1996

The Purdue PSE Kernel: Towards a Kernel for Building PSEs

Sanjiva Weerawarana

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

John R. Rice
Purdue University, jrr@cs.purdue.edu

Ann C. Catlin

Margaret G. Gaitatzes

See next page for additional authors

Report Number:
96-082

THE PURDUE PSE KERNEL: TOWARDS A KERNEL FOR BUILDING PSES

Sanjiva Weerawarana, Elias N. Houstis
John R. Rice, Ann C. Catlin, Margaret G. Gaitatzes,
Cheryl L. Crabill, Shahani Markus & Tzvetan T. Drashansky

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR 96-082
December 1996

PPK: Towards a Kernel for Building PSEs

Sanjiva Weerawarana, Elias N. Houstis, John R. Rice, Ann C. Catlin, Margaret G. Gaitatzes, Cheryl L. Crabill, Shahani Markus and Tzvetan T. Drashansky

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, USA.

Abstract

Problem Solving Environments (PSEs) are very high level software environments that provide all the facilities for dealing with some class of problems. It is clear that building PSEs is a costly endeavor both in terms of the person-years required and the diversity of knowledge and expertise required. This paper is about the Purdue PSE Kernel (PPK), a software framework designed to assist in the development of PSEs.

PPK assumes a fairly general model of PSEs where PSEs are viewed as a collection of communicating, cooperating entities. The architecture of PPK is designed to provide all the infrastructure needed to build application PSEs that adhere to this model. This model is realized in terms of an electronic notebook for user interaction with the PSE, an object manager for storing all the problem and solution components and a software bus for supporting the communication and integration needs of the components of the PSE. An embedded, customizable programming language is provided within the electronic notebook to allow users to "program" a problem solving process by specifying a high level script. This base architecture of PPK is augmented with a set of domain-specific toolkits which provide the required infrastructure in key areas such as symbolic computation, computational intelligence, computational geometry and numeric computation. In addition, a high level composition framework allows users to compose PSE from existing PSE components.

This paper describes the overall design of PPK and a prototype of the basic PPK framework that we have developed. An example PSE built using PPK has demonstrated its viability as a kernel for building PSEs.

1 Introduction

A Problem Solving Environment (PSE) is a computer system that provides all the computational facilities necessary to solve a target class of problems [1]. These features include advanced solution methods, automatic and semi-automatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems, so users can run them without specialized knowledge of the underlying computer hardware or software. By exploiting modern technologies such as interactive color graphics, powerful processors, and networks of specialized services, PSEs can track extended problem solving tasks and allow users to review them easily.

Overall, they create a framework that is all things to all people. They solve simple or complex problems, support rapid prototyping or detailed analysis, and can be used in introductory education or at the frontiers of science. PSEs provide users with tools and facilities for "multifidelity" simulation. That is, initially one may wish to perform conceptual, symbolic simulations. Later a preliminary numerical simulation may be performed and even later a detailed simulation requiring many hours or days of simulation time may be performed.

It is obvious that building such software systems is a monumental task [2]. In addition to the sheer quantity of program code required, the task is further complicated by the diversity of knowledge required to build complex PSEs. It is clear that a strong computer science background is needed to build such PSEs, yet it is not computer scientists who will (or should) be building these PSEs. Application scientists must be able to build (or compose) PSEs for their work without having to spend person-years of effort on computing tasks which have basically nothing to do with their problem domain. Clearly, a solid base infrastructure upon which application scientists can build their PSEs is needed. The *Purdue PSE Kernel (PPK)* is a software framework (infrastructure) designed to assist PSE builders in their task.

PPK assumes a fairly general model of PSEs where PSEs are viewed as a collection of communicating, cooperating entities. The architecture of PPK is designed to provide all the infrastructure needed to build application PSEs that adhere to this model. This model is realized in

terms of an electronic notebook for user interaction with the PSE, an object manager for storing all the problem and solution components and a software bus for supporting the communication and integration needs of the components of the PSE. An embedded, customizable programming language is provided within the electronic notebook to allow users to "program" a problem solving process by specifying a high level script. This base architecture of PPK is augmented with a set of domain-specific toolkits which provide the required infrastructure in key areas such as symbolic computation, computational intelligence, computational geometry and numeric computation. In addition, a high level composition framework allows users to compose PSE from existing PSE components.

While PSEs are a special type of software, they do share many properties with other large scale integrated software environments such as Microsoft Office. Clearly, there already exist many software frameworks which support the development of such systems, including Microsoft OLE [3], OpenDoc [4], and CORBA [5]. There is also a large body of existing work on distributed communication environments such as RPC [6], MPI [7] and Glish [8]. However, due to a variety of reasons that will be described later in detail, none of these systems completely addresses the somewhat unique infrastructure needs of problem solving environments. A recent project called PSEWare [9] by a consortium lead by Indiana University is researching kernels for building PSEs as well and appears to be very similar to the PPK project. PSEWare is also described later.

In this paper we describe the overall architecture of PPK and a partial prototype that has been completed. This paper is organized as follows: the next section describes the problem solving process used by scientists using computation as the primary technique for solving some problem. The goal of PPK is to provide a complete software framework to support building software that follows this process. Section 3 provides an overview of PPK. Sections 4 to 10 describe the various components of PPK: the overall architecture, the software bus, the object manager, the electronic notebook, the domain specific toolkits, the PSE component browser and the PSE composer. Section 11 briefly describes a basic PSE built using PPK. Section 12 describes related work and compares it to PPK. Finally, section 13 draws some conclusions and indicates briefly our future plans for PPK.

2 Problem Solving Process

In this section we consider the problem solving process used when computation is the primary technique for solving some problem. We consider activities from both the user's viewpoint and the "system's" viewpoint. What we refer to as the system is the sum total of all software/ hardware which is involved in computationally solving the problem.

Initially, the user must define the problem to the system. In a PSE, this specification is declarative (i.e., only indicates the required information and not what to do with it or how to do something with it), symbolic (i.e., in some abstract form) and in terms that are natural to the problem domain. At this level only the essential features of the problem are specified; there is no indication of how it is to be solved or any other solution scheme related information provided.

If a PSE already exists for solving this type of problems, then the user must interact directly with the PSE and solve the problem. Since in this paper we are concerned with the situation where a PSE is not already available, we will ignore this case from now on.

Suppose that while there is no existing PSE for solving the specific problem, there does exist a (large) collection of problem solving components (i.e., a workbench), including those that are needed to solve the current problem. Then, the user must first combine some of these components to form a custom PSE and then apply it to solve the problem at hand. In this case, the user must be able to "browse" the available components, "select" the appropriate ones, and "connect" them to form a custom PSE. Then, to solve the problem, the user transfers the declarative problem specification to the PSE and interacts with the PSE appropriately.

The final scenario is when only some of the components needed to solve the problem are already available from a component database. The other needed components must be custom developed. Thus, the user must "build" the components by writing program code in some form and then connect the components together to form the PSE, as in the previous scenario.

How does the problem solving environment thus assembled finally solve the problem? The process (Figure 1) involved can typically be decomposed to the following five stages:

- Declarative problem specification: As described above.
- Computational script: The problem specification must be transformed to some solution algorithm which when run will result in solving the problem. The computational script can be viewed as a high level, pseudo-code specification of this algorithm. In some instances, this script may not be explicit, but it is usually present nevertheless.
- High-level programming language program: The computational script must be executed by either interpreting it directly or by translating it to a program in some traditional high level programming language.

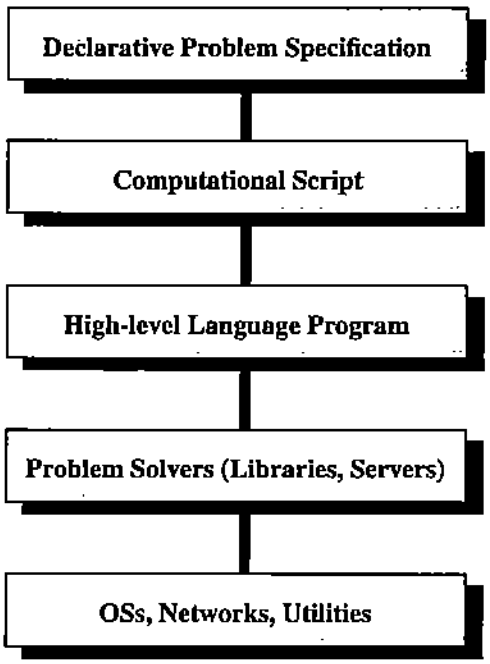

Figure 1 Levels of computation in a PSE.

- Problem solvers (libraries, servers): The problem solvers are the components that do the real work for solving the problem. These are invoked from the high level language or by the script interpretation process.
- OSs/networks/utilities: The lowest level is the traditional computing platform on which the problem solvers execute.

3 Overview of PPK

The goal of the Purdue PSE Kernel project is to develop a software kernel that can be used to build PSEs that support the problem solving process described above. This goal is realized by the following components:

- PSE architecture: PPK defines and supports a powerful, extendable PSE architecture. All the components of PPK and the resulting PSEs assume and support this architecture.
- PSE component database and browser: The component database and browser allow users to view existing PSE components as well as to install new components into the component database.
- PSE composer: The composer is essentially a very high-level programming facility where the user programs in a data flow manner. Composing components selected from the component database is expected to be the likely approach to building custom PSEs.
- Electronic notebook: The problem solving process typically involves multiple steps and a solution path determined by trial-and-error. The electronic notebook serves as the central recording and access environment for monitoring, controlling and steering this process. An embedded programming environment allows users to program (or script) a sequence of operations to be performed during a problem solving process.
- Object manager: The problem solving process involves many data objects including the problem input objects, the solution objects and the output objects. The object manager is the database which manages these components for the user and for the PSE components.
- PSE component builder: The components ("tools") of a PSE are what provide the real computing muscle to it. The component building process involves using the appropriate data object standards for input and output and implementing the component's internal functionality using whatever toolkits are provided by the environment.
- Language kernel: The language kernel is a toolkit with which one can build the application specific language with which the user may interact with the PSE.
- Software bus: The software bus is the underlying "glue" that supports the integration and operation of the PPK framework outlined above.

Figure 2 illustrates the layered organization of these components in PPK. The PPK prototype currently built supports the "lower level" infrastructure of PPK comprising of the software bus, the object manager and the notebook. In the following sections we discuss the architecture

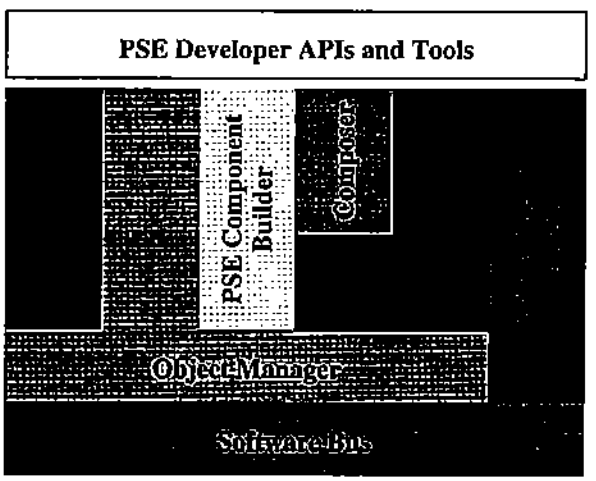

Figure 2 Layered Architecture of PPK.

of PSEs supported by PPK and each of these lower level components; the "higher level" components (the component builder, browser and composers) will be described in a separate document. The language kernel is an intermediate level component of the PPK framework which is embedded in the notebook. A brief discussion of the language framework will be provided in the notebook section.

4 PSE Architecture

The architecture of PSEs supported by PPK is also based on the levels of computation as illustrated in Figure 1. That is, a PSE build with PPK will have roughly the layers present illustrated in Figure 1. Each layer in this model contains a collection of tools. The tools interact within a level via well-defined object interfaces. For example, at the high-level programming language level, the word "tool" may refer to a function while "object interface" may refer to some "standard" data structures and function signatures. Interaction across levels occurs in some abstract specification language or by some automated process that translates a set of tools and objects from one layer to the representations used at a different layer. Clearly the key then is to allow the integration of the various pieces to form the comprehensive, integrated system that provides problem solving facilities to the user. Software that provides such integration frameworks is typically called "middleware" and PPK can be viewed at as a middleware system for PSEs.

Building a PSE using PPK requires one to customize it by configuring the core components (software bus, notebook and object manager) of PPK appropriately and by developing any necessary tools. The result is a customized framework into which application-specific components can be integrated conveniently.

5 Software Bus

The "software bus" model is used in the lowest level glue that binds components of PPK together. The software bus concept is an attempt to emulate the hardware bus mechanism that provides a standard hardware interface to attach additional capabilities to a machine. In the hardware bus, new units describe their capabilities to the bus controller, which then passes the information along to other units in the bus. In the PSEBus software bus, software components register their exported services with the software bus and rely on the software bus to invoke these services when requested by interested clients. The software bus is responsible for the application of any representation translators as required for the valid invocation of the service. The location and instantiation of service providers is also managed by the software bus, thereby relieving application components of the need to be aware of the global application topology. Thus, the software bus provides a mechanism where two or more tools can interoperate with each other without having explicit knowledge about each other and also provides the infrastructure for managing a set of distributed tools. Other features include security (all communication can be made fully secure), support for multiple communication protocols at the lowest level and compression for dealing with large data objects.

PSEBus is implemented as a fully-distributed, multi-threaded set of libraries that will be linked with the application components. When PSEBus is instantiated, it creates a set of threads for handling software bus tasks and supports the communication needs of one or more user threads. Components (clients) are identified with a logical name (string) which provides a level of indirection by separating the logical identity of the component from its physical attributes such as executable file name, file path and host name. The communication facilities available in PSEBus include both blocking (synchronous) and callback send/receive facilities for both connection-oriented and connection-less communication. A component may communicate in datagrams, objects (data structures) or messages (message header and zero or object arguments).

PSEBus provides all the communication facilities needed to implement PPK. Specific functionality used by each tool will be illustrated there.

6 Object Manager

The problem solving process involves many data objects including the problem input objects, the solution objects and the output objects. The object manager is the persistent database which manages these components for the user and for the PSE components. Since components of PPK-based PSEs interact with one another via these objects, the object manager also serves as the shared workspace that facilitates component interaction. The PPK object manager is basically an object oriented database kernel that supports persistently storing and retrieving of PPK objects under direct user control or by other components via the object manager remote interaction API (application programming interface). In addition to the object manager kernel, a graphical user interface exists for browsing and manipulating the database of objects.

6.1 PPK Objects

PPK objects are a data abstraction that allows users to store and retrieve arbitrary types of data. Each data object has associated with a set of meta-data that describes the object and its associated properties:

- *name*: a (hierarchical) name is associated with each object and is used to locate an object within an object manager. The full name of an object is a combination of the location of the object manager, the object's type (below), its hierarchical name and its version number (below). The full name of an object uniquely identifies and locates it on a network.
- *type*: each object is tagged with a type name to describe the content of the data associated with the object. The type name itself is written using an extension of the MIME (Multipurpose Internet Mail Extensions [10]) standard for describing content types on the Internet. The extension allows us to represent multilevel types rather than just the two-level model supported by MIME. A multilevel (i.e., hierarchical) type name is used to represent the inheritance hierarchy of an object's type in a string format. For example, a two-dimensional domain in PDELab would be given the type name "pdelab/domain/2d."
- *immutable*: every object is immutable; i.e., cannot be destroyed once created. This property, along with versioning (below) allows users to record the history of a problem solving process rather than just the end result, a very useful capability.
- *version*: immutability of objects means that when an object is "changed" and the change committed, a new object with the same name but a different version number is created. Many versions of an object may exist at any given time; hence the version number is a part of the object's unique name.
- *dependencies*: objects typically are dependent on other objects. The dependencies property of an object records the full names of other objects that it is dependent on. For example, a mesh object may depends on the domain object over which the mesh is defined.
- *parameters*: objects typically also have parameters, which may be typed or untyped depending on the object. For example, a circular domain may be defined as a domain object with center at (x,y) with radius r. This object then has (typed) parameters x, y, and r. Parameters must be bound to values before parameterized objects can be used in most computations.
- *bindings*: parameters of an object may have predefined possible value sets. These are represented as bindings sets associated with the object.
- *multiple representations*: an object typically has many ways in which it may be represented. For example, a circle of radius r centered at the origin may be represented by the equation x^2+y^2 or be described as the area enclosed by the boundary defined by x = sin(t), y = cos(t) for t = 0, 2*pi. It may also be described by an image in some format. Each such representation is identified by the type name of the data associated with the representation, where the type name is written using the (extended) MIME syntax. For example, an image (in GIF format) representing a domain would be called the "image/gif" representation of the domain. The approach PPK follows unifies both presentation representations of an object (such as "image/gif" and a computational representation (such as "binary/pdelab/domain"), allowing us to treat these two cases uniformly. A client who requests an object "knows" what possible representations it might require and requests them. For example, the notebook would request image or text representations for an object embedded in the notebook, whereas the domain editor would request a binary representation for a domain.

The goal of the PPK object model is to support arbitrary types of data which may have varying requirements and needs. The approach we have chosen for this allows us to treat both presentation and computational versions of the data uniformly without bias. Each client who requests an object must know a priori what possible representation(s) it can accept and request one of them. This is not an unreasonable requirement as an existing client presumably always know what it needs. To help a clients in their selection, facilities exist for listing the available representations and also for requesting one of a set of possible representations.

6.2 Object Manager Interface

The (graphical) object manager interface allows users to interactively browse the database of objects in a given object manager. Users can select between browsing by name and browsing by type, each of which mode gives higher priority to the given classification. In addition to browsing, users can view certain representations of objects (for example, image/gif representations) and also perform some operations (such as renaming and copying). Drag and drop facilities are supported between the object manager interface and the notebook to allow users to conveniently move objects to the notebook. Instances of the interface are shown in section 8.

6.3 Implementation

The prototype implementation of the object manager is on top of the Unix file system as the underlying persistent store. Within an object manager, an object is mapped into the file system as follows: the object's type is used to locate the directory in which the object is stored by replacing each '/' with a '.'. For example, "pdelab/domain/2d"

would become "pdelab.domain.2d." Within each type directory, an object named "/tmp/demo/circle" would be stored in the subdirectory named "tmp/demo/circle" of the type directory. Within each object's directory, a separate directory is created for each version and is named by the version number. In addition, a file is created to record the latest version number associated with this object. Within each version of an object, each of its representations is stored in a separate file named by the representation type name changed as before. This file is generated by serializing the objecting into the file using whatever serialization was registered with the object manager / software bus for that object. For example, the representation "binary/pdelab" would be stored in the file "binary.pdelab." A separate file ("representations.config") is used to list the currently available representations for an object. Several other files ("dependencies" and "parameters") are used to store other information such as the dependencies and parameters of an object.

The graphical user interface is implemented using the X11/Motif window system and basically supports browsing through this database. It also allows one to perform some limited operations on the objects, such as viewing them.

The object manager API allows remote client programs to perform all their interaction with the object manager via a function call interface. In addition to get/set type functions for all properties of objects, there are several query functions to query the database for information.

7 Electronic Notebook

The problem solving process typically involves multiple steps and a solution path determined by trial-and-error. The purpose of the electronic notebook is to serve the purpose of a *notebook* for recording and tracking this process. While most high level problem solving systems have languages for recording the problem and the solution algorithm, they typically do not allow one to record the entire process, including the various iterations that did not produce correct results. The objective of this approach is to mimic the physical laboratory notebook (notepad) that scientists commonly use in their day-to-day activities. This notebook records not only what worked, but also the steps and iterations involved with achieving that success. The electronic version of the notebook can, and must, of course extend the capabilities of the physical notebook that scientists use daily.

The PPK electronic notebook is a free-format multimedia document editor that allows users to record arbitrary content (including, but not limited to, text, graphic, audio and video). In addition, the notebook serves as an intelligent, high-power calculator for the problem domain by providing the user with a set of tools (i.e., the tools in the PSE that the notebook is part of) to create an manipulate objects that arise naturally in the domain. That is, from the point of view of the user, the notebook *is* the PSE. Since most problem solving processes involve multiple algorithmic strategies, an embedded programming language allows users to program (or script) a sequence of operations to be performed to solve a problem. In the rest of this section we describe the multimedia editing functionality of the notebook, how PPK objects are included in the notebook, the embedded programming language, and how tools interact with the notebook at run-time.

7.1 Multimedia Editor

Note-taking is clearly the first and most fundamental task of the electronic notebook. In the PPK context, note-taking means being able to record arbitrary PPK objects which arise in the process of defining and solving problems, and being able to introduce arbitrary *notes* (either textual or other) about these objects.

In order to mimic the look-and-feel of the physical notebook, the notebook is a paged, free-format editor where one may insert *annotations* at any place on any page. Two types of annotations are supported: embedded annotations and reference annotations. Embedded annotations are annotations whose media content is embedded in the notebook. For example, a piece of text is an embbed annotation as could be an image. Reference annotations on the other hand are annotations whose useful media content is maintained somewhere else and only referred to by the notebook. For example, an audio clip and a Web (HTTP) reference are reference annotations. An embedded annotation is used to indicate the presence of a reference to the user by using the prior as a label or handle. PPK objects are included in the notebook as embedded annotations and are discussed in the following section.

The notebook is necessarily a *user* interface, where there user is human. However, in cases when a PPK-based PSE is a part of a larger process, the end-user of the PSE may not be an end user at all. In order to support these scenarios, the architecture of the notebook allows one to disregard the graphical user interface and to treat the balance as a problem-specific object container and calculator.

7.2 Embedding PPK Objects

The notebook allows one to insert arbitrary PPK objects into it as embedded annotations. Embedded objects do not migrate into the notebook; they still live within the object manager. The notebook simply receives the name and type of the object to be embedded and

requests appropriate representations for it from the object manager.

When embedded, objects need a *visual (presentation) representation* so that the user may "see" the object inside the notebook. Each object can define the types of possible visual representations it wants (for example, plain text, image in GIF format, or image in Postscript format) and the notebook selects one to use based on user preferences and other configuration parameters. It is important to note that the object manager does not distinguish between a visual representation and a data representation of an object; they are both of equal privilege. The receiver has the responsibility and option of requesting and using the appropriate forms.

Object types that can be embedded in the notebook must be configured into it. This is done by using the notebook Tcl [11] based configuration language to write a script that indicates the object type, the bitmap to use as an icon, the possible visual representations, the possible actions and how to implement each of the actions.

Note that objects are *non-editable* within the notebook itself. That is, one must always invoke an external tool by performing an action on the object to edit or perform any changes to an object and/or its parameters. This is different from Microsoft OLE, for example, where embedded objects may be edited *in situ*. These and other issues are discussed in Section 9.

Objects are rendered within a notebook in a manner that indicates the objects parameters and dependencies as well as any bindings to the parameters. When an action is invoked on an object, its current closure is determined and sent to the tool which is receives the action request.

7.3 Programming in the Notebook: PPKScript

Clearly problem solving is not a matter of invoking a set of pre-existing tools. The ability to write a script that invokes tools and performs other operations is vital in large scope problem solving environments. To this end, the notebook supports an embedded programming language model that can be used to write problem solving scripts.

The architecture of this component supports being decoupled from the notebook itself. That is, the embedding of the PPK programming language in the notebook is a special case; the language and the associated functionality can be embedded in any containing software system. This ability is useful as it allows one to integrate the problem solving capabilities of PPK into any system, not just the PPK notebook which is intended for (human) users.

The goal of the PPK language, PPKScript, is to be somewhat analogous to what JavaScript [12] does for HTML, the language used to author documents for the World Wide Web. JavaScript defines an object model that maps HTML documents into JavaScript object structures and then provides the "usual" language facilities such as loops, conditionals, functions and data structures for writing programs that manipulate these objects. PPKScript allows one to access and manipulate PPK objects embedded in the notebook and to write programs with all the usual language facilities.

This is achieved by defining a clean mapping of PPK objects to PPKScript. This allows one to access the data and meta-data fields of PPK objects at a programming level of PPKScript. Then the other features of PPKScript, such as variables, block structured scope, loops, conditionals, functions and objects can be used to implement arbitrary problem solving scripts. PPKScript also has a language level interface to PSEBus to allow problem solvers to write scripts that interact with multiple problem solving tools available in the PSE.

This is the PPK provided language framework. However, for each PSE built with PPK, one can customize PPKScript by defining a set of "built-in" functions/objects that are available to the problem solver. This customization can be done in PPKScript as well as in "foreign" languages which are imported into the runtime environment.

7.4 Implementation

The prototype implementation of the notebook is based on the Window-Icon-Mouse-Pointer (WIMP) model. That is, the user navigates and interacts with the notebook using traditional interaction technologies. An implementation using novel technologies such as pen input, tactile manipulation, speech recognition and generation is possible and very much desired.

The notebook is implemented using X and Motif. The architecture achieves a clean separation between the notebook's core functionality and its graphical user interface component so that it can easily be ported to other GUI toolkits and also so that it can be used without a GUI.

The kernel of the notebook manages all the annotations in the notebook, including layout, event delivery and storage. Notebooks are stored by serializing the state of the notebook to a file, which permits full recovery of state upon reactivation. Event delivery is done by locating the annotation(s) to which the event should be delivered and then by converting the physical events (such as mouse/key clicks) to logical events (such as view/edit). The operations to be done by an annotation in response to an event is configured into the notebook and then activated at this time. Such activation involves contacting the appropriate tool and delivering one or more messages.

The PPKScript language kernel is not yet integrated with the notebook. We are currently experimenting with

re-using an embeddable language kernel (such as from Tcl or Scheme or Java) as the core language to build upon.

The notebook API allows tools to interact with the notebook via messages. This API allows tools to inform the notebook when new objects are created and also for the notebook to inform the tools when users request operations on objects.

8 Example: PDELab PSE

We have started to build a new version of our PDELab [13] PSE as proof-of-concept for PPK. This system consists of PDEBook, PDEOm, PDEBus and various PDE tools. The first three of these are the PPK notebook, object manager and software bus customized for the PDELab problem solving environment. In this section, we will briefly describe the process used to develop these customized components and also discuss the changes needed to take an existing PDE tool (such as domain editor or a mesh editor) and convert it to be a PPK accessible tool. Note that while the configurations described below are mostly static (i.e., done before the PPK-based PSE is running), the architecture fully supports completely dynamic configurations.

8.1 Configuring the Software Bus

The software bus plays a central role at development time of a PPK-based problem solving environment. At run-time, the software bus is responsible for activating the processes that are the tools of the PSE and also for delivering messages between them. The software bus must hence be configured with the set of tools that can be activated at run-time (the configurations can be made dynamically as well). This task is fairly simple; one must basically list the major components (configured notebook and object manager) as well as all the tools that are to be used by the PSE and indicate the path to their program files as well as any invocation options. Figure 3 shows part of the configuration script for PDELab.

8.2 Configuring the Notebook

To configure the notebook, one must first configure the set of PSE tools that are available. For each tool, the tool's textual name (i.e., text label shown to the user), its programmatic name (i.e., the name by which it is identified in the software bus) and its icon (i.e., a bitmap that is displayed to identify the tool) must be given. The location of the program binary for the tool itself and its options etc. are given in the software bus configuration (above).

The notebook must also be configured for the types of objects it is expected to handle. An object configuration

```
# find the install location of pdelab
set pdelab $env(
PDELAB)

# find the install location of ppk
set ppk $env(PPK)

# standard I/O streams for a process
set ios {{stdin in      0 file /dev/tty}{stdout out    1 file /dev/tty} \
      {stderr out    2 file /dev/tty} {pdebus-in in  3 pipe pdebus} \
      {pdebus-out out 4 pipe pdebus}}

# debug level for the notebook and the om
set debuglevel 2
set omdebuglevel $debuglevel
set nbdebuglevel $debuglevel

# notebook config file location
set nbconfigfile $pdelab/lib/pdebook/config

# om config file location
set omconfigfile $pdelab/lib/pdeom/om_config.rc

client "object-manager"  $ios \
   "x-client-loc://localhost/$pdelab/bin/pdeom" -debug 3 \
   -config $omconfigfile

client "notebook" $ios \
     "x-client-loc://localhost/$ppk/bin/notebook" \
     -debug $nbdebuglevel -configfile $nbconfigfile

# register all the tools
register client "meshtool2D" $ios \
     "x-client-loc://localhost/$pdelab/bin/meshtool2D"
register client "domaintool2D" $ios \
```

Figure 3 Configuring the software bus.

starts with its extended-MIME type. Then, the list of actions that are possible must be given along with how each action must is to be implemented. Each action implementation indicates what tool is responsible for supporting that action and what message(s) should be sent to that tool to implement that action. Finally, the possible visual representation types for the object type must also be indicated so that notebook can allow the user to select an appropriate one. Figure 4 shows the configuration script use to configure the "pdelab/domain/2d" object type in PDELab.

Once PPKScript is implemented, additional configuration would be needed to define the built-in functions for PDEScript, the PDELab scripting language.

These are all the configurations that are needed to configure the notebook for the PDELab application. Figure 5 shows an instance of the PPK/PDELab notebook in action. The application PSE developer does not need to be aware of any of the internals of the kernel nor to write code in low level languages. The kernel delivers the notebook as a pre-compiled executable which loads in a detailed config-

```
# object representations commons to most objects
set standardreps {text/pdespec text/plain
                  image/xwd image/xpm image/gif image/xpm}

# set up the pdelab/domain/2d type.
set actions {
   {Edit domaintool2D edit <OBJECTURL> <OBJECTTAG>}
   {View domaintool2D view <OBJECTINSTANCE>}
   {"Define Boundary Conditions" domaintool2D do bc \
                                 <OBJECTINSTANCE>}
   {"Generate Grid" gridtool2D do grid <OBJECTINSTANCE>}
   {"Generate Mesh" meshtool2D do mesh <OBJECTIN-
STANCE>}
}
ppknb_type "pdelab/domain/2d" domain.xpm $standardreps
$actions
```

Figure 4 Configuring the notebook.

uration file to learn all necessary information about the tools and objects of the PSE it is working for.

8.3 Configuring the Object Manager

Unlike in the notebook case, the object manager does in fact need to be aware of data-structure level knowledge of the data it is working with. The kernel hence provides the object manager as a library with hooks to allow the application-dependent data types etc. to be defined into the object manager. This application initialization function is responsible for registering and defining the various representation types that will be used by the various objects. In addition, it indicates the function used to serialize data of that type to/from a byte-stream format. This function is used when data types are transmitted via the software bus to the notebook or any other tool as well as to store objects persistently in the file system.

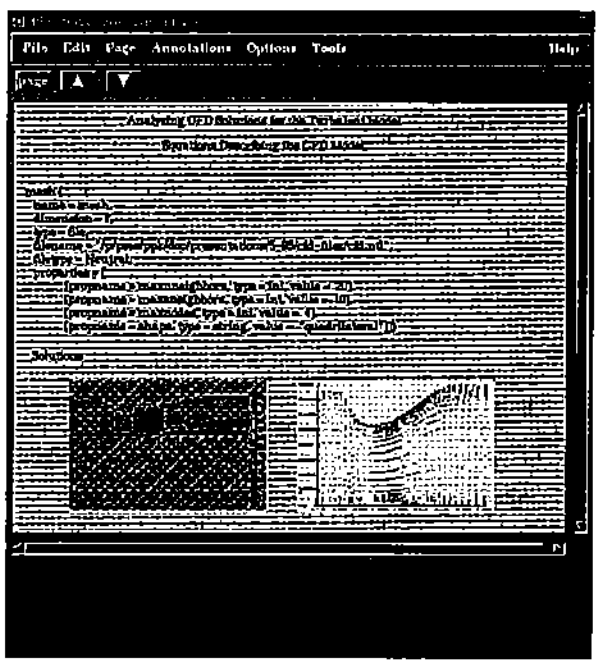

Figure 5 PPK/PDELab Notebook in action.

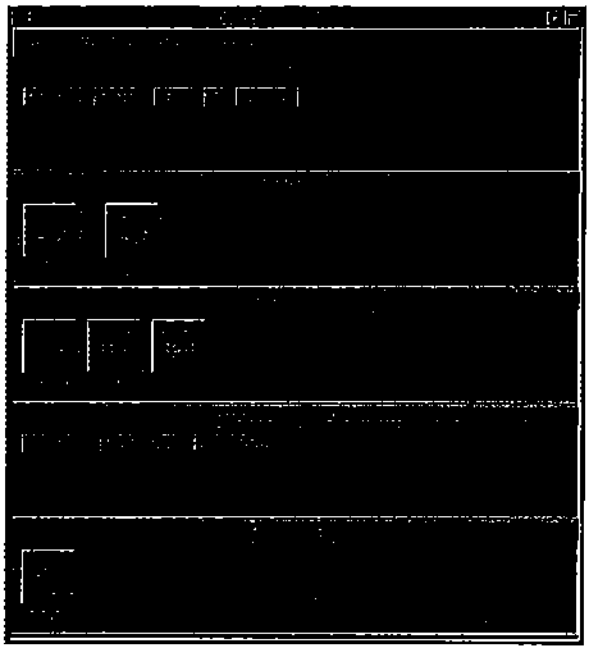

Figure 6 PDELab Object Manager GUI.

In addition to configuring the data management aspects of the object manager, one must also configure the browsing features. This is done by indicating the icons to be used to represent each type of object when browsing the object manager's data space graphically. Figure 6 shows an instance of the object manager's graphical interface.

8.4 Configuring the Tools

To incorporate a tool as a PPK-compliant tool, one must (re-)engineer the tool to invoke several initialization functions and then register appropriate event handlers. The steps involved are the following:

- As soon as the process starts up, invoke the software bus initialization function to initialize communications via the software bus.
- Initialize the notebook and object manager APIs by invoking the appropriate function.
- Register the types of objects that will be communicated by this tool by installing the type serializers (as in the object manager case) to the software bus.

- Write the functions that will be invoked when the user requests certain operations on the object type(s) this tool is responsible for (for example, edit or view).
- Register the even handling functions with the software bus.
- Change the "saving" or "committing" function of the tool to insert the saved object into the object manager by using the appropriate functions from the object manager.

9 Related Work

9.1 Microsoft OLE/COM

Microsoft's Object Linking and Embedding (OLE) [3] is a set of services that provides a powerful means to create documents consisting of multiple sources of information from different applications. Objects can be almost any type of information, including text, bitmap images, vector graphics, and even voice annotation and video clips. OLE associates two major types of data with an object: presentation data and native data. An object's presentation data is information needed to render the object on a display device, while its native data is all the information needed for an application to edit the object.

PPK objects can also be of any type and can contain full multimedia information as well. In contrast with OLE, PPK uses one unified model for object data which does not distinguish between "presentation" and "native" uses - it is up to who requests a specific representation to do use it appropriately. Unlike OLE, PPK does not allow one to embed an application directly in another application; it only provides a mechanism to link applications together via the objects shared between them. The linking facilities in OLE and PPK are comparable.

9.2 CORBA

The Common Object Request Broker Architecture (CORBA) [5], is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

The (ORB) is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

The PPK software bus is an ORB in CORBA terminology. While it is currently not CORBA compatible, we expect to make it be so in the future, allowing PPK components to interoperate with any component via a CORBA-compatible ORB. There are no CORBA analogies to the other functionality present in PPK.

9.3 PSEWare

PSEWare [9] is a toolkit for building problem solving environments and hence in closest to the PPK work. The project is just starting and hence is in its early stages. The descriptions give here are from the project overview as given in their Web pages and may or may not be exactly what is being developed in this work.

The PSEWare project is addressing four major issues in the general area of building PSE frameworks:

- PSEs that support the transformation of symbolic problem definitions to parallel object-oriented programs that can be executed efficiently on a variety of sequential and parallel architectures
- Object-oriented libraries of parallel program templates or archetypes that can be refined to obtain specific applications by using ideas such as inheritance
- User interface archetypes for scientific and engineering PSEs that can be refined to construct a PSE for a specific problem domain
- Technologies for collaboration and ubiquitous distributing computing focused on the Internet, the Web and Java, with the goal of applying these technologies to distributed collaborative PSEs.

PSEWare is a multi-institutional collaborative project involving California Institute of Technology, Indiana University, New Mexico State University, Los Alomos National Labs, Drexel University and University of California at Irvine.

While PSEWare is similar in mission to PPK, we are unable to make a direct comparison as PSEWare is yet in concept stage.

10 Conclusion

Building problem solving environments has long been recognized as a complex task and hence the need for powerful, integrated PSE development frameworks is a high priority. The PPK project methodically develops a PSE model and then provides a complete framework consisting of a software bus, an electronic notebook, an object manager, a customizable language kernel, a software component builder, a component browser and an application composer.

The experience in building the PDELab PSE has illustrated the power and flexibility of the PPK architecture. While more implementation is pending, we feel confident that PPK is a significant step forward towards full functional PSE development frameworks.

11 References

[1] J.R. Rice and R.F. Boisvert, "From Scientific Software Libraries to Problem Solving Environments," *IEEE Computational Science and Engineering*, 3 (3), IEEE Computer Society, CA, USA, 1996.

[2] S. Weerawarana, "Problem Solving Environments for Partial Differential Equation Based Applications," Ph.D. Thesis, Dept. of Computer Sciences, Purdue University, 1994.

[3] http://www.microsoft.com/devonly/strategy/ole/ole.htm

[4] http://www.cilabs.org/

[5] http://www.corba.org/

[6] W.R. Stevens, *UNIX Network Programming*, Prince Hall, 1990.

[7] http://www.netlib.org/mpi/

[8] V. Paxson and C. Saltmarsh, "Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems," *Proc. Winter 1993 USENIX Conference*, Usenix Association, 1993, pp. 271-276.

[9] http://www.extreme.indiana.edu/pseware/

[10] N. Borenstein and N. Freed, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," *Internet Request for Comments 2045*, http://www.graphcomp.com/info/rfc/rfc2045.html, 1996.

[11] J. Ousterhaut, "Tcl: An Embeddable Command Language," *Proc. Winter 1990 USENIX Conference*, Usenix Association, 1990, pp. 133-146.

[12] http://www.netscape.com/eng/javascript/

[13] S. Weerawarana, E.N. Houstis, J.R. Rice et. al., "PDELab: An Object-Oriented Framework for Building Problem Solving Environments for PDE Based Applications," Proc. 2nd Annual Object-Oriented Numerics Conference, RogueWave Software, OR, USA, 1994, pp. 79-92.