

A SOFTWARE DEVELOPMENT ENVIRONMENT FOR SIMULATION PROGRAMMING

Richard Reese
Assistant Professor
Department of Computer Science
Stephen F. Austin State University
Nacogdoches, TX 75962

Sallie Sheppard
Associate Professor
Department of Computer Science
Texas A&M University
College Station, TX 77843

A prototype simulation software development environment (SSDE) has been developed at Texas A&M University as a means of providing a flexible system for the development of simulation software. It utilizes an underlying structure composed of a series of data bases and a language. This framework provides the means of hiding the file system, hiding the development support tools, and automating the transition between the different phases of development. The prototype SSDE includes several tools developed specifically to support simulation programming. This paper presents the rationale for software development environments and gives an overview of the prototype which has been constructed.

1. INTRODUCTION

Recently the field of computer science has seen the introduction of software development environments. A Software Development Environment (SDE) has been defined as "... a set of techniques to assist the developer(s) of a software system, supported by some (possibly automated) tools, along with an organizational structure to manage the process of software production" (Wasserman 1981). SDEs with their supporting tools and organizational structure have been designed to help cope with the complexities of software development and the associated costs.

The support software of today evolved from a myriad of simple, isolated programs to the more interrelated and complex tools present in many environments. These tools have typically been designed to assist the implementation of a single specific phase or aspect of the program life cycle. The availability of such a large variety of tools and the software sophistication implied in their use, however, has tended to distract the developer from the purpose of the target software. As a result, potential users of software tools frequently fail to employ these tools either because they were unaware of their existence or because it was simpler to do without the support than to deal with the added complexity of using them.

SDEs seek to overcome these difficulties by providing an integrated environment in which the appropriate tools are made available but with as many as possible of the implementation details hidden from the user. The term "hiding" refers to the process of decreasing the user's awareness of tools or supporting file structures by reducing the need to explicitly access these entities. The ultimate SDE can be envisioned as a system which automatically produces the end product given the requirements of the software that is desired by the end user. Although this goal is beyond the state of the art at this time, current research is investigating techniques to make such systems a reality.

The purpose of this paper is to present an experimental SDE which has been developed specifically to support simulation programming at Texas A&M University. First three other major SDE efforts will be briefly summarized to provide background on current state of the art. Next the experimental simulation system will be presented along with a description of the use of specific tools. Finally the preliminary conclusions drawn from the use of the prototype relating to the utility and feasibility of SDEs to support simulation programming will be discussed.

2. PREVIOUS RESEARCH

SDEs have historically been concerned with managing the complexity of software development. Initially the environments were very unstructured and had few tools. The file systems implemented

This material is based upon work supported in part by the National Science Foundation under Grant No. ECS-8215550

in these early SDEs were primitive with compilers and editors being the primary tools included.

The UNIX* PWB (Programmers' Work Bench) was one of the first SDEs to emerge and win wide acceptance (Bourne 1978). A major feature which contributes to the usefulness of the UNIX PWB is the common and consistent tool interface which its simple concept of a file provides. A major weakness of UNIX has been the lack of information sharing by the system tools. As a means of alleviating this weakness, many SDEs developed since UNIX PWB have been incorporating an underlying database (DB). Many of the tools present in these newer environments interface with the DB which serves as a repository for knowledge about the environment. Tools which have some knowledge of the environment can enhance the power and utility of the environment by providing new and improved services.

Interlisp is another SDE which has been in use for a number of years (Teitelman and Masinter 1981). Based upon the Lisp programming language, Interlisp has been widely used within the artificial intelligence community and in the development of expert systems. Interlisp has demonstrated the advantages of tightly integrating the environment and the language and is very easy to extend. The technique of incorporating into one software development phase the process of coding, debugging, testing, and maintenance is also supported. One drawback to the Interlisp system is that its complex interfaces tend to require that the user have considerable expertise with the system prior to performing any productive work.

Another important recent contribution to the SDE development is the Ada^R Programming Support Environment (ASPE) (Buxton 1980). The U.S. Department of Defense recognized that the tools and environment which are used to develop a project are at least as important as the language itself. Therefore they sponsored the development of a set of requirements which define a collection of software tools to support Ada applications. This ASPE is defined in several levels to promote transportability of the environment to various hardware. At the inner level is the host operating system. The next layer, called the Kernel Ada Programming Support Environment (KAPSE) provides the logical-to-physical mapping of resources for the remaining layers. The next layer, the Minimal Ada Programming Support Environment (MAPSE), contains what is considered to be the minimal set of tools while the outermost layer, APSE, includes advanced tools to support various phases of the life cycle and specific projects. Thus the tools themselves can be designed and implemented in a machine independent fashion with appropriate implementations of the KAPSE providing the linkage to the actual hardware on which the APSE exists. Several organizations are currently working on implementations of APSEs.

* UNIX is a trademark of Bell LABS.

^R Ada is a registered trademark of the U.S. Department of Defense.

3. PREVIOUS SIMULATION ENVIRONMENTS AND TOOLS

Recent interest in software development environments has pointed out that there is more to developing a software system than simply choosing the most appropriate language. Tools are needed throughout the life cycle of the project to support the system developers. The evolution of SDEs to support simulation software development has followed that of general application SDEs. Initially a number of tools were developed to support specific aspects of simulation programming. A recent paper by Sheppard (1983) notes the use of various software engineering tools and techniques as they apply to simulation programming. Other simulation support tools include program generators (Mathewson 1981) and a database system which allows queries before, during and after program execution (Standridge 1981). In general other SDEs for simulation also apply only to a single stage of the software life cycle (Callender 1980, Clarkson 1980). None of these systems, however, provide state of the art integrated software development environments for simulation.

A simulation SDE has been designed and prototyped at Texas A&M University as part of a larger project whose primary goal is the development of a microprocessor-based distributed digital simulation system (Sheppard, Phillips and Young 1982). The research objectives of the project are (1) to conceptually design a simulation language based upon distributed processing, (2) to implement these concepts through construction of an executable simulation system, and (3) to evaluate the feasibility and utility of distributed simulation. A companion paper (Wyatt, Sheppard and Young 1983) gives the rationale for distributed processing in simulation and describes the development of a distributed simulation language.

The primary purpose of constructing a prototype simulation software development environment (SSDE) as part of this project was to provide a means to explore the feasibility and utility of SDEs for simulation programming. The SSDE has been implemented on a Texas Instruments (TI) 990/12 mini-computer using the DX10 operating system. The implementation language is TI Pascal with the TI DBMS being used to provide DB support.

4. THE PROTOTYPE SSDE

The prototype SSDE developed at Texas A&M University is composed of a system of Data Bases (DB) and a framework which permits the sharing of environmental data, provides the basis for supporting specific design methodologies, permits the file system and the tools to be hidden, and allows for multiple views of the software being developed (Reese 1983). This SSDE uses the concept of a design plan as the framework or control mechanism of the development process. A design plan (DP) is defined as a user modifiable, hierarchical set of instructions used by the SSDE to govern and direct the software development process. Design plans are used to provide a framework for the SSDE. A set of design plans are constructed to support specific software life cycle models. Each plan invokes a series of tools or other design plans appropriate for specific life cycle phases or sub-phases.

For example, to support generation of code the design plan may first invoke an editor to allow the user to create a module. This will be followed by the invocation of the compiler and then the linker, loader, and possible analysis tools as appropriate. Although similar means of combining these tools exist in most environments, they are not uniformly supported or used throughout the environment.

In addition, the design plan can be used to enforce life cycle phase transition requirements. For example, a requirement specification design plan can invoke the appropriate tool which would check the requirement specifications for completeness and consistency prior to entry to the design phase. This capability can be used to provide management better control of a project.

4.1 SSDE Overview

Figure 1 gives a functional overview of the prototype SSDE. The core of the SSDE consists of a

user interface which handles user/system communications, and an SSDE administrator which supervises the overall operation of the SSDE. There are three types of DB files in the system:

1. A design DB which provides generic templates for various software development methodologies.
2. A support tool DB which contains the tools used to develop the target software (including, for example, compilers, editors, and verification and analysis tools).
3. A target DB consisting of the software being developed and its documentation.

The system supports three types of users as illustrated in Figure 1. First is the target developer (which in the SSDE is a simulation programmer) who is responsible for creating the software system under consideration (i.e., the simulation model). A second type of user is the design plan devel-

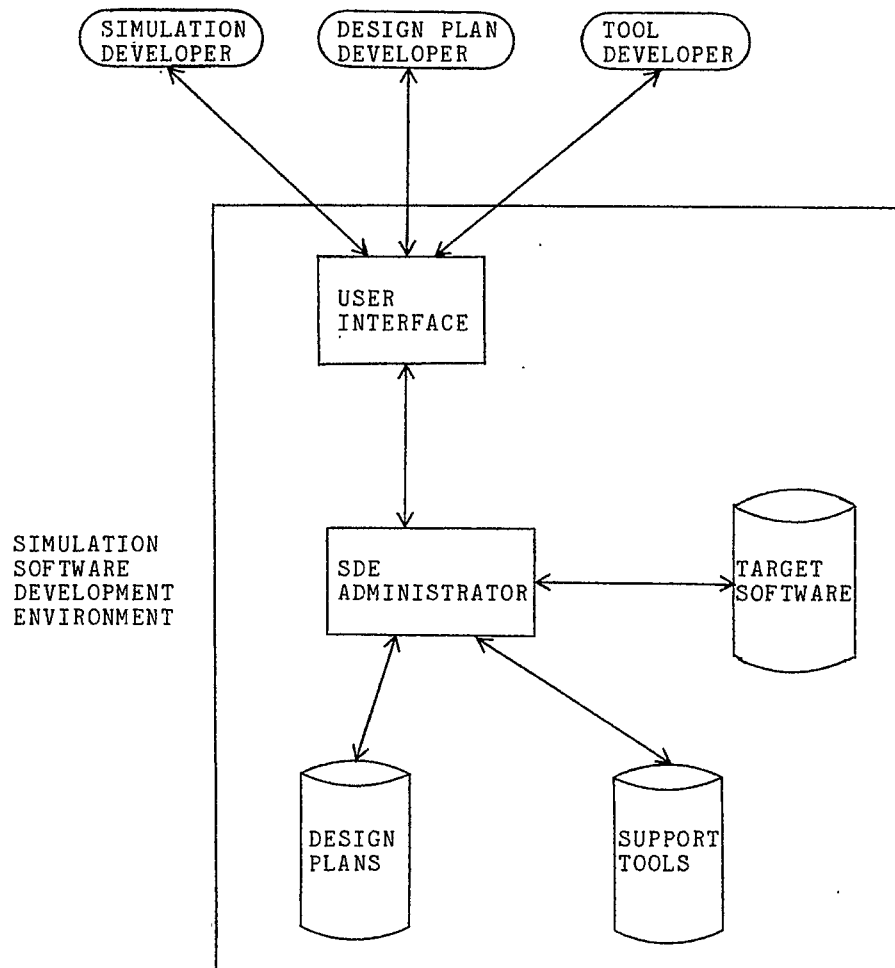


Figure 1: Functional Overview of the Simulation Software Development Environment.

oper who creates and maintains design plans supporting specific methodologies or sequences of operations. The third user is the support tool developer who develops and maintains support tools for use by specific design plans. Each of these types are described in more detail below.

4.1.1 Simulation Developer

The design plans guide the simulationist through the software development process in a tutorial fashion. This is achieved by invoking tools as specified by the DPs. The simulation programmer is automatically placed into the appropriate development state, such as editing, by the governing DP without the need to explicitly invoke the editor. These DPs may be organized in a hierarchical fashion in order to guide the user through the development process.

The decision to place the user into a specific state is determined by the DP. The decision may be based upon DPs which range in complexity from very simple to very complex. Should the user decide that the current state is inappropriate, commands are available to allow him to redirect his effort.

Once the activity of the current state is completed, the user is placed into the next state as directed by the DP. When a given DP is complete, another DP is executed as determined by a higher-level DP. This process continues until the project is complete. Throughout the process the evolving simulation program in its various representations is stored in the target DB.

4.1.2 Design Plan Developer

The DP developer creates design plans which control the system. The role of the DP developer in the SDE is somewhat analogous to the database administrator in database environments. The design plan developer interacts with the simulation users and the SSDE for the purposes of defining the database and for creating and maintaining design plans. The design plan developer creates these DPs and defines the DB using the various commands and support tools of the environment. The design plan developer may even create design plans for the specific purpose of aiding him in the completion of his task - namely the development of design plans. These special purpose design plans may help in the creation of design plans, the maintenance of the DB, or support of other areas.

4.1.3 Support Tool Developer

The support tool developer is responsible for the development and maintenance of the support tools. The role of the support tool developer is similar to the role of the simulation programmer in that both are responsible for creating and maintaining software. However, a different set of design plans may be used since different software development methodologies will in general be appropriate for the two kinds of software.

4.2. Design Plans

The design plan is implemented with a command language which is interpreted by the SDE administrator. It consists of a declaration section and a series of procedures which affect the operations.

Each procedure may invoke support tools similar to the manner in which external procedures are handled in other languages.

By allowing the user to modify these design plans, the user is given more flexibility in the development of procedures for which he is responsible. Allowing the developer to modify the design plans recognizes the individuality of all users. The creation of unique design plans is important in such development phases as testing.

The language used by the SSDE is a unified one in that the commands used by the design plans may also be invoked by the user. The command set, which is typically limited to the command level, may also be used in design plans. A well developed set of design plans and a complementary set of support tools can guide the user through the development process. The user will hopefully have little need to redirect the effort. If it necessary the user can take a different course of action. This might include for example, re-execution of a previously terminated life cycle phase or the examination of completed work.

The design plans support the evolutionary development of software. Should a new tool be needed or a change in the development methodology be deemed necessary, modification of the design plans can accommodate these changes in a relatively straight forward manner.

4.3 Data Base

An interesting aspect of the design plans is the declaration section. This in effect declares the schema for the target DB. The declarations are composed of components which are in turn composed of elements and other components. Components roughly correspond to the concept of records in Pascal while elements correspond to the parts of the record. At the same time the declarations describe the structure of the target DB. While the implementation method is DB dependent, components correspond to records of a DB and the elements are the line items of the records.

These declarations are dynamically modifiable. An addition of an item to the declaration will result in the automatic addition of this item to the appropriate component.

The data base and the associated declarations provide an easy method of accessing environmental information. Access to the DB is at the component and element level. The design plan developer is in effect defining the level and type of access to the DB. In UNIX the common interface is the file. In this system it is the component and element.

5. SAMPLE TOOL USAGE

The prototype SSDE does not contain a full set of tools capable of supporting all phases of simulation software development. Sufficient tools have been included, however, to give the flavor of system operation. The coding phase has been emphasized in the prototype because of the importance of this phase and because tools typically used in coding are well-defined.

The simulation implementation language supported by the prototype SSDE is SIMPAS which is a Pascal-based process-oriented discrete simulation language (Bryant 1981). This language was chosen because although it offers typical simulation facilities, it is simple and straight forward.

Specific tools available in the prototype include a SIMPAS compiler, a SIMPAS support editor, a SIMPAS style assessment tool, and a SIMPAS metric tool (under development). Since SIMPAS is based

upon Pascal, these tools will work as well on Pascal programs. The SIMPAS compiler, which is implemented in Pascal, was obtained from the University of Wisconsin and was adapted to run on a TI 990/12 minicomputer.

The SIMPAS support editor is screen-oriented and incorporates many of the ideas developed in the GANDALF project (Habermann 1980). The most useful feature of this editor is its ability to insert templates (such as those shown below in Table 1) into the simulation program at the dis-

COMMANDS	TEMPLATES
INclude	INCLUDE <name-1> [, <name-2>]...;
Start Simulation	START SIMULATION(<status>);
Event	EVENT <event-name>[<formal parameter list>; <label-part> <type-part> <var-part> <procedure and function decl part> BEGIN <statement-list> END;
Schedule	SCHEDULE<event-name>[<actual parameters> [NAMED <ev_ptr>] { NOW AT <time-expression> DELAY <time-expression> BEFORE <ev_ptr> AFTER <ev_ptr> }]
Cancel	CANCEL <ev_ptr>
DEStroy	DESTROY <ev_ptr>
DELeTe	DELETE <ev_ptr>
Reschedule	RESCHEDULE <ev_ptr> { AT <time-expression> DELAY <time-expression> BEFORE <ev_ptr> AFTER <ev_ptr> NOW }
Queue Member	<entity> = QUEUE MEMBER <attribute-1> : <type-1>; <attribute-2> : <type-2>; . . . END;
Queue Of	<queue-type> = QUEUE OF <entity>;
INsert	INSERT <e_ptr> [{FIRST LAST BEFORE <e_ptr> AFTER <e_ptr> } IN <queue>
REmove	REMOVE [THE] [{FIRST LAST}] <e_ptr> FROM <queue>
FORall	FORALL <e_ptr> IN <queue> [IN reverse] DO BEGIN <statement list> END

Table 1: SIMPAS Templates.

creation of the programmer. The set of templates currently in use in the support editor are for SIMPAS and Pascal statements. Sample templates are shown in Table 1.

To insert the templates into the program the user positions the cursor at the appropriate point on the screen and depresses a command key. This causes a list of template commands to appear at the bottom of the screen from which the desired template is selected. The editor then automatically inserts the template at the cursor position and prepares to accept user inputs to modify the template for the specific instance.

Figures 2 and 3 illustrate this template insertion process. Figure 2 displays the state of the editor after the cursor has been positioned at the insertion point as indicated by the dollar sign and after the command key has been depressed. The command key is implemented in the prototype via a function key on a TI 911 video terminal. The depressing of the command key results in the display of the template commands at the bottom of the screen. Note that the commands, when entered, can be abbreviated with only the use of the capitalized letters in the command name. The insertion command is entered at the bottom of the screen as illustrated in Figure 2 with the entry of the "S" for the schedule command. Figure 3 shows the inserted text after the execution of this command. The SSDE is now ready to replace the <event name> with the specific user input, along with optional parameters and scheduling

information necessary to complete the instruction.

This template insertion method has a number of advantages. The most obvious one is that the user need not key-in standard statement sequences. This avoids unnecessary effort and helps to reduce the number of syntax errors. The templates also assist the programmer should he forget the exact form or syntax of a specific statement.

These templates can be created by the user and are not limited to SIMPAS or Pascal templates. By modifying the command and template libraries the user can alter old templates or add new ones. Table 2 illustrates other template uses. Note that they are not limited to programming language statements.

A SIMPAS style assessment tool was included as part of the support tool DB. A style assessment tool is used to automatically quantize the style of a program. While style assessment tools have been criticized (Glass 1983), they can be used by knowledgeable programmers to provide a positive, albeit limited, feedback. The style assessment tool included in the prototype was patterned after (REES 1982).

A simple metric tool based primarily upon Halstead's "Software Science" (Halstead 1983) is being developed to illustrate the use of metrics in SDEs. The inclusion of metric software in the SDE allows the metric designer to have access to information from all phases of the life cycle.

```

      .
      .
      .
WHILE PORT_IS_OPEN DO
  BEGIN
    CASE DOCK_NUMBER OF

      DOCK_ONE:
        BEGIN
          $
        END;

      DOCK_TWO:
        BEGIN
          :

```

```

INclude, Start Simulation, Event, Schedule, Cancel, DESTroy, DElete,
Reschedule, Queue Member, Queue Of, Insert, REMove, Forall, Program,
PRocedure, FUnction, RECord, Begin/end, IF, While, Repeat, CAsE, For
S

```

Figure 2: Screen As the Schedule Command is Selected.

```

.
.
WHILE PORT_IS_OPEN DO
BEGIN
CASE DOCK_NUMBER OF
DOCK_ONE:
BEGIN
SCHEDULE<event-name>[<actual parameters>]
[NAMED <ev_ptr>]
{ NOW |
AT <time-expression> |
DELAY <time-expression> |
BEFORE <ev_ptr> |
AFTER <ev_ptr> }
END;
DOCK_TWO:
BEGIN

```

Figure 3: Screen After Completion of the Insertion Process.



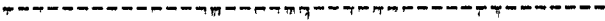
COMMANDS	TEMPLATES
Documentation	AUTHOR; DATE CREATED; DATE LAST UPDATED; VERSION: PURPOSE: INPUTS: OUTPUTS:
Big Box	
Small Box	
Line	

Table 2: Other Possible Templates.

Early metric designers usually had to rely on either static analysis of the source code or dynamic execution traces. Metric tools implemented within SDEs, however, can monitor the evolution of software - from the specification stage through to maintenance.

6. CONCLUSIONS AND FUTURE RESEARCH

Experimentation with the prototype has established the basic feasibility and utility of the simulation software development environment. In particular the prototype SSDE was able to support the following useful facilities and functions:

1. A user-friendly interface which can be tailored to the specific needs of different users with the construction of appropriate design plans.
2. The capability to support all phases of the software development process.
3. An integration of individual existing tools with newly developed tools to support specific aspects of the software development process.
4. Access to this collection of tools without the user being required to know and understand many of the details of the tools.
5. Support of multiple target languages (e.g. SIMPAS and Pascal in the prototype).

The most useful implementation concepts explored in the prototype were the design plans and database usage. A well designed set of DPs can reduce the need on the part of the simulationist to explicitly use the available commands since the DPs can be designed to invoke the required tools at the appropriate times. Further research is needed to establish criteria for determining the adequacy of a design plan set and in identifying any inherent limitations of the design plan concept in meeting the needs of the users.

Databases have been used extensively in the prototype SSDE as repositories of information regarding the evolving simulation program as well as the support tools and design plans. The feasibility of using available database management systems in accessing these databases was established in the prototype. The use of a DBMS, however, does affect the portability of the SSDE. Additional experimentation is needed relating to database organization and interface to determine optimum organization and access methods for specific information requirements.

The prototype has emphasized the coding phase of software development. Additional study is needed to design and develop support tools for other life cycle phases. Many existing tools can simply be incorporated into the SDE. The prototype provides a testbed for this proposed research and other experimentation in the area of software development.

REFERENCES

- Bourne SR (1978), UNIX Time-Sharing System: The Unix Shell, The Bell System Technical Journal, Vol. 57, No. 6 Part 2, pp. 1971-1990.
- Bryant RM (1981), SIMPAS User Manual, Computer Science Department and Madison Academic Computing Center, University of Wisconsin-Madison, Madison WI.
- Buxton JN (1980), Requirements for Ada Programming Support Environments, Department of Defense, Washington, D.C.
- Callender EE (1980), An Overview of Software Design Languages, Proceedings of the 1980 Summer Computer Simulation Conference, AFIPS Press, Arlington VA, pp. 251-256.
- Clarkson WK (1980), Structured Design and Programming for Simulation, Proceedings of the 1980 Summer Computer Simulation Conference, AFIPS Press, Arlington VA, pp. 257-263.
- Glass RL (1983), Letter to the Editor, SIGPLAN Notices, Vol. 18, No. 7, pp. 11.
- Haberman AN (1980), Tools for Software System Construction, Riddle WE and Fairly RE (eds), Software Development Tools, Springer Verlag, Heidelberg.
- Halstead M (1977), Elements of Software Science, Elsevier, New York.
- Mathewson SC (1981), A DRAFT II/SIMON Manual, Department of Management Science, Imperial College, London.
- Rees, MJ (1982), Automatic Assessment Aids for Pascal Programs, SIGPLAN Notices, Vol. 17, No. 10, pp. 33-42.
- Reese RM (1983), A Language Directed Software Development Environment, PhD Dissertation, Texas A&M University, College Station TX.
- Sheppard SV, Phillips DT, Young RE (1982), The Design and Implementation of a Microprocessor-Based Distributed Digital Simulation System, NSF Grant No. ECS-8215550.
- Sheppard SV (1983), Applying Software Engineering to Simulation, Simulation, Vol. 40, No. 1, pp. 13-19.
- Standridge CR (1981), Using the Simulation Data Language (SDL), Simulation, Vol. 37, No. 3, pp. 73-81.
- Teitelman W, Masinter L (1981), The Interlisp Programming Environment, Computer, Vol. 14, No. 4, pp. 25-33.
- Wasserman AI (1981), Toward Integrated Software Development Environments, Ed. Wasserman A.I., Tutorial: Software Development Environments, IEEE, New York, pp. 15-35.
- Wyatt DL, Sheppard SV, Young RE (1983), An Experiment in Microprocessor-Based Distributed Digital Simulation, Proceedings of the 1983 Winter Simulation Conference.