1-1-1994

The Simulated Bread Board Imaging system: A Study of computer simulation as an engineering tool

Gregory R. Conway

The Simulated Bread Board Imaging System: A Study of Computer Simulation as an Engineering Tool

Gregory R. Conway

Computer Science Department, Rochester Institute of Technology, Rochester, NY
Joseph C. Wilson Center for Research and Technology, Xerox Corporation, Webster, NY

A thesis, submitted to The Faculty of the Computer Science Department at RIT,
in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approvals:

Allen Ted Retzlaff	9/19/94
Dr. Allen Ted Retzlaff, Xerox Corporation	**Date**
John Shaw	9/19/94
Dr. John Shaw, Xerox Corporation	**Date**
Peter G. Anderson	19 Sept 94
Dr. Peter Anderson, Rochester Institute of Technology	**Date**

September 19, 1994

Table of Contents

Appendix A - The SBBI User's Manual

Appendix B - The SBBI Visual Artifacts

Appendix C - Sample SBBI Shell Script

Appendix D - Sample Prints

Bibliography

Abstract

A software system was developed to aid in the translation of customer print quality requirements into formal design specifications during the development of new printers and copiers. The system is designed to allow the user to generate a series of prints with various deviations, or *visual artifacts*, introduced into them. These artifacts simulate the effects of design considerations on the expected output at the image output terminal. The resulting prints can be used to incorporate customer feedback and requirements early in the development cycle. In so doing, it is expected that the need for hardware prototyping can be reduced or eliminated. The result is a product which is cost effective, market timely, and customer oriented.

Computing Reviews Classification Information

Categories and Subject Descriptiors: I.6.0 (Simulation and Modeling), J.2 (Physical Sciences and Engineering), I.4.0 (Image Processing), D.2.7 (Distribution and Maintenance), H.1.2 (User/Machine Systems).

Keywords: Simulation, Modeling, Image Processing, Printers, Engineering, Design, Maintenance

Chapter 1

Introduction

A computer program has been developed to aid in the determination of copier and printer design requirements early in the product development cycle. The program is known as the **Simulated Bread Board Imaging (SBBI)** system. The software is intended to mimic the output characteristics of printers and copiers. The goal is to simulate the effect of design configurations on the print quality of the output image. This allows customers to comment on these image quality effects *before* significant resources have been spent designing and building prototype hardware. Once the user has worked with the customer to determine his or her requirements, the information is sent to Xerox engineers, who translate these requirements into formal design specifications. Chapters 2 and 3 discuss the historical, business and technical aspects of the SBBI system. The remainder of this chapter lays the groundwork for those chapters by defining several key concepts and terms.

1.1 Images

An *image* is a digital computer graphics file containing text or pictures. The SBBI program reads an image as input, modifies the image according to the user's specifications, and writes the resulting image as output. The input image corresponds to the paper placed on the glass copying surface of a copier or the raw data stream sent to a printer. The output image corresponds to the copy produced by the copier or the paper print produced by the printer.

1.2 Visual Artifacts and Properties

A *visual artifact* is any deviation between the input image and the output image. These deviations affect the appearance or quality of the print in some quantifiable fashion. Notice that nothing in the definition of visual artifact implies that the artifact must have a *detrimental* effect on the image appearance. Some customers may find that the inclusion of certain visual artifacts actually *improves* the output image. Many vendors provide software and hardware solutions for customers seeking to improve or enhance images. The SBBI system could be used as a tool to aid in the development of such solutions. However, the SBBI system is typically used to help engineers *minimize* the introduction of visual artifacts, so that the output print becomes a faithful reproduction (within the confines of the customer's requirements) of the input image.

In hardware printers, visual artifacts are introduced by engineering considerations. For example, if the paper does not move through the print engine at *exactly* the right speed, then some areas of the output print will have dots (toner particles or ink drops) too close together while others will have them too far apart. This artifact is commonly

referred to as **banding** (see Appendices B and D). The effects of banding can be reduced by incorporating better motors and paper handling mechanisms. The inclusion of this additional hardware, however, may present engineering obstacles that drive the cost beyond the customer's range. Therefore, it is important to understand *exactly* how much of each artifact the customer is willing to accept, and how much he or she is willing to spend to remove or minimize the remainder. This information is then given to the design engineers who can develop the system in accordance to the customer's requirements. In this sense, the SBBI system supports the *Total Quality Management* initiative at Xerox. Appendix B describes each of the visual artifacts currently supported by the SBBI system.

A *property* is any piece of information needed to describe and implement an artifact. For example, the banding artifact has two properties, the *frequency* of the bands (which determines how many bands will appear per unit distance) and the *amplitude* of the bands (which determines the prominence of each band). The SBBI system imposes no limit on the number of artifacts it can support, or on the number of properties each one of those artifacts may require.

1.3 Modeling and Simulation

The SBBI system is a case study in two related, but separate, fields of study: modeling and simulation. While these two words are often used interchangeably in the computer science field, they shall be restricted to a more precise definition for the purposes of this discussion. The American Heritage dictionary[1] defines a *model* as "a tentative description of a system or theory that accounts for all of its known properties." It defines *simulate* as "to have or take on the appearance, form, or sound of; imitate." A model is a set of equations, theories, and data that describe a system. A simulation is the exercise of a model.

The distinction between a model and a simulation is important because the SBBI program contains elements of both. The modeling component includes the determination of how to understand and mathematically express the presence and character of visual artifacts commonly found in printers. This is an on-going research initiative at many companies, including Xerox. Most of the modeling in the SBBI system came from previous research initiatives[2,3,19]. The bulk of the work described in this paper is the development of a simulator that incorporates these models to simulate a print engine.

The intent of this project was *not* to model the physical behavior of various print engines. To do so would require the modeling of the actual hardware at an extremely fine level of granularity. For example, the system might have to model the electrostatic forces that hold the toner particle to the paper prior to fusing. The number of variables and parameters of such a model would exceed the scope of this initiative. While such research efforts have been undertaken in other segments of Xerox in the past[2], the goals of this system were much more macroscopic. The intent of the SBBI system was to develop a tool for determining the print quality requirements of Xerox customers. To the

customer, the scientific explanation for visual artifacts is not as important as the fact that the artifact exists at *all* in the output image. Therefore it is sufficient for the system to present to the customer a realistic interpretation of the output image, without concern for modeling the mechanical and physical processes that created the image.

1.4 Users and Customers

The *user* is the individual who sits at the computer console and interacts with the SBBI system. Presumably, the user understands the visual artifacts and what design considerations might give rise to them. The *customer* is the person, group, or organization that needs a new Xerox print engine. The user shows the customer sample prints incorporating each of the visual artifacts and aids the customer in understanding each artifact and how each artifact can be minimized (and at what cost). The customer then provides the user with feedback detailing what level of each artifact is acceptable. Finally, the user passes this information on to the engineering department, where these requirements are combined with the requirements of other potential users and resolved into formal design specifications.

Chapter 2

History and Perspective

This chapter provides the reader with a historical perspective on the SBBI system. The use of modeling and simulation as scientific and business tools is discussed, followed by an explanation of the specific Xerox initiatives that eventually lead to the SBBI system. Finally, an overview of the SBBI system is provided that highlights its unique advantages over these previous initiatives, and introduces the reader to its features and capabilities.

2.1 The History of Modeling and Simulation

Modeling and simulation go back to a time long before modern computers. Scientific modeling dates back to the very beginnings of scientific thought. In fact, science itself is the effort of mankind to apply reason and mathematics to describe, in an orderly fashion, the workings of the world around us. Every scientific theory or equation ever expressed is a *model* of some aspect of our universe. For example, Ohm's law is a *model* of the behavior of a simple electronic device known as the resistor. (Ohm's law states that the voltage across a resistor, V, is equal to the product of the current passing through the resistor, I, and the resistance of the resistor, R. See Figure 2-1).

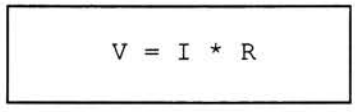


Figure 2-1
Ohm's law as a scientific model

Models are merely predictions and approximations of the behavior of some object or system. Before a model can be accepted, it must be tested. Historically, this has been done using one of two techniques: experimentation or simulation. *Experimentation* is the direct study of the actual object or system to see if it corresponds to the model. For example, to determine if Ohm's law does in fact describe the behavior of simple resistors, scientists have measured the three variables expressed in Figure 2-1 and checked the results against the model.

However, experimental verification is not always practical. For example, it is difficult to test models about the behavior of particles on the edge of a black hole, because humans cannot yet reach black holes. It may be too dangerous to study other models, such as those about the conditions inside a tornado's vortex. Still other times, experimentation may be too costly or slow. For example, it is not economically desirable to build complete airplanes in order to test models of airplane design and flight.[18] Out of these needs was born the practice of *simulation.*

Simulation is the exercise of a model without direct use of the object or system the model represents. Simulation does not have to be carried out on a computer (storms can be simulated in a wind tunnel, and dummies simulate human beings in automobile safety testing). However, computers have become an increasingly valuable tool for conducting simulations. Computers bring a wide range of benefits to bear on simulation problems. Some of the benefits of computer simulation, summarized in Figure 2-2, have already been identified in other research efforts[4].

1. Manpower savings over traditional methods.
2. Study may not be feasible any other way.
3. Savings in materials (no consumables).
4. Faster answers, more consistent results.
5. Increased flexibility (software is malleable).
6. Increased accuracy.
7. Increased range of operation.
8. New results not available before.
9. Improved results (due to standardization).
10. Increased understanding of the system modeled.
11. Explicitly-stated assumptions and constraints.

Figure 2-2
Advantages of computer simulation over traditional methods

Every computer simulation is designed to meet some objective. The objective is the ultimate answer (or answers) the user hopes to acquire with the aid of the simulator. The answer might be as simple as a test of Ohm's law, or as complex as a study of the feasibility of landing a space craft on Mars. The objective of most simulation experiments belongs to one of four major categories[5]. These categories are shown in Figure 2-3. The SBBI system is an example of the first category of simulation, because it is designed to answer *what if* questions regarding print engine designs (e.g., "*what* effect would it have on the customer's perception *if* the frequency of the banding is increased?").

1. Comparison of a finite number of strategies regarding an individual real-life problem.
2. Simulation for developing functional relationships.
3. Use of simulation for validating and evaluating newly developed analytical methods.
4. Use of simulation for educational purposes.

Figure 2-3
Objectives of computer simulation

Computer simulations have existed since the earliest days of digital computers, but it was not until the 1960s and 1970s that they began to take on a prominent role as a viable tool for science and business. While basic simulation principles had been applied for decades, it was not until this time that computers grew powerful enough to handle complex models. With the advent of these new computers came the possibility of extracting scientific and business value from computer simulation techniques. One of

the first uses of computer simulation was as a business tool for corporate America[5]. Executives began to use computer simulations to help determine the optimal way to spend time, money, and resources. The software gave them the opportunity to explore business scenarios completely without concern for waste or failure. In this way the executive could determine the option most likely to yield the desired results.

The use of computer simulation as an engineering tool has also been evolving for many years. Engineers specialize in building real-world systems from precise mathematical descriptions. These mathematical descriptions constitute models of the actual system. When incorporated into computer simulations, these models provide a valuable tool to help engineers test and develop systems. Simulation can be used in every step of the development process to ensure that the final product is evolving to meet the original specifications. The Boeing Corporation has used simulation effectively to design complex airplane surveillance systems[10] as well as to develop large portions of the 777 aircraft[18]. Studies have proven that simulation can detect problems early in the design phase, reducing cost overruns and product delivery delays[11]. Some engineers have used computer simulations to develop systems with optimal designs[12].

The SBBI system is different from such studies in the *way* that it brings simulation to engineers. Rather than tell them *if* they are doing their job right, the SBBI system is designed to help them determine *how* to do their job right. In this case, doing the job right means designing and building a print engine that meets the customer's requirements for print quality the first time. Thus, the SBBI system is a proactive, rather than reactive, engineering tool.

2.2 Precursors to the SBBI system

The SBBI system benefited from a legacy of simulation tools applied to engineering tasks[2,3,13,14,15,16,18]. Xerox has been relying increasingly on computer simulations to reduce cost and solve problems for several decades. This section describes several of the studies and programs initiated previously that contributed to the design of the final SBBI program.

2.2.1 Human Vision Studies [7]

"A picture is worth a thousand words."
- Anonymous

Appendix D shows a series of prints with a progressive amount of *banding* introduced. (Recall that banding is introduced when the paper does not move at a uniform speed through the print engine.) The first print, labeled **Banding Example: f = 100 a = 0**, shows the base print as it might appear from a perfect copier (in this case, **f** refers to the frequency of the banding, and **a** refers to the amplitude). The second print, labeled **Banding Example: f = 100 a = 2**, introduces a small amount of banding to the

image. Notice the impact that even this small amount of banding has on the overall perceived *quality* of the print. This is a result of the way humans see.

Human vision is only now beginning to be understood. While humans enjoy the benefit of five senses, up to 90% of human perception comes directly from sight. For a company like Xerox, whose core business is printing and imaging, this percentage is crucial. Humans are capable of discerning extremely small discrepancies in visual material.

The level at which visual deviations can be detected, and the affect of those deviations on the viewer's opinion of the image, are highly subjective. Not all viewers react the same way to the same visual information. These divergent perceptions can be attributed not only to the differences between the subject's brains, but also on the intended use of the image. For example, it may be acceptable for a print destined for a family album to have some small amount of graininess. However, the same level of graininess may be unacceptable for a high altitude military reconnaissance photograph, or the cover of a fashion magazine. For this reason, it is critical to incorporate the customer's subjective reaction to sample prints before the final product goes into production.

2.2.2 The Xerox Image Processing Toolset[14]

The Xerox Image Processing (XIP) toolset* is a set of image processing tools developed at Xerox. The XIP tools come in two standard varieties: executable libraries that can be linked with user programs (usually written in C, although any program that follows the C calling convention could presumably be linked with XIP modules), and stand-alone executable programs (capable of being incorporated into a UNIX™ shell script). This makes it possible to build programs utilizing XIP in either compiled (C) or interpreted (UNIX shell script) format. XIP is both a collection of existing imaging routines and a platform for the development of new imaging routines. The XIP libraries contain functions for reading and writing images in a variety of formats and translating them into a simple, easily manipulated, internal format. XIP routines have been written to introduce each of the visual artifacts supported by the SBBI system[3], as well as many others. XIP calls are usually sandwiched between two other calls, one that reads data files in some external format (such as gif, tiff, or jpg) and produces a new version in the internal format; and one that does the reverse. Figure 2-4 illustrates a simple XIP UNIX shell script. The XIP system is an on-going research project within Xerox.

* Several of the toolsets used by the SBBI system are still under active development at Xerox and are considered *Xerox private data*. To protect the company's interests, the names of these toolsets have been altered for the purposes of this discourse. The following toolsets, referenced in this paper, are known by different names within Xerox: The Xerox Image Processing Toolset (section 2.2.2), the Xerox Image Metrics Toolset (section 2.2.3), the Xerox Color Calibration Toolset (section 2.2.4), the Xerox Image Simulation Environment (section 2.2.5), and the Modular Simulation Package (section 2.2.6).

```
read_jpg foo.jpg | scale 10.0 | rotate 90.0 | write_tiff foo.tif
```

Figure 2-4
Sample XIP script

2.2.3 The Xerox Image Metrics Toolset [15]

The Xerox Image Metrics (XIM) toolset is a set of imaging routines developed at Xerox for studying the quality of printed images. The toolset is designed to objectively measure the quality of a prints according to certain predefined metrics. The XIM package provides distributed services, as well as a library of metrics routines. The SBBI system does not currently use XIM, although hooks for its introduction are included. XIM could be used, for example, to perform the translation of subjective customer comments into objective print quality metrics, or to produce a self-tuning system (see section 4.3).

2.2.4 The Xerox Color Calibration Toolset [16]

The Xerox Color Calibration (XCC) toolset is a set of routines for handling color calibration. The toolset provides routines for translating color schemes from one encoding into another, and producing standard color calibration images suitable for various classes of printers. The SBBI system uses XCC features to perform color separation and management. The XCC tools read and write the same internal format graphics files as the XIP routines.

2.2.5 The Xerox Image Simulation Environment [8, 9]

Several years ago Xerox Webster Research Center (now known as the Joseph C. Wilson Center for Research and Technology) began work on a unifying infrastructure for the modeling and simulation of image output terminals (printers and copiers). The goal of the system was to develop an environment of hardware and software that would become the basis for all future models and simulations of print engines within Xerox corporate research. That simulation environment was not applicable to this particular project, since it was directed at studying *physical* characteristics of print engines. The system was designed to facilitate the study of *how* the toner is laid down on the paper, and timing the movement of the paper from component to component. Because of the extraordinarily fine level of detail involved in such simulations, a great deal of processor power and time may be required (depending, of course, on the actual model and the size of the image). For example, a typical Sun Sparc™ computer might require an hour or more to model the printing of an image a few square millimeters in area.

The goal of the SBBI system was to provide a viable engineering tool for the analysis of customer print quality requirements. This necessitates the creation of full-

page images. The system is designed to be capable of rapid feedback and analysis. For example, the SBBI system provides an on-screen preview mode that allows the user to experiment with the values of properties before producing the final set of prints. If the SBBI system is to be responsive under such circumstances, it must be capable of producing output images within a few minutes to an hour. The Xerox image simulation environment was not intended to handle this type of rapid simulation.

While these tools were not ultimately incorporated into the SBBI system, the Xerox Image Simulation Environment project does illustrate the fundamental importance Xerox places on using simulation as a means of adding value to research and development. In this regard, it laid some of the groundwork for later applications of computer simulation, including the SBBI system.

2.2.6 The Modular Simulation Package [20,21]

A FORTRAN program was developed at Xerox in the early 1980s for modeling print engines. The system was designed to allow the user to plug in FORTRAN modules that described the behavior of various mechanical subsystems. These models mathematically described the inputs, outputs, and operation of the subsystem. The system was designed to configure itself at startup and to dynamically attach each subsystem that was to become part of the simulation.

Like the Xerox Image Simulation Environment, the Modular Simulation Package was designed to simulate the system at a physical level. For this reason, it required a long time to execute. Furthermore, because the models were written in FORTRAN the system would not have integrated well with the UNIX and C libraries used by the SBBI system. The Modular Simulation Package and the SBBI system are similar, however, in that both utilize self-configuring elements to enhance system operation (see section 2.3).

2.2.7 Thermal Ink Jet (TIJ) Printhead Simulation [2]

A computer simulation was developed at Xerox in 1992 for the study of thermal ink jet (TIJ) printheads. The simulation centered on a problem quality testers were having with some initial versions of a thermal ink jet printhead. A computer model was developed to mimic the behavior of the printhead in order to study the problem.

The TIJ simulator was designed to read an input image and convert it into standard XIP internal format. In the process, the image would be rasterized and some color correction would be applied. The system would then apply several custom XIP routines to simulate the operation of the TIJ printhead. The resulting images were stored and printed for analysis.

The images produced by this simulation proved useful in determining the cause of the malfunction. Eventually, the solution developed with the aid of the simulator was

incorporated back into the production version of the printhead. While this project was originally begun to study a *specific* problem, the success of the effort lead to a more general approach (see section 2.2.8). Many of the elements of this first simulation were carried over into this next version, and from there into the SBBI system.

2.2.8 Full-Page Image Simulation [3]

The full-page image simulation system was developed as a continuation of the thermal ink jet printhead simulation (see section 2.2.7). The intent of this effort was to produce an environment for modeling the behavior of image output terminals. Unlike the thermal ink jet printhead simulation, which modeled the behavior of each and every drop of ink placed on the page, the full-page image simulation system was designed to model only the *appearance* of the final page. The goal was to create a system to aid Xerox in measuring and quantifying a potential customer's image quality requirements early in the development stages, before significant resources had been committed to the product[3].

In intent, the full-page image simulation system, which was the immediate predecessor of the SBBI system, was similar to the SBBI system. Both systems rely upon UNIX shell scripts, which call XIP and XCC imaging routines, to produce prints used to determine customer print quality requirements. The similarity, however, ends there. The SBBI system is more than just a new user interface applied to an old architecture, it is a fundamentally different architecture with unique features and capabilities.

2.3 The SBBI System - An Overview

In the past, Xerox has built prototype machines to demonstrate print engine designs, and the effect of these designs on print quality, to potential customers. This process is expensive and time-consuming. While no study has ever been carried out to determine *exactly* how much money Xerox has spent building such machines, estimates range from hundreds of thousands to millions of dollars. To reduce this cost and quicken the pace to product, a software simulation was required to aid Xerox in understanding the print quality requirements of potential customers *early* in the development cycle. The SBBI system is intended to fulfill this role.

The operational goals of the SBBI system are shown in Figure 2-5. The first goal reflects the impetus for the entire project. The system was designed to fill a business need; and if it cannot fill that need, then it is a failure. The second goal states that if the new system is no faster or cheaper than the old system, then it has no value to the corporation. The third goal reflects the fact that this system is intended for use by individuals who may not be fluent in UNIX or C; and if they can't understand it, then it is useless to them as a tool. Chapter 4 will return to these points to examine how well the SBBI system has met each of these goals.

```
1.  Provide a viable tool for quantifying customer print
    quality requirements.
2.  Provide business value to Xerox.
3.  Provide a user-friendly environment.
```

Figure 2-5
Operational goals of the SBBI system

The SBBI system is a true client/server program featuring a small server and a full-feature, self-configuring client. This section provides an overview of the key operational and architectural features of the system.

The most prominent operational and architectural feature of the SBBI system is the client/server structure of the program. This was a desirable feature because of the processing requirements of image processing. The images which the SBBI system manipulates are large, often exceeding 100 MB in size. Each such image has millions of pixels, and each pixel may consume many bytes of storage, depending on the color depth of the image. The effective processing of such images requires a computer with a high speed CPU and many megabytes of memory and disk space. While such computers exist within Xerox, they are not common. The client/server architecture of the SBBI system allows the server, which handles all the image processing, to reside on a remote computer. In practice, this remote computer is usually a high-power workstation well equipped for image processing. The client, which provides all the user interaction, resides on the user's workstation. This reduces the computational requirements of the overall system and makes it available to a wider base of users.

The use of a client/server architecture is not without complications. The SBBI system requires that the server be able to access input images. If the server is running on a remote computer, these images may not be accessible to it. In order to operate transparently, the client computer and server computer must *share* the directories where the images are read and written. This means that one or both computers must access the directory over a network such as NFS (the Network File System) or AFS (the Andrew File System). As a practical example, suppose the input file is located in the directory /tmp on the client computer, which is a local directory not imported or exported over the network. When the server attempts to read the file, it will attempt to locate it in /tmp on the server's computer. This would cause the server to generate an error message. The correct way to use the SBBI system is to locate the files in a common, shared directory. Many sites, including Xerox and RIT, NFS mount user's home directories to a collection of workstations, so that users can log in from any workstation and access the same files. Such directories are acceptable locations for input and output files, since both the client and server machine can access the same files with the same path names.

The server uses standard UNIX sockets to communicate simultaneously with one or more clients. The server has been designed to perform all of the work necessary to make the client connection seamless. To the user, the server appears to be running on his or her local computer, although in reality it may be running on a computer miles away. The server is designed to detect and catch errors (including those generated by UNIX when the shell script executes) and pass appropriate information back to the client for processing. The server also automatically change directories and user ids to match those of the client, so permissions and relative path names work as the user expects.

The operational metaphor that the client provides is both simple and intuitive. At its core is the concept of artifacts and boxes. The user views the artifacts as being located in one of two boxes: a box that contains all the available artifacts and a box that contains the artifacts that the user wants to apply to the input image. Conceptually, the box containing all the available artifacts has an infinite number of each artifact. The goal of the user is to move artifacts *from* the box with all the available artifacts (the available artifacts box) and *into* the box of artifacts to be applied (the applied artifacts box). To help the user do this, the system provides a small set of simple operations for moving, deleting, and rearranging artifacts (see Figure 2-6).

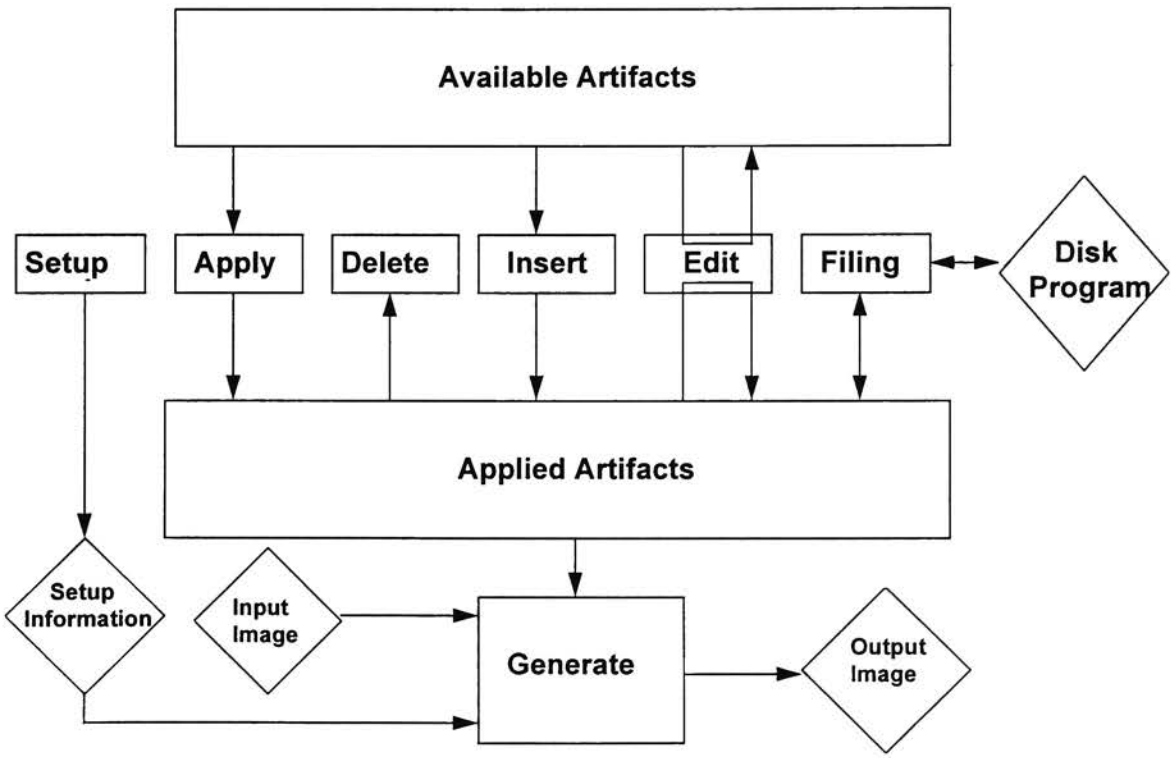


Figure 2-6
An architectural view of the SBBI system

Artifacts are only allowed to flow *from* the available artifacts box *into* the applied artifacts box. At no time do artifacts move in the opposite direction, from the applied

artifacts box into the available artifacts box. Because the available artifacts box can be viewed as holding an infinite number of each type of artifact, there is no need to move artifacts back into it. Similarly, when artifacts are deleted from the applied artifacts box, they do not need to be saved, so they can be discarded. The process of moving an artifact from the available artifacts box into the applied artifacts box is called **apply**, because it causes that artifact to be *applied* to the output image.

The only time an artifact leaves one box and returns to the same box is during an **edit** operation. This allows the user to not only modify the values of properties in the applied artifacts box, but to set new default values for properties in the available artifacts box. Thus, once the user has marked the value of a property as some value, say 10.0, it will remain 10.0 each and every time that artifact is selected until the next time it is changed. The edit operation is invoked implicitly when the user selects an artifact from either box. The system does not provide a means to restore the original default value of the property automatically once it has been changed.

A **delete** operation is provided to allow the user to remove an artifact previously applied to the image. This causes the artifact to be removed from the applied artifacts box and discarded. The system provides no means of undeleting an artifact, or for deleting an artifact in the available artifacts box (short of editing the configuration files, see Appendix A). A **delete all** operation is provided for removing all the applied artifacts with one operation.

An **insert** operation exists for placing an artifact into the middle of the applied artifacts list. This is an important feature because the order of application of artifacts can be significant. In this sense, the applied artifacts can be thought of as a *program* whose processing is directed by the selection of artifacts. Taken together, the apply, insert, and delete operators allow the user to control not only which artifacts are applied, but also the order in which they are applied.

The **filing** operation can be used to take all the artifacts in the applied artifacts box and place them into a disk file. These disk files can then be read back in at a later date. In effect, this allows the user to save a *state* of applied artifacts and retrieve this state at will. As Figure 2-6 illustrates, the filing, insert, and delete options can only be used to modify, save, and restore the contents of the applied artifacts box, not the available artifacts box.

The **setup** operation allows the user to specify the names of the input and output files, as well as what form the output file should take (on-screen preview or PostScript). From here the user can also specify if the output file should be compressed, and the scaling factor to apply to the image when the output mode is preview (this allows the user to scale the image so that the processing will take less time and so the image will fit on-screen). Setup is the only operation that does not manage the movement or configuration of artifacts.

To produce the output image, the user utilizes the **generate** operation. This causes the system to read the input image, apply the artifacts, and generate an output image. This output image can be displayed on screen or sent to a PostScript printer. The setup option allows the user to specify the format of the output file.

The SBBI system has the ability to generate multiple prints with a single run. This batch mode of operation is an important convenience when generating a series of prints. The SBBI system allows the user to set the "value" of one or more (numeric) properties not to a *single* value, but to a *range* of values. To use this feature, the user must enter three pieces of information: the start value, the stop value, and the number of prints. The system will then generate a series of prints, incrementing or decrementing the properties as needed for each print. For example, the user can specify that some property, call it **amplitude**, should have a start value of 1 and a final value of 5. The user then tells the system (in the setup box) to generate three prints. This will cause three prints to be generated, the first with an amplitude of 1, the next with an amplitude of 3, and the final with an amplitude of 5. If more than one property is configured in this manner then *all* the properties will be updated with each iteration of the loop. The system provides no mechanism for handling the *nesting* of these loops.

The SBBI system provides the user not only with a predefined set of artifacts, but minimum/maximum and default values for each property associated with that artifact. This type of design relieves the user of the burden of memorizing artifacts and properties, and prevents him or her from entering invalid values for properties.

One of the most useful features of the SBBI system is the self-configuration that both the client and server undergo at run time. Under most client/server applications, fundamental changes to the system require the editing and recompilation of both the client and the server. Under the SBBI system, the most common type of change is the inclusion of a new artifact or property, or the editing of an existing artifact or property. To simplify this task, both the client and server read ASCII configuration files that allow them to build an internal database of artifacts and properties. The client's database allows the client to determine the name of each artifact, along with the name, type, default value, minimum value, maximum value, and help information for each property. The server's database allows the server to construct the parser required to build shell scripts. Chapter 3 discusses the technical details of self-configuration. Appendix A discusses how to modify the configuration files to alter the behavior of the system.

The SBBI system has been designed to shield the user from the need to know *how* the imaging is done (see Chapter 3) and the details of each artifact. The layout of the interface has been designed to reflect the operational metaphor (Figures 2-6 and A-1). The available artifacts are listed in the left hand box, while the artifacts are listed in the right hand box. Artifacts conceptually move from left to right. The buttons which facilitate this movement lie in the middle of the screen, between the two boxes. Each operation has a separate and clearly-marked button associated with it. The system provides instructions at the bottom of the screen, as well as status messages below the

instructions. These status messages are designed to orient and guide the user through the operation of the program. Finally, help information for each property can be provided. Each property in the configuration file has an optional component containing the path name of an ASCII file containing help information for that property. When the user selects a property and clicks on the help button, the system will start up a simple text editor to display the contents of the corresponding help file (if it exists).

Figure 2-7 summarizes the key operational and architectural features of the SBBI system. The final product builds on the strengths of the ink jet printhead and full-page image simulations, while adding features and capabilities never implemented in either of these systems. The new additions make the SBBI system not just a valuable research tool, but a viable engineering and business one as well.

Feature	**Benefit Analysis**
Client / Server Architecture	Reduces computing resource requirements.
UNIX Socket IPC	Portable to wide range of platforms.
Error Handling	Catches user and programming mistakes. Displays meaningful error messages, even if the error originated on the server side.
Simple, consistent metaphor	Easy to learn, easy to maintain.
Filing	Applied artifacts (state) can be stored and retrieved
Automatic generation of multiple prints, with varying property values	User can generate a controlled series of prints in one operation.
Default values, range checking, and help information for properties	No memorization of property names, types, permissible values, or meanings required.
Self-configuration	New artifacts can be added more easily.
Help information	No memorization of cryptic property meanings

Figure 2-7
Architectural and Operational Features of the SBBI system

Chapter 3

The Design and Implementation of the SBBI System

Xerox has invested a significant amount of time and money providing hardware and software tools for image processing. These tools are well-designed and well-tested, but not always user-friendly in practice. The use of these tools in research efforts like the full-page image simulation project required the user to create and maintain complex shell scripts by hand. The user was often required to memorize cryptic arguments to XIP and XCC programs. From the development standpoint, it was desirable to leverage the strengths of these tools while adding new features to make the system more user-friendly and easier to extend. Four key design goals for the SBBI system were identified early in the development effort (see Figure 3-1). These goals constituted the foundation for the development of the SBBI system.

1. Leverage existing hardware and software
2. Integrate existing tools with each other and the rest of the system.
3. Permit the easy inclusion of new artifacts and properties.
4. Provide a user-friendly environment.

Figure 3-1
Design goals of the SBBI system

3.1 Assumptions

The SBBI system has been designed to be adaptable to changing business needs. It was necessary, however, to identify those assumptions that were required in order to fix the initial design and work toward some recognizable final configuration. Otherwise, the development of the system could become an open-ended effort, with no constraints regarding where the effort should begin or end.

3.1.1 Build for the trained user

The user should understand the basic principles of modern print marking engines. In particular, the user must understand the nature of the artifacts that the program uses. This is required to explore with the customer (who in all likelihood is *not* the user) each of the types of artifacts, what causes the artifacts, and the consequences of removing or reducing the artifacts. For example, the user should be able to explain to the customer that banding is formed by non-uniform movement of the paper through the print engine,

causing some bands of toner or ink to fall too close to the adjoining bands, and others to fall too far apart. This effect can be minimized with certain types of designs, but the consequence may be a higher cost and/or an unacceptable increase in one of the other artifacts. The user must be able to explain these considerations to the customer to determine the acceptable tolerances of *all* appropriate artifacts.

3.1.2 Leverage existing components

The system should utilize existing Xerox imaging tools. These tools are well documented and highly optimized for this type of imaging. In particular, it should leverage the Xerox Image Processing toolset and the Xerox Color Correction toolset. These programs are ideal because most of the required routines are already present in a stable and optimized environment. In addition, these tools have been designed to accommodate the inclusion of new routines, such as might be required to implement additional SBBI visual artifacts.

3.1.3 Standard platform

The system should run on Sun workstations running MIT's X Window System™ version 11 and Sun's *Open Windows*™ window manager. This platform was chosen because Sun workstations are readily available within Xerox and the X Window System is portable to a wide range of alternate platforms. In addition, many of the tools used by the SBBI system run under X Windows and Open Windows, including the text editor used for displaying run logs and the XIP display program used to handle on-screen previews.

3.1.4 Standard Language

It is assumed that this product will be in use for some time. The decision was made, therefore, to do the development in the C programming language (ANSI C). While C is not the most expressive language for this type of application, it does have the advantage of being widely used and supported within Xerox. In addition, C is the most natural language to use for the development of UNIX and X Windows applications (because the UNIX and X Window System libraries are themselves written in C).

3.1.5 External Printer Service Vendors

The SBBI system produces output both on-screen and in PostScript files. These files can be printed on any PostScript device, or converted into other page description languages using appropriate tools. At times, users may wish to send these files to external vendors for lithographic (or other) printing. These vendors must understand that the presence of visual deformations is intentional and must *not* undertake to remove or minimize them. To do so would undermine the intent of the entire process.

3.2 A Brief Tour of the Development Process

Because this was a research project in the truest sense, it was not clear at the beginning what the optimal design would be. Accordingly, a rapid prototyping development style was adopted. Within a few days of initiating the project a prototype was developed that implemented the basic features of the SBBI system. Over the course of the next few months this version was modified and fleshed out to determine the optimal configuration of the final system.

3.2.1 Understanding the SBBI data structures

Simulations are designed to mimic the appearance and actions of real systems. To do so, the simulation must describe the system completely and drive the system properly. In programming terms, the description of the system is facilitated by the proper choice of data structures, while the operation of the system is facilitated by the proper program structure and organization. The design of expressive data structures, and convenient routines to manipulate them, is a critical first step in any simulation program[6].

As the exact number and type of artifacts in the final system is not fixed, the SBBI system needed a flexible and powerful way to manipulate these entities. At its most basic level, the system maintains an unknown number of artifacts, each with an unknown number of properties. A system of single linked lists was used to describe and manage this structure, with each artifact being tied to the artifact next to it in the chain, and each artifact having an independent chain of properties associated with it. Figure 3-2 pictorially illustrates this relationship, with As representing artifacts and Ps representing properties.

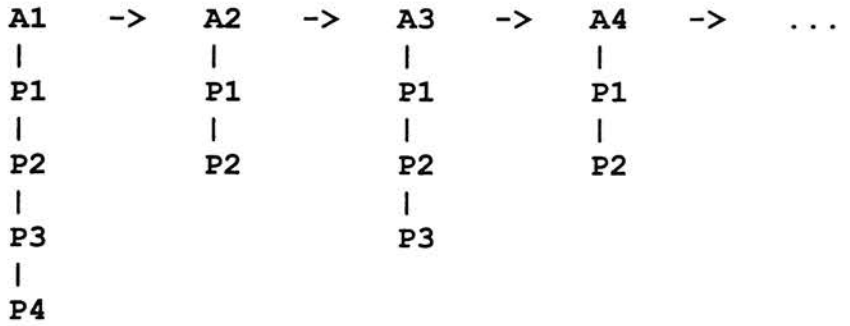


Figure 3-2
Visualizing Artifacts and Properties

In the first version of the program, a set of types and routines were adopted to create, manage, edit, and delete this data structure. Because this data structure contains single linked lists, the structure lends itself well to techniques such as recursive traversal. Figure 3-3 illustrates how this particular data structure can be easily managed using recursive programming techniques. The routines shown in Figure 3-3 can be used to

delete the entire list in one function call. Routines were also developed to look up each artifact and property by name and position and return pointers to the appropriate entity within the table.

```
void delete_matrix (ARTIFACT *a)
{
      void        delete_property (PROPERTY *);

      if (a != NULL) {
            delete_matrix (a->next);
            delete_property (a->property_root);
            free (a);
      }
}

void  delete_property (PROPERTY *p)
{
      if (p != NULL) {
            delete_property (p->next);
            free (p);
      }
}
```

Figure 3-3
C code fragment illustrating the manipulation of the SBBI data structure.

3.2.2 SBBI client/server design

Manipulating large images requires a great deal of memory and processing power. Many of the images that the SBBI system handles are hundreds of megabytes in size. The introduction of artifacts to the image may require fairly intense computations at each and every pixel. The problem is compounded if the introduction of the artifact requires floating point mathematics. Add to this the potential for generating several prints at a time and it's possible to estimate that execution times will run into days or even weeks. One way to address this problem is to run the image processing on a high-speed processor. However, not everyone at Xerox has (or can afford) that kind of computer. The client/server architecture of the SBBI system alleviates this problem by allowing the client (which provides all the user interaction) to run on one machine while the server (which does all the image processing) runs on another.

Ironically, one aspect of this first crude form of client/server architecture (which consisted of two processes communicating via a standard UNIX pipe) was actually carried over into the final product: the decision to pass information from client to server and back again as simple ASCII text. Originally, the intention was to implement some form of remote procedure call (RPC) to tie the client to the server in the final product. After further discussion, however, it was decided that ASCII communication through simple UNIX sockets was the most portable form of IPC currently available. Commercial socket packages are available for almost every major make of computer, but commercial

remote procedure call packages are more difficult to find. Since portability was a design consideration (see Figure 3-1), the decision was made to use ASCII client/server communication.

3.2.3 The switch to X Windows

The next major version of the system ran under X Windows and was developed using the *xview* toolkit. It was at this time that the system began to take on many of the characteristics it has today. The visual metaphor of presenting the user with two large list boxes (one showing all the artifacts the system understands and one showing those the user has chosen to apply to the image), along with a set of buttons for moving artifacts from one list box to the other, made its first appearance in this version. This version also presented users with a dialog box where properties could be edited, and used UNIX sockets as the IPC mechanism.

3.2.4 Shell Scripts

Most of the imaging carried out by the SBBI system is done using XIP and XCC toolsets. These toolsets provide two principle interfaces, compiled C libraries and executable, stand-alone UNIX programs. The SBBI system utilizes the latter. By building and executing shell scripts, the SBBI system can keep up to date with changes in these toolsets without requiring recompilation. Furthermore, the compiled library version of the XIP toolset is currently undergoing a *significant* overhaul. The library is expected to undergo specification, interface, and design changes that may render it incompatible with the SBBI architecture. Using shell scripts shields the SBBI system from these mechanical details. Finally, design reviews indicated that the performance gain that could be achieved by linking in compiled code (as opposed to calling compiled, stand-alone programs) was on the order of ten to twenty percent. These modest gains were not worth the additional development time required to convert the system.

3.2.5 Client self-configuration

The next version of the system featured a self-configuring client. The goal was to minimize the amount of coding required to implement new artifacts. To achieve this, a system was developed to allow the client to read the information about artifacts and properties from three configuration files: a system, a user, and a project. The system configuration file is stored in a special, shared location. This is the first configuration file read by the client. The user configuration file is stored in the user's home directory. This is the second configuration file read by the client. The project configuration file is stored in the current working directory. This is the final configuration file read by the client. Each configuration file contained the names of artifacts and a collection of associated properties. Each property contained a name, a type, a default value, a minimum value, and a maximum value. When duplicates artifacts or properties appeared in the configuration file, the contents of the new file take priority over the contents of the old

file. This design allows the SBBI owner to set up standard, default values. If individual users find these values unsuitable, they can override these defaults by creating an alternate configuration file in their home directory. Similarly, if the same user desires different defaults for different projects, he or she can place separate configuration files in various project directories.

3.2.6 Looping and Filing

The looping mechanism introduced in the next version allows the user to specify not a single fixed value for each property, but start and stop values along with the number of prints to generate. When the user clicks on the **generate output file(s)** button, the system will produce the specified number of prints, automatically incrementing (or decrementing) the value of these properties as it goes. Thus, if the user selects to vary one property from 1 to 9 and generates 5 prints, the first print will have a value of 1 for that property, the next will have a value of 3, the next will have a value of 5, and so on up to a value of 9. If the user requests 9 prints, then the artifact will go from 1 to 2 to 3 and so on up to a value of 9. This is useful for generating a set of prints that vary in some simple, linear fashion along one or more properties. Note that the user can enter as many variable properties as he or she wishes, but the system provides no mechanism for nesting these loops.

A filing feature was also provided to allow the user to save the state, or set of applied artifacts (those artifacts that are set to be applied to the image along with their properties), to a disk file; or retrieve a previously saved state. The initial implementation of this feature allowed the user to save the artifacts appearing in either list box, as well as merging artifacts stored in disk files with the current set of applied artifacts. These features were removed when user testing revealed that the distinction between applied artifacts and available artifacts, and the distinction between loading and merging, was too vague.

3.2.7 Server self-configuration

The final version of the SBBI system features a self-configuring server. The server reads information from ASCII files about the name of each artifact, and a description of how to form the appropriate lines in the shell script to handle that artifact. This description takes the form of a simple, C format style string enclosed in double quotes. Using this string as a guide, the server can build calls into the script to perform the image processing.

3.2.8 Cleanup and Testing

The remainder of the development time was spent cleaning up and debugging the system to prepare it for release. Several volunteers (see section 4.4) were allowed to experiment with the system and report questions, concerns, and bugs. Several minor

changes were made to the system during this stage, mostly designed to make the system more user friendly. Some of the changes included the simplification of the filing option (see section 3.2.5), the ability to correctly manage multiple instances of the same artifact in the applied artifacts list box, the inclusion of titles in the preview mode display, and the introduction of several new artifacts. The system was also extended to perform additional consistency checks before executing shell scripts (such as verifying the existence of files and insuring no place holder artifacts are in the applied artifacts list box when the user clicks on the **generate output file(s)** button). Status and help messages were added at this time to guide the user through the correct use of the SBBI program.

User testing indicated some concern about the way limits were implemented. Prior to this change, the system would check the values of numeric properties against the maximum and minimum values specified in the configuration files. In the event that a field was out of range, the system would adopt the appropriate minimum or maximum value as the value of that property and continue. Specifically, if the user value was less than the minimum value, then the value of the property became the minimum value. If the user value was greater than the maximum value, then the value of the property became the maximum value. Users commented that this silent change was confusing, so the system was extended to ignore the new value (leaving the value unchanged) if the user entered a value outside the acceptable ranges. At the same time, a dialog box is displayed warning the user of the error and informing him or her of the correct maximum or minimum value.

Finally, conversations with users lead to the inclusion of help information for each property. This information takes the form of an ASCII help file associated with each property. The client configuration file was modified to include a field for each property indicating the name of the corresponding help file. When the user needs help understanding the purpose of a property, he or she can click on the help button and the system will bring up a text editor to display the contents of the help file. The help file itself can be edited using standard tools such as vi(1).

3.3. The Architecture of the SBBI system

The SBBI system is divided into two primary conceptual units, the client and the server. The server runs in the background and performs all image processing requirements for any number of potential clients. The client runs on the user's desktop computer and drives the user interface. The two programs communicate with each other through UNIX sockets. Figure 3-4 represents the primary architectural features of the SBBI system graphically. The figure is not designed to show every conceivable relationship within the system, but to provide a visual aid in understanding the *major* subsystems and *primary* data flow patterns. The key in the upper right hand corner shows the meaning of each shape. Figure 3-4 clearly shows that only data is shared between the client and the server (that is, the interprocess communication routines pass ASCII data, rather than calling remote procedures - see section 3.2.2). The client passes the server the **artifact description language,** or ADL. The server returns to the client

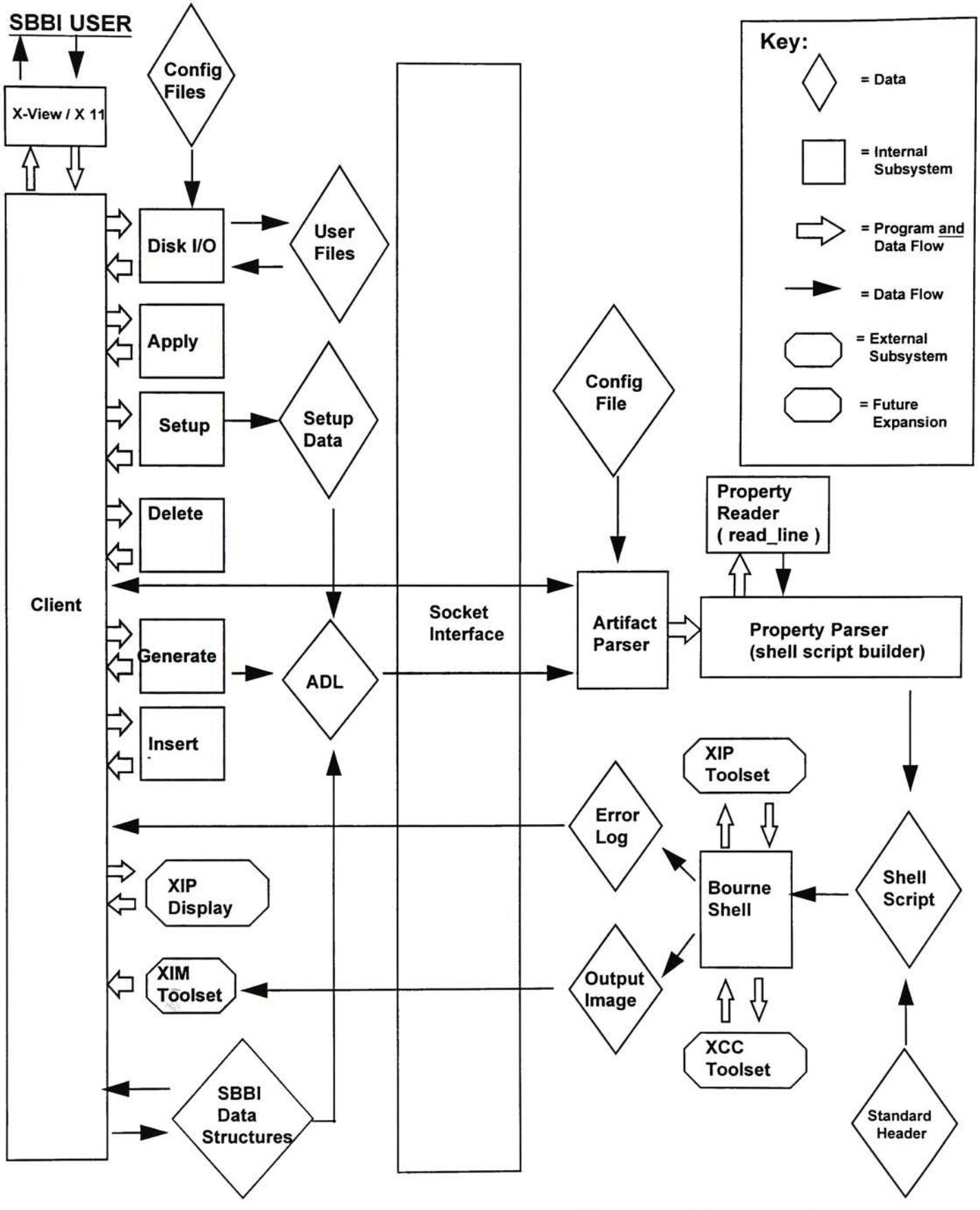


Figure 3-4

The SBBI System Architecture

two pieces of information, the output file and an error log file. In addition, greetings are exchanged at start-up in the form of *hello* messages (see section 3.3.2). This allows the client to verify that a server is present before beginning user interaction.

3.3.1 The SBBI server

Startup

The SBBI server runs as a UNIX daemon process. Upon startup, it performs a number of actions to prepare for the work it will eventually be required to perform. These actions are outlined in Figure 3-5. Once these actions are complete, the server waits in the background for a client program to connect.

1. Verify that the server is running with root privileges. The server requires the ability to switch to the id of the client user in order to operate. Otherwise, users could accidentally read and destroy the files and images of other users. For example, the user could arrange to make the output file /etc/passwd, which would have disastrous consequences. In order to switch to the id of the connecting user, the system needs to run as root.

2. Perform housekeeping operations to become a daemon. This includes calling the fork(2) system call to place itself into the background, changing to the root directory, ignoring BSD style job control signals, and removing itself from its tty associations and process groups.

3. Process the ASCII configuration file (see section A.5.2)

4. Open up a socket at a pre-defined TCP/IP port and listen for incoming connections.

Figure 3-5
Startup actions of the server

Client connections

Once a connection is established, the server uses the fork(2) system call to create a copy of itself. The new copy is responsible for handling the connection to the client. The original copy closes its connection to the client and returns to wait for a new connection. Without this action, the server would only be able to handle a single connection from a single client at a time.

The new server process now begins processing information from the client. The first line the server receives tells it the user-id of the client user, the name of the connecting machine, the current time, and other valuable information. Before processing any more information, the server uses this information to switch to the user-id of the connecting user. The server goes on to read some other parameters from the client, such as the client's current directory (used to locate the input files and place the output files in the proper location), the name of the input file, the name of the output file, the output mode, the scaling factor, and whether or not the user wants to compress the output images.

The security hole presented by changing user-ids based on ASCII information passed over a tcp/ip socket is considerable. Anyone familiar with the layout of the transmission from the client to the server could simply tell the server that his or her user-id is zero, thus gaining root access to a "programmable" server. The SBBI system addresses this concern in several ways. The server has been designed to refuse processing if the transmitted user-id is less than 100. This effectively protects the system accounts (including root). Furthermore, the server builds shell scripts with only a small set of predefined calls. Finally, the script written by the user is always set to mode 700, turning off the setuid bit. It should be noted that the SBBI system is a research prototype, and is designed for use in an open, trusted environment. While every reasonable effort has been made to ensure security, the possibility exists for security violations.

Next, the server places a copy of a standard header file into the shell script it is building. (The server performs its imaging by calling XIP and XCC modules in a shell script that it builds and executes). The standard header contains comments about the structure of the shell script (so it can be identified and edited later if desired) as well as critical environment variables required by the XIP and XCC routines. Note that the fact that the header file is stored separately means that the paths and environment variables can be changed later without requiring a recompilation of the server. This is useful because the SBBI system utilizes tools which are under active development at Xerox.

The name of each artifact received by the server is looked up in the database of artifacts built by the server during configuration. When the proper entry is found, the corresponding format string is copied to the shell script. This format string tells the server what to copy to the script to implement that artifact. The format strings used by the server is similar, but not identical, to the format strings used by the printf family of C library routines. The string is copied to the script verbatim *except* when a substitution marker is encountered. At that point, a corresponding property is read from the client and the value of that property is substituted into the script at that point. Table 3-6 shows the possible substitution markers and their meanings.

Marker	Meaning
%f	Read and insert a floating point property
%d	Read and insert an integer property
%s	Read and insert a string or filename property. This string will be enclosed in double quotes.
%b[string]	Read a boolean property. If that property is TRUE, then recursively substitute in the format string enclosed in braces.
%!b[string]	Read a boolean property. If that property is FALSE, then recursively substitute in the format string enclosed in braces.

Figure 3-6
Server format string substitutions

As an example, consider the following format string, associated with a fictional artifact known as *the snow effect*:

```
"snow -size %d -prob %f %b[-file %s]"
```

Furthermore, assume that the client has been configured to understand the *snow effect* artifact, and expects to handle four properties associated with that artifact. The first property is the size of the white dots (perhaps in pixels), the second is the probability that any given dot will be turned to snow. Assume the *snow effect* artifact can also output statistical information to a file for debugging, so the third property is a boolean indicating if the debugging file should be created, and the fourth property is the name of the debugging file. (How the client would process and present this information to the user is the subject of section 3.3.2).

When the artifact is transmitted to the server, the corresponding format string is located. Using this as a guide, the server begins writing a suitable line into the shell script. First, it places a pipe character as the first character in the line, to connect this artifact to the preceding artifacts. Then, the server begins copying the format string to the script. When the %d sequence is encountered, the server determines that an integer property is required and reads that property from the client. The value of that property is then inserted into the script and the server resumes the process. When the server encounters the %f sequence, it reads and copies into the script (as a floating point number) the value of the next property. When the server encounters the %b, it reads a boolean property from the client. If

this property is true, the server copies the section enclosed in braces to the output file (including further substitutions). If the boolean property is false, the server skips the entire string enclosed in braces. Note that the server correctly deals with nested braces, and will take care to read property values that are skipped (although they are not copied to the shell script). Finally, the server places a line continuation character into the script (\ ) and goes on to read the next artifact. Figure 3-7 illustrates several lines that could be generated in the shell script based on the *snow effect* format string.

```
| snow -size 4 -prog 0.5 -file "debug.out" \
| snow -size 10 -prog 0.01 \
| snow -size 6 -prog 0.99 \
```

Figure 3-7
Sample shell script lines

It is essential that the format string read by the server match the order and type of the properties transmitted by the client. This requires that the configuration files for the client and server be in synchronization at all times. Otherwise, the server might fail to read a property (causing that property to be read and parsed as an artifact) or may substitute a floating point property when an integer property was needed.

Running the script

Once the server has processed all of the artifacts, it writes a few lines of epilogue code to the script. These lines contain comments and a check to verify that the output file was written. The exit status of the script is one if the output file was generated and zero otherwise. The SBBI system then calls the fork(2) system call to create a new process for the script. After forking *but before actually calling the execl(2) system call to run the script*, the server does one more piece of housekeeping: it arranges to have the standard error stream redirected to a temporary file. Any error messages generated by the script is thus saved and can be displayed on the user's screen. In the event that an error occurs while executing the script, the client will open up a text editor to display a log file. This log file contains the entire contents of the script, the error message(s) generated by UNIX, and comments. Once the script has executed, the server returns the exit status to the client, cleans up, and exits. Appendix C contains a sample script similar to those generated by the SBBI program.

The SBBI server file naming convention

The SBBI server creates several files during its execution. Some of these files are designed to be persistent (meaning they are left behind in the file system even after the server and client exit) while others are transitory (meaning they are temporary and are usually removed automatically by the system). Figure 3-8 shows the file names that would be generated for a typical run where the user has entered an output image file name of **elvis**.

Filename	Purpose
elvis	Output image file (if no looping)
elvis-1 elvis-2 . . elvis-n	Output image files (if looping). Each successive file represents the output image from that iteration. Thus, elvis-1 represents the print generated from the first iteration, and elvis-2 represents the print from the second iteration.
elvis.run	Shell script file (if no looping)
elvis-1.run elvis-2.run . . elvis-n.run	Shell script files (if looping). Each successive file represents the shell script generated from that iteration. Thus, elvis-1.run represents the shell script generated by the first iteration, and elvis-2.run represents the shell script from the second iteration.
elvis.run.error	Error log file (if no looping)
elvis-1.run.error elvis-2.run.error . . elvis-n.run.error	Error log files (if looping). Each successive file represents the error log from the execution of the script for that iteration. Thus, elvis-1.run.error represents the error messages from the execution of the script created during the first iteration.

Figure 3-8
SBBI server output files

3.3.2 The SBBI client

Startup

Upon startup, the SBBI client initializes the *xview* toolkit by creating and positioning the dialog boxes, list boxes, edit fields, and buttons.

The system then attempts to locate and establish communication with the server. If the client is run with no arguments, the server is assumed to be running on the local machine. If an argument is given, the client uses this argument as the name of a remote machine and attempts to locate a server on that machine. The client establishes this communication by sending a *hello* message to the server. If no response comes back, or if the socket system calls return an error indication, then the client issues an error message to the user and exits. It is important that the client attempts to establish this connection before beginning user interaction, as there is no means in the user interface for connecting to another server once the program begins. Without this feature, the user might spend a considerable amount of time configuring artifacts, only to find that no server is present to generate the output image.

The client next loads the list of artifacts and properties from various system configuration files. These configuration files list each artifact, each property associated with that artifact, the type of the artifact, the default value for that property, the minimum and maximum values that the property can take on, and an optional UNIX path name to an ASCII file containing help information for that property. For more information on the structure and location of the configuration files, see Appendix A. Each artifact and property is read and copied into the SBBI linked-list data structure shown in Figure 3-1.

The client actually maintains two such lists. One maintains the state of the artifacts in the available artifacts list box, the other maintains the state of the artifacts in the applied artifacts list box. Both share the same basic structure: a set of artifacts with properties (each property having a specific value). The difference is that the available artifacts list is created at start up and never grows or shrinks (although values within it can change as the user enters new default values), while the applied artifacts list starts out empty but grows and shrinks depending on the actions of the user. Both lists share a common need to be able to look up and return pointers to, and information from, specific members.

The main window

The system now brings up the main window, which consists of an **available artifacts** list box, an **applied artifacts** list box, a set of buttons, and some directions at the bottom of the screen (see Figure A-1). This main window forms the primary interface to the client. Users configure artifacts they wish to apply to the image from the available artifacts list box, set the values of each of that artifact's properties, and apply (add) them to the applied artifacts list box.

The entire interface is designed to be simple and mouse driven. A status line at the bottom of the screen tells the user exactly what the system is doing at all times. Pressing the *help* key while the mouse is on any button, list box, or

menu item brings up a small help box. In addition, instructions are provided at the bottom of the main screen as an instant reminder to the user.

The system maintains a separate state (set of properties and values for each property) for each artifact in each list box, and allows the user to edit these values at any time. When an artifact is selected, the system locates and returns a pointer to the corresponding node in the list. This pointer allows the system to display and modify the contents of that node. The nodes are looked up by name and position. This allows the user to enter two copies of the same artifact in the applied artifacts list box, something that would not work if the artifacts were looked up by name only.

Each time the user selects a new artifact or property, the contents of the old property (if it was being displayed and edited) are copied back to that property and the new values copied into the **edit properties** dialog box. Values are also copied back out when the user selects **Generate output file(s)** and **Insert artifact**. This ensures that the state of the linked list is always in synchronization with the user's expectations.

The edit properties dialog box also provides a **Property Help** button. When the user clicks on this button, the system locates the currently selected property to retrieve the help file path and name. This information is loaded from the configuration file during startup. The help file is an ASCII file containing information about the property. The system will then start a suitable text editor (in read-only mode) to display the contents of the help file.

The operation of the **Apply this artifact**, **Insert artifact**, **Delete this artifact**, and **Delete all artifacts** buttons is straightforward. The **Apply this artifact** button makes a copy of the current artifact and appends it to the end of the artifact chain in the applied artifacts list. The **Insert artifact** button simply locates the last artifact selected from the applied artifacts list and inserts a copy of the last artifact selected from the available artifacts list at that location. The **Delete this artifact** and **Delete all artifacts** buttons recursively free the memory consumed by the appropriate node(s) and patch up the remaining pointers.

The main window has a **Filing** button. This button brings up a dialog box with a single edit field (for the file name) and three buttons: one for saving the currently applied artifacts, one for retrieving a previously saved set of applied artifacts, and a cancel button (to return the user to the main window). Note that the system does not support the saving and loading of the available artifacts list. This list does not change often and can be tuned using the SBBI configuration files. Each file is written out as an ASCII file identical in structure to the structure of the configuration files. This allows these files to be used interchangeably with configuration files and reduces the number of file formats the program must be capable of parsing.

Finally, the main window has a **Setup** button. The **Setup** button invokes a dialog box that allows the user to enter the input and output files, the output file type, the resolution of the output file, the number of prints, the preview mode scaling, and whether or not the user wants to compress the output file. These fields are designed to have sensible default values (for example, the system provides default input and output file names that include the current working directory, as this is the most likely place the user will look for the files).

While the creation and maintenance of these user interface objects constitutes a significant portion of the code, this code shall not be discussed at length in this paper. This is because they are, at best, mechanical details not critical to the understanding of the code. The user interface could just as easily been written using Motif, Xt, or even Microsoft Windows™ toolkits. Readers interested in the programming interface to these objects are invited to read one of the X Window System books listed in the bibliography.

Generating the output file(s)

When the user clicks on the **Generate output file(s)** button, the client prepares to send the currently applied artifacts to the waiting server. First, the client makes a number of consistency checks. It is useful to allow the client to make some of these checks as the client has direct and immediate access to the user's screen. Thus, the client can pop up status messages and warning dialog boxes to draw the user's attention to the problem. It is an unfortunate consequence of this design that the client ends up doing some of the initial work in the image processing stage, so the division between client duties and server duties becomes slightly muddied. This seemed an acceptable tradeoff to make the system more user-friendly.

The first thing the client does is save any properties from the edit properties dialog box. This is to ensure that any last minute changes to one of the applied artifacts is updated properly before the prints are generated. The system then checks the applied artifacts list for properties of type **FILENAME**. Those of type filename have a mode associated with them from the configuration file of either **READING**, **WRITING**, or **NO-CHECK**. Each of these is defined as a constant in the client header file. Those marked as files of type "reading" are checked to make sure they exist (before the server references them in a shell script). Those marked as files of type "writing" are checked to make sure that they do not exist, to prevent the server from accidentally overwriting them. Those marked as "no-check" have neither check performed on them. The system now verifies that the input file exists. This is done separately as the input and output files are not properties of specific artifacts. Finally, the system verifies that the number of prints is between 1 and 20, inclusive. The choice of 20 as a maximum number of prints is arbitrary but came from the realization that each print may

take hours to run, and more than 20 prints from a single run would likely be a typing error on the part of the user. These consistency checks are summarized in Figure 3-9.

```
1.  Save current edit properties
2.  Verify file names in the applied artifacts list.
      a.  Those of type READING must already exist.
      b.  Those of type WRITING must not already exist.
      c.  Those of type NO-CHECK may exist.
3.  Verify that the input file (from the setup dialog box)
    exists.
4.  Verify that the number of prints is between 1 and 20,
    inclusive
```

Figure 3-9
Consistency checks performed when the user generates output files

The system now determines if looping is enabled. To do so, the system determines that the **number of prints** field is greater than 1, and that at least one of the properties has a variable value (a start and stop value as opposed to a single value). Those properties that are variable are initialized to the start value to prepare for the first print.

The system now prepares to generate the required number of prints. For each print, the system first establishes a connection to the server. If the connection could not be established (for example, if the server died between the time the client first checked for it and now) then the error message "Could not connect to server" is generated in the status line and the routine exits, conceptually returning the user to the main window. If the client is able to establish the connection, it passes the server the information shown in Figure 3-10. This information is used by the server to switch user ids, move to the proper directory, and produce the shell script it will ultimately attempt to execute. Some of the information is not currently used by the server, but could be used to generate log files and statistical information about client and server usage. A [SETUP] and [SETUP END] statement pair bracket this header transmission.

```
1.  The hostname of the user's computer, the user-id of the
    user, and the process-id of the client.
2.  The input file name
3.  The output file name
4.  The output resolution
5.  The compression flag (on or off)
6.  The output mode (preview, final, or shell-script).
7.  The preview mode scaling factor
8.  The current working directory
```

Figure 3-10
The control information sent to the server

The client now sends the server each of the artifacts listed in the applied artifacts list. Each artifact, and its properties, is sent as an ASCII string. The client must send *every* property of each of the applied artifacts *every* time. This is important because the parser on the server side is relatively simple and expects a fixed number of properties with each artifact. While this reduces the error handling capability of the SBBI system, it makes the introduction of new artifacts and properties much easier. The server is under no obligation to *use* every property it encounters, only to *read* it.

The format of the transmitted artifacts and properties (known as the ADL) is similar to the format of the SBBI configuration files. Each property name is bracketed by curly braces. This is followed by a [BEGIN] statement, the properties, and an [END] statement. Each property takes the form shown in Figure 3-11.

```
[PROPERTY NAME] TYPE VALUE
```

Figure 3-11
Syntax of a property line in the ADL

Property name is the name of the property as found in the configuration files. The type is either BOOLEAN, INTEGER, FLOAT, STRING, or FILENAME. The value is the actual value of the property (if looping is enabled, this is the current value of the property for *this* iteration). STRING and FILENAME values are enclosed in double quotes so that spaces may be included. An entire transmission might look like the example shown in Figure 3-12.

```
[elf.wrc.xerox.com 100 4368]
[SETUP]
/tmp/input
/tmp/output
300
TRUE
1
0.1
/tmp
[SETUP END]
{Snow Effect}
[BEGIN]
[Size] INTEGER 2
[Probability] INTEGER 10
[END]
[DISCONNECT]
```

Figure 3-12
Sample ADL transmission

Once the client has transmitted all of the artifacts and properties, it waits for a response from the server. This response can be either **OK** or an error message. If the return status is OK, and the output mode is **preview**, then the client will invoke the XIP program **xipdisplay** to display the contents of the output file to the user. If the return status is OK, and the output mode is not **preview**, then the user may send the resulting PostScript file to any suitable printer. In the event of an error, the client will place an error message on the status line and bring up the Open Windows program **textedit** to display the contents of the log file for the user. This file contains the entire shell script, along with the output of the script (including error messages). This information can be used by the user to determine what went wrong and take any corrective action required.

If looping is enabled, the client now updates (changes) the value of each of the variable properties and begins the next print. Once all the prints are done, the client returns to the main window to await the next user request.

Xerox Image Metrics

The Xerox Image Metrics (XIM) toolset is a set of routines and metrics developed by Xerox for gauging image quality. Section 4.3 discusses the possible benefits of the inclusion of the XIM toolset in the SBBI system. The XIM system has not been incorporated into this version of the SBBI system, but hooks for its introduction exist and Figure 3-4 clearly shows where such a new subsystem would fit within the greater framework of the SBBI system.

Chapter 4

Conclusions

Figure 4-1 summarizes the development of the SBBI system. The system began as a set of existing but unrelated tools and technologies for managing image processing. These tools were tied together into a new, unified infrastructure and supplemented with extensions and new features to produce a tool uniquely suited for studying customer print quality requirements.

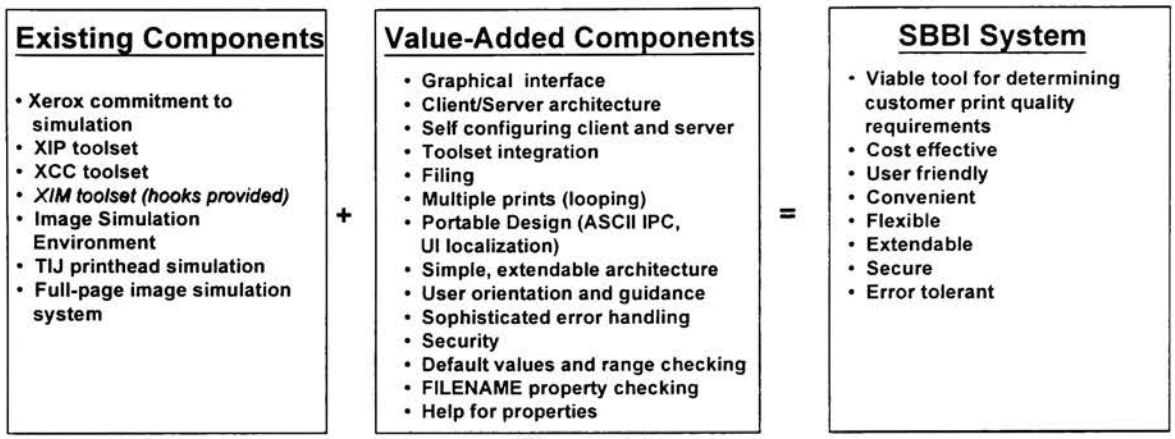


Figure 4-1
Summary of the development of the SBBI system

The success of the SBBI system can be measured from two perspectives, the user's perspective and the programmer's perspective. The user's perspective includes the items shown in Figure 2-5, while the programmer's perspective includes those items shown in Figure 3-1.

4.1 The User's Perspective

The SBBI system is well suited for measuring customer print quality requirements. The artifacts have been carefully designed to reflect the visual artifacts commonly found in prints from actual machines. The system provides a wide range of such artifacts, and can be easily extended to include more. The user can generate prints illustrating these artifacts to the user and can then solicit customer feedback on the prints. Working with the customer, the user can explain the factors that create these artifacts and determine acceptable levels for each. These requirements can then be submitted to the engineering staff where they can be translated into design specifications. For example, the user can issue a report to the engineers stating that the customer requires a banding effect of not more than two bands per inch. In time, the system can be expanded to include the XIM toolset, which will allow these subjective user responses to be translated

into objective measurements, eliminating the need for human intervention at this critical stage.

The system is user-friendly. The metaphor provided, both in architecture and operation, is simple and consistent. Artifacts are provided in a list box where they can be selected with a mouse click. The properties associated with that artifact can also be selected with a mouse click. The system describes the artifacts and properties in understandable terms, informs the user of the type of the property, and enforces range checking. The SBBI program also provides help and status messages to guide the user through the proper use of the system. The user can choose to view the image on-screen before committing to the final print, which can save time and effort. Help information for properties can be easily provided and retrieved. Finally, the system provides convenient short-hand notations for such common tasks as generating multiple prints and compressing the output image. Taken together, these features provide the requirements for an effective user interface[10]: orienting and leading the user, enhancing the productivity of the user, anticipating the needs of the user, allowing the user to focus on the task at hand, giving the user control, and encouraging the user.

4.2 The Programmer's Perspective

To the programmer, it is useful if the system utilizes existing tools to simplify its design and speed its development. The SBBI system leverages two principle toolsets, XIP and XCC. It also provides hooks for leveraging the XIM toolset. Both of these toolsets have been integrated into a single package that not only utilizes them transparently, but extends and enhances them by introducing new features and capabilities not available in either package.

The SBBI system can be extended. The client is self configuring and can be extended with a simple text editor. The server features a simple parser and table lookup that allows the easy inclusion of new artifacts. A template function has been provided for implementing new artifacts (see Figure 3-7).

The system is graphical and user friendly. It has been written to implement visually a close approximation of the architectural metaphor (the SBBI system provides two list boxes, with push buttons to move artifacts from one to the other). It has been written to run under the X Window System, making it not only graphical but highly portable. The system provides consistency and range checking, the ability to generate multiple prints with a single command, filing operations, and help messages - all of which makes the user's job easier.

The SBBI system is portable. It has been written entirely in ANSI standard C. The use of the X Window System means the system can run unchanged on computers from IBM PCs to high-end mainframes. It uses standard UNIX sockets and ASCII

messages to share information. The X calls have been isolated to a few functions within the client's source code to facilitate porting.

The SBBI system was designed to meet a specific business need within the Xerox corporation. The *ultimate* success or failure of the system lies in how well it fills that need. Negotiations are currently under way to transfer the SBBI system to an internal Xerox group specializing in studying customer needs and requirements. This group will test the program to determine its viability as a tool to aid in their efforts. The final result of their testing will not be known for some time.

4.3 Future Work

It would be useful if the SBBI software could handle the translation of subjective customer comments into actual design specifications. To accomplish this, two components would need to be added: the XIM toolkit (to translate subjective responses into objective metrics), and a system for translating these objective metrics into design specifications. The XIM toolkit would be called to process the output image and generate statistical information regarding the *quality* of the image. An expert system (written perhaps in Prolog) could be employed to handle the translation of this information into design specifications. This expert system would accept as input a database of facts regarding print engine components, the desired print quality information, and rules for the connection of components. The expert system could then suggest configurations that would yield the desired results.

The SBBI system could be ported to the C++ programming language. Such a change should wait until C++ becomes more widely used and accepted within Xerox than it is today, but it would hold unique potential to make the program more expressive. Graphical user interfaces are, by their nature, ideal candidates for object oriented design. The *union* used by the SBBI system to keep the track of the value of a property, along with an integer telling the system which of the fields of the union to use to access that value, is an awkward but common C work-around for the derived classes of C++. Under C++, a class could be created for each type of property, one which derives the features common to all properties and explicitly adds those features needed to implement that type. This would also make it easier to declare new types of artifacts that are similar to previous artifacts but with small differences. The programmer could simply derive a new class from the existing class and change the functionality of the key members accordingly.

The client could be ported to another windowing environment (such as Macintosh™ or Microsoft Windows™). This would require the inclusion of a suitable sockets library (either written in-house or commercially acquired). The server would not have to change to introduce this feature, and the changes to the client are limited to the user interface code (most of which is found in a single source file).

A version that utilizes the C library versions of the XIP and XCC toolsets would provide greater speed and accuracy. The introduction of this feature has been facilitated by the inclusion of conditional compilation flags in the server source code (see Figure 3-7). The change would require that the statements that produce the shell script be substituted with the building of a linked list of functions to call to perform the same work. Such a change should wait until the C source code for the XIP and XCC system stabilizes.

The processing of the configuration files could be handled using LEX and YACC. Currently, the SBBI system processes these configuration files using the strtok(3) library call. As the configuration files grow more elaborate and complex, the processing of these files might benefit from a more complete parser design.

The system could be changed to allow the user to select a server machine from the user interface (perhaps through the setup dialog box). This addition would allow the user to switch servers during a session, or connect to a new server if the original server went down. The system could maintain a list of known default server computers, which the user could then simply select with a single click of the mouse. The system could also provide an edit field, where the user could type in the name of a server machine.

4.4 Acknowledgments

The author wishes to acknowledge the assistance of the Xerox corporation, and the following individuals, for making this project a reality. Dr. Peter Anderson of RIT, Dr. Ted Retzlaff of Xerox (WCR&T) , and Dr. John Shaw of Xerox (WCR&T) for acting as advisors and project coordinators. Dr. Martin Abkowitz, Dr. Milan Stolka, Dr. Leonard Brillson, and Dr. Jim Larson of Xerox (WCR&T) for providing the flexible work hours required to see this project through. Dr. Homer Antoniadis and Dr. Howard Stark of Xerox (WCR&T) for support and guidance in getting this project started. Dr. John Shaw, Dr. Ted Retzlaff, and Dr. Howard Stark for testing the SBBI software. Mr. Robert Conway of Logical Operations for editorial assistance. Ms. Wendy Weigert, for support and encouragement from start to finish.

4.5 Dedication

This work is dedicated to my mother, Shirley R. Conway, who took so much pride in my education. Your gift will last a lifetime.

Appendix A

The Simulated Bread Board Imaging System User's Manual

A.1 Introduction

The Simulated Bread Board Imaging (SBBI) system is designed to allow the user to work with a customer to determine the customer's requirements before design and production begins on a new print engine. The goal is to understand the customer's image quality requirements completely, and to translate those requirements into formal design specifications.

The user of the SBBI system introduces *visual artifacts* into an input image to produce an output image that can be displayed on the screen or saved and sent to a printer (laser, ink-jet, or lithographic). Simply stated, a visual artifact is any deviation between the input and output images. These deviations might correspond to actual deviations introduced by various hardware configurations. The trained user introduces these visual artifacts to determine the range of values acceptable to the customer. For example, the user might show the potential customer ten prints, each with a different amount of graininess. The customer can tell the user which prints are acceptable for his or her application. The user then passes this information along to the engineering team, where these subjective comments are translated into formal design specifications. It is assumed that the user is familiar with the artifacts being introduced and can work with the customer to determine the optimal design criteria.

A.2 Starting the SBBI system

The SBBI system is actually made up of two separate programs. The first program, called *sbbi-server*, runs in the background on a host computer. This is the program that actually does all the image manipulation. The second program, called *sbbi-client*, usually runs on the local machine where the user is sitting. The two programs communicate with each other over the network. This type of design is called a client/server design. In this case, the program that interacts with the user is the client. The program that carries out work on behalf of the client, such as the image processing done by the sbbi-server program, is called the server. Such a design confers unique capabilities to the system, including:

- Image processing, which can be a slow and difficult process, can be carried out on a remote (possibly faster) computer.

- When porting the program to a new architecture, only the front end (the client) needs to be rebuilt. The back end is independent of the front end implementation (assuming the networking protocols don't change).

- One server process can connect to multiple clients, reducing resource requirements on the client side.

A.2.1 Starting the SBBI Server

The SBBI program is currently designed to only run under UNIX workstations, such as the Sun Sparc line of computers. Before running the SBBI client program, a copy of the SBBI server must be running on one of the local Sun workstations. The choice of which computer will host the server is arbitrary, but because the server does all the image processing it usually works best if a fast workstation is used as the server host.

Steps for running the SBBI Server

1. Log onto the machine which will act as the host for the SBBI server program.

2. Use the *su* command to become root (the SBBI server must be run by root). For more information on the *su* command, see the su manual page in section one of the UNIX manual (eg, "man 1 su"). The server will generate an error message if it is started by someone other than root.

3. Move to the directory where the SBBI server is located. Currently, that is

   ```
   /tilde/conway/thesis/bin
   ```

4. Run the program *sbbi-server*, as in:

   ```
   # sbbi-server
   ```

   Here, the pound sign (#) represents the UNIX prompt for the root user, which may or may not be the same on your machine. The program will place itself into the background and return control immediately to the user. If an SBBI server is already running on this machine, the program will give you the following error message:

   ```
   Sorry, there seems to be another process using the SBBI tcp/ip port.
   Is it possible you already have a server running?
   ```

   If you get such a message, you can either use the currently running SBBI server, or kill the current server and start a new one. Note that killing a

running server will disconnect any users connected to it, so use that option with extreme care. To kill a running server, use the **-kill** flag to the **sbbi-server** command line, as in:

```
# sbbi-server -kill
xxxxx: killed
#
```

The system responds by giving the process-id (a number, represented here as "xxxxx") followed by the message "killed". If you get the following message:

```
Cannot determine the pid of any running servers.
If there is one running, you'll have to kill it by hand.
```

then the system has been unable to locate the server that is causing a problem. (This can happen when a server program quits prematurely). In that event, seek the assistance of an experienced UNIX user or administrator.

5. Exit the root shell by typing "exit", as in:

```
# exit
```

On some systems, it may be necessary to use control-d to exit the root shell.

A.2.2 Starting the SBBI client

Once a server is running on the appropriate machine, the client can be started up. Note that the *sbbi-client* program requires X-Windows to run. X11R4, X11R5, Motif, and OpenLook are examples of the X Window system. Other Sun windowing systems may or may not be derivatives of the standard X11 window system. Contact an experienced UNIX user or administrator to find out if such windowing systems are compatible.

Steps for running the SBBI client:

1. Log onto the machine where you wish to run the client. Keep in mind that this machine must be the same machine that the server is running on, or connected to that machine via the network. Most Xerox internal Suns are connected to the same network.

2. Move to the directory where the SBBI client is located. Currently that is:

```
/tilde/conway/thesis/bin
```

3. Run the program **sbbi-client**, as in:

```
% sbbi-client &
```

Here, the percent sign (%) represents your UNIX prompt, which may or may not be the same on your machine. The SBBI client is usually run in the background explicitly, by placing an ampersand (&) after the program name on the command line.

This will attempt to connect to an SBBI server running on the local machine. To connect to a *remote* server, place the machine name hosting the server after the program name, as in:

```
% sbbi-client foobar &
```

Here, foobar is the name of the remote machine hosting the SBBI server.

One of the first things the SBBI client will do is attempt to locate and talk to the SBBI server (see section 3.3.2). If it cannot find a server, the following message will appear:

```
I can't find a server on machine "foobar".  To connect to a
remote server, specify the machine name on the command line, as in:

        sbbi-client elmo.wrc.xerox.com
```

In this event, either start a server on the intended machine (as the error message directs), or connect to a different server residing on another machine.

Note: The SBBI server is a standard X11 application. This means that like any X11 application, the program can be "run" on one machine and displayed on another. To accomplish this, set the environment variable DISPLAY to be the name of the machine where you want to display the program, followed by :0. For example:

```
%  setenv DISPLAY elf:0
```

Here, the system has been told to send the program's display to the machine known as **elf**. Assuming that X11 is configured and running on the machine elf, and that permissions have been set properly (via the xhost command), the SBBI client will now display its output to, and get its input from, the computer elf. In effect, this means it is possible to run the server on one computer, the client on another, and display on a third. This is a useful trick for running the SBBI system from machines with commercially available X11 servers but no native-mode version of the SBBI client (such as the IBM PC and Apple Macintosh).

A.3 Using the SBBI System

Once the server has been started and a connection established by the client, the SBBI system functions as one cohesive piece of software running on the computer the user is currently using.

The client program presents you with one large window sectioned into four main parts. On the left is a list box showing the visual artifacts the system currently provides (the **available artifacts** list box). On the right is an empty list box showing the artifacts currently being applied to the input image (the **applied artifacts** list box). In the center is a set of buttons that control the movement of artifacts between the available artifacts list box and the applied artifacts list box. At the bottom is a set of brief instructions and a status line (which is usually not visible until the first action is initiated by the user). An actual screen dump from the program appears in Figure A-1.

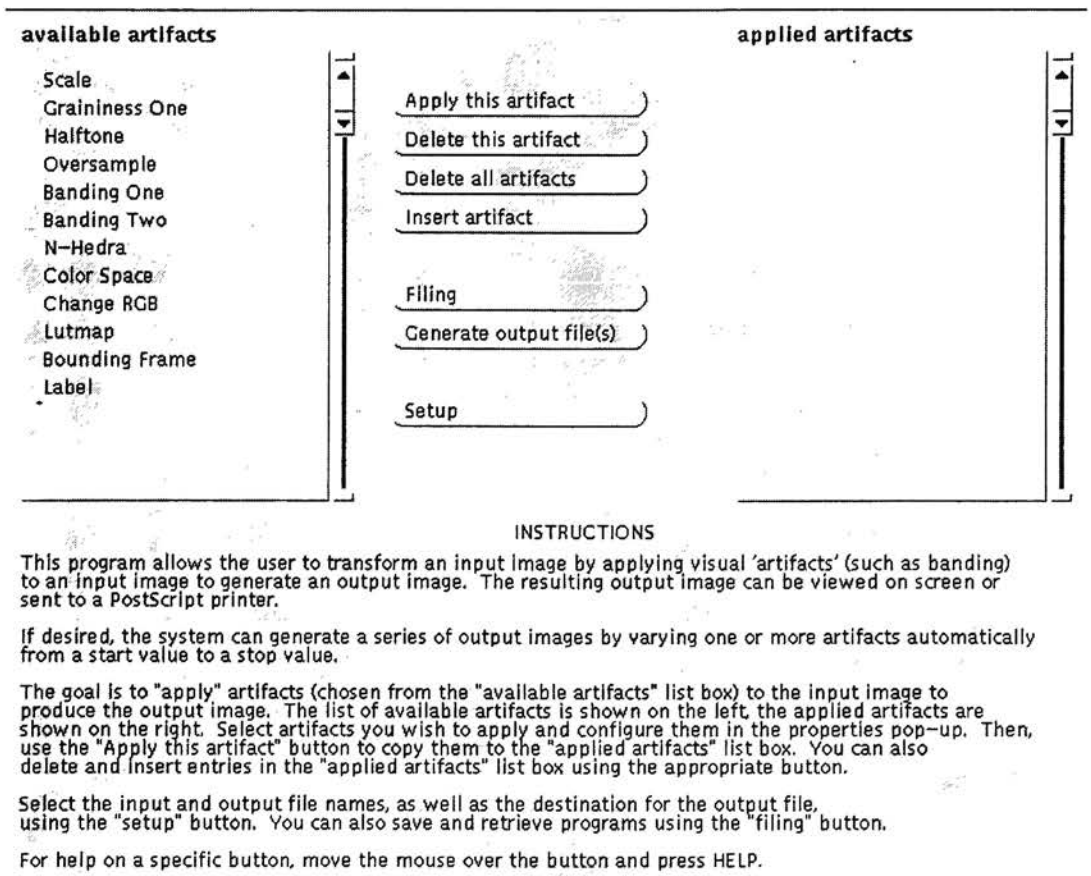


Figure A-1
The SBBI client window

A.3.1 Understanding Visual Artifacts

In hardware print engines, visual artifacts are introduced by numerous (often conflicting) design goals. It is these visual artifacts that the customer is responding to during the course of the discussion with you. For the purposes of simulation, the visual artifacts are introduced artificially by the software. For a description of each of the artifacts supported by the SBBI system see appendix B.

Visual artifacts often require several pieces of information to be introduced properly into the image. Consider a fictional visual artifact to be introduced called the "snow effect." Such an artifact might introduce random white spots across the surface of the image. Each spot is composed of a block of white dots either one, two, or three dots (pixels) on a side. To introduce this artifact to a base image, the system would need to know the frequency of the dots (expressed, perhaps, as a one-out-of-X chance that any given dot will turn white) and the size of the dots (expressed as an integer N in the range one to three). In the terminology of the SBBI program, the "snow effect" is termed a *visual artifact*, and the probability (X) and size (N) are termed *properties* of that visual artifact. Each property has current and default values associated with it, a type, a minimum allowed value, a maximum allowed value, and a path to an ASCII help file for that property. Figure A-2 illustrates these relationships for the "snow effect" artifact (see section A.5 for more information).

Property Name	Type	Current Value	Min. Value	Max. Value	Help File
Probability	INTEGER	100	1	1000	/SBBI/prob.hlp
Size	INTEGER	2	1	3	/SBBI/size.hlp

Figure A-2
Properties of the "snow effect" visual artifact

Each artifact can have as many properties as is required to fully describe it. The SBBI system supports the property types shown in Figure A-3.

Type Name	Description	Examples
BOOLEAN	Logical TRUE or FALSE	TRUE, FALSE
INTEGER	Numbers (no fractional component)	0, 1, 101, -100
FLOAT	Numbers (fractional)	0.0, 3.1415, 100.0
STRING	Sets of characters or numbers	HELLO, THIS IS A TEST
FILENAME	UNIX path names	/tmp/test, /usr/local/foobar

Figure A-3
Valid SBBI property types

A.3.2 Using the SETUP Dialog Box to Configure the Prints

The first thing you should do after running the client is configure the print. To do so, move the mouse pointer over the **Setup** button and click the left mouse button. This open the **setup** dialog box shown in Figure A-4.

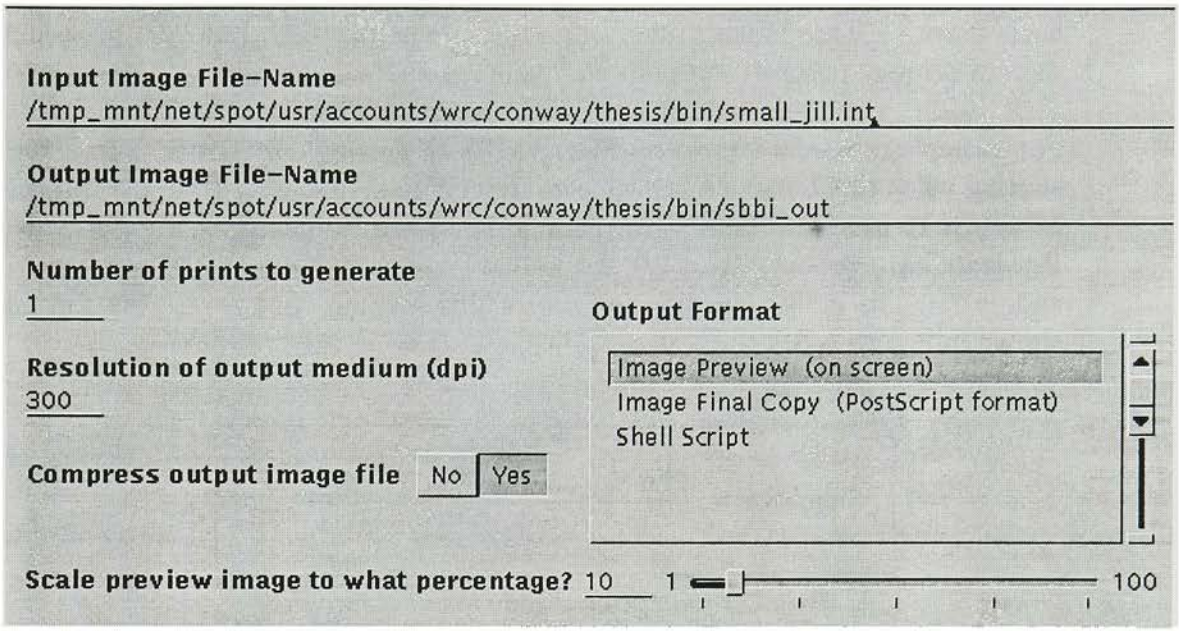


Figure A-4
The SETUP dialog box

This dialog box allows you to configure the specifics of the current set of prints. A complete description of each of the fields in this dialog box follows.

Input Image File-Name

The input file name field tells the system where to locate the file that contains the base input image. This image corresponds to the paper placed on, for example, the glass surface of a copier. It is to this image that the artifacts will be introduced.

Output Image File-Name

The output file name field tells the system where to write the output of the program. The type of the output file depends upon the setting of the **output format** list box (see Figure A-6).

Number of prints to generate

The number of prints to generate field tells the system how many prints you want to produce. This field is directly related to the **start value** and **stop value** fields in the properties dialog box. The SBBI system is capable of a simple form of looping, whereby it can generate series of prints automatically, where each print varies slightly from the others in the value of one or more properties. You enter the start value and stop value for the property (or properties) that are to vary with each print. The system then generates the specified number of prints, incrementing the property from its start value to its final value.

For example, consider a property **PROBABILITY** of type **FLOAT**. You enter a starting value of 1.0 and a final value of 10.0. You then enter a 10 in the **number of prints to generate** field. This causes the system to generate 10 prints. The first print has a probability of 1.0, the second has a probability of 2.0,
and so on. To determine the *step size* of this looping, simply use the formula shown in Figure A-5.

$$\text{Step Size} = \frac{(\text{Final Value} - \text{Start Value} + 1)}{\text{Number Of Prints}}$$

Figure A-5
Calculating the Step Size of a loop

Resolution of output medium (dpi)

This field tells the system what resolution the output printer will have (in dots-per-inch). This information must be encoded into the PostScript file for proper spacing and calibration. Typical values are 300, 600, 1200, and 2400. Only a single value is needed, as the system assumes the same number of dots per inch horizontally and vertically.

Compress output image file

Use this switch to tell the system if it should automatically compress the output file using the standard UNIX compress(1) program. This feature exists to reduce disk space requirements for storing large image files, which can easily run tens or even hundreds of megabytes. Select **yes** if you want compression; select **no** if you want the output file stored in its normal fashion.

Scale preview image to what percentage

This slide bar can be used to reduce the size (scale down) the input image before processing to make large images visible in their entirety during the on-screen preview. Without this feature, the output file might not be visible in its entirety during previews (which may or may not be acceptable for your application). Note that this scaling happens for screen preview purposes only, the original input file on disk remains intact and full size. The value of this field is a percentage, such that a value of 10 means that the input image will be scaled to 10% (1/10) of its original size. This option *cannot* be used to increase the size of the input image (ie, the value must be between 1 and 100 inclusive), although the *scale* artifact does exist for that purpose.

Output format

This list box controls the type of the output file. The system allows the three output formats shown in Figure A-6.

Output Format Name	Output file	Comments
Image Preview (on-screen)	*user-specified*	Output file is in proprietary Xerox format (XIP)
Image Final Copy (PostScript)	*user-specified*	Output file is in PostScript Level I
Shell Script	*user-specified* + .run	Output file name has **.run** appended to it. Format is an ASCII Bourne shell script

Figure A-6
Output Modes

The third output format, Shell Script, has a special meaning. The SBBI system operates by generating and executing UNIX Bourne shell scripts that call proprietary Xerox imaging routines. Under the first two output formats, Image Preview and Image Final Copy (PostScript), these files are created, executed, and deleted automatically. The Shell Script output format causes the system to generate the shell script, but not to execute or delete it. This allows the user to review the contents of the shell script. Under this mode, the shell script has the same name as the output image, with an additional **.run** placed at the end (see Figure 3-8).

Once these fields have been configured, you can begin configuring artifacts to be added to the system.

A.3.3 Using Visual Artifacts

When an artifact is selected from the available artifacts list box, the SBBI program pops up a dialog box listing all the properties of that artifact and the current value for each property. The properties are listed in a list box, and as each is selected the information at the bottom changes to reflect the current value of that property. This dialog box is termed the **properties dialog box**. Figure A-7 shows a typical properties dialog box.

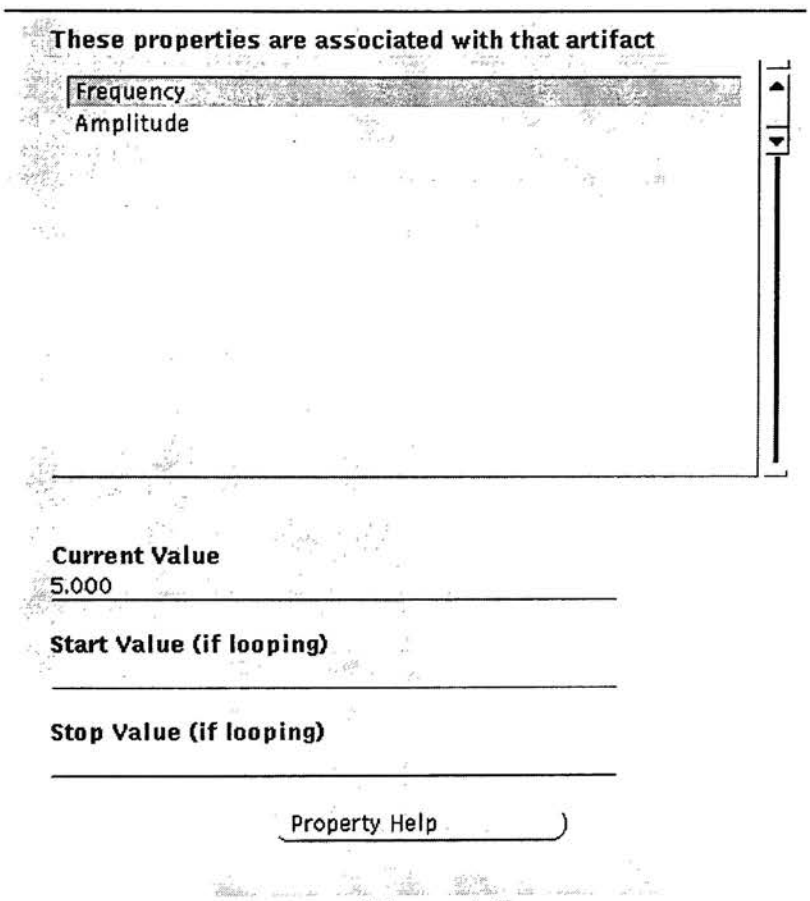


Figure A-7
The PROPERTIES dialog box

To enter a new value, click the mouse on the *current value* field and type in the new value. If multiple prints are being generated (see section A.3.2), you may enter start and stop values using the properties dialog box. For help on any property, select the property and click on the **Property Help** button. In the event that the proper help file has not been specified or does not exist, an error message will appear in the status line.

NOTE: You are encouraged to arrange the dialog boxes such that they do not overlap. The properties dialog box is redrawn each time an artifact is selected from either the available or applied artifacts list box. This may cause the properties dialog box to interfere with your ability to quickly move between artifacts, as the main window will need to be continually brought to the foreground. Tiling the windows across the workspace will eliminate this problem.

Applying an artifact to the input image

To apply an artifact to the input image, select the desired artifact from the available artifacts list box. The properties dialog box for that artifact will open. Enter a value (or start and stop values) for each property. Click the **Apply this artifact** button in the middle of the window. The artifact now appears at the end of the applied artifacts list box on the right. Note that artifacts are applied to the input image in the order that they appear (from top to bottom) in the applied artifacts list box. For most applications, the order of the application of artifacts is not important.

Editing an artifact's properties

To edit an artifact from either list, click on the artifact. This opens the properties dialog box. Enter the new value for each property and close the dialog box (or select the next artifact).

Removing an artifact from the input image

To remove an artifact currently applied to the input image, click on the artifact in the applied artifacts list box. The properties dialog box for that artifact will appear. Click on the **Delete this artifact** button in the middle of the window. The artifact will disappear from the applied artifacts list box. It is not possible to delete artifacts from the available artifacts list box once the program starts running (see section A.5 for information on how to remove artifacts from that list box).

Removing all artifacts from the input image

To remove all the artifacts in the applied artifacts list box, click the **Delete all artifacts** button in the middle of the window.

Inserting an artifact into the *middle* of the applied artifacts list box

To insert an artifact into the middle of the applied artifacts list box, click on the artifact that currently occupies the desired position. For example, if you wish to insert an artifact between the second and third artifacts, creating a new third artifact, click on the artifact that is currently in the third position. The properties dialog box for that artifact appears. Now, click on the artifact you wish to apply from the available artifacts list box. The properties dialog box for that artifact appears. Finally, click on the **Insert artifact** button. The new artifact appears at the desired insertion point.

A.3.4 Filing

To save time later, it is sometimes useful to save the contents of the applied artifacts list box (or the *state* of the SBBI system) for future use. This state can then be read back in at a later date for further use. The filing button can be used to store the current, or retrieve an old, state. The filing dialog box appears in Figure A-8.

Figure A-8
The FILING dialog box

Saving the current set of applied artifacts

To save the current state, click on the **Filing** button in the middle of the window. The filing dialog box appears. Enter the file name in the **file name** field and click on the **Save applied artifacts** button at the bottom of the dialog box.

Retrieving an old set of applied artifacts

To retrieve an old state (previously stored using the save option), click on the **Filing** button in the middle of the window. The filing dialog box appears. Enter the name of the file in the **file name** field and click on the **Load applied artifacts** button at the bottom of the dialog box.

If the system cannot find the file name specified, a message box will appear informing you of this fact. At that point the user may either enter another file name or close the dialog box without loading.

Returning to the main window

The filing dialog box is designed to return to the main window automatically after saving or retrieving a state. If you wish to return to the main window without performing either action, use the **cancel** button.

A.3.5 Generating the Output File(s)

Once you have entered and configured a set of visual artifacts in the applied artifacts list box and configured the prints using the **setup** feature, you are ready to generate the output image(s). Click on the **Generate Output Images(s)** button in the middle of the window. The status line at the bottom of the screen informs you of the progress of the run. As each print is generated, its file name will appear in the status line. In preview mode, the preview program will automatically appear for each output file.
If all goes well, the following message will appear in the status line:

```
Generation of output file(s) is complete
```

If all does not go well, an error message will appear in the status line and the system will attempt to pop-up a text editor (Open Look's "textedit" editor) showing the exact contents of the shell script built by the system and a log of the output generated by UNIX when the script was executed.

Properties of type FILENAME are checked before execution begins. If the file is to be read (that is, if the file is an input file of some sort) the system makes sure that the file is present and readable before beginning execution. If the file is to be written (that is, if the file is an output file of some sort) the system makes sure that the file does not exist. The type of file (input or output) can be specified in the SBBI configuration file(s). See section A.5 for more information on this feature.

A.4 Using the Output File(s)

Once the output file or files have been created, you can view or use them you see fit. PostScript files would normally be sent to one of the numerous PostScript printers available within Xerox. The user should check that the output resolution of the printer matches that of the output file (as determined in the setup dialog box). If the file is a preview file, then it is stored in Xerox internal proprietary format (XIP) and can be viewed using appropriate tools. If the file is a shell script, then it can be modified and executed as a stand-alone application.

A.5 Configuring the SBBI System

Both the client and server have been written to be self-configuring. Rather than compiling knowledge about artifacts and properties into the code itself, this information is read from ASCII configuration files by the programs when they begin execution. The advantage of this approach is that the user can edit these configuration files using standard text editors to alter the behavior of the system. Note that to introduce a new artifact or modify the type of a property is is necessary to modify the configuration files for both the client and the server. To modify the help file location, default value, minimum value, or maximum value of a property, editing the client configuration file is sufficient.

A.5.1 Configuring the client

When the client executes, it reads information about the names of available artifacts and properties, their types, their default values, their minimum and maximum values, and help file locations from various system and user configuration files. These configuration files have a syntactically rigid but expressive structure. By default, the system reads the contents of the configuration files from three different locations, shown in Figure A-9.

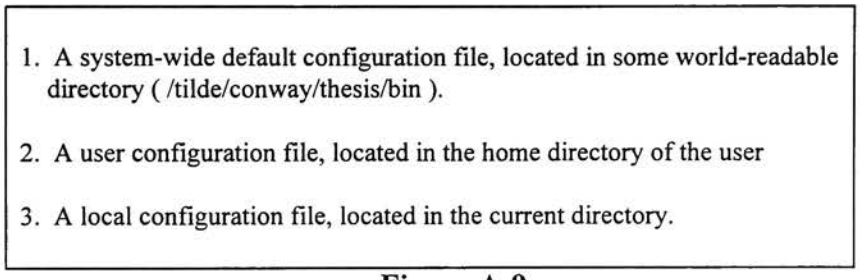


Figure A-9
Configuration file locations

In all three cases, the name of the configuration file is **.SBBIrc**. The SBBI system reads each configuration file in turn (starting with the system-wide one and ending with the local one). Each configuration file's contents *merge* with the contents of the previous files (in memory, not on the disk). Where two artifacts with the same name exist, the new artifact overrides the former artifact. Where two properties of the same name exist for the same artifact, the new property overrides the former property. Artifacts not previously encountered are appended to the list of available artifacts. Thus, each file *merges* with the contents of the previous configuration file(s).

Each line in the configuration file can take only one of four possible forms. They may be blank lines, comment lines, artifact lines, or property lines. A blank line, as its name implies, is an empty line in the file. These are ignored. A comment line begins with a pound sign (#) in the first position. These too are ignored. A new artifact is an

artifact name bracketed by curly braces ({ and }). For example, the following line introduces a new artifact (surrounded by comments) :

```
# This is the "snow effect" artifact

{snow effect}

# Here are its properties:
```

Properties always associate with the most recently declared artifact. Property lines have the property names bracketed by square brackets ([and]) followed by the type (**BOOLEAN, INTEGER, FLOAT, STRING, FILENAME**) and a default initial value. Integer and floating point properties are then followed by minimum and maximum values. File names are followed by either **READING**, **WRITING**, or **NO-CHECK**. Properties marked as reading files will be checked by the system to make sure they exist and are accessible before generation of output files begins. Those marked as writing will be checked to make sure old files of the same name are not overwritten by accident. Files marked as no-check have neither check performed on them before generation of output file(s) begins. The last item of any property line is an optional UNIX path and file name (in quotes) to an ASCII help file associated with that property. Properties of type STRING and FILENAME have their default values enclosed in double quotations. Properties of type BOOLEAN have default values of either TRUE or FALSE. Figure A-10 shows a more complete SBBI configuration file.

```
#
# This is a sample configuration file.  This file must be called
# .SBBIrc for the system to see it, and should reside in either
# the user's home directory or current directory when the sbbi-client
# program is run
#
# Gregory R. Conway
# Wednesday, July 20, 1994
#

{Snow effect}
[Probability] INTEGER 5 1 10 "/tilde/conway/thesis/bin/help/snow-effect-probability"
[Size] INTEGER 1 1 3 "/tilde/conway/thesis/bin/help/show-effect-size"

{Scale}
[Scaling Factor] FLOAT 1.0 0.01 100.0 "/tilde/conway/thesis/bin/help/scale-factor"

{Transparency Effect}
[Input File] FILENAME "/tilde/conway/in" READING "/tilde/conway/thesis/bin/help/input"
[Generate Output File] BOOLEAN TRUE "/tilde/conway/thesis/bin/help/trans-file"
[Output File] FILENAME "/tilde/conway/out" WRITING "/tilde/conway/thesis/bin/help/output"

#
# End of configuration file
#
```

Figure A-10
Sample SBBI Configuration File

A.5.2 Configuring the server

Unlike the client, the server configuration information is stored in a single file (`/tilde/conway/thesis/bin/server-parser`). The server needs to know how to construct lines in the shell script to implement the artifacts applied by the user. To accomplish this, the server associates a format string with each artifact that it understands. This format string tells the server how to build the appropriate line in the shell script. In general, the format string is copied verbatim to the shell script, except when substitution markers are encountered. Substitution markers are represent place-holders within the format string, and are replaced by the value of specific properties. Substitution markers can be identified because they begin with a percent sign, much like format strings used by the printf family of C library routines. Figure 3-6 shows the substitution markers currently supported by the SBBI server.

The configuration file can contain blank lines and comment lines (comment lines begin with a pound sign). The name of the artifact appears at the beginning of the line in curly braces ({ }), and the corresponding format string appears on the same line enclosed in double quotes. To introduce a new artifact, an XIP, XCC, or similar module must first be written to handle the artifact. Then, an entry is added to the server configuration file to teach the server how to write appropriate calls to the module. Finally, the client configuration file(s) must be updated so the user can be asked appropriate questions regarding the application of the artifact. Figure A-11 shows a sample server configuration file.

```
#
# XEROX PRIVATE DATA.  All rights reserved.
#
# Sample SBBI server configuration file.
# Thursday, September 15, 1994
#

#
# These support the artifacts and properties shown in Figure A-10
#

{Snow effect} "snow -prob %d -size %d"
{Scale} "scale -factor %f"
{Transparency Effect} "trans -input %s %b[-output %s]"
```

Figure A-11
Sample server configuration file

Important

The casual user can render the SBBI system unusable by incorrectly modifying the configuration files. While this does no permanent damage to the system, it may require that the original configuration files be restored from some backup media. Exercise care when modifying the configuration files. For security reasons, the server configuration files should be owned by root and set to mode 644.

Appendix B

The Simulated Bread Board Imaging System Visual Artifacts

The SBBI system currently supports the inclusion of nine visual artifacts. Each of these artifacts is listed in Figure B-1. Note that the SBBI system is designed to make the inclusion of new artifacts easy, so this list may or may not be complete. Most of these artifacts have corresponding routines in either the XIP or XCC toolkits.

Artifact Name	Purpose
Scale	Causes the image to grow or shrink by a given factor. A scaling factor of 2 doubles the size of the image, while a scaling factor of 0.5 cuts the image size in half.
Graininess	Introduces noise into the image such that the edges are not as crisp and well defined.
Halftone	Applies a halftoning pattern to a contone image.
Oversample	Similar to scale, but exactly duplicates pixels to increase size (without dropping or adding scan lines).
Banding	Introduces a pattern of alternating light and dark regions that simulates the effect of motion variation in the paper handling mechanism.
N-Hedra	Performs transformation between color spaces of arbitrary size.
Color Space	Converts an image into one of several pre-defined color spaces.
Change RGB	Converts between Xerox RGB and XCC RGB.
Lutmap	Performs color correction.

Figure B-1
The SBBI system visual artifacts

Appendix C

Sample SBBI Shell Script

Figure C-1 shows a typical shell script as it might be generated by the SBBI system. The script is fully documented, and contains definitions for all the environment variables needed for the XIP and XCC toolsets. The return status of the script is designed to tell the server if the script executed properly. *This script is illustrative only and contains no calls to actual XIP or XCC routines.*

```
#!/bin/sh
#
# This script was generated by the SBBI system.  This program is built
# up from user specifications and run by the SBBI server.  This file
# may be deleted once execution of the script is complete, as it will
# be re-created by the SBBI system when needed.
#
# XEROX PRIVATE DATA.  Not for distribution or sale.  For information
# regarding the use of this program, or the SBBI system, please contact:
#
# Greg Conway
# Xerox Corporation
# Joseph C. Wilson Center for Research and Technology
# 800 Phillips Road  Bldg. 114-39D
# Webster, NY 14580
# USA
#

# Set up the PATH environment variable

PATH=/bin:/usr/bin:/usr/ucb:/usr/local/bin:/XIP/bin:/XCC/bin

# Set the library path for the XIP routines

LD_LIBRARY_PATH=/XIP/lib:/usr/lib:/lib

# Set the XIP and XCC environment variables

XIP_HOME=/XIP
XIP_TEMP=/tmp
XCC_HOME=/XCC
XCC_TEMP=/tmp

# export the world

export PATH LD_LIBRARY_PATH XIP_HOME XIP_TEMP XCC_HOME XCC_TEMP

# Here is the program, taken from the user specifications

rm -f foo.tif

read_jpg foo.jpg \
| scale -factor 10.0 \
| write_tiff foo.tiff

# See if the script executed OK

if [-s foo.tif]; then
    exit 0
else
    exit 1
fi
```

Figure C-1
Sample SBBI script

Appendix D

Sample Prints

The following six pages contain prints that illustrate the effect of banding. Banding is caused by the uneven movement of paper through the print engine, resulting in some "rows" of dots being closer together than they should be, while others are further apart. This causes the streaking effect visible in the prints. The property values used to generate each print are found in Figure D-1.

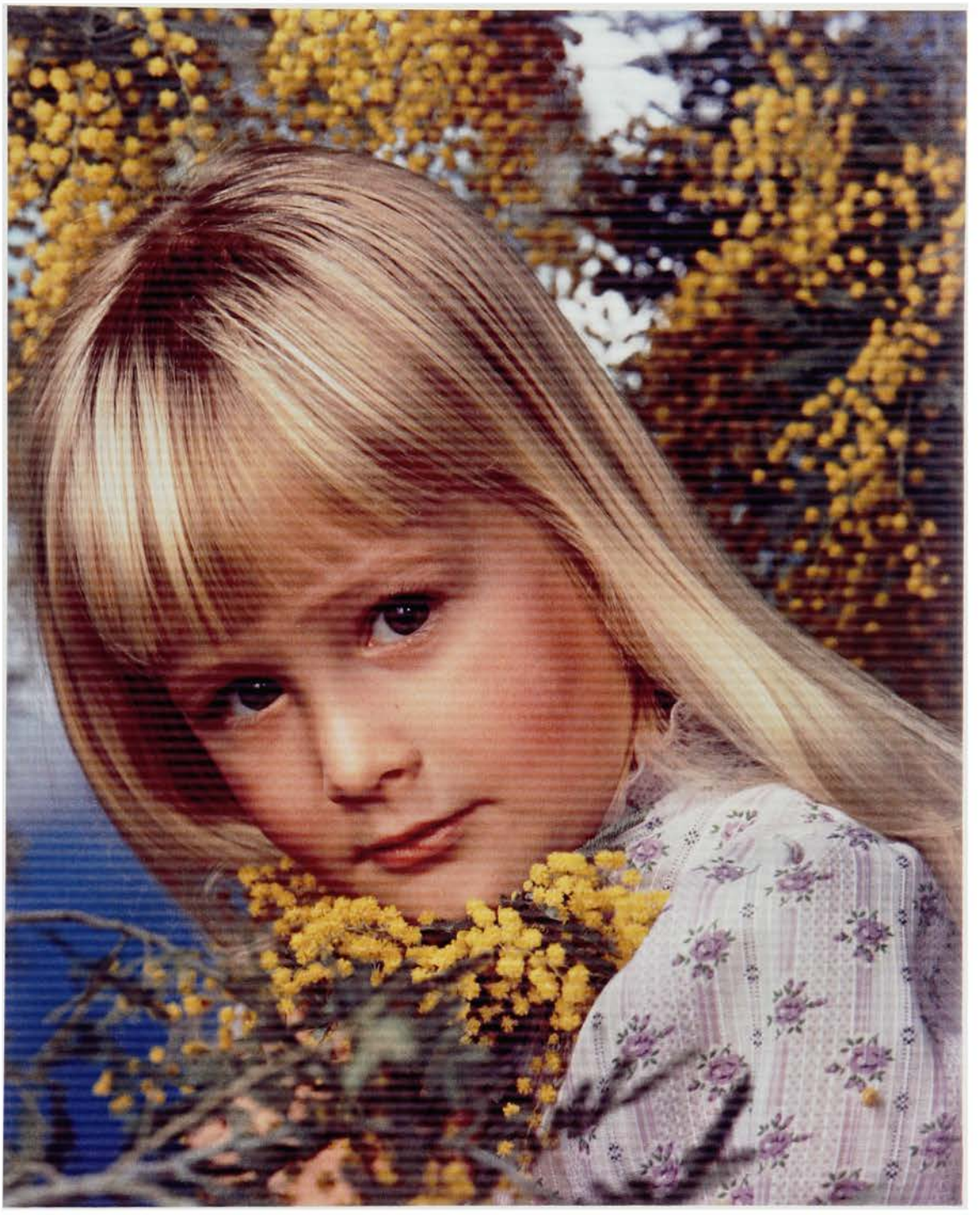
Print Name	Frequency	Amplitude	Page No.
Banding Example: f=100 a = 0	100	0	67
Banding Example: f=100 a = 2	100	2	69
Banding Example: f=100 a = 4	100	4	71
Banding Example: f=100 a = 6	100	6	73
Banding Example: f=100 a = 8	100	8	65
Banding Example: f=100 a = 10	100	10	77

Figure D-1
Banding property parameters for sample prints

Banding Example: f=100 a=0

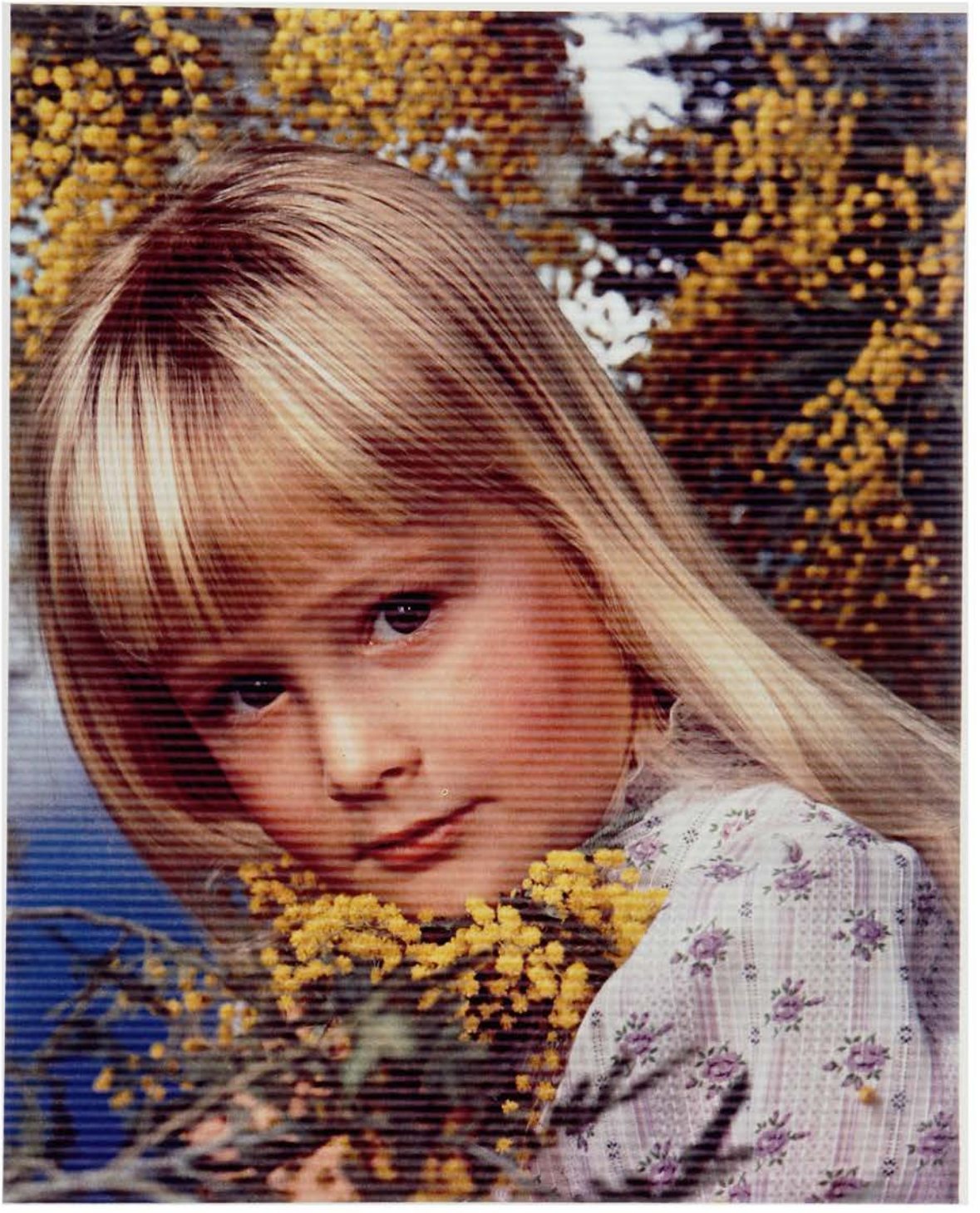
Banding Example: f=100 a=2

Banding Example: f=100 a=4



Banding Example: f=100 a=6

Banding Example: f=100 a=8



Banding Example: f=100 a=10

Bibliography

1. "The American Heritage Dictionary (2nd College Edition)," *Houghton Mifflin*, 1983.

2. Torpey, P., Xerox Internal Report #X9200514, 1992. *Xerox Corporation.* 800 Phillips Rd. 114-44D, Webster, NY 14580.

3. Shaw, J. and Stark, H., "Full-Page Image Simulations for Xerographic IOTs," Xerox communication, 1994.

4. Ingels, D., "What Every Engineer Should Know About Computer Modeling and Simulation," *Marcel Dekker, Inc.*, 1985.

5. Dogramaci, A. and Adam, N., "Current Issues in Computer Simulation," *Academic Press,* 1979.

6. Lehman, R. S., "Computer Simulation and Modeling: An Introduction," *Lawrence Erlbaum Associates*, 1977.

7. Clapp, R. E., "Visual Simulation", *Proceedings of the 20th Annual Simulation Symposium*, IEEE Press, 1987.

8. O'Brien, J. and Chen, I., Xerox Internal Report #X9100260, 1991. *Xerox Corporation.* 800 Phillips Rd. 114-21D, Webster, NY 14580.

9. Jeyadev, S., Chen, I., Duke, C., Lanzattella, C. B., Marshall, S., Nash, S., O'Brien, J., Xerox Internal Report #X9100259, 1991. *Xerox Corporation.* 800 Phillips Rd. 103-05B, Webster, NY 14580.

10. Heil, D., "Simulation and Systems Engineering: An Affirmation," *Proceedings of the 18th Annual Simulation Symposium*, IEEE Press, 1985.

11. Bunker, W. M., "Simulate - Then Solder: Simulation in Electronic System Design," *Proceedings of the 10th Annual Simulation Symposium*, IEEE Press, 1977.

12. Engel, Dr. Robert D., "Using Simulation to Optimize Solar Greenhouse Design," *Proceedings of the 17th Annual Simulation Symposium*, IEEE Press, 1984.

13. Boike, D. G. and Ernst, E. H., "Modeling and Simulation in Product Development," *Proceedings of the 16th Annual Simulation Symposium*, IEEE Press, 1983.

14. Bollman, J., Xerox Internal Report #X9400079, 1994. *Xerox Corporation.* 800 Phillips Rd. 128-27E, Webster, NY 14580.

15. Rasmussen, D. R., Xerox Communication. *Xerox Corporation.* Jefferson Road 801-27C, Rochester, NY 14623.

16. Steinhour, J., Maltz, M., Rolleston, R., Xerox Internal Report #9400210. *Xerox Corporation.* 800 Phillips Rd 128-29E, Webster, NY 14580.

17. Powell, J., "Designing User Interfaces", *Microtrend Books*, 1990.

18. "21st Century Jet", *WXXI Television.* Contact Dr. Ted Retzlaff, Xerox Corporation, 800 Phillips Rd 128-29E, Webster, NY 14580..

19. Retzlaff, A., Shaw, J. Xerox Communication. *Xerox Corporation.* 800 Phillips Rd. 114-22D, Webster, NY 14580.

20. Ulrich, D., Xerox Internal Report #X8201553, 1982. *Xerox Corporation.* Technical Information Center, 800 Phillips Rd., Webster, NY 14580.

21. Ulrich, D., "Modular Simulation Package for Product Design Studies," *Proceedings of the 16th Annual Simulation Symposium*, IEEE Press, 1983.

Suggested Reading

Shaw, J. and Retzlaff, T., "CPTools Reference Manual". Xerox Internal Report #X9300317, 1994. *Xerox Corporation.* 800 Phillips Rd. 114-22D, Webster, NY 14580.

Kernighan, B., Ritchie, D., "The C Programming Language (second edition)," *Prentice Hall*, 1988.

Kernighan, B., Pike, R., "The UNIX Programming Environment," *Prentice Hall*, 1984.

Stevens, W. R., "UNIX Network Programming," *Prentice Hall*, 1990.

Heller, D., "XView Programming Manual for XView Version 3," *O'Reilly & Associates*, 1991.

Jones, O., "Introduction to the X Window System," *Prentic Hall*, 1989.

Scheifler, R., Gettys, J., "X Window System (second edition)," *Digital Press*, 1990.

Shannon, R. E., "Introduction to Simulation," *Proceedings of the 1992 Winter Simulation Conference*, Association of Computing Machinery Press, 1992.

Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F., "Computer Graphics: Principles and Practice," *Addison-Wesley Publishing*, 1990.

Hoover, S. and Perry, R., "Simulation: A Problem-Solving Approach," *Addison-Wesley Publishing*, 1990.

Edwards, S., "Simulation as a Mind Set," *Simulation at the Frontiers of Science*, The Society for Computer Simulation, 1986.