SDA: A Novel Approach to Software Environment Design and Construction†

Kouichi Kishida†, Takuya Katayama†, Masatoshi Matsuo†, Isao Miyamoto††, Koichiro Ochimizu†*, Nobuo Saito‡‡, John H. Sayler†††, Koji Torii****, Lloyd G. Williams*****

Abstract

A Software Designer’s Associate (SDA) is a workstation-based collection of tools which support: 1) the description, evaluation and comparison of software system architectural designs, and 2) cooperation among, and management of, a team of software designers [Ridd87]. Each Software Designer’s Associate is a specific instance of a generic facility which supports a team member’s design activities, cooperation among team members, and overall team management. It provides a framework for the integration of tools supporting the use of various notations within the context of a particular set of technical and managerial methods. These tools, notations and methods may be adapted to support the needs of a particular project or the habits of an individual developer by selecting the particular tools to be added to the generic facility.

The Software Designer’s Associate project is a joint effort involving a consortium of researchers from academic and industrial organizations in both Japan and the United States. This paper describes the concept of Software Designer’s Associates and the cooperative, international project which is intended to lead to the realization of that concept.

1. Introduction

The past few years have seen an increasing interest in software environments (see e.g., [PSDE1] and [PSDE2]). A number of projects whose goal is the construction of a prototype advanced software development environment have recently appeared. The emergence of new technologies, particularly knowledge-based approaches, object management systems and sophisticated user interfaces has helped to motivate this interest and several projects incorporate one or more of these technologies. This paper describes one such effort, the Software Designer’s Associate project.

A Software Designer’s Associate (SDA) is a workstation-based collection of tools which support: 1) the description, evaluation and comparison of software system architectural designs, and 2) cooperation among, and management of, a team of software designers [Ridd87]. Each Software Designer’s Associate is a specific instance of a generic facility which supports a team member’s design activities, cooperation among team members, and overall team management. It provides a framework for the integration of tools supporting the use of various notations within the context of a particular set of technical and managerial methods. These tools, notations and methods may be adapted to support the needs of a particular project or the habits of an individual developer by selecting the particular tools to be added to the generic facility.

A Software Designer’s Associate may be viewed as a specialized software environment, one whose facilities are restricted to support for preliminary design activities. Each instance of a Software Designer’s Associate is expected to be part of a network of workstations which support a variety of software development activities. The other workstations in the network may also be Software Designer’s Associates or they may be workstations supporting other software creation and evolution activities, such as coding, testing or maintenance. A Software Designer’s Associate will, therefore, be part of an overall software environment whose capabilities are distributed over a network of workstations and individuals. In addition, since a given network may contain more than one instance of a Software Designer’s Associate, it is likely that design activities will also involve a team of individuals whose activities are distributed over several workstations.

The Software Designer’s Associate project is unique in several respects. First, rather than redesigning an environment and its associated tools from scratch, we are relying on the fact that many of the tools needed to populate a Software Designer’s Associate already exist and more design-support tools are on the near-term technology horizon [Ridd87]. Thus, the principal focus in this project is to provide a host environment which supports the consolidation and integration of existing tools.

Second, the scope of the Software Designer’s Associate is restricted to the preliminary design phase. This restriction is...
contrary to the current trend of environments which support all phases of software creation and evolution. It reflects our view of a software environment as composed of several components with facilities for coordinating activities which take place on those components. History has shown that focusing on a small subset of related software creation and evolution activities makes it possible to provide more sharply focused support (both in software and hardware) for those activities. Thus, by creating an environment composed of such components, we can provide high-quality support which is still comprehensive in scope. The Software Designer's Associate represents one such component. As noted above, an SDA is expected to be part of a network of workstations which support a full range of software development activities.

Finally, the Software Designer's Associate project is a joint effort involving a consortium of researchers from academic and industrial organizations in both Japan and the United States. Researchers from Japan represent Keio University, Osaka University, Shizuoka University, and the Tokyo Institute of Technology. United States researchers represent the University of Colorado at Denver, the University of Hawaii, and the University of Michigan. With sponsorship from seven Japanese companies, a special, non-profit organization, the SDA Consortium, has been established to provide funding and other support for this project. Work on the project is being coordinated by Software Research Associates, Inc., Japan. Other consortium-based projects (e.g., [Tay186]) have reported a synergistic effect which we have also observed. To our knowledge, however, this is the only such project involving cooperation between Japanese and American researchers.

Realization of the Software Designer's Associate concept is to be accomplished on a three-year schedule which involves conceptual definition, prototyping and evaluation and, finally, full-scale implementation of the Software Designer's Associate.

This paper describes the concept of Software Designer's Associates and the cooperative, international project which is intended to lead to the realization of that concept. We begin by describing the requirements for a Software Designer's Associate. Section 3 describes the three-year project plan. Section 4 introduces the model-based approach to environment integration which is being used as a basis for defining a Software Designer's Associate and Section 5 describes the details of the conceptual models which will be used. In Section 6 we describe several prototyping efforts which are intended to explore and elaborate various aspects of the implementation of Software Designer's Associates. Finally, Section 7 describes some related projects.

2. Requirements for a Software Designer's Associate

The basic, high-level requirements for a Software Designer's Associate were established in [Ridd87]. First, an SDA must provide support for the description, evaluation and comparison of software system architectural designs. Description of architectural designs will require support for the use of several types of languages, including definitional languages, structural languages and behavioral languages [Ridd87]. Evaluation of architectural designs requires the capability to assess design properties such as functionality, portability or complexity, as well as overall system characteristics which are influenced by the design, such as suitability (possibly assessed through the use of prototypes or cost/benefit (possibly aided by estimation tools such as COCOMO [Boeh81]). Comparison of alternative designs requires these evaluation capabilities, together with additional support for analytical comparison as well as empirical studies.

Second, an SDA must support cooperation among the individuals who make up the design team as well as management of their activities. Since members of the design team may be distributed across a network of workstations (and at different locations), an SDA will need to support distributed communication as well as (coordinated) access to design objects. Management capabilities include support for project estimation, planning and scheduling, together with feedback capabilities to allow monitoring of progress and comparison with original projections.

Finally, an SDA must support adaptation: the creation of a specific instance of a Software Designer's Associate which is tailored to a particular organization, project or individual. Adaptation requires the ability to consolidate and integrate existing technologies so that they may be used, both individually and in combination, from a single environment. Consolidation and integration will, in most cases, require some modification of the tool and/or its internal and external interfaces.

3. Project Plan

The three-year goal of this project is to produce a working version of a Software Designer's Associate. To achieve this goal, the project is divided into three phases of approximately one year each. The approach during the first year has been "top-down," consolidating and amalgamating existing technology. Products of the first year's activities include a specification and architectural design for the Software Designer's Associate as well as some preliminary prototyping to resolve difficult issues and demonstrate the feasibility of high-risk implementation concepts.

During the second year, the prototyping effort will be expanded to more fully elaborate the specification and preliminary design developed during the first year. It is likely that alternative versions of some portions of the system will be prototyped. These alternative implementations, as well as the overall prototype itself, will be compared and evaluated for their effectiveness in meeting the requirements of Software Designer's Associates. This approach may be characterized as "bottom-up" with guidance provided by the specification and design developed during the first year.

The third year will be characterized by a return to a "top-down" orientation. The information gathered during the development and evaluation of prototypes during the second year will be used to guide a redesign of the Software Designer's Associate. This new design will then define the final, full-scale, implementation.
4. Model-Based Integration

Many of the tools needed to populate a Software Designer's Associate already exist and more design-support tools are expected to become available in the near future [Ridd87]. Because of the current and near-term availability of such tools, the focus of this project is on the development of technology to aid in the consolidation and integration of existing tools rather than on the development of new tools.

A central issue in developing a Software Designer's Associate is, therefore, that of integration. External integration in the Software Designer's Associate is to be achieved through the development of three conceptual models:

- a software process conceptual model that defines the characteristics common to a variety of software development and evolution processes
- a software products conceptual model that defines the organization and content of the designs, documentation, code, etc. and the relationships among these products and
- a tool collection conceptual model that provides a user-oriented view of the set of tools.

These conceptual models unify the facilities provided by the tool collection by describing how those tools contribute to developing the various products required during the course of a software project.

Internal integration will be achieved by developing a corresponding set of semantic models which describe interactions among the products, the process and the tool collection. The semantic models are:

- an analysis semantic model that describes the feedback capabilities provided by the SDA,
- a tool interaction semantic model that describes interactions among the tools and the rules governing their use, and
- a substrate semantic model that defines, among other things, the dynamics of the information repository, the tool invocation control mechanisms and the adaptation mechanisms for SDAs.

The semantic models contribute to establishing a common base for the tools by describing the information they share, the interfaces among them, and the ways that they interact.

The implementation of these semantic models will, in turn, be defined through the development of three realization models:

- an analysis management system model that describes the conduct of informal and formal experiments to analyze or assess designs
- an object management system model that describes the information repository and tool invocation control facilities provided by the substrate and

Figure 1: Relationships Among Conceptual, Semantic and Realization Models (adapted from [Ridd87])
In this figure, the bold arrows indicate that there are, in general, influences among each of the sets of models and the lighter arrows indicate particularly strong relationships among certain pairs of models.

5. Conceptual Models

As described in Section 4, integration in a Software Designer's Associate is to be achieved using a set of models which describe, at several levels, the software process, its products and the tools used. To date, the conceptual models have been most thoroughly developed and these are presented here.

5.1 Software Products Conceptual Model

The main role of the software products conceptual model is to define: the collection of artifacts associated with the software process, the information about these artifacts that is required during the software process, and the relationships which exist among the artifacts and among the associated information.

To accomplish these goals, the software products conceptual model must:

- Provide for the description and analysis of partial products, and provide appropriate views of these descriptions and analyses for the different potential audiences who will use the Software Designer's Associate.
- Provide a view of the product's use in the target (or development) system, and its role in the development process.
- Provide for adaptability, both on the macro and the micro level.
- Be extensible in order to include new types of tools, methods or product fragments.
- Provide for reusability.

The nature of each of these models and their roles in developing a Software Designer's Associate are described more fully in [Ridd87]. The models exhibit a high degree of interdependence, as shown in Figure 1. In this figure, the bold arrows indicate that there are, in general, influences among each of the sets of models and the lighter arrows indicate particularly strong relationships among certain pairs of models.

### Table 1: Software Products Conceptual Model

<table>
<thead>
<tr>
<th>N</th>
<th>QV = (QV1, QV2, ..., QVn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(QV1) x P(QV2) x ... x P(QVn) ⊇ Q</td>
<td>the set of state variables of O</td>
</tr>
<tr>
<td>AV = (AV1, AV2, ..., AVm)</td>
<td>the set of attributes of O</td>
</tr>
<tr>
<td>P(AV1) x P(AV2) x ... x P(AVm) ⊇ A</td>
<td>the set of attributes of O</td>
</tr>
</tbody>
</table>

5.1.1 Software Products Conceptual Model Description

The software products of an SDA are modeled as abstract objects. Each abstract object represents a discrete product of the software process. The concrete products are technical, project or managerial artifacts and documentation. Example technical artifacts include specifications, design representations, and test plans. In addition, software products include tools and tool scripts. Example project artifacts include standards and procedures. Example managerial artifacts include plans, schedules, work breakdown structures, PERT charts, cost estimations, and analysis results.

Formally, then, a software product, O, is a 5-tuple <N, QV, Q, AV, A> as shown in Table 1.

Each object, or software product, is defined by a name, a set of state variables, a set of states (relations over the state variables), a set of attribute variables, and a set of attributes (relations over the attribute variables). State variables will generally be used to represent the status of objects during design, while attribute variables will generally be used to express information about the use of an object in a design. Associated with each of these variables is a set of values.

An example of a state variable, QV1, is the degree of completion or specification of an object. A possible set of values for QV1 is (not-begun, experimental, finished). An example of an attribute variable, AV1, is the role that an object plays in a design. A set of values associated with AV1 might be (documentation object, design fragment object, analysis object). Attribute variables will also be used to capture process and tool related information, for example, which tool produced the object, through which method. Thus, a typical set of attribute variables for an object might include: user identification, ancestor and descendent objects, object version or revision, associated processes, and tool(s), and accessor information.

A state of an object is a relation over the state variables of the object, many of which may not participate. Moreover, some states comprise subsets of values of one or more state variables. States may be used to model computations, as in the TOPD [Hend75] and DREAM [Ridd78] modeling systems. Attributes (of an object) are identical to states, except that they are relations over the attribute variables of the object.

To illustrate the notion of object states, consider an object named MODULE_M with two state variables, QV1 and QV2, representing the documentation status and coding status of MODULE_M, respectively. Let (not-begun, active, finished).
official) be the set of values for QV1 and let (not begun, active, finished, tested) be the set of values for QV2. Then the set of states of MODULE_M could include:

- NOT_BEGUN, which reflects both state variables having the value "not begun;"
- ACTIVE, which is reached if either state variable has the value "active;" 
- FINISHED, which reflects both state variables having the value "finished;" and
- BASELINED, which is similar to FINISHED, except that QV1 also has the value "official" and QV2 also has the value "tested."

Some, or all, of the states of an object may be hidden from one or more users of the SDA. For example, a manager may only be interested in whether MODULE_M is in the NOT_BEGUN or BASELINED states. In addition, there may exist visibility restrictions on some of the state variables themselves.

5.2 Software Process Conceptual Model

The principal role of the software process conceptual model is to provide a basis for unification of the tool collection's facilities. It performs this role by describing how the tools contribute to developing the products which are produced during a software creation and evolution effort. Since the Software Designer's Associate is to be a generic facility which can be adapted to specific needs and/or projects, the software process conceptual model must define characteristics common to a variety of software creation and evolution processes.

To achieve these goals, the software process conceptual model must:

- Support the definition of a set of tools to populate a Software Designer's Associate.
- Provide for independence between tools and details of specific software methods.
- Incorporate a distributed, team-based view of software creation and evolution.
- Reflect a broad range of current software methods.

These characteristics help to define a generic software process model. A particular software development process will be an instance of the software process conceptual model, tailored to the needs of the product being developed, the organization, and the individual developer.

5.2.1 Software Process Model Description

The software process model described here focuses on development-in-the-large [Ridd86]. Development-in-the-large is analogous to programming-in-the-large in that it focuses on the various activities associated with software development and their interconnection. These activities (e.g., design, development of a prototype, or testing) are typically carried out by teams of individuals and they often take place concurrently and asynchronously. They will, in general, be performed by different individuals who may be distributed over different locations and/or different hardware configurations. Communication and synchronization between these processes is accomplished via message-passing; the nature and content of the messages is determined by the activities and the needs of the developers.

The approach advocated here is one of behavioral, rather than procedural, modeling and, thus, differs from process programming [Oste86], [Oste87]. We describe the software process in terms of abstractions of software creation and evolution activities. This approach is described as behavioral because the descriptions of activities focus on the effects which the activities produce rather than the specific procedures (or algorithms) used to produce those effects, i.e., what an activity is supposed to do rather than how it is to be done. This allows describing software development at any desired level of abstraction and makes it possible to rigorously model activities, such as design, whose details are poorly understood. The use of behavioral descriptions to describe the software process is similar to the the behavioral modeling of software systems found in the DREAM system [Ridd78].

The software process model (SPM) describes software development as a set of activities:

\[ SPM = \{\text{activity}\} \]

where an activity is defined to be a 4-tuple consisting of a set of preconditions, an action, a set of postconditions, and a set of messages:

\[ \text{activity} = (\{\text{precondition}\}, \text{action}, \{\text{postcondition}\}, \{\text{message}\}) \]

Activities may be performed by different individuals and at different locations. Activities which are not constrained to be sequential by their pre/postconditions may take place concurrently.

An action is a software development task which may be performed by invoking a particular tool or sequence of tools or may be carried out by a developer without automated assistance. An action might result in the creation or transformation of one or more software objects.

Activities may be hierarchically organized by building complex activities from simpler ones. This allows modeling the software process at whatever level of abstraction is appropriate for the purpose at hand. Thus, we may combine several activities to produce a composite activity:

\[ \text{composite activity} := <\text{activity}> | <\text{activity}> <\text{composite activity}> \]

Note that this definition requires the activities which comprise composite activities to be performed sequentially. If two or more activities can take place in parallel, they are not considered to be composable. Composite activities are provided as a convenience for describing higher-level activities which can be performed as sequences of lower-level activities (e.g., an edit-compile-debug cycle). The preconditions for a composite activity are those which are defined for the initial activity in the sequence. The postconditions for a composite activity are formed from the union of the postcondition for the last activity in the sequence with any unresolved postconditions for intermediate activities.

Since higher-level activities may also consist of lower-level activities which are performed in parallel, we introduce complex activities:

\[ \text{complex activity} := <\text{activity}> | <\text{activity}> | <\text{activity}> \Delta <\text{complex activity}> \]

where the \( \Delta \) ("shuffle") operator represents the interleaving of concurrent activities. An example of a complex activity might be

\[ (\text{activity 1}, \text{activity 2}, \text{activity 3}) \]

2 If activity A1 consists of two subactivities (ab) and A2 consists of (cd) then the set of behaviors described by shuffling A1 and A2 (A1 \( \Delta \) A2) is (abcd, abcd, abcd, abcd, abcd).
be a preliminary design step which includes development of an
architectural design, a preliminary user manual and initial test
plan, all performed in parallel. The preconditions for a
complex activity are formed from the union of the
preconditions for each individual activity. The postconditions
of a complex activity are the union of the postconditions for
each component.

The use of preconditions and postconditions here follows that
in the Inscape Environment [Perr87] and is also similar to that
used in MARVEL [Kais87]; they are used as points of
interconnection between software development activities.
Each precondition or postcondition represents a fact about the
activity or the software process as a whole. As with Inscape,
while our use of preconditions and postconditions resembles
Hoare's input/output predicates [Hoar69], their interpretation
is significantly different. In particular, preconditions and
postconditions are not used to prove particular properties of
the software process. Instead, they represent assertions about
the properties of various activities (as opposed to properties of
objects, as in Hoare semantics) and the interconnections
among those activities.

In both Inscape and MARVEL, preconditions and
postconditions are associated only with those actions which
are performed by tool. Here, we relax that restriction to
allow modeling of activities which are performed by human
developers. Thus, preconditions and postconditions may exist
at several levels of formality and/or restrictiveness. An
individual developer may, for example, perform a sequence of
activities in which some (or all) of the preconditions or
postconditions (e.g., consistency among design objects or
completeness of a particular design description) are
temporarily unsatisfied. When that individual's work is
integrated with the work of others, however, all
postconditions associated with that sequence of activities
would typically be required to be satisfied.

An action may have several preconditions, all of which must
be satisfied, to the appropriate level of formality, before that
action can be performed. Following the approach used in
MARVEL, an action may also have several mutually exclusive
postconditions, only one of which will be satisfied when the
task is completed. The choice of which postcondition is
satisfied will depend on the result produced by the action. For
example, a design element may be subject to a consistency
check. If the check succeeds, the postcondition will be
"consistent;" if it fails, the postcondition will be "error." The
existence of error conditions provides a means of backtracking
or 'exception handling' for activities or groups of activities.

As noted above, messages provide a means of communication
and synchronization among various activities. They also
provide a visibility mechanism for project management in that
messages can be associated with project milestones. A
message may be simple, (e.g., notification that an activity has
completed) or complex (e.g., a software object which must be
communicated from one activity to another).

Following Wile and Allard [Wile87], we define a message to
consist of two parts: the message object itself, and a message
event.

$$\text{message} = (\text{message\_body}, \text{message\_event})$$

A message event specifies the sender and receiver(s) of the
message, as well as the times at which the message was sent and
received.

Separation of messages into a message body and a message
event makes it possible to explicitly model the action
of sending or receiving a message, an important feature for
team coordination and project management.

The software process model describes software development
as a set of activities. Specification of the activities to be
included and the order in which they are to be performed
defines a particular software process or method. The set of
possible activities forms an "alphabet" for software process
description and describes a class of software creation and
evolution activities (e.g., a specification-based versus
prototyping approach). A particular process is defined using a
language (the process description language) over this alphabet.
A given development effort would correspond to an individual
string (a particular sequence of activities) in this language.

In practice, a software process (or a software method) is
defined by specifying the activities to be performed, including
their preconditions and postconditions. The pre- and
postconditions are derived from the rules for using the target
method. They determine the order in which activities may be
performed. A more extensive discussion of this model,
together with examples of its application to software process
and method description appears in [Wili88].

5.3 Tool Collection Conceptual Model

The purpose of the tool collection conceptual model is to
provide a user-oriented view of the set of tools [Ridd87]. This
model supports the definition of the logical parts of the tool
collection as well as the facilities that each part should provide.
Identification of the logical parts of the tool collection and their
required facilities will make it possible to identify components
which are common to the tool collections that make up various
instances of Software Designer's Associates and assist in the
definition of the Software Designer's Associate substrate.

The tool collection conceptual model must, therefore:

- Emphasize the identification of a substrate for Software
  Designer's Associates.
- Assist in defining patterns of tool usage and relating
  them to patterns of software creation and evolution
  activities.
- Help define natural information structures and tool
  interfaces.
- Assist in defining the tool collection's logical parts and
  the facilities that each part should provide.

The particular tools which populate the Software Designer's
Associate will be defined by the software process and software
products conceptual models. The tool collection conceptual
model provides a general framework in which to embed those
specific tools. By keeping this framework (i.e., the
information structures and tool interfaces) independent of any
particular method it will be easier to identify tools which are
applicable to a variety of methods (and should, therefore, be
included in the substrate) and to integrate tools which are
specific to a particular method into the general framework.

5.3.1 Tool Collection Conceptual Model

Description

The tool collection conceptual model presented here is based
on the division of tool functions into logical functions and
specific instances. Thus, a tool collection conceptual model (TCM) is described as a set of tool functions:

\[ \text{TCM} = \{ \text{tool Function} \} \]

where a tool function is defined as

\[ \text{tool Function} = \{ \text{logical function}, \{ \text{instance} \} \} \]

Logical tool functions include capabilities such as editing or compiling while instances are used to identify particular realizations of those capabilities. Attaching a set of instances to a given logical function indicates that there may be more than one tool corresponding to a given logical function. Thus, for a logical function such as editing, one might choose as instances any one of several text editors (e.g., vi or emacs) or another appropriate editor, such as a data-flow diagram editor. Or, for compilation, one might choose instances such as a C compiler or an Ada™ compiler.

A particular tool instance is described by specifying its \[ \text{input object type(s)}, \text{the tool itself, and the output object type(s)}. \]

\[ \text{instance} = \{ \{ \text{input object type}, \text{tool}, \text{output object type} \} \} \]

In addition to the ability to invoke individual tools, users of a Software Designer's Associate should be able to compose or combine tools to synthesize overall tool functions, carry out more complex tasks, or to relate individual tool functions to overall software creation and evolution activities. Thus, as with activities, tool functions may be composite or complex. These are defined as follows:

\[ \text{composite function} := \langle \text{tool function} \rangle \]

\[ \text{complex function} := \langle \text{tool function} \rangle \{ \text{tool function} \} \]

As with the process conceptual model, the set of all possible tool functions forms an "alphabet" for describing composite or complex tool functions or their combinations. A given function is defined using a language (the tool composition language) over this alphabet. The particular function corresponds to a string in the tool composition language. Composite and complex tool functions help to define "scripts" for assembling tools from tool fragments or creating high-level functions from simple tools.

5.3.2 Classes of Tools

A Software Designer's Associate actually includes two classes of tools: design/management tools and integration/adaptation tools. Design/management tools are used to produce software designs and related objects and to coordinate and manage team activities. General design tools are tools which are useful across several different software development methods and organizational environments (e.g., a text editor and formatter or electronic mail). Specific design tools support a particular software method or organizational environment (e.g., a JSD editor or COCOMO). Integration/adaptation tools are those tools which are used to produce a specific instance of a Software Designer's Associate. These tools include capabilities such as a tool interface generator or tool-building tools (e.g., lex and yacc).

General design tools and integration/adaptation tools are part of a generic Software Designer's Associate and are included in the substrate. Specific design tools are added (with the assistance of integration/adaptation tools) to create a specific instance of a Software Designer's Associate.

6. Prototyping

Prototyping is an especially valuable tool in cases where the requirements for a system are uncertain or where there are significant risks attending the implementation of one or more parts of the system. In developing an environment such as a Software Designer's Associate, there are uncertainties in the requirements for supporting a variety of software methods as well as in the requirements for a user interface. Since the focus of this project is on the integration of existing, state-of-the-art tools rather than on creation of new tools, there is also significant risk associated with developing a system which allows straightforward integration of a variety of tools which, in general, will have been implemented for different environments.

Thus, as described in Section 3, prototyping will play a central role in the development of the Software Designer's Associate as a vehicle for clarifying requirements and reducing risk. In this section we describe the objectives of Software Designer's Associate prototyping efforts and discuss several current prototyping projects. These prototyping projects are being carried out at different institutions by researchers with widely different backgrounds. This diversity makes it possible to approach various aspects of the Software Designer's Associate in an experimental fashion, comparing several different points of view.

While each of the prototyping activities described below explores certain aspects of a Software Designer's Associate, it is not expected that any of them will lead directly to an implementation. Rather, each effort will contribute information to be used in specifying the requirements and design of a Software Designer's Associate. Some activities are also aimed at resolving questions associated with the implementation of certain critical or high-risk components of the SDA.

6.1 Prototyping Objectives

The principal objectives for the first year's prototyping efforts fall into the following categories:

- **Models:** Construction of the Software Designer's Associate is to be based on a set of conceptual, semantic and realization models which describe the software process as well as its products and the tools used in their production. Prototyping efforts will be aimed at evaluating the effectiveness of these models and languages to be used in their description.

- **Design Processes:** Instances of Software Designer's Associates will serve as specialized environments to support preliminary design activities. Despite the large number of existing design methods, however, software design activities are poorly understood. In order to provide a better understanding of the design process and its products (as well as provide a more realistic foundation for the Software Designer's Associate) prototyping efforts will explore these issues by focusing on existing design methods and attempting to elucidate their commonalities as well as their differences.

- **Architecture:** Architectural issues relate to the information repository, the user's view of the Software
Design processes and products often take on an hierarchical structure and, in these cases, the attribute grammar formalism can be used as a basis for their description. At Tokyo Institute of Technology, an environment called SAGE [Shin88] has been developed for supporting software development using a language, AG, designed for hierarchical and functional programming based on attribute grammars [Kata81]. The use of SAGE will facilitate the exploration of languages for describing SDA models and their use in describing design processes.

In order to use AG for software process modeling, however, it will be necessary to add capabilities for scheduling and monitoring of activities which are represented as AG functions which transform input products to output products. Although a process description using attribute grammars specifies what activities should be performed and how they are related, it says nothing about how they are scheduled, executed and monitored. Activity scheduling and monitoring facilities for an AG-based implementation must, therefore, make it possible to:

- determine dependencies among activities and use this information to determine which activities must be performed serially and which can be performed in parallel;
- show which activities are to be performed at a given time together with information on which tools are to be used.
- If several activities can occur, concurrency control is provided by the scheduler/monitor; and
- keep track of the status of various activities and provide feedback on the status of the design process.

Current prototyping efforts are also addressing these issues.

At Keio University, the use of Modula-2 as a software process description language is being explored. In this approach, each activity is described by a single module. For this purpose, the Modula-2 language has been extended to provide for module nesting. This makes it possible to more completely describe the hierarchical structure of activity decomposition.

6.2.2 Design Processes

The Software Designer's Associate will be a "generic" environment, capable of supporting many different software creation and evolution methods. Each instance will, then, support a particular method. In order to develop such an environment, it will be necessary to abstract the activities of the different design methods. Before this abstraction can be performed, it will be necessary to extract essential activities (or characteristics) of existing methods. Some of these activities will be common to many methods; others will be specific to a given method. The common activities will be described in a generic fashion and then elaborated into a concrete description.

In order to avoid ambiguity, it is desirable to describe each activity as formally as possible. At Osaka University, to explore issues in this arena, formal descriptions will be expressed in ASL (Algebraic Specification Language) [Higa84]. An interpreter for ASL/F, a subset of ASL [Inou86], will be used for constructing prototypes.

6.2.3 Architecture

Prototyping efforts which address information repository issues are based on reconstruction of the Zodiac system developed at Software Research Associates [Sako86]. The existing Zodiac system supports design and generation of application software for small business computers. The current environment includes some of the functionality planned for the SDA, but does not embody the concepts of integration and adaptation which are necessary for a Software Designer's Associate. Reconstruction of this environment will make it possible to address issues of information repository schema and object management sub-system architecture. This effort will also serve to validate the model-based integration approach and indicate the actual feasibility of implementing a Software Designer's Associate.

The Prototyper's Workbench, developed at the University of Hawaii [Miy87], is being used to construct prototypes of a Software Designer's Associate based on the various language schemas proposed. The Prototyper's Workbench is a multiple model-based prototyping environment which can accommodate a number of different models and formalisms. Prototypes developed using this environment are aimed at exploring implementations for a software process management sub-system with a strong emphasis on the user interface.

Prototypes developed using the Prototyper's Workbench will be exercised by potential users of the Software Designer's Associate to determine the suitability of the user interface and its underlying process management sub-system. Feedback from these users will be used to help refine the requirements...
for Software Designer's Associates in a "user-oriented" design approach.

Other prototyping efforts which address architectural issues are aimed at developing a mechanism which can naturally support the expression and accumulation of knowledge about the design process. Initially, knowledge about the design process is to be expressed in a rule-based formalism based on extensions to Prolog developed at Shizuoka University [Ochi88]. A Prolog rule is used to represent a primitive design activity and predicates correspond to either tool invocations or human design activities. Arguments specify the software objects involved in the activities. The backtracking feature of Prolog has been extended to represent controlled iteration by introducing a new notation for backtracking scope. This approach will be used to help define substrate mechanisms for tool integration and for connecting tools to the information repository.

The two approaches which address process management issues take very different approaches and address different issues. The results of these efforts will be evaluated and combined with the lessons learned from other prototyping activities in the actual implementation.

7. Related Work

Several existing software environment research projects have one or more features in common with the SDA project. FASET [Nobe86], the Prototyper's Workbench (PWB) [Miya87] and Leonardo [Mark86] are all concerned with preimplementation activities. FASET is concerned primarily with the development of tools to support formal specification techniques while the PWB is intended to support requirements analysis and definition activities based on a prototyping approach. Leonardo is expected to encompass a broad range of techniques which may include both formal specification and prototyping. In both MARVEL [Kais87], [Barg87] and Arcadia [Tayl87], [Tayl86], the software process is a central concern. MARVEL encodes "knowledge" about the software process using a rule-based approach to describe preconditions and postconditions on tool invocations. In Arcadia, the software process is to be described using process programs [Oste87], [Oste86]. Both MARVEL and Arcadia also make use of sophisticated object management systems to mediate the application of tools to software objects and to provide specialized views of those objects. The description of human activities in the design process is a central feature of Leonardo. And, the incorporation of advances in user interface technology is an important aspect of the Arcadia project.

The Software Designer's Associate project is unique in its consideration of all of these concerns and in its use of a model-based approach to the design and construction of a software environment. In the SDA project, we are attempting to provide support for a wide variety of software design paradigms and methods, incorporate consideration of both human and automated design activities, and take advantage of advances in object-management and user-interface technologies.

8. Conclusions

This report has described the Software Designer's Associate concept and the cooperative, international project which is intended to lead to the realization of that concept. In this concluding section, we discuss the contributions of the SDA project from two different points of view.

First, from the technical point of view, we expect that this project will produce several significant contributions to the software engineering community. As described in Section 4, one of the principal features of this project is the introduction of a "model-based" approach to environment integration. If, as we expect, this approach is successful, it will represent a major breakthrough in establishing a framework for environment architectures. Even if this approach is not completely successful, however, our research will contribute to understanding the role of the software process and software process models in the design and construction of software environments.

Second, as described in the introduction, the SDA project represents a new level of international cooperation in software engineering research. This approach has involved a certain amount of overhead. Following the inauguration of the project in January 1987, it was necessary to spend a significant amount of time achieving a common understanding among the participants. Meetings among the international participants were held approximately once every six weeks and meetings between academic and industrial participants in Japan were held monthly. This overhead appears to be inevitable for this type of effort which involves cooperation on an international basis as well as between academia and industry. On the other hand, the unique combination of perspectives and expertise afforded by this project structure has more than offset the disadvantages. And, as the project matures, we expect that these advantages will multiply.

To make the results of this project available to the software engineering community at-large, the SDA Consortium has agreed to place the results, including environment prototypes, in the public domain.

9. Acknowledgements

The Software Designer's Associate project is based on concepts which were initially articulated by Bill Riddle. We gratefully acknowledge the contributions of individuals from the participating institutions as well as from several industrial organizations. The following people have contributed to the definition and design of the Software Designer's Associate: T. Daitoku (PFU Corporation), H. Fukase (ASCII Corporation), Y. Hayashi (SRA, Inc.), T. Kawashima (University of Hawaii), K. Kitagawa (Hirata Industrial Machinery Co., Ltd.), S. Murai (Freelance Technical Writer), K. Nomura (Osaka University), T. Nomura (Japan Information Processing Service Co., Ltd.), M. Norisada (NEC Software, Ltd.), A. Ohki (Shizuoka University), M. Ohki (Japan Information Processing Service Co., Ltd.), T. Ohta (Shizuoka University), M. Okada (NEC Software, Ltd.), R. Okamoto (Kobe Computer Service, Inc.), H. Sakoh (SRA, Inc.), Y. Shinoda (Tokyo Institute of Technology), Y. Sugita (SRA, Inc.), S. Subramanian (University of Michigan), K. Takeda (University of Hawaii), T. Yamoka (Keto University). The sponsors of the SDA Consortium are: ASCII Corporation; Hirata Industrial Machinery Co., Ltd.; Japan Information Processing Service Co., Ltd.; Kobe Computer Service, Inc.; NEC Software, Ltd.; PFU Corporation; Software Research Associates, Inc.; and Toden Software, Inc.
10. References


