Structured Programming (1991) 12: 43-60

Structured Programming

Prototyping-Oriented Software Development—Concepts and Tools

Gustav Pomberger[1], **Walter Bischofberger**[2], **Dieter Kolb**[3], **Wolfgang Pree**[1], **Holger Schlemm**[3]

[1] Institut für Wirtschaftsinformatik, Johannes Kepler University, Altenbergerstr. 69 A-4040 Linz, Austria
e-mail: K2G0190@AEARN

[2] UBILAB (UBS Informatic Laboratory), Union Bank Switzerland, CH-8021 Zürich, Switzerland
e-mail: proto@ifi.unizh.ch

[3] Siemens AG, ZFE IS SOF, Otto-Hahn-Ring 6, D-8000 Munich, Germany
e-mail: glas%moony@ztivax.uucp

Abstract. It is often assumed—and current reports from research and industry confirm this assumption—that a prototyping-oriented development methodology can ameliorate some of the weaknesses of the life cycle-oriented development approach.

Specialists have not arrived at a consensus on what methods and tools are necessary for supporting prototyping-oriented software development. Based on the results of a several year long research project, this paper explains the authors' concept of prototyping in the area of software development and what tools are necessary to support it.

Key Words: prototyping, software development environment, software engineering environment, software life cycle, architecture prototyping, user interface prototyping, incremental software development

1. Drawbacks of the Conventional Software Design Methodology

When people are confronted with the need to solve a complex task, they attempt to systematically decompose the process of solving the problem, i.e., to define an approach model. Such an approach model regulates the chronological sequence of the solving process. It decomposes the solving process into distinct steps that are intended to make a stepwise planning, decision and implementation possible.

We apply this basic idea from the field of systems engineering to the design process for software products. Analogous to other types of projects, software projects are thus subdivided into individual *project phases*. The phases and their chronological sequence are collectively termed the *software life cycle*, which has become a classical term in the field of computer science.

Software design using the life cycle paradigm is based on the principle of *top-down decomposition* of so-called *black boxes*, i.e., stepwise refinement (see Figure 1).

The prerequisites for the individual phases are well-defined inputs (usually in the form of documents). These inputs are processed in the respective phases with methods and tools availed by software engineering [18], and the results are passed on to the next phase.

The phases are clearly defined, and a given phase is terminated only when its outcome is validated and verified.

One characteristic of the software life cycle paradigm is that during the analysis and specification phases the system is described *from the outside*, i.e., in terms of *what* the system is to achieve. *How* the system is to achieve the requirements is left open.

The system itself is viewed as a black box whose outward effect is precisely defined and whose internal structure remains hidden. During the system design phase the procedure is analogous in that one specifies the components into which the system is to be decomposed, *what* the components must do, and how they must work together. When the process is completed, i.e., the interfaces of all the system components are defined, the designers need only tend to

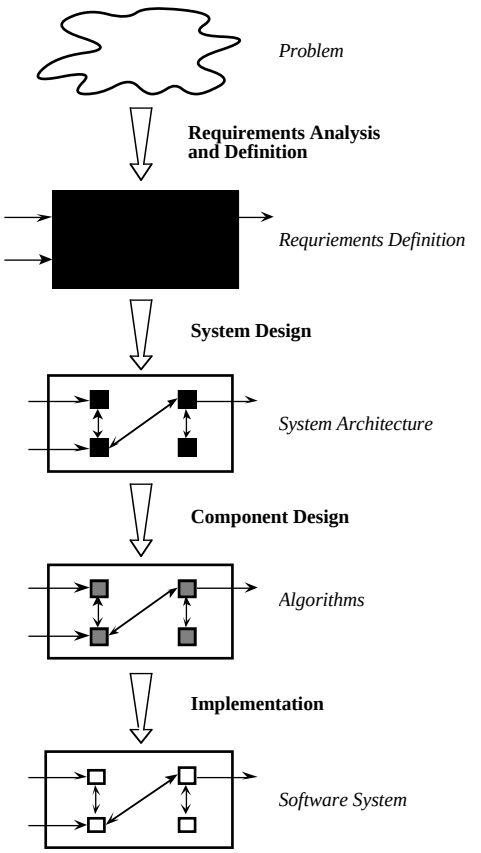

Figure 1. Stepwise refinement according to the life cycle paradigm (see [30]).

component specifications in order to achieve the design of the algorithmic structure of the system components. The process supports a successive reduction in complexity of both the design process and the product itself. The algorithms are then translated into a programming language and tested individually. The decomposition is followed by a synthesis of system components.

Studies have shown that the software life cycle paradigm is the most widespread software development methodology used today and that it has in general proven to be useful. Practice, however, shows the limits and the drawbacks of this paradigm.

The model is based on the (false) assumption that (as a rule) the development process is linear and iterations between phases occur only as exceptions. Such iterations are included in the description of the life cycle-oriented development methodology, but it is unclear when and according to what criteria the iterations take place.

Strict application of this development method requires that one phase can only be begun after the preceding phase is completed, that is, when the respective intermediate products are available. In reality, however, a complete specification or a suitable system architecture can seldom be produced straight off. Usually the later phases have a strong impact on the earlier ones.

The strict separation of the individual phases is an unacceptable idealization. In reality the activities of the phases overlap and interaction between phases is much more complex than exhibited in the paradigm.

The strictly sequential approach leads to tangible products or components being available only at a late stage. Yet experience shows that the validation process is insufficient without experiments close to reality. Furthermore, modifications requested by the client can only be expressed relatively late, and integrating them at that stage can lead to substantial overhead.

In short, the life cycle-oriented software development paradigm has proven itself in the field and serves as an established basis for an engineering methodology, although with serious drawbacks. It describes a sequence of activities and the nature of the intermediate products (results of phases). The activities established in this paradigm are certainly necessary—although perhaps not sufficient—and cannot be circumvented by any other ingenious methodology.

The use of this methodology in practice has taught us that we cannot achieve the goal of system development completely independently of the type of solution chosen. But exactly that is assumed in the life cycle paradigm. In order to allow us to escape this problem, the mutual effects of *what* the goal is and *how* to get to it must be given more consideration. The strict sequential proceeding, the manner in which the prescribed activities are performed, and the nature of the intermediate products form the weak points of this methodology and leave room for improvement.

It is often assumed—and current reports from research and industry confirm this assumption—that a *prototyping-oriented development methodology* can ameliorate some of the weaknesses of the life cycle-oriented development approach.

2. The Prototyping-Oriented Software Development Methodology

Reports on software projects in which prototypes were constructed in order to clarify user requirements and design problems have been appearing in the literature for more than a decade. An analysis reveals that there is no consensus on the terms *prototype* and *prototyping*. The same is true of what is known as *prototyping-oriented design methods.*

There is obvious agreement that software prototypes must be operational. Whether this operationality must be direct or simulated is the subject of controversy. The experts are nearly unanimous that software

prototypes—in contrast to other prototypes, e.g., in the realm of hardware—must be *realized quickly* and *cheaply*. Some writers advocate the design of *real prototypes* in the sense of templates that have all the relevant characteristics of the planned product and are then used as the specification for the actual product development process—prototyping in the classical sense. Others believe that software prototyping is a *bottom-up* process: a few (simple) basic functions are implemented quickly, tested by the user, and improved; then additional user requirements are implemented, and the cycle continues until the product is finished.

Since there is no generally accepted definition of *prototype* and *prototyping*, we first need to establish what we mean by the terms.

Bernhard Boar [5] has defined *prototyping* as a specific strategy for performing requirements definitions wherein user needs are extracted, presented, and successively refined by building a working model of the ultimate system quickly and in its working context.

A useful definition is given by Connell and Shafer [7]: "A *software prototype* is a dynamic visual model providing a communication tool for customer and developer that is far more effective than either narrative prose or static visual models for portraying functionality. It has been described as:

- functional after a minimal amount of effort
- a means for providing users of a proposed application with a physical representation of key parts of the system before system implementation
- flexible modifications require minimal effort
- not necessarily representative of a complete system."

This definition is deliberately general. Its purpose is to establish that experimenting with models is very useful in the development of large software systems and that the general proceeding should be similar to that of development in other technical areas. For further definitions see [9] and [12].

2.1 The Prototyping-Oriented Software Life Cycle

A prototyping-oriented development paradigm is not radically different from a purely phase-oriented one. Furthermore, the two are to be viewed more as complementary than as alternative. The new aspects are that this model is explicitly not linear but iterative and that it specifies where and how this iteration is not just possible but necessary. The modified life cycle paradigm is shown in Figure 2 (see also [2]).

The prototyping-oriented development methodology (as we see it) differs mostly from the conventional

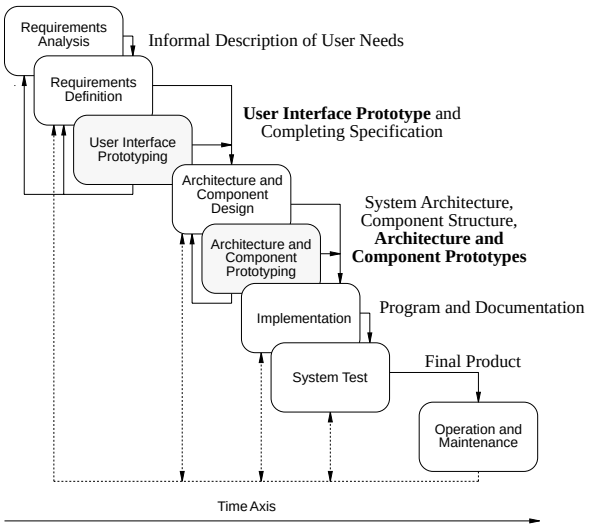

Figure 2. Prototyping-Oriented Software Life Cycle.

development methodology in its activities and the results produced in the individual phases. Although the distinction of phases is maintained, problem analysis and specification overlap a great deal, and design, implementation and testing very much blend into one another (see Figure 2). The phases are thus no longer phases of a continuous development. We therefore no longer speak of phases in the sense of the classical software life cycle. Instead we refer to *activities* which can no longer be clearly separated as was customary.

First a prototype is created (in a tool-supported manner) based on the results of preliminary activities. This prototype permits experimentation that reflects the user's application of the product in order to evaluate whether the user's requirements have been met. In an environment close to reality, the software developer and the user can test whether the system model has errors, whether it is what the user had in mind, and whether modifications are necessary.

This leads to another significant difference between the two approaches. In the conventional model implementation is carried out *as late as possible* and only after all the details of the specification and design processes have been clarified. In the prototyping-oriented model implemention of a prototype is carried out *as soon as possible*. Experience shows that it is more likely to achieve the goal if the requirements specification and system architecture are developed stepwise, based on a model that permits the dynamic representation of system behavior, i.e., based on an executable prototype and not merely on static descriptions.

In this way the risk of a bad decision is reduced. Even more, the *learning effect* that is achieved by

experimenting with the results of the phases presents a new dimension in quality assurance.

Whether the various prototypes (for user interfaces, system architecture, or individual components) are throw-away or reusable is not important for the general methodology. The goals are risk reduction, quality assurance and learning from experimentation. With respect to the expensive production process, a reusable prototype is desirable; i.e., an evolutionary development strategy would be best. The quality of the prototyping tools is the primary factor in determining whether this is possible.

3. Tools for the Production of Prototypes

Prototypes must be produced quickly and cheaply. The reason that the idea of prototyping arose so late is that tools for the efficient production of prototypes are difficult to produce and thus were not available in sufficient quality and number. The tools used are central to both the prototype production process and to the modification and extension process. Typical classes of prototyping tools are discussed below (see also [20], [19]).

Generators

Generator tools generally consist of two components: an *editor* with which specifications are written and a *compiler/interpreter/run-time system* for checking the syntactic correctness of the specifications and transforming them into an operational system.

The software (or prototype) developer only has to specify what has to be done; he is not concerned with the realization.

There are different methods to specify the behaviour of the prototype abstractly, for instance declarative languages (e.g., the user interface specification language, UISL, described in [15]), attributed grammars (examples in [22]), and graphic specifications (e.g., the dynamic interface creation environment, DICE, described in [21]).

Typical examples of the generators that can be employed for prototyping include user interface generators, generators for lexical and syntax analyzers, and generators for information systems.

Generators are particularly suitable as prototyping tools because they permit specification on a high abstraction level, rapid implementation, and fast and simple modification.

The drawback of generators is often their lack of flexibility—prototypes that cannot be described in the specification language cannot be generated, even if they are very similar to describable prototypes—and in the usually unsatisfactory efficiency of the generated code.

Ordinary module and procedure libraries are suitable for prototyping only to a limited extent because modules and procedures represent an abstraction level that is too low. Even the smallest adaptation has to be made at the code level, and the software engineer still has to develop large parts of the application himself.

Generic applications are a significant step forward. A generic application has the standard I/0 behavior of an application, e.g., for managing windows, lists, input/output masks and graphical objects. In order to produce a specialized prototype for an application, the generic application is extended with application-specific parts. The positions where extensions can be made (*hooks*) are predetermined by the generic application.

Modern implementations of generic applications are based on object-oriented programming languages. These are known as *Application Frameworks* (e.g., MacApp [28], ET++ [27], Application Kit [17], and Smallkit [25]). Compared to implementations with conventional languages, they have the advantage that extensibility is not limited to those parts that were foreseen. The inheritance mechanism of object-oriented languages makes significant extensions of generic applications easily possible.

Application frameworks are thus tools that support above all evolutionary prototyping. The production of prototypes is more difficult than with generators, for parts of the prototype have to be written by the developer. However, this is greatly simplified by powerful building blocks and the flexibility of object-oriented languages.

Architecture Simulators

The result of the design process is the system architecture, i.e., a specification of all necessary system components and their interrelationships. Architecture simulators are tools that permit the verification of the mutual effects among specified components before they are realized. A tool to support architecture verification must offer the following functions:

- simulation of the information flow between system components
- linking the architecture design with user interface prototypes
- display of current system state and selection of the next system state
- automatic logging of state transitions during simulation
- automatic playback of already executed simulations
- support of an incremental implementation process

The development of tools for architecture prototyping is in its infancy. A possible approach is described in Section 4.

Database Management Systems and Fourth Generation Systems

The production of prototypes for information systems requires

- tools for (interactive) describing database schemata
- tools for generating a database
- tools for testing a database
- tools for modelling a convenient user interface

Graphic-oriented tools for defining the schemata are most suitable for prototyping. The database must be generated directly from the schemata definition by means of a generator.

Data definition languages (DDLs) are less comfortable than graphical schemata editors, but they are nonetheless suitable as prototyping tools. The expressive power of DDLs matches and sometimes exceeds that of graphical schemata editors. Often a normal text editor is used for the creation of schemata specifications. This has the serious drawback that structural errors are detected only when the database is generated. Interactive database languages are also suitable for prototyping. They lend themselves to both the data definition and the formulation of data manipulations.

Fourth generation systems are important tools for the production of prototypes of information systems. They usually integrate a schemata editor, a user interface generator and an application generator, permitting the definition of database schemata and user interfaces and the generation of applications as well as the creation and deletion of relations and associative queries.

Programming Languages for Prototyping

Complete prototypes can seldom be produced exclusively with generators and reusable software. Highly complex parts still need to be written by hand.

The choice of a programming language can have a significant impact on the effort invested in production of a system component and on its modifiability and structuredness. Problem-specific programming languages—such as APL for mathematical-statistical problems, Snobol for string processing tasks, or Lisp in the area of symbolic programming—permit simpler and more rapid implementation of algorithms in the application area for which the language was conceived than is possible in other languages. Such an implementation of a system component usually does not achieve the efficiency and structuredness that are expected of the ultimate software product. However, other quality requirements are placed on prototypes.

The most important characteristics that a programming language should possess in order to be employed for prototyping are:

- interpretation instead of compilation
- abstraction concepts that permit the construction of reusable components and easy extensibility
- imbedding in a development environment

4. Topos—A Toolset for Prototyping-Oriented Software Development[1]

The preceding sections made a number of statements about the prototyping-oriented software development methodology and explained what kinds of tools can be employed for the production of prototypes. Based on a scenario for *prototyping-oriented incremental software development*, which is built on fundamental methodological considerations, this section presents a concept for a prototyping-oriented software development environment and discusses its implementation.

4.1 Basic Concepts

Software can be developed with algorithmic languages or with high level prototyping tools. It is obvious that the abstraction level on which a software system is realized has a profound influence on the required development effort and on the (system) architecture of the resulting application. With an increasing abstraction level, the programmer is less bothered with the design and implementation of an application. In the ideal case he just specifies what functionality it should provide, while the knowledge about the design and implementation is coded in the tools used to execute the high level formalisms.

Development at a high abstraction level is more economical than in an algorithmic language. Experience has shown that the amount of code which has to be written as well as the development costs can be reduced by several orders of magnitude.

Unfortunately there are few formalisms and tools for developing entire software systems on a high abstraction level, and their application areas are limited because software systems written for the same application area usually differ in a myriad of details which cannot be standardized and captured by a tool.

---

[1] This project was a cooperation between the University of Linz, the University of Zürich and Siemens Munich.

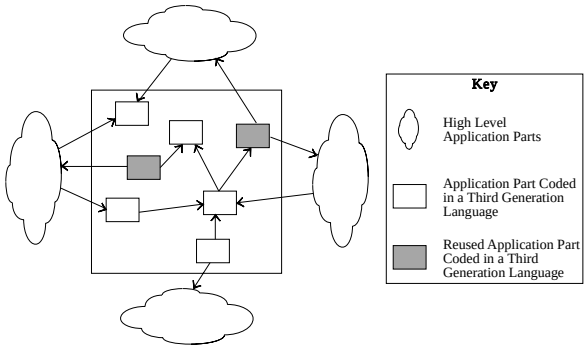

Figure 3. A hybrid system architecture.

Because of this shortcoming, many formalisms and tools were fashioned which make it possible to develop standardizable parts of an application on a high abstraction level (high level parts) and to connect them with other parts written in a third generation language (low level parts). Depending on how much of the final application can be written on a high abstraction level, these formalisms and tools can help to pronouncedly speed up the development process.

Software systems developed with such formalisms and tools have a hybrid architecture; i.e., they consist of parts written in different formalisms. For this reason they are called *hybrid software systems* (see also [3]). For an example of a hybrid architecture see Figure 3.

Today two paradigms are applied to the practical development of software systems with hybrid architectures. Under the first, the *conventional sequential paradigm*, the high level formalisms and tools are utilized to speed up the design and implementation. Under the second, the *evolutionary prototyping paradigm*, the high level formalisms and tools are used to develop reusable prototypes in an iterative process. During this process the evolving prototypes are discussed with the client, thus helping to determine the requirements. If the prototypes cover only parts of the final application (which is usually the case), the other parts are developed conventionally and integrated with the reusable prototypes.

The combination of prototyping and the conventional sequential paradigm is well established and frequently used in practical projects. This approach is successful in building hybrid software systems where the conventionally developed part is not too complex.

During the development of novel complex hybrid sytems in the Topos project, it became more and more obvious that the prototyping approach was helpful during specification, but that methods and tools were missing to support the subsequent realization process. Because of their novelty and complexity, many low level parts were implemented by following the exploratory programming paradigm. Some of the changes applied to them also implied changes of the reusable prototypes. The resulting realization process therefore consisted of a dynamic evolution of application parts written in an algorithmic language and reusable high level prototypes.

This parallel evolution of high and low level parts proved tedious for the following three reasons:

1) The manual management of complex hybrid software systems is tiresome and error-prone.
2) It is difficult to determine if a planned architecture consisting of different reusable high level prototypes as well as reused and customized conventionally developed building blocks will work properly.
3) It becomes expensive to change prototypes once they are integrated with the conventional parts because the reintegration is time consuming and error-prone.

An approach consisting of prototyping *and* exploratory programming would have alleviated these problems, but unfortunately there were neither methods nor tools allowing for a combination of the two paradigms.

Because of these problems and the “obvious” solution, the goal of a subproject of Topos increasingly became to study the integration of prototyping and exploratory programming. The results are *prototyping-oriented incremental software development*, a paradigm, as well as methods and tools which were developed to support it.

Figure 4 shows a general representation of a prototyping-oriented incremental software development process. It consists of three cyclical processes. The novelty is that two processes (*prototyping* and *exploratory programming*) are integrated as subprocesses into one *cyclical prototyping-oriented incremental software development process* (the third cyclical process).

The paradigm is very flexible and suited to describing almost every kind of software development process. The conventional sequential paradigm as well as prototyping and exploratory programming are special cases thereof, whereby some of the cyclical processes are carried out only once or omitted completely. While the subsumption of various time-tested paradigms under a new one proves its orthogonality and generality, the cases which were not considered by the old paradigms but can be be mastered by applying the new one give it its relevance. These new cases include, but are not limited to, the use of exploratory programming during prototyping and the evolution of reusable prototypes during the final implementation of the application.

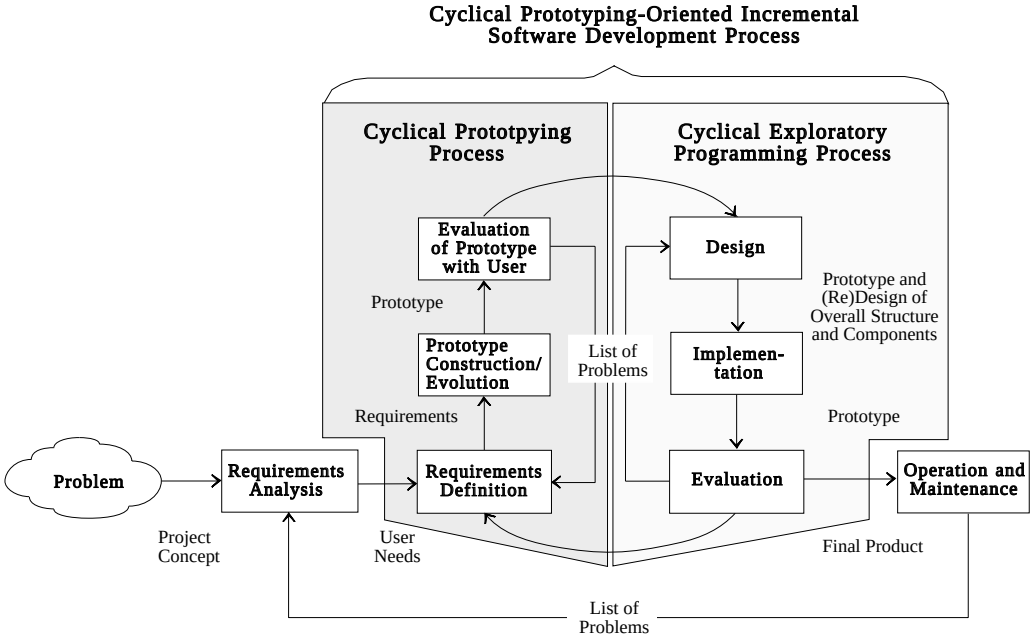

Figure 4. The prototyping-oriented incremental software development process.

There are two cases in which *exploratory programming* is important *during prototyping*. First, high level prototypes occasionally have to be enhanced with chunks of algorithmic language code because they do not provide enough functionality themselves. Second, feasibility risks and possible performance problems sometimes require the realization of low level application parts in order to ensure that a system can be built as specified with the high level prototypes.

The second case considered by the prototyping-oriented incremental paradigm is that many *prototypes* which were built as illustrations for the customer *evolve during the subsequent realization*. This can happen because the prototypes were partly mock-ups, because they were functionally incomplete, because the requirements change, and for many other reasons.

It is obvious that in all of these cases management, integration, and validation problems arise unless adequate tool support is available. Unfortunately almost no such tool support exists. There are excellent prototyping tools and programming environments, but the required combination of these tools has not yet been accomplished.

Two approaches could be taken in order to build adequate tools. The first is to develop an *integrated tool set* consisting of prototyping tools and a programming environment for exploratory programming. The second is to build a *tool serving to connect different existing prototyping tools* with a programming environment.

Comparing the two approaches, it can be seen that successfully taking the first approach results in a comfortable, integrated tool which can be used to develop applications for a well defined application area.

The second approach mentioned above results in a more loosely coupled extensible tool kit providing less comfort but facilitating the development of a wider range of applications. This approach was taken in the Topos project.

A tool for prototyping-oriented incremental software development has to support prototyping and exploratory programming effectively. Furthermore it has to provide features for the management of complex hybrid software systems, for the validation of hybrid system architectures, and for fast execution of a hybrid software system under development after changes were applied to any part of it (i.e., short edit/run cycles).

4.2 The Structure of Topos

The considerations that led to Figure 2 (prototyping-oriented software life cycle) and Figure 4 (prototyping-oriented incremental software development process) make it clear that we need tools that support 1) the requirements analysis and specification phases by enabling the construction of user interfaces, and 2) architecture verification by permitting the realization of prototypes of system architectures and their execution together with the user interfaces prototypes. In addition we need tools for the management of generated compo-

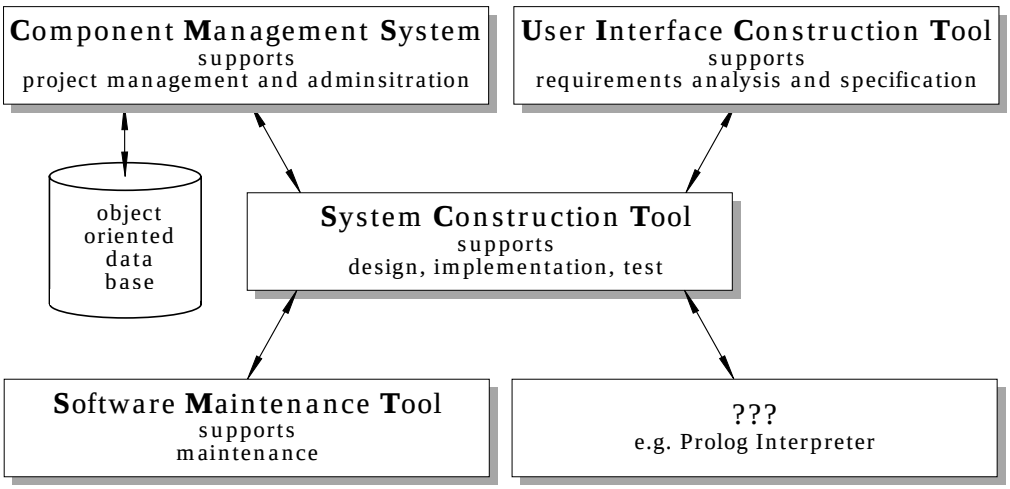

Figure 5. The coarse structure of Topos.

nents (prototypes, documents, modules, etc.) and tools that support incremental implementation.

The coarse structure of Topos thus takes the form shown in Figure 5.

We discuss only the tools that relate directly to prototyping, i.e., UICT (User Interface Construction Tool) and SCT (System Construction Tool). A description of CMS (the Component Management System) can be found in [24].

4.3 The User Interface Construction Tool (UICT)

UICT is a tool that supports exploratory and evolutionary prototyping and enables the easy construction of prototypes for dialog-oriented software systems. The user interfaces of such systems can be described in terms of finite automata defined by a set of *states* and possible *state transitions*. From the viewpoint of the user these states are defined by *graphical representations* on the communication medium, i.e., the screen or printed output, and by their associated *functionality*. A state transition is invoked by the reaction of the user to a particular state and/or by the system itself. Each software system has a set of special states: the initial state and final states, e.g., the "stop state" or various error states.

UICT permits a flexible configuration of the graphical representations and the simulation of state transitions. The user interfaces supported by UICT meet the most modern requirements. Such user interfaces presuppose high resolution graphic monitors with a mouse and are based on user interfaces such as those of Sun workstations and Macintosh computers. The design concepts can be transferred to other machines that have similar characteristics.

The most important elements are environments consisting of *windows*, *subwindows*, *pop-up menus*, *dialog boxes* and *lists*.

Every user interface is associated with one window in which, from the user's point of view, the application runs. The position and form of the window must be defined.

Subwindows as well as the remaining area must be definable in such an (application) window. The subwindows are usually of various types and sizes. The type of a window determines its form and its basic functional behavior. Within these limits, however, a free definition of the functionality should be guaranteed. This functionality is determined more exactly by specification of behavior on user input. On the one hand, the temporary changes in the behavior of an interface must be specifiable, e.g., that a pop-up menu or a dialog box is to appear on the screen when a text or a graphical symbol is to be selected, etc. On the other hand, state transitions need to be specified, e.g., that a new text, another list or a picture is to be displayed when the number and type of subwindows is to change, etc. In addition it must be possible to define procedures that are invoked on the occurrence of certain events.

The graphical representation of the state of an application window, i.e., the displayed subwindows and the remaining area, combined with the functionality that is defined for the window as a whole is termed the window's *environment*. In the sense of finite automata every window has a particular environment that is to be displayed first, the *start environment*. The functional properties of the start environment's subwindows determine to which other environments the user can make a transition. An interface can thus be perceived as a directed graph of environments.

Figure 6. Supported interface: environment *Main*, menu *MainBorrower*, environment *Search*, sheet *ParamSearch.*

We distinguish five types of subwindows:

- *Text Editor subwindow*
- *Graphics Editor subwindow*
- *ListMenu subwindow*: a static or dynamic menu with a scrollable list of entries, one of which can be selected at a time. Depending on the specification of the environment, this can cause a different list to be displayed, a procedure to be called, etc.
- *MaskButton subwindow*: a subwindow containing input/output masks with fields for editing text, push buttons, slide controllers, etc.

These four types cover the standard types of interactions of most applications. To keep the system open to prototyping of even more sophisticated user interfaces, we provide another type:

- *Empty subwindow*: a universal subwindow whose functionality is managed by user-supplied code.

The left side of Figure 6 shows an environment consisting of a single subwindow of type Empty subwindow. In reaction to the right mouse button being pressed, a pop-up menu (borrower menu) is displayed. If the line “search book” is selected, the environment *Search* is displayed. The right side of Figure 6 shows this environment. It consists of two List-Menu subwindows (“Authors” and “Subjects”) and a Text Editor subwindow. When the right mouse button is pressed, a dialog box (“Set additional parameters”) is displayed.

The coarse structure of UICT and its interaction with the Component Management System and the System Construction Tool are shown in Figure 8.

UICT is a generator system in the sense of Section 3 and includes the tools *DICE*, *UI Translator*, *UI Interpreter* and *M2/C Code Generator*.

To allow UICT to be experimentally verified as quickly as possibly, the overhead of implementing DICE (Dynamic Interface Creation Environment) was initially postponed. Instead, an easily implemented specification language, UISL (User Interface Specification Language), was defined. Figure 7 shows the UISL-Code for describing the prototype illustrated in Figure 6. The goal of the design of UISL was to provide a simple, compact and extensible formalism with a high abstraction level and high documentation value, allowing UICT to be applied productively even without a graphical editor such as DICE.

To check the syntactic correctness of a prototype specification formulated in UISL, a *Translator* was developed. This tool stores syntactically correct specifications as *UI Tables.*

In order to be able to experiment with the prototype of a user interface, the UI Tables can be processed by the UI Interpreter. If the functionality of the created user interface elements does not suffice, the prototype description can be augmented with procedure invocations.

```
APPLICATION LibInfoSystem:

WINDOW
  TITLE (" Library Information System");
  POS (0, 100);
  EXT (920, 650);
  ICONFILE (Icons/LIS.icon);
END WINDOW;

ENVIRONMENT Start: START;
  SUBWINDOWS (LStart, EmStart);
END Start;
...
ENVIRONMENT Search:
   SUBWINDOWS (AuthorsSearch, SubjectsSearch,
       ParmSearch, ResultsSearch, EditSearch);
...
SUBWINDOW MainBorrower: LISTMENU;
  STARTLIST (MainBorrower);
  EXT (EXTEND, 18);
  MENUBUTTONACTION (MENU (MainBorrower));
END MainBorrower;
...
SUBWINDOW AuthorsSearch: LISTMENU;
  STARTLIST (AuthorsSearch);
  EXT (460, 140);
  MENUBUTTONACTION (MENU (Search));
END AuthorsSearch;
...
MENU MainBorrower:
  MENUPAGE
    TITLE ("Borrower Commands");
    ("list borrowed copies",
       SWITCHENVIRONMENT (BorrowedBorrower)),
        ("search  book",  SWITCHENVIRONMENT
(Search)),
    ("restart", SWITCHENVIRONMENT (Start)),
    ("quit", QUIT);
  END MENUPAGE;
END MainBorrower;
...
SHEET ParmSearch:
  TITLE ("Set additional parameters");
  STRINGINPUT (RETURNONCRT, TITLE ("More
       Authors"), FORMAT(40));
  STRINGINPUT (RETURNONCRT, TITLE ("More
       Subjects"), FORMAT(40));
   SELECTION (TITLE ("Type"), "Any", "Book",
...)
   SELECTION (TITLE ("Language"), "English",
...)
     INTEGERINPUT  (RETURNONCRT,  TITLE
("Published
       After 19.."), FORMAT(2));
  STRINGINPUT (RETURNONCRT, TITLE ("Code"),
       FORMAT(8));
END ParmSearch;
...
LIST AuthorsSearch:
  TITLE ("Authors");
   ("Aho A.V."), ("Carlson E.D."), ("Clocksin
M.F."), ("Cox B.J."), ...;
END AuthorsSearch;
...
END LibInfoSystem.
```

Figure 7. UISL-code to generate the interfaces of Figure 6.

The M2/C Code Generator makes it possible to not just simulate procedure invocations, but, in the event of an implementation (in Modula-2 or C) to actually execute them. Thus a UICT prototype can be transformed in an evolutionary manner into the complete application.

UICT was developed on Sun Workstations and ported to Siemens PC 16-20 machines. Modula-2 was the implementation language. In order to be able to use the program components that were already implemented in C (particularly the Window Management System), a Modula-2 layer was wrapped around each C component. A Modula-2 environment that permitted the invocation of C programs was used for this purpose.

UICT is implemented so that various window systems and hardware can be used as its base. For this purpose, a virtual window management system was defined that had previously been implemented with the Siemens *S/Windows System*. A description of the implementation can be found in [15].

Dynamic Interface Creation Environment (DICE)

With DICE one can define a user interface by using solely graphical means, thus abandoning any textual input, and lifting restrictions of UISL:

- the interface can consist of any number of windows
- more interface items (see Figure 9) are provided

When the *"Edit Window"* button in DICE's Control Panel is pressed (see Figure 9), an empty window is opened. To insert user interface elements, one has to choose the proper element in the control panel and mark the position where the item is to be placed in the window.

Besides the usual interface elements (*Buttons*, *Text Fields*, *Popup Items* and scrollable *Subwindows*, as shown in Figure 9) there are two elements for grouping interface items—*Cluster* and *Expander*. They allow a comfortable specification of the window layout and of the behavior of the contained elements if a window is resized.

DICE provides undoable editing functions as known from state-of-the-art WYSIWYG editors: moving and resizing of the elements, cut/copy/paste between windows and different DICE prototypes, etc. The attributes of interface elements are defined in specific attribute sheets.

Since the behavior of a user interface is above all

determined by its dynamics, it is not enough just to describe screen layouts. DICE offers the following possibilities to portray the dynamic behavior of the user interface prototype:

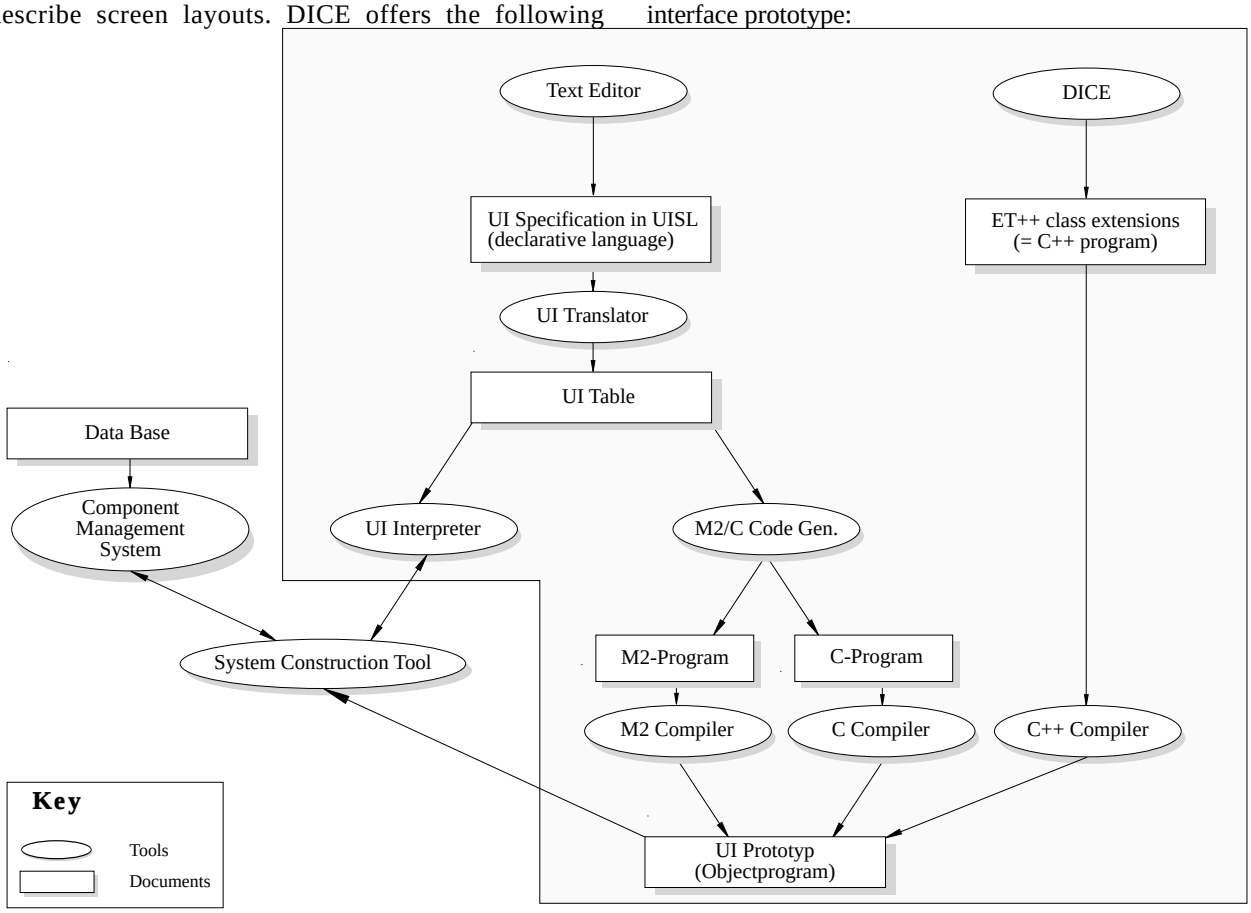

Figure 8. UICT system overview.

Figure 9. DICE's Control Panel.

- *Without programming*: Certain messages are assigned to each user interface element (e.g., the messages "Open" and "Close" to a window, the messages "Enable", "Disable" and "SetText(...)" to a text field, etc.). From each interface element that can be activated (Buttons and Menu items), any number of such predefined messages to other elements can be specified by using a special tool, the *Message Editor*. When the prototype is tested and an interface element has been activated, the messages specified for that element are sent to their receivers. The execution of these messages effects the corresponding change in the user interface. Thus rudimentary dynamics can be realized without programming effort.
- *With object-oriented programming*: For a specified prototype, C++ classes (ET++ subclasses—see [27]) can be generated. These classes can be enhanced with special functionality in an object-oriented way. This kind of code generation separates changes in the user interface from coded functionality as far as possible.
- *With conventional programming*: A protocol was developed that allows the prototype to be connected with other UNIX processes using the UNIX Interprocess Communication mechanism. Special functionality of the user interface can thus be enhanced with conventional programming.

4.4 The System Construction Tool

SCT (see [3, 4]) was developed to support a software development process as depicted in Figure 4. The following goals were pursued in implementing SCT:

1) development of a comfortable exploratory programming environment for a programming language with strong type checking
2) extension of exploratory programming towards exploratory designing
3) support of various programming paradigms[2] (e.g., modular, object-oriented, functional, and logic programming)
4) support of the prototyping-oriented incremental paradigm by providing a mechanism for execution of hybrid software systems consisting of Modula-2 code and high level parts written in other formalisms

[2] A discussion of the advantages of implementing software systems in various programming paradigms is beyond the scope of this article. Such discussions can be found in [14] and [15].

The first goal was attained by developing an *interpretive programming environment* for Modula-2 which provides most of the standard functionality known from exploratory programming environments such as Smalltalk-80. In order to provide satisfying execution speed during exploratory programming, a *hybrid execution mechanism* was realized allowing for interpretation of modules currently being implemented and for direct execution of implemented and tested modules.

To achieve the second goal a further execution mechanism, the Simulator, was added to the hybrid execution system. The Simulator makes it possible to verify a system architecture before all the corresponding software components are completely implemented or after changes to the architecture of an existing application were planned. This is done by *simulating the control and data flow* through the newly designed module interfaces while all existing parts of the application under development are interpreted or directly executed.

The third and the fourth goals were fulfilled by realizing a *flexible mechanism to add other execution tools to SCT's hybrid execution system*. Using this mechanism, a developer can easily add other interpretive execution tools in order to execute hybrid software systems written in various programming languages and high level formalisms.

After adding a Prolog interpreter, for example, it is possible to execute hybrid software systems consisting of parts written in Prolog and Modula-2. Another typical example is the integration of the interpreter of the User Interface Construction Tool with the hybrid execution mechanism of SCT.

We chose the following approach to present the individual components of SCT: First, the concept of hybrid execution of hybrid software systems is explained. Then the individual components of SCT are described, beginning with the Configuration Manager, which is the base of all other components.

The Concept of Hybrid Execution of Hybrid Software Systems

Hybrid execution of a software system means executing different components of a software system in different ways. Hybrid execution is no new concept, and many available tools provide hybrid execution mechanisms which combine interpretive and direct execution. Good examples are Lisp and Prolog development environments, e.g., [29], [16], as well as Saber-C [13], [23], [14]. Hybrid execution by

transparent integration of interpretation and direct execution results in fast execution and a comfortable development environment.

SCT goes one step further by providing simulation as an execution mode to complement interpretation and direct execution. While interpretation allows for short turnaround times during implementation and direct execution provides run-time efficiency, simulation makes it possible to dynamically validate a system architecture before all the corresponding components are completely implemented. This is done by simulating the control and data flow through newly designed modules for which only the interfaces exist.

Hybrid software systems are software systems consisting of parts written in different formalisms (languages). Three groups of tools supporting the development of such software systems can be distinguished today:

- Tools belonging to the first group provide a code generator which takes the description of a high level part as input and produces code in an algorithmic language. This code is then integrated manually with the other application parts. (For examples see [11], [8].)
- Tools in the second category provide an interpretive development environment for one high level formalism and a procedural interface allowing the connection of application parts developed in an algorithmic language with the interpreted high level parts. These tools require that the compiled application parts be linked to the execution tool. (For examples see [15] and [26].)
- Tools belonging to the third group provide various cooperating interpretive execution tools, one of them usually being an interpreter for an algorithmic language. (For examples see [1] and [10].)

Under the prototyping-oriented incremental paradigm high and low level parts dynamically evolve in parallel. In order to manage such an evolution process, it is important that the software system under development be immediately reexecutable after changes were applied to either high or low level parts.

This requirement can only be met by tools belonging to the third group, while the tools of the first and second group require several generation, compilation and integration steps before reexecution.

SCT belongs to the third group. The difference between SCT and other tools of this group is that the set of cooperating interpretive execution tools is not predefined or otherwise restricted.

The Configuration Manager

In order to facilitate the management of complex hybrid software systems during their development, a *configuration manager*, CM, was written which handles all information required to evolve hybrid software systems in the context of hybrid execution.

The following kinds of information are handled by CM:

- *documents* containing information about a software system (Modula-2 code, input formalisms for any kind of added execution tool, and any kind of documentation)
- *attributes* for classifying these documents
- *paths* indicating the location of the documents in the UNIX directory tree
- *object code libraries* needed to execute the software system
- information about added execution tools

While documents managed by CM are regular UNIX files, all unnecessary details about the file system are hidden from the user and the documents are presented in the form of a list.

Instead of ordering the documents in a directory tree, *attributes* are assigned to the *documents*. Every document can be described by any number of attributes and every attribute can be assigned to any number of documents. Based on these attributes, various kinds of selection mechanisms can be applied to obtain a list with a subset of the managed documents.

The advantage of the attribute-based organization over a directory tree is that documents can be organized in more than two dimensions and that it is possible to search the dimension best suited to finding the required documents. Projects, for example, can be logically organized into overlapping subprojects using attributes, which is obviously not possible with a directory tree.

While the developer works without bothering about the underlying file system, CM needs information to find the documents. For this reason it maintains an ordered list of *paths*. When system, user or project data is loaded into CM, every document is located by sequentially searching all directories indicated in the path list until a file with the name of the document is found.

UNIX applications frequently use functions provided by *object code libraries*. In order to execute such applications, CM manages an ordered list of object code libraries and files which are linked to SCT like the object code for directly executable modules.

Other execution tools (e.g., UICT) can be connected as server processes or they can be linked directly to SCT. CM provides several services that are beyond mere organization of other documents (e.g., documentation and input documents for added

execution tools).

When SCT is started, the configuration manager is the first tool which is activated, as depicted in Figure 10. The scrollable list at the right side displays all attributes, and the scrollable list in the middle all documents defined on system and user level.

Figure 10. The Configuration Manager's user interface.

The buttons on the left side serve to enter global commands which do not affect a selected module or attribute, and the buttons below the scrollable lists serve to select modules and attributes as well as to issue commands referring to selected documents.

The Implementor

An Implementor serves to edit and browse one module and to execute software systems which have this module as their root. Any number of implementors may be active. The only restriction is that only one Implementor can be used to start an execution. Implementors are created in the Configuration Manger by pressing the *Edit* or *Execute* buttons.

Figure 11 shows the user interface of an Implementor. It consists of a line of push buttons serving to control analysis, execution and debugging along the top and a comfortable text editor at the bottom.

Figure 11. Overview of an Implementor's user interface.

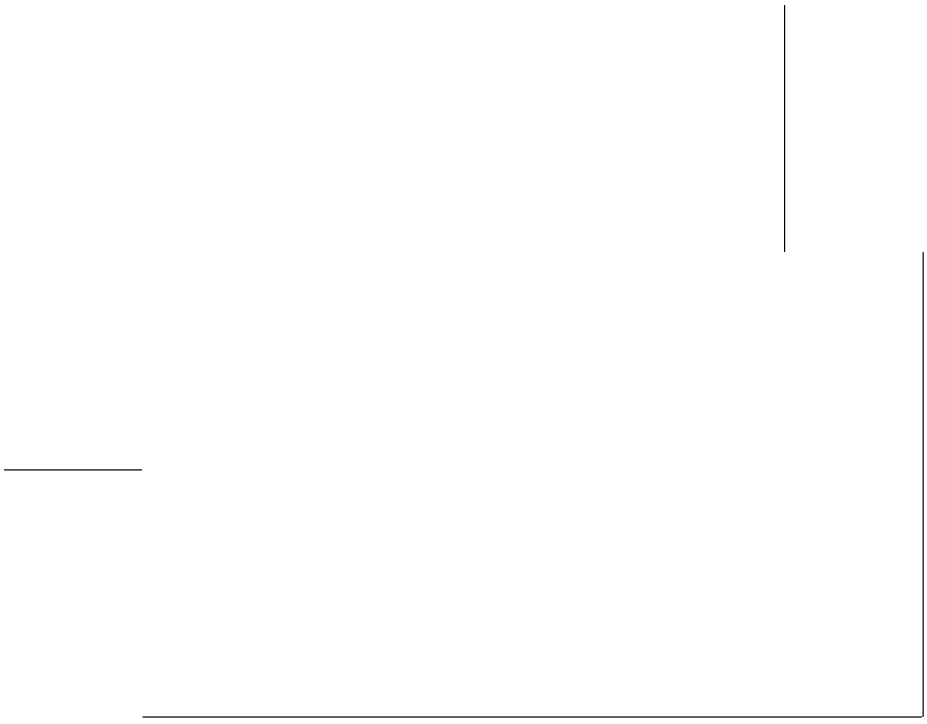

Figure 12. Browsers displaying the procedure call chain (left) and refined global run-time data (right).

Analysis and Execution. The Implementor carries out a threefold analysis of modules: it checks their syntactical correctness, extracts data which can be used for browsing them, and prepares them for execution.

Before a module can be executed, the implementation parts of all interpretable modules from which

something is imported have to be analyzed because a procedure imported from one of them could be called. Furthermore, if a developer wants to use the browsing tools to get information about implementation parts of directly executable modules, their implementation parts have to be analyzed, too.

The execution of interpretable and directly executable modules can be triggered by pressing the execute button. Execution monitoring is supported by the possibility to set breakpoints and by stepwise execution.

To support exploratory programming, the Browser tools make it possible to get various information about the program system (e.g., the procedure call chain, complex data structures, import/export dependencies, constants, etc.) and to edit the run-time data (see Figure 12).

The Simulator

The Simulator can be used to *execute software systems which are designed but not implemented*, i.e, for which only the definition modules are written. Such an execution mode is useful to validate (prototype) system architectures by playing with realistic scenarios. These scenarios can be stored in history files and replayed after a change to the system architecture was applied in order to revalidate it. The Simulator's user interface is shown in Figure 13.

From a very abstract point of view, the Simulator provides only two features. First, it makes it possible to start an execution by manually calling any procedure currently known to SCT. Second, it allows for execution of simulable procedures.

In order to simulate a software system, two prerequisites have to be fulfilled. First, it must be possible to simulate a procedure call. In the Simulator this is done by selecting the procedure to be called in the *Procedures* subwindow, instantiating its local data with the *Activate* command, editing the input data and dispatching the call with the *Execute* command. Second, it must be possible to return from a simulated procedure call. In the Simulator this is done by issuing the *Return* command in the *Procedure Call Chain* subwindow.

Simulation is supported by interrupting the execution as soon as a simulable procedure is called. During such an interrupt the Simulator displays the parameters of the simulated procedure, the currently active procedure call chain, and the names of all procedures which were already called from the current

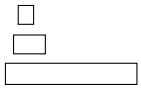

Figure 13. The Simulator's user interface.

procedure in the course of the simulation. During the simulation of a procedure it is possible to simulate procedure calls carried out by it (by applying the Simulator's first feature), to browse and edit its local

data, and to return from it.

All activities related to the current simulation and/or testing activities are logged by the Simulator. At the end of a simulation this information can be saved in a history file, which makes it possible to replay the simulation later.

The Simulator supports a developer in carrying out many different tasks during software development.

The Simulator's procedure call mechanism can be used to test modules and application parts of any degree of completion. In order to test a module, the developer calls its interface procedures repeatedly with different kinds of input parameters and he checks the output parameters for correctness. All test cases for a module can then be stored in a history file and replayed after the module was changed. In the replay mode the Simulator compares the expected with the effective output parameters, making it possible to retest modules without much effort.

During testing the Workspace Manager tool described in the next section comes in handy because the complex data structures which are sometimes needed as test data can be managed and do not have to be reconstructed for every test run.

Another possible application of the Simulator is the *experimentation with system libraries, reusable code, and interpretive high level tools*. This kind of experimentation is especially important during exploratory programming and the development of hybrid software systems. The Simulator supports such experimentation in two ways. First, it enables a developer to test any single feature immediately with the procedure call mechanism. Second, it supports him in connecting libraries, interpretive execution tools and any other kind of existing code by simulating the connecting parts without the need to write a single statement.

The user interface of the Simulator, depicted in Figure 13, consists of three groups of push buttons aligned along the top and six subwindows filling the rest of the window.

The three subwindows aligned along the left side serve to browse through modules, the procedures defined in them and the parameters of these procedures. The *Modules* subwindow contains a list of all currently available modules. If a module is selected, a list of all procedures defined in this module is displayed in the *Procedures* subwindow. Finally, the selection of a procedure results in the displaying of its interface in the *Parameters* subwindow.

The three subwindows aligned along the right side serve to display data related to the current execution. The *Local Data* subwindow shows the parameters of the currently simulated procedure. The *Called Procedures* subwindow displays a list of all procedures which have already been called from the current procedure during the current simulation. The *Procedure Call Chain* subwindow displays the list of the currently active simulated and interpreted procedures.

The Workspace Manager

Using SCT, it is possible to execute parts of an application, to provide them with input parameters and to inspect the output parameters. Furthermore, it is possible to edit complex data structures during a break of an active execution and to manipulate complex data structures with *Copy* and *Paste* commands. In such an environment it would be useful to have some kind of workspace into which data structures can be stored (pasted) if the developer is interested in keeping them after the current execution is terminated. The Workspace Manager provides such a workspace.

Most interpretive programming environments provide only one global workspace where all global data is stored. This is all a developer needs if he uses the workspace to store output parameters of application parts for later use. Besides one global workspace, SCT also provides one workspace for every simulated module. These workspaces are used to simulate the global data of the corresponding modules, and they can be accessed from the simulator.

The user interface of the workspace manager is very similar to the user interface of the Browser. We do not include an illustration.

Adding Other Execution Tools

In order to integrate interpretive execution tools into a hybrid execution mechanism, four prerequisites have to be fulfilled:

1) The formalism executed by the interpretive execution tool has to provide some mechanism to define the cooperation with other formalisms.
2) The interpretive execution tool has to be an open system. This means that it has to provide an interface which can be used to receive requests for certain services from other tools. In addition many interpretive execution tools have an interface which allows them to request certain services from other execution tools.
3) The interfaces of the various tools must match semantically and the formalisms in which they are formulated must be compatible so that meaningful synchronization and an exchange of data is possible.
4) A common platform has to exist which makes it possible to integrate the interfaces provided by the different tools and the tools themselves.

SCT's hybrid Modula-2 execution system is a common

platform for integrating any kind of interpretive execution tools running under UNIX.

In order to integrate an interpretive execution tool into SCT's hybrid execution system, its interface for communicating with other tools has to be added to the current application. This interface enables other tools and low level application parts to send requests to the newly added tool using the hybrid procedure call mechanism. If a request has to be dynamically bound, this can be achieved by using SCT's ProcCall-Dispatcher interface, which provides dynamic binding.

Implementation. Topos was implemented on Sun Workstations using Modula-2, C and C++ as implementation languages. The user interface was built with ET++, an object-oriented application framework implemented in C++. A description of the most interesting implementation aspects can be found in [4].

5. Conclusions and Future Directions

We have been using UICT since spring 1987 and SCT since fall 1989 in various projects at our institute. In this time we developed several small to medium-sized prototypes. Both tools have also been applied in pilot projects by Siemens Munich AG.

The feasibility and usefulness of hybrid execution and incremental development of hybrid software systems was verified by integrating SCT with UICT, a Prolog interpreter, and a relational database system. With the resulting hybrid development environment we were able to develop prototypes of impressive functionality in a few days.

We are currently doing research work in the area of the synthesis of object-oriented programming and a prototyping-oriented development approach. As such, the extension of Topos with tools that support object-oriented programming is underway.

References

1. Acius Inc. (1987) Documentation of 4th Dimension.
2. Bischofberger WR, Keller RK (1989) Enhancing the Software Life Cycle by Prototyping. Structured Programming, Vol. 10, No. 1
3. Bischofberger WR, Pomberger G (1989) SCT: a Tool for Hybrid Execution of Hybrid Software Systems. in Proceedings of the First Conference on Modula-2, Bled, Yugoslavia
4. Bischofberger WR (1990) Prototyping-Oriented Incremental Software Development—Paradigms, Methods, Tools and Implications. Doctoral Dissertation, Johannes Kepler University of Linz, Austria
5. Boar B (1983) Application Prototyping: A Requirements Definition Strategy for the 80s. John Wiley
6. Bobrow DG (1985) If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms. IEEE Transactions on Software Engineering, Vol. 11, No. 11
7. Connell JL, Shafer LB (1989) Structured Rapid Prototyping. Prentice Hall International, Yourdon Press
8. Cossey GR (1987) Prototyper; Users Guide. SmethersBarnes Inc, Portland, Or.
9. Floyd Ch (1984) A Systematic Look at Prototyping. In Approaches to Prototyping, Springer-Verlag
10. Goodman D (1988) Danny Goodman's HyperCard Developer's Guide. Bantam Books
11. Hulot, J-M (1987) Exper Interface Builder. User's Guide
12. Jörgensen AH (1984) On the Psychology of Prototyping. In Approaches to Prototyping, Springer-Verlag
13. Kaufer S, Lopez R, Pratap S (1988) Saber-C, An Interpreter-Based Programming Environment for the C Language. In Proceedings of USENIX, San Francisco
14. Kaufer S, Kendall S, Mullings H (1989) Saber-C: The Programmer's Edge. Sun Technology, Summer 1989
15. Keller RK (1989) Prototypingorientierte Systemspezifiaktion—Konzepte, Methoden, Werkzeuge und Konsequenzen. Verlag Dr. Kovac (in German)
16. LPA Ltd. (1987) LPA MacPROLOG: Reference Manual
17. NeXT Inc. (1989) Technical Documentation: Reference. NeXT Inc. Redwood City, CA
18. Pomberger G (1986) Software Engineering and Modula-2, Prentice Hall International
19. Pomberger G (1988) Integration von Prototyping in Software-Entwicklungsumgebungen. In Anleitung zu einer praxisorientierten Software-Entwicklungsumgebung. Verlag Angewandte Informations Technik (in German)
20. Pomberger G (1989) Methodik der Software-Entwicklung. In Handbuch der Wirschaftsinformatik (Kurbel, Strunz ed.), Poeschl Verlag (in German)
21. Pree W (1990) DICE—An Object-Oriented Tool for Rapid Prototyping. In Proceedings of TOOLS Pacific '90, Sydney
22. Rechenberg P, Mössenböck H (1989) A Compiler-Generator for Microcomputers. Prentice-Hall International
23. Saber Inc. (1988) Saber-C Enhances Development for C Programmers. The Sun Observer, Vol. 1, No. 4
24. Schmidt D, Pomberger G, Bauknecht K (1989) The Topos Component Management System. Institutsbericht Nr.89.08, Institut für Informatik der Universität Zürich
25. Stritzinger A (1991) Smallkit—A Slim Application Framework. To be published in the Journal of Object Oriented Programming
26. SUN Inc. (1986) SUN Unify Documentation
27. Weinand R, Gamma E, Marty R (1988) ET++ — An Object Oriented Application Framework in C++. OOPSLA 88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11
28. Wilson DA, Rosenstein LS, Shafer D (1990) Programming with MacApp. Addison-Wesley
29. Xerox Corporation (1983) Interlisp Manuals
30. Zave P (1989) A Compositional Approach to Multiparadigm Programming. IEEE Software, September 1989