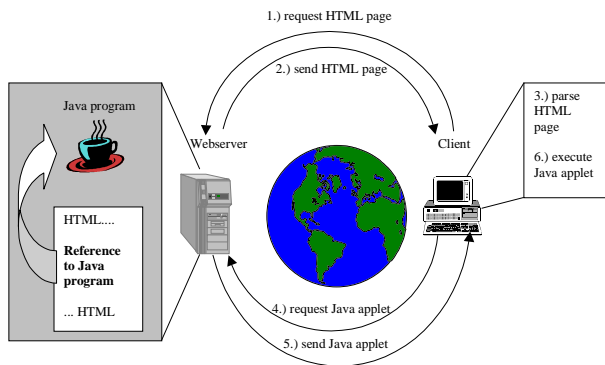

Figure 2. Downloading and executing an applet

Using this setting has several advantages: no explicit installation is needed on the client computer; only a Java-enabled browser is needed which is available on most computers anyway; SDE is world-wide accessible via the Internet; users do not have to maintain the software since they always download the newest version.

Additionally, SDE can also be downloaded and run as a local applet or as a stand-alone application. This is interesting for use on computers without Internet connection or if only a slow connection is available. Using SDE as a local applet requires a Java-enhanced browser. For the application version a Java Virtual Machine (JVM), i.e. the Java runtime environment is necessary.
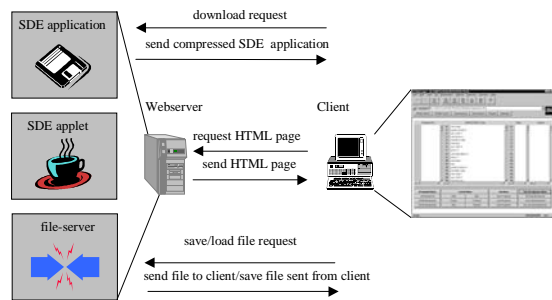
Figure 3 shows the global architecture of SDE.

Figure 3. Global architecture

Applets only have restricted access to the client computer, e.g. file access is not possible. To provide storing and loading of SIMPLESEM files for the applet version, we implemented a file server that runs on the host holding the SDE applet. Users of the applet version can load from and store to the server. The application version uses the file system directly. Despite the slightly different capabilities, which are due to security restrictions, both versions are included in a single source. Features become available or are disallowed dependent on whether SDE executes as an applet or as an application. Configuration, however, is not necessary.

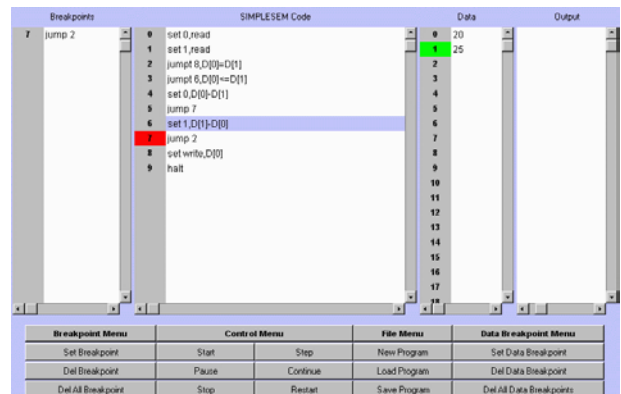SDE consists of a simple line-oriented editor, a SIMPLESEM interpreter, and a graphical debugger.

Figure 4. SDE screenshot

Figure 4 shows a sample SDE session. The editor/code component shows the SIMPLESEM program code in the code memory C (here the program given in Figure 1). The next instruction to be executed is the instruction in line 6. This is indicated by the changed background color of this line. A breakpoint has been set at line 7, which is shown in the breakpoint window on the left. Additionally the line number part of line 7 in the code window indicates the breakpoint by having a different background color. A data breakpoint is set on location 1. The different background color of location 1's line number in the data window indicates that the breakpoint was triggered at the current state of execution. This means that automatic execution was suspended and SDE waits for user

input. The user now can check the execution state, change data values, edit the program, change breakpoints, continue with the execution, etc. These functionalities are accessible via the various windows in combination with the buttons at the bottom of the main window. SDE offers the following functionalities:

Editing. SDE programs can be typed directly into the editor window. The editor is line-oriented and offers basic editor functionalities such as cut and paste, line deletion and insertion, etc.

Breakpoints. Breakpoints can be inserted into the program to suspend automatic execution at defined points.

Execution control. The standard behavior is that programs are executed automatically until user input is required (read register is accessed), the execution hits a breakpoint, or the user hits a control button (pause, stop, restart). In such cases the user can take further actions, e.g. provide input, set/delete breakpoints, continue execution, single-step through the code, etc.

Data breakpoints. Data breakpoints are set on D memory locations. If an instruction attempts to modify a location with a data breakpoint, execution is suspended.

Loading and storing files. SIMPLESEM code can be stored to, and loaded from, files. In case of the application version this is done on the local file system. The applet version is not allowed to access the file system. So loading and storing is done via a remote file server that runs on the applet's server. This server is part of SDE.

A detailed SDE user's guide is included in [8]. To round out the discussion, Figure 5 shows the internal architecture of SDE.
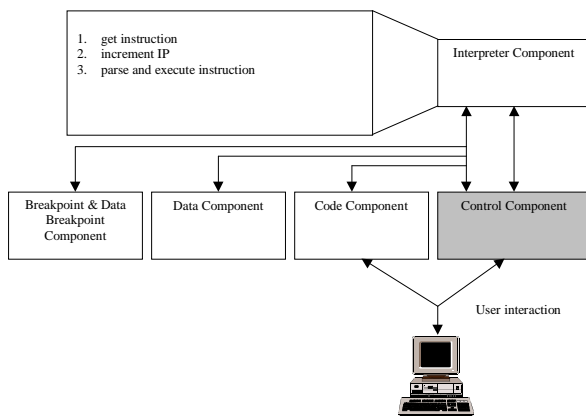

Figure 5. SDE's internal architecture

The control component provides the user interface for the communication between the user and the other components. Additionally, it is in charge of the communication between the components. The code component implements the source code editor. The two breakpoint components store and display breakpoints. They provide a specialized interface to the interpreter to support fast breakpoint checking.

The interpreter component (Figure 6) gets lines of code from the code component and feeds them into its lexical analyzer. The lexical analyzer breaks the input into tokens, which are the input for the parser. The parser combines multiple tokens into a SIMPLESEM instruction and checks its syntactical correctness. The interpreter then executes the instruction. Needed data values are retrieved from, and sent to, the data component.
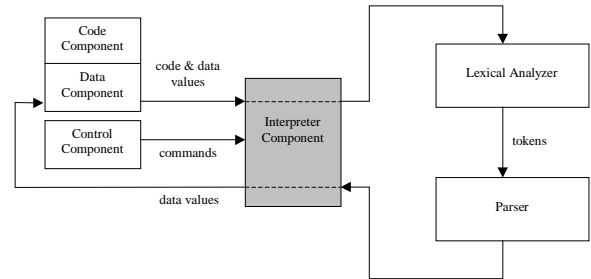

Figure 6. Interpreter interaction

To allow the user to interrupt or pause execution, as described above, the interpreter is implemented as a Java thread. When the user starts a SIMPLESEM program, an interpreter thread is created. It cooperates with the lexical analyzer and parser which are threads themselves. Interrupting and pausing requests are done by suspending the relevant threads. Later, these threads can be destroyed or resumed, e.g. in case of continue or single-step operations. This multi-threaded architecture ensures highly interactive responsiveness. A detailed description of SDE's implementation is given in [8].

Using SDE for teaching

Abstract concepts are always difficult to teach to beginners in a subject. For example, beginning students always have difficulty with the concept of recursion: "how is it possible to call a procedure that you are defining?" they ask. By visualizing the execution of recursive procedures, the student can clearly see how recursion starts, how it proceeds, and how it finally terminates. The student can see the cost of the mechanism and can compare it visually with equivalent implementations based on iteration. The visual impact is immediate. To debug a nonterminating recursive procedure by tracing it on paper is orders of magnitude more difficult than by running it in an interpreter. Indeed, the initial demo program loaded with SDE is a recursive factorial program. It is a good demonstration of how a simple program gives rise to complicated dynamic behavior. Getting one instruction wrong can lead to infinite recursion. By modifying selective instructions in the program the instructor can demonstrate common mistakes and their— sometimes dramatic—consequences.

The basic idea of SDE is to visualize abstract concepts that have no physical counterparts. Most areas of computer

science deal with similarly abstract concepts and can benefit from similar supporting software tools.

Conclusion

We have described the SIMPLESEM Development Environment for the study of the semantics of programming languages. SDE is being used in our university for our Programming Language Concepts course. This is the first time that the course is being taught with such support.

Interestingly, even though SDE has just been developed and has not even been announced officially, it has already been "discovered" by several instructors around the world (on at least three continents!) and is being used in supporting their teaching. Traditionally, instructors of similar courses share their experiences and are also supported by textbook authors. The material shared is usually static in nature: model syllabi, copies of transparencies, etc. A software tool such as SDE enables new ways of sharing. In particular, specially interesting SIMPLESEM programs (such as our factorial demo program) that demonstrate specific issues such as recursion or procedure parameters may be provided on the WWW.

SDE also enables new kinds of homework assignments that were not possible before because of the tedium of handling anything but very small SIMPLESEM programs. Furthermore, homework assignments can take the form of applets that are run by students and submitted automatically to the instructor's site. The assignments can be evaluated automatically and the student notified immediately. Such mechanisms enable one instructor to support a larger number of students than is usually possible.

SDE was implemented in Java and mainly intended for use with browsers over the WWW. Java and the WWW/browser infrastructure were chosen to provide ubiquitous access and immediate availability of new versions. Additionally, this setting was intended to free users from installation and maintenance work. Since Java supports portability, we assumed we could meet these goals easily. This turned out to be overly optimistic. There were two main reasons that lengthened the duration of the project considerably: a new Java release and the Java environments of the browsers. During the implementation Java changed from version 1.0.2 to 1.1.x. This introduced some incompatibilities, e.g. deprecated interfaces, which caused a rework of SDE. The browsers and the platforms they run on have slightly different environments, especially in terms of GUI programming. These differences were big enough to turn Java's "write once, run everywhere" approach into a "write once, test everywhere" one and caused considerable delay due to compatibility and stability tests. These problems will be with us until Java stabilizes. Nevertheless we know of no other approaches that offer similar benefits. SDE is available over the World-wide Web (http://www.infosys.tuwien.ac.at/pl-book/simplesem/).

References

[1] ACM, "Special issue on campus-wide computing," *Communications of the ACM*, Vol. 41, No. 1, January 1998.

[2] Arnold, K., Gosling J., "The Java programming language," Addison-Wesley, Reading, Mass. and London, 1996.

[3] El-Rewini, H., Mulder, M.C., Freeman, P.A., Stokes, G.E., Jelly, I., Cassel, L., Lidtke, D.K., Russo, S., Krämer, B.J., Haines, J.E., Turner, J., "Keeping pace with an information society," *IEEE Computer*, Vol. 30, No. 11, November 1997, pp. 46–57.

[4] Ghezzi, C., Jazayeri, M., "Programming Language Concepts," 3rd edition, John Wiley, New York, 1997.

[5] Krämer, B.J., Wegner, L., "From custom text books to interactive distance teaching," *ED-MEDIA/ED-TELECOM 98*, Prentice-Hall, Englewood Cliffs, NJ, 1998.

[6] Lee, E.A., Messerschmitt, D.G., "Engineering an education for the future," *IEEE Computer*, Vol. 31, No. 1, January 1998, pp. 77–85.

[7] Vetter, R.J., "Web-based education experiences," *IEEE Computer*, Vol. 30, No. 11, November 1997, pp. 139–141.

[8] Winzer, A., "A SIMPLESEM interpreter in Java," technical report, master thesis done at Distributed Systems Group, Technical University of Vienna, Austria, 1998.