INTEGRATING THE SPICE-PAC SIMULATOR WITH THE OSA90/hope™ DESIGN ENVIRONMENT

R.M. Biernacki, J.W. Bandler, S.H. Chen and P.A. Grobelny

SOS-92-5-R

July 1992

INTEGRATING THE SPICE-PAC SIMULATOR WITH THE OSA90/hope™ DESIGN ENVIRONMENT

R.M Biernacki,* J.W. Bandler,* S.H. Chen* and P.A. Grobelny

Abstract This report presents Spicepipee™, a pipe-ready version of SPICE-PAC. Spicepipe comprises SPICE-PAC and the means to communicate between OSA90/hope™ and SPICE-PAC. OSA90/hope is a circuit design environment which features powerful design optimization and statistical capabilities, including Monte Carlo analysis and yield-driven design. It also has built-in frequency-domain circuit simulation including harmonic balance. SPICE-PAC is an optimization-structured version of the popular circuit simulator SPICE. It contributes excellent time-domain simulation and noise analysis. Spicepipe utilizes the high-speed Datapipe™ technology available in OSA90/hope as the vehicle for the interconnection. The Datapipe technology, based on UNIX pipes, creates and maintains fast communication channels between OSA90/hope, (the parent) and processes created by OSA90/hope (children). Spicepipe is OSA90/hope's child designed to explore the SPICE-PAC simulators from within the OSA90/hope. Combining tools working in different domains results in a mixed frequency-time-domain circuit design environment. Spicepipe augments OSA90/hope with the capabilities of SPICE-PAC time-domain simulation, noise analysis and SPICE device models. OSA90/hope extends SPICE-PAC's simulation-only capabilities by advanced pre- and post-processing of data, flexible input and output as well as by optimization. The utilization of Spicepipe in solving circuit design problems is illustrated by three circuit examples involving simulation, optimization, Monte Carlo analysis and yield-driven design in both the frequency and time domains.

The authors are with the Simulation Optimization Systems Research Laboratory and the Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada L8S 4L7.

* R.M. Biernacki, J.W. Bandler and S.H. Chen are also with Optimization Systems Associates Inc., Dundas, Ontario, Canada L9H 6L1.

This work was supported in part by the Optimization Systems Associates Inc. and by the Natural Sciences and Engineering Research Council of Canada under Grants OGP0042444 and OGP0007239.

CONVENTIONS

In this report we use *different* `fonts` to distinguish different concepts. Here are the general conventions:

- We use the Times Roman font for regular text.
- We use the *italic font* to indicate the names of programs, C and Fortran functions and subroutines (Fortran subroutines names are usually capitalized.)
- We use the `Prestige Elite font` to show keywords and examples.
- We use the `Prestige Elite Small font` for file listings (to save space).

I. INTRODUCTION

Complex engineering problems may require us to combine circuit simulation with other software tools supporting optimization, statistical analysis, higher or lower level simulations. This report describes Spicepipe, a pipe-ready version of SPICE-PAC. Spicepipe allows the user to explore SPICE-PAC simulation capabilities from within the OSA90/hope circuit design environment.

We introduce OSA90/hope [1] and SPICE-PAC [2], an optimization-structured version of the popular circuit simulator SPICE [3]. We also introduce the library for inter-program pipe communication (IPPC) [4], the basis of the high speed Datapipe technology [1, 4] featured in OSA90/hope. We describe how we have used the Datapipe technology to interconnect SPICE-PAC with OSA90/hope. We provide illustrative examples of the utilization of Spicepipe. We assume that the reader is familiar with OSA90/hope and that he or she understands the SPICE-PAC input file language. Detailed understanding of the Datapipe technology is not necessary.

Spicepipe, connecting SPICE-PAC to the OSA90/hope design environment augments OSA90/hope's circuit design capabilities of simulation and modelling with those available in SPICE-PAC, most importantly, time-domain simulation, noise analysis and additional device models. OSA90/hope, as a circuit design environment, provides the user with harmonic-balance frequency-domain simulation, statistical analysis and, most importantly, very flexible and versatile optimization. As a result, Spicepipe creates a mixed frequency-time-domain design environment. This environment features versatile simulation and optimization, statistical analysis and yield-driven design in both domains.

To interconnect SPICE-PAC with OSA90/hope we have used the OSA90/hope Datapipe technology and Datapipe Server implemented with the help of the IPPC library for inter-program pipe communication [4]. The IPPC library employs the concept of UNIX pipes [5] which are fast interprocess communication channels. The channels interconnect OSA90/hope, called the parent process, with processes created by OSA90/hope, called its children. Spicepipe is simply an

OSA90/hope child designed to organize and control the exchange of information between OSA90/hope and SPICE-PAC. The Datapipe Server and the IPPC library are integral parts of OSA90/hope.

In Sections II and III we provide a short introduction to OSA90/hope and SPICE-PAC, respectively. Section IV describes the IPPC library, the engine of the Datapipe technology. Section V explains the general structure of the interconnection between OSA90/hope and SPICE-PAC. Sections VI, VII and VIII discuss the interfacing driver, SPICE-PAC driver and the *create_file* program, respectively. The interfacing driver, SPICE-PAC driver and the *create_file* program are the interfacing components of Spicepipe. The interfacing driver provides the interface between OSA90/hope and SPICE-PAC. The SPICE-PAC driver organizes the tasks performed by SPICE-PAC. The *create_file* program is used to create a disk input file for SPICE-PAC. In Section IX we explain the details of the OSA90/hope input file for Spicepipe. Section X includes three circuit design problems. We optimize an LC transformer to illustrate the communication between OSA90/hope and SPICE-PAC. For an NMOS inverter we explore production yield estimation with time-domain specifications. The last example uses a second-order RLC circuit to demonstrate mixed frequency-time-domain optimization. We first perform optimization of the nominal circuit and then continue with yield optimization. We used a Sun SPARCstation 1 as the platform to perform these experiments. Our conclusions and acknowledgment are in Sections XI and XII, respectively.

II. OSA90/hope [1]

OSA90/hope - ***O***ptimization ***S***hell ***A***ssembly/***h***armonic ***o***ptimization ***p***ersonal ***e***nvironment is a general purpose Computer-Aided Design (CAD) system developed by Optimization System Associates Inc. OSA90/hope contains three simulators for DC, AC and large-signal harmonic-balance analyses and several state-of-the-art optimizers. It features statistical Monte Carlo analysis, yield optimization and expression processing capability operating on scalars, vectors and matrices. OSA90/hope is also equipped with several ready-to-use Datapipe protocols [1, 4, 5]. Datapipe can connect external programs with the OSA90/hope internal simulators and/or optimizers (see Fig. 1). The external programs are then called child programs.

The user communicates with OSA90/hope by means of an input file [1]. The input file describes the circuit under consideration as well as the operations requested by the user. The Datapipe connections are also defined in the input file. In order to connect external programs through OSA90/hope the user does not need to access the OSA90/hope source code. Furthermore, the internal organization of the Datapipe transfer mechanism is transparent to the user.

To have a better feeling of what Datapipe protocols are, let us consider the COM Datapipe protocol in more detail. This will not only acquaint us with the protocols in general but will also serve as an introduction to utilizing Spicepipe which uses the COM protocol. The COM protocol is capable of transferring character strings such as data file names, messages, keywords, or even the entire contents of a file.

The COM Datapipe, like any other Datapipe, is defined in the OSA90/hope input file within the `Expression` and/or the `Model` block. The syntax for the COM protocol is as follows.

```
Datapipe:
      COM                    FILE="filename"
      N_INPUT=n              INPUT=(x1, ..., xn)
      N_OUTPUT=m             OUTPUT=(y1, ..., ym);
```

where COM is the keyword identifying the protocol, `filename` represents the name of the binary (executable) child program, n is the total number of inputs to be passed from OSA90/hope to the child, and m is the total number of outputs to be passed from the child back to OSA90/hope. The inputs `x1, ..., xn` can be specified by constant values, optimization variables and labels previously defined in the input file. Labels defining character strings are also allowed. The outputs `y1, ..., ym` must be specified by unique label names which have not been used as other identifiers. The outputs can also include character string labels.

As the syntax of the COM protocol shows the only thing the user has to do in order to define a COM Datapipe is to define the input and output for the Datapipe.

III. SPICE-PAC [2]

SPICE-PAC was developed by W.M. Zuberek of the Department of Computer Science, Memorial University, Newfoundland. It is a simulation package that is upwardly compatible with the popular SPICE circuit simulator [4]. SPICE-PAC accepts the same circuit description language as SPICE (with a few minor exceptions) and provides the same circuit analyses. It also supports a number of extensions and refinements which are not available in the original SPICE program.

One of the extensions, particularly interesting for our application, is the notion of, so called, circuit variables. Circuit variables are those attributes of circuit elements that can be modified during subsequent simulations and in optimization. Circuit variables are defined within the extended circuit description which is a part of the input file. In SPICE-PAC, likewise in OSA90/hope, the input file is the main means of communication between the user and the system. The extended circuit description is separated from the "basic", or SPICE like, part of the input file by the .END/EXT keyword and terminated by the .END keyword. Circuit variables are defined by the .VAR lines,

```
.VAR variable_name
```

where variable_name is either a simple element name for those elements which have one attribute only, e.g., the resistance of a resistor or the capacitance of a linear capacitor, or a composite name which is used for multi-attribute circuit elements, e.g., parameters of semiconductor devices. In fact, the idea of introducing circuit variables as an extension of SPICE evolved directly from optimization related applications of SPICE.

The main difference between SPICE and SPICE-PAC lies, however, in the internal organization of the programs. While SPICE is a program with a fixed flow of operations, SPICE-PAC is a collection of loosely coupled simulation primitives. The simulation primitives can be composed in many different ways, according to a particular application. Typical examples of simulation primitives include reading the circuit description, performing a specific circuit analysis, changing values of circuit elements, or redefining analysis parameters. In SPICE-PAC each simulation primitive constitutes a subroutine. The names of the subroutines are: *SPICEA*, *SPICEB*, ..., *SPICEY*, where, e.g., *SPICEA* reads and processes the circuit description, *SPICEB* defines circuit variables, *SPICER* performs circuit analysis. Each of the subroutines invokes a number of SPICE-PAC's internal subroutines and functions which appear "invisible" to the user. To utilize SPICE-PAC the user has to write a program called SPICE-PAC driver in which the simulation primitives are used as functional blocks. The SPICE-PAC driver organizes the flow of operation among the primitives so that the behaviour of the system meets the specifications. A detailed description of the primitives and their functions is given by Zuberek in [6].

The modular structure of SPICE-PAC makes the package very attractive, especially for specific applications. SPICE-PAC is particularly useful in optimization-oriented applications, where its modular structure may significantly increase the efficiency of the system. While creating a driver for SPICE-PAC, the user can program or include any additional tasks such as statistical post-processing or graphical output facilities.

Our version of the SPICE-PAC driver is discussed in Section VII. The availability of SPICE-PAC is described in Appendix A.

IV. FUNDAMENTALS OF THE OSA90/hope DATAPIPE™ TECHNOLOGY

This section of the report is to outline basics of UNIX pipes, the IPPC library, and the OSA90/hope Datapipe mechanism.

A pipe is an I/O channel intended for use between two cooperating processes. One process writes into the pipe, while the other process reads from the pipe. UNIX, as the operating system, controls buffering of the data and synchronization of the two processes. The system call *pipe( )*

creates a pipe and returns two file descriptors, one for the read side and the other one for the write side of the pipe. These descriptors, being file identifiers, may be used in *read()*, *write()* and *close()* calls just like any other file descriptor. If a process reads from a pipe which is empty, it waits until data arrives; if a process writes into a pipe which is full, it waits until the pipe is emptied somehow. Once the pipes have been created by the call to *pipe()* the process uses the *fork()* system call to create a copy of itself. The new, so called child copy of the process, then calls the system shell to execute the desired child program. See the SPARCstation user's manual [5] for more details.

The IPPC library [4] was created to facilitate the process of establishing high speed data connections between OSA90/hope and one or more external programs. IPPC employs the concept of UNIX pipes and includes subroutines that open, initialize, close, read from and write to a pipe.

The OSA90/hope Datapipe mechanism is built on the basis of the IPPC library. The basic form of inter-program pipe communication is between two programs: a parent program and a child program, and the communication is not repetitive (i.e., the child is not called iteratively). This corresponds to the following sequence of calls to the IPPC subroutines in the parent program.

```
cid = pipe_open("child_program");            /* open pipe and activate the
                                                child program */
pipe_initialize(cid);                         /* initialization */
pipe_write(buffer, size, n_item, cid);        /* send data to child.  Upon
                                                receiving all necessary
                                                data, the child program will
                                                start processing */
pipe_read(buffer, size, n_item, cid);         /* get data from child after
                                                the child program is
                                                finished */
pipe_close(cid);                              /* close the pipe */
```

where `cid` is the child identifier and `"child program"` is the name of the child to be activated. The `buffer`, `size` and `n_item`, refer to the buffer for the data in the parent program, size of the data item and the number of the data items to be sent, respectively.

More general and useful is the iterative case, where instead of loading and activating the child program in each iteration (the *pipe_open( )* function) we do it only once outside a loop in the parent program.

```
cid = pipe_open("child_program");         /* open pipe and activate the
                                             child program */
for (i = 0; i < 100; i++) {               /* 100 iter. in the parent
                                             program */
   pipe_initialize(cid);                  /* initialization */
   pipe_write(buffer, size, n_item, cid); /* send data to child; upon
                                             receiving all necessary
                                             data, the child program will
                                             start processing */
   pipe_read(buffer, size, n_item, cid);  /* get data from child after
                                             data processing in the child
                                             program is finished */
   /* data processing in parent program here */
}
pipe_close(cid);                          /* exit child and close the
                                             pipe */
```

The use of pipe communication by the child program is the same regardless whether the parent calls the child iteratively or non-iteratively. This is due to an infinite loop which should be set up by the user in the child program. The loop will perform the desired data processing as long as it is requested to do so by the parent program which sends an initializing signal to the child in each iteration. Eventually, by sending a closing signal to the child the parent will terminate data processing and close the pipe. General use of pipe communication in the child program is as follows.

```
for (;;) {                                /* set up an infinite loop */
   pipe_initialize2();                    /* initialize (synchronize with
                                             the parent) */
   pipe_read2(buffer, size, n_item);      /* get data from parent */

   /* data processing in the child program */

   pipe_write2(buffer, size, n_item);     /* send data to parent */
}
```

It should be emphasized that the infinite loop must be present in the child program even if the child is going to be called only once. The main reason for this is that the *pipe_initialize2()* function is used to receive not only the synchronization signals but also the termination signal.

The IPPC library also allows one parent program to communicate with several child and grandchild programs. Up to 127 concurrently running children created by any parent are allowed.

For further details and examples of utilization of the IPPC library see [4, 5].

V. THE STRUCTURE OF THE INTERCONNECTION

While approaching the problem of coupling OSA90/hope and SPICE-PAC we had to solve a number of problems. Some of them were purely implementation related problems while the others were more conceptual ones. An example from the first group could be the problem of how to pass the optimization variables from OSA90/hope to SPICE-PAC and then the SPICE-PAC responses back to OSA90/hope. Another problem was how to initialize SPICE-PAC to perform specific circuit analyses and avoid such an initialization in subsequent calls. An example of a more conceptual problem was how to deal with two input files, one for OSA90/hope and another one for SPICE-PAC. Dealing with two independent input files would be quite inconvenient for the user. Ideally, it is preferable to work with only one input file.

We concentrated our effort on designing Spicepipe in such a way that the user would not be required to do any programming. Furthermore, we assumed that the user should be able to invoke SPICE-PAC from OSA90/hope in a completely "invisible" manner. To this end we created a version of the SPICE-PAC driver which is linked, together with SPICE-PAC, to a short interfacing driver forming altogether Spicepipe: a pipe-ready version of SPICE-PAC. The structure of the connection between OSA90/hope and SPICE-PAC is shown in Fig. 2. The interfacing driver, see Fig. 2, was created on the basis of the general child template described in [1]. The SPICE-PAC driver organizes SPICE-PAC's simulation primitives. The *create_file* child, called through the separate Datapipe channel is employed to help the user in dealing with two input files, one for SPICE-PAC and another one for OSA90/hope.

The interfacing driver, SPICE-PAC driver and the *create_file* child are discussed in detail in the following sections.

VI. INTERFACING DRIVER

The IPPC library, described in Section IV, likewise the template for an OSA90/hope child [1], are written in C. SPICE-PAC, on the other hand, is written in FORTRAN. The most straightforward approach to combine SPICE-PAC and OSA90/hope is to call SPICE-PAC from inside a small C program whose primary task is to maintain communication between the two systems. We called this C program an interfacing driver. It is the *main()* function of Spicepipe.

The interfacing driver was created on the basis of the general child template but we extended that template by adding some error checking and dynamic memory allocation. The program is able to report a number of the most common errors, e.g., syntax errors in the SPICE-PAC input file, unknown analysis types passed to SPICE-PAC, etc. The interfacing driver also performs data type conversion. The conversion is necessary because SPICE-PAC works in double precision and OSA90/hope in single precision arithmetic. Therefore all the output from SPICE-PAC has to be cast from double to single precision floating point numbers.

The listing of the interfacing driver source code can be found in Appendix B. Appendix C contains a short discussion on how to pass data from C to FORTRAN routines.

VII. SPICE-PAC DRIVER

Here we discuss the structure of our version of the SPICE-PAC driver and the analysis types available through this driver.

The structure of the driver is shown in Fig. 3. The two paths of operation flow in Fig. 3. correspond to the first and subsequent calls to SPICE-PAC. If the user requests a single simulation of a circuit using SPICE-PAC, there will be only one call to SPICE-PAC and the operation flow will follow the path for the first entry to SPICE-PAC. If the user wants to perform optimization or multiple simulations of the same circuit, the second and all subsequent calls will skip the initialization operations of SPICE-PAC. Such organization significantly saves CPU time.

During initialization, SPICE-PAC first tries to open its input and output files. If the files have been successfully opened, SPICE-PAC will call the *SPICEA* subroutine to read the input file. Next, the program checks if the input data from OSA90/hope is consistent with definitions in the SPICE-PAC input file. The input data, if it exists, consists mainly of sweep or optimization variables. If the information is consistent *SPICEB* is called to define the corresponding SPICE-PAC circuit variables (see Section III). A call to *SPICEM*, defining the temperature for subsequent analysis, completes the initialization process.

The successive SPICE-PAC tasks are performed in every call. First, the values of sweep or optimization variables are updated by a call to *SPICEU* and then *SPICER* is called to perform the requested analysis. Before returning control to the interfacing driver and then to OSA90/hope, SPICE-PAC saves the results of the analysis in its output file. This takes place, however, only if the user requested SPICE-PAC to do so by setting an additional flag in the OSA90/hope input file. We will describe this flag in detail in the section on the OSA90/hope input file for Spicepipe.

If an error is detected upon a call to any of the invoked subroutines, SPICE-PAC sets the error flag and returns control to the interfacing driver immediately. The described SPICE-PAC driver, based on the standard SPICE-PAC driver created by W.M. Zuberek [6], is intended to support general usage of SPICE-PAC. The user, however, can extend this driver or adjust it to a particular application.

Our implementation of the SPICE-PAC driver allows the user to request one type of analysis at a time, out of the following: DC transfer curve analysis, transient analysis, AC analysis, noise analysis, distortion analysis or Fourier analysis. If two or more analyses are required the user has to create a separate Datapipe communication channel for each of the analysis types.

The source code of our SPICE-PAC driver is listed in Appendix D.

VIII. THE *create_file* CHILD

Both OSA90/hope and SPICE-PAC require their own input files. Having two input files is thus necessary. However, they are interdependent and to synchronise the activities of both systems the files have to be consistent. To allow the user to work with one input file only we have created a separate pipe-ready executable child program for OSA90/hope named *create_file*. *create_file*, if called from OSA90/hope through the Datapipe mechanism, will create a disk file, specifically an input file for SPICE-PAC. The name of the file as well as its contents are both character string type arguments for the second Datapipe in Fig. 2. In other words the user can define his or her SPICE-PAC input file inside the OSA90/hope input file as a character string.

The following example illustrates the usage of the *create_file* child.

```
...
Expression
   char file_name[] = "testfile.txt";
   char file_contents[] = "This is the text which will constitute the
contents of the 'testfile.txt' file.";
   Datapipe:
      COM               FILE = "create_file"
      N_INPUT = 2       INPUT = (file_name, file_contents)
      N_OUTPUT = 1      OUTPUT = (in_name[13]);
End
...
```

We use the `COM` Datapipe protocol with two inputs: `file_name` and `file_contents`. `file_name` contains the name of the file to be created and `file_contents` contains its contents. Both are defined as character string variables in the lines preceding the Datapipe definition. The Datapipe returns as its output `in_name`, which is equal to the Datapipe's input `file_name`. Having the created file name returned as the output from the *create_file* Datapipe is very convenient. For example, instead of organizing the OSA90/hope input file in such a way so that the *create_file* is executed whenever necessary it is enough to specify the *create_file* output `in_name` as input to the Datapipe that relies on the existence of the `in_name` file. Then, whenever OSA90/hope evaluates the other Datapipe it will first determine its inputs or in other words it will automatically execute the *create_file* child. We use this mechanism to relate the *create_file* Datapipe with the Spicepipe Datapipe.

It should be stressed that *create_file* is not actually a part of Spicepipe nor is it absolutely necessary to use Spicepipe. *create_file* was designed to facilitate the process of creating and editing the input file for SPICE-PAC from OSA90/hope design environment. Nevertheless, if the SPICE-PAC file already exists and editing is not necessary, *create_file* is redundant. Of course, *create_file* can be used in other applications, not only with Spicepipe.

The source code of the *create_file* program is listed in Appendix E.

IX. OSA90/hope INPUT FILE FOR Spicepipe

As we know, both OSA90/hope and SPICE-PAC communicate with the user by means of input files. In each system the input file describes the circuit under consideration as well as the operations requested by the user. In the preceding section we learned how to use *create_file* to combine two input files, one for OSA90/hope and the other one for SPICE-PAC. The resulting file is an OSA90/hope input file which contains information required by both systems. In this section we examine this file in detail. After reading this section the user should be able to create his or her own OSA90/hope input files utilizing Spicepipe.

The OSA90/hope input file for Spicepipe is in fact an ordinary OSA90/hope input file except for two, specific to Spicepipe, aspects. First, the OSA90/hope input file for Spicepipe must define the Spicepipe Datapipe and second it must be consistent with the corresponding input file for SPICE-PAC. We will analyze these aspects first, and then we will support the analysis with a complete circuit example.

A. The Spicepipe Datapipe

Once the SPICE-PAC input file for a given problem has been created, either externally or through the *create_file* child program, the user has to organize the transfer of data between SPICE-PAC and OSA90/hope. In order to do so a `COM` datapipe channel between the two packages has to be opened and input and output to this Datapipe have to be specified. The syntax of the COM datapipe protocol was described in Section II. The Datapipe invoking Spicepipe is defined as follows.

```
Datapipe:
        COM         FILE = "Spicepipe"
        N_INPUT = n INPUT = (in_name, dump, out_name, ana_name, input)
        N_OUTPUT= m OUTPUT  = (output);
```

where *Spicepipe* is the name of the executable child program, (see Fig. 2.), n is the number of inputs to the pipe and m is the number of outputs from the pipe. n and m are determined by the contents of INPUT and OUTPUT. In INPUT there are five fields of data:

1. in_name is a required character string which specifies the name of the SPICE-PAC input file.
2. dump is a required float flag which tells SPICE-PAC whether the SPICE-PAC output file should include results of circuit analysis or not (if dump=0 the results will not be stored, otherwise SPICE-PAC will save results of each circuit analysis in its output file).
3. out_name is a required character string which specifies the name of the SPICE-PAC output file.
4. ana_name is a required character string which specifies the type of analysis to be performed by SPICE-PAC.
5. input is an optional float vector specifying input variables.

The vector input may contain an arbitrary number of float type variables. The length of the input vector together with in_name, dump, out_name and ana_name (each counted as 1) determine n. Therefore n is 4 if the input field is absent, or n is 4 plus the number of float variables in the input vector if it is present.

Two last fields require some more explanation. ana_name informs SPICE-PAC which analysis should be performed. The valid values of ana_name are: ".dc", ".tr", ".ac", ".no",

".di" and ".fo", also: ".DC", ".TR", ".AC", ".NO", ".DI" and ".FO". These correspond to the DC, transient, AC, noise, distortion and Fourier analyses respectively. An attempt to set ana_name to a different value would cause the child program to terminate and return an error message.

The input field defines which variables from the OSA90/hope input file should be passed to SPICE-PAC. The primary usage of input is to transfer optimization variables, though some other sweep or even fixed parameters may be passed as well. In the case of optimization the transfer of variables takes places not once but as many times as required by the optimizer (see Section VII).

Finally, N_OUTPUT and OUTPUT of the Datapipe definition are used to describe data which is sent back from SPICE-PAC to OSA90/hope. N_OUTPUT is equal to the number of floating point variables to be received by OSA90/hope and OUTPUT is a set of placeholders defining space for these floating point variables.

B. Consistency of the input files

As already mentioned in Section VIII, the input files for OSA90/hope and SPICE-PAC have to be consistent. It is not surprising if the systems are to cooperate. There are basically three things that have to match in both input files. They are the circuit variables, the type of the analysis and the output from the Spicepipe Datapipe.

The input vector in the OSA90/hope Spicepipe Datapipe definition (described in the previous subsection) and SPICE-PAC circuit variables (described in Section III) must correspond to each other. If there are no SPICE-PAC circuit variables the input vector in the OSA90/hope Spicepipe Datapipe definition must not be defined. If there are a number of SPICE-PAC circuit variables defined in the SPICE-PAC input file the input vector must contain exactly the same number of OSA90/hope variables. Furthermore, the order of the variables in the input vector must correspond to the order in which the corresponding SPICE-PAC circuit variables are defined. If, for example, the following circuit variables were defined in the SPICE-PAC input file

```
...
RE 3 0 150
RB 2 5 950K
RC 4 5 5K
CB 1 2 100UF

...

.END/EXT
.VAR RE
.VAR RB
.VAR RC
.VAR CB
.END
```

then, in the corresponding OSA90/hope input file the user should define a four element `input` vector of variables, e.g.,

```
...

re = ?150?;  rb = ?950e3?;  rc = 5e3;  cb = 100e-6;
input[1:4] = [re, rb, rc, cb];

...
```

The names of variables in the SPICE-PAC and OSA90/hope input files do not have to be the same. Notice, that `re` and `rb` are OSA90/hope optimization variables (they are defined with a pair of question marks) and `rc` and `cb` are not. The consequence of the specific ordering of the variables in `input[1:4]` is that `re`, `rb`, `rc` and `cb` correspond to SPICE-PAC's `RE`, `RB`, `RC` and `CB`, respectively.

To match the analysis types for OSA90/hope and SPICE-PAC is simple. The `ana_name` character string variable in the Spicepipe Datapipe input determines the analysis to be performed by SPICE-PAC. The selection is restricted to what has been declared in the SPICE-PAC input file. If the following two lines, defining AC and DC analysis respectively, were included in the SPICE-PAC input file and there were no other analysis type declarations

```
...

.AC LIN 20 5M 100M
.DC VCC 5 12 1

...
```

the available values for ana_name would be ".ac" and ".dc" (or ".AC" and ".DC"). An attempt to set ana_name to a different value would cause Spicepipe to return an error message and terminate.

The output from the Spicepipe Datapipe is the third and last place where the two input files must be consistent. The Spicepipe Datapipe output is strongly related to the SPICE-PAC .PRINT statement(s). The .PRINT statement in the SPICE-PAC input file defines the output for a given analysis. Different .PRINT statements can be defined for different analyses. For example, the following lines

```
...

.AC LIN 20 5M 100M
.DC VIN 0 5 0.1
.PRINT AC VR(4) VI(4) VM(4)
.PRINT DC V(3)

...
```

define outputs for the AC and DC analyses. In this example, the AC analysis returns the real, imaginary and the magnitude values of the voltage at node 4 for 20 equally spaced frequency points in the range from 5MHz to 100MHZ. To maintain consistency, the output of the Spicepipe Datapipe should allocate space for 60 (3x20) floating point numbers. Consequently, N_OUTPUT in the Spicepipe Datapipe should be set to 60. It may look as follows.

```
... N_OUTPUT = 60     OUTPUT = (VR4[1:20], VI4[1:20], VM4[1:20]) ...
```

In the case of the DC analysis the output contains 51 values of the voltage at node 3, which may be declared as

```
... N_OUTPUT = 51     OUTPUT = (V3DC[1:51]) ...
```

C. An illustrative example of the OSA90/hope input file for Spicepipe

Consider an OSA90/hope input file for optimization of the modulus of the input reflection coefficient of an LC transformer circuit. The input file for the problem could be as follows.

```
#define DUMPON        1
#define DUMPOFF       0
Expression
  char cir_contents[]=
" ****************************
 * TRANSFORMER SIMULATION *
 ****************************
VG    1 0 AC 1
RIN   1 2 3
C6    2 0 1
L5    2 3 1
C4    3 0 1
L3    3 4 1
C2    4 0 1
L1    4 5 1
ROUT  5 0 1
.PRINT AC VR(2) VI(2)
.AC LIN 21 0.079578H 0.187644H
.END/EXT
.VAR L1
.VAR L3
.VAR L5
.VAR C2
.VAR C4
.VAR C6
.END
";
   char cir_name[]="lc6.cir";
   char out_name[]="lc6.out";
   char ac[]=".ac";
   Datapipe:
        COM                      FILE = "create_file"
        N_INPUT = 2              INPUT = (cir_name, cir_contents)
        N_OUTPUT = 1             OUTPUT = (char in_name[8]);
   input[1:6]=[?1?, ?1?, ?1?, ?1?, ?1?, ?1?];
   DataPipe:
        COM                      FILE = "Spicepipe"
        N_INPUT = 10             INPUT = (in_name, DUMPOFF, out_name, ac, input)
        N_OUTPUT = 42            OUTPUT = (Vinr[1:21], Vini[1:21]);
! Calculate the reflection coefficient |Ref| using Ref=2*Vin/Vg-1 where Vg=1
   Refr[1:21]=Vinr+Vinr-1;
   Refi[1:21]=Vini+Vini;
   Refm[1:21]=sqrt(Refr*Refr+Refi*Refi);
   i=0;
End
Sweep
   Title = "Reflection Coefficient"  i: from 1 to 21 step 1 Refm[i];
End
Specification
   Refm=0;
End
```

There are two Datapipes in the file, *create_file* and Spicepipe (with *Spicepipe* as the executable program). We use the *create_file* Datapipe to create the SPICE-PAC input file, see Section VIII. There are two character string inputs to this Datapipe: `cir_contents` and `cir_name`. `cir_contents` defines the contents of the SPICE-PAC input file and `cir_name` defines its name, *lc6.cir* in this case. The output of the *create_file* Datapipe is a character string variable containing the name of the created file, just like `cir_name`. This output is then used as a part of the input to the Spicepipe Datapipe which chains in a way the two Datapipes. This makes the *create_file* Datapipe be executed before the Spicepipe Datapipe is invoked. There are six SPICE-PAC circuit variables defined in the input file, describing all L and C elements in the circuit. The `.AC` and `.PRINT` statements call for the real and imaginary parts of the AC voltage at node 2 at 21 frequencies as the output from SPICE-PAC.

The Spicepipe Datapipe, in order to be consistent with the SPICE-PAC input file, asks for the AC analysis (`char ac[]=".ac"`) and provides a six element vector `input` containing the values for SPICE-PAC L and C elements. It is worth noticing that the values assigned to the L and C elements in the "basic" part of the SPICE-PAC input file will first be read by SPICE-PAC and then they will be overwritten by the updating procedure with the values from `input` sent from OSA90/hope (see Fig. 3). The `N_OUTPUT` is equal to 42 (2×21) and `OUTPUT` provides space for 42 floating point numbers. `DUMPOFF`, predefined as 0, forces SPICE-PAC not to save the results in its output disk file. To evaluate the modulus of the reflection coefficient we use the expression processing capability of OSA90/hope. The modulus of the reflection coefficient is finally given as the `Refm` vector. The `Sweep` block defines the display, and the `Specification` block defines the error functions for optimization.

X. APPLICATIONS

A. Optimization of an LC transformer

An LC transformer optimization, referred to already in Section IX, is chosen to demonstrate communication between OSA90/hope and SPICE-PAC. We want to optimize the modulus of the input reflection coefficient $|\rho|$ for the transformer of Fig. 4. We use 21 equally spaced points in the frequency range from 0.079578Hz to 0.187644Hz. All L and C elements in the circuit are optimizable. The input resistance is R_{in}=3Ω and the output resistance is R_{out}=1Ω.

We use the *create_file* child to create the input file for SPICE-PAC. We use the expression processing capability of OSA90/hope to calculate the reflection coefficient, providing relevant formulas in the input file. The maximum value of $|\rho|$ before optimization was 0.66. After 62 iterations, using the minimax optimizer, it is decreased to 0.076. The values of the L and C elements before and after optimization are listed in Table I. The diagrams of $|\rho|$ as a function of frequency before and after optimization are shown in Fig. 5.

We also solved the problem entirely by OSA90/hope. The results are practically the same. Small differences are most likely due to different numerical algorithms used in both simulators. The CPU times used running OSA90/hope with the Spicepipe connection to SPICE-PAC and OSA90/hope alone were approximately the same.

B. Time-domain response and Monte Carlo analysis of an NMOS inverter

An NMOS inverter with depletion load [7], shown in Fig. 6, is used to illustrate the utilization of Spicepipe to perform Monte Carlo analysis with time-domain specifications. Monte Carlo analysis is organized within the OSA90/hope design environment but the actual circuit simulations are performed by the SPICE-PAC time-domain simulator. We used the level 1 option of the SPICE-PAC MOS transistor model [3, 7] to model the transistors. We selected channel length, channel width, threshold voltage and transconductance of both load and inverting transistors as the statistical parameters. In reality, production variations of the threshold voltage and transconductance are the most notable ones. We assumed normal distributions of the parameters with no correlations for simplicity. See Table II for transistor model parameters and statistical distributions assumed for statistical variables. We selected the inverter's propagation time $t_P < 2.5$ns as the acceptability criterion. The propagation time t_P was computed by an additional child program. The inverter was excited by a trapezoidal signal and its output was connected to another inverter of the same type to simulate a more realistic load. We did not include statistical variations in the load inverter. We also did not include the interconnection capacitances. The production yield, estimated using 200 outcomes, was 79.5%. The time-domain response of the nominal circuit as well as the Monte Carlo results are presented in Fig. 7.

C. Mixed frequency-time-domain optimization of an RLC circuit

This example demonstrates the mixed domain optimization capability available through Spicepipe. We want to find a second-order model, with the schematic of Fig. 8, of a fourth-order system when the input to the system is an impulse. We consider the time interval from 0 to 10 seconds. The fourth-order system time-domain response is given analytically by

$$V_{out}(t) = \frac{3}{20}e^{-t} + \frac{1}{52}e^{-5t} - \frac{1}{65}e^{-2t}(3\sin 2t + 11\cos 2t). \qquad (1)$$

The diagram of this response is shown in Fig. 9. In addition, we impose a frequency-domain specification on the insertion loss INSL of the modelling circuit. We want INSL to be less than 20dB in the frequency range from 0.1Hz to 0.4Hz. C_1, R_1 and R_2 are optimizable variables; R_{in}, R_{out} and L_1 are fixed.

We used the OSA90/hope ℓ_1 optimizer to perform optimization of the nominal circuit. The time-domain response of the second-order circuit was matched to (1). The maximum difference between the desired response (1) and the model response was reduced from 0.15 to 0.01, which satisfied our specifications. INSL satisfied the 20dB specification in the whole frequency range of interest.

Having found the optimum ℓ_1 solution to the problem we performed statistical analysis of the circuit. The Monte Carlo estimate of the production yield at the solution of the nominal problem was 50%. After 30 yield optimization iterations the yield was increased to 90.5%. We used the OSA90/hope yield optimizer with 50 outcomes to optimize yield. To estimate yield, before and after optimization, we used 200 outcomes. Table III lists the values of the optimization variables and assumed standard deviations for statistical variables. Fig. 10 shows the results of Monte Carlo analysis performed before yield optimization. The error between the time-domain response of the second-order circuit and the desired response (1) as well as `INSL` are plotted. The corresponding curves generated after yield optimization are plotted in Fig. 11.

The listings of the OSA90/hope input files for all three problems are provided in Appendices F through H.

XI. CONCLUSIONS

We have described Spicepipe, a new child for OSA90/hope. Spicepipe, integrating OSA90/hope with SPICE-PAC, provides the user with all the features of OSA90/hope extended by the time-domain and noise analyses contributed by SPICE-PAC. Spicepipe augments the OSA90/hope modelling capabilities by the device models featured in SPICE-PAC. On the other hand, exploiting SPICE-PAC simulators through the OSA90/hope design environment is more flexible and efficient. The responses evaluated by SPICE-PAC and returned to OSA90/hope can be post-processed using expressions giving the user the flexibility of utilizing other than predefined responses. Expression processing makes it also possible to impose algebraic relations among SPICE-PAC circuit parameters. Probably the most important feature of Spicepipe, however, is that by combining tools working in different domains it provides the user with the unique capability of simulating and optimizing a circuit in the frequency and time domains simultaneously.

We have used the Datapipe technology and the Datapipe Server to interconnect SPICE-PAC with OSA90/hope. We introduced OSA90/hope and SPICE-PAC. We discussed the capability of OSA90/hope circuit design environment. The modular structure and the simulation primitives of SPICE-PAC were also discussed. We pointed out the superiority of SPICE-PAC over SPICE for specialised or optimization-related applications. The OSA90/hope input file for Spicepipe was described in detail. We emphasized the Spicepipe Datapipe definition and the consistency of the OSA90/hope and SPICE-PAC input files. We illustrated the utilization of Spicepipe with three design problems involving simulation, optimization, Monte Carlo analysis and yield-driven design in both the frequency and time domains.

To combine OSA90/hope with SPICE-PAC no additional reprogramming of OSA90/hope was required. For SPICE-PAC we had to create the driver organizing SPICE-PAC simulation primitives, but a driver has to be written for SPICE-PAC anyway. Having to write such a driver requires that the user possess some programming knowledge as well as a good understanding of SPICE-PAC simulation primitives.

XII. ACKNOWLEDGMENT

The authors wish to express their appreciation to Dr. W.M. Zuberek of Memorial University of Newfoundland, St. John's, Newfoundland, Canada, the author of SPICE-PAC, for his assistance, helpful discussions, technical suggestions and comments.

REFERENCES

[1] *OSA90/hope™ Version 2.0 User's Manual*, Optimization Systems Associates, Inc., P.O. Box 8083, Dundas, Ontario, Canada L9H 5E7, 1992.

[2] W.M. Zuberek, "SPICE-PAC version 2G6c an overview," Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1C 5S7, Technical Report 8903, 1989.

[3] A. Vladimirescu, K. Zhang, A.R. Newton, D.O. Pederson and A.L. Sangiovanni-Vincentelli, "SPICE Version 2G - User's guide," Department of Electrical Engineering and Computer Sciences, University of California, Berkeley CA 94720, 1981.

[4] J.W Bandler, Q.J. Zhang, G. Simpson and S.H. Chen, "IPPC: a library for inter-program pipe communication," Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada, Report SOS-90-10-U, 1990.

[5] *Programming Utilities and Libraries*, SPARCstation 1 Users Manual, Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043, pp. 21-26, 1988.

[6] W.M. Zuberek, "SPICE-PAC version 2G6c user's guide", Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1C 5S7, Technical Report 8902, 1989.

[7] D.A. Hodges and H.G. Jackson, *Analysis and Design of Digital Integrated Circuits.* New York: McGraw-Hill, Inc., 1988.

[8] *1.2 Sun FORTRAN Manual Set*, SPARCstation 1 Users Manual, Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043, 1988.

TABLE I
LC TRANSFORMER CIRCUIT:
L AND *C* ELEMENT VALUES BEFORE AND AFTER OPTIMIZATION

Element	L_1 (H)	C_2 (F)	L_3 (H)	C_4 (F)	L_5 (H)	C_6 (F)
Value Before Optimization	1	1	1	1	1	1
Value After Optimization	1.041	0.979	2.340	0.780	2.937	0.347

TABLE II
NMOS INVERTER:
MODEL PARAMETERS AND DISTRIBUTIONS ASSUMED

Name	SPICE-PAC Variable	Mean Value	Standard Deviation (%)
Both transistors:			
Transconductance	KP (A/V^2)	20.0	6.0
Body factor	GAMMA (V$^{1/2}$)	0.37	-
Body doping	NSUB (cm^{-3})	5.0×10^{14}	-
Gate oxide thickness	TOX (m)	0.1×10^{-6}	-
Junction depth	XJ (m)	1.0×10^{-6}	-
Lateral diffusion	LD (m)	1.0×10^{-6}	-
Zero-bias bulk capacitance	CJ (F/m^2)	70.0×10^{-6}	-
Zero-bias perimeter capacitance	CJSW (F/m)	220.0×10^{-12}	-
Gate-drain overlap capacitance	CGDO (F/m)	345.0×10^{-12}	-
Gate-source overlap capacitance	CGSO (F/m)	345.0×10^{-12}	-
The load transistor:			
Threshold voltage	VTO (V)	-3.0	12.0
Gate width	W (m)	5.0×10^{-6}	2.0
Gate length	L (m)	12.0×10^{-6}	2.0
Drain diffusion area	AD (m^2)	100.0×10^{-12}	-
Source diffusion area	AS (m^2)	25.0×10^{-12}	-
Drain area perimeter	PD (m)	40.0×10^{-6}	-
Source area perimeter	PS (m)	15.0×10^{-6}	-
The inverting transistor:			
Threshold voltage	VTO (V)	1.0	12.0
Gate width	W (m)	10.0×10^{-6}	2.0
Gate length	L (m)	7.0×10^{-6}	2.0
Drain diffusion area	AD (m^2)	100.0×10^{-12}	-
Source diffusion area	AS (m^2)	100.0×10^{-12}	-
Drain area perimeter	PD (m)	35.0×10^{-6}	-
Source area perimeter	PS (m)	40.0×10^{-6}	-

- indicates elements assumed fixed and non-statistical.

TABLE III
YIELD OPTIMIZATION OF THE RLC CIRCUIT

Element	Before Optimization	After ℓ_1 Optimization	After Yield Optimization	Standard Deviation (%)
R_{in} (Ω)	1.00	1.00	1.00	-
R_1 (Ω)	0.50	0.92	1.00	5
C_1 (F)	0.50	0.43	0.44	5
L_1 (H)	1.00	1.00	1.00	5
R_2 (Ω)	2.00	0.28	0.26	5
R_{out} (Ω)	1.00	1.00	1.00	-

- indicates elements assumed fixed and non-statistical.

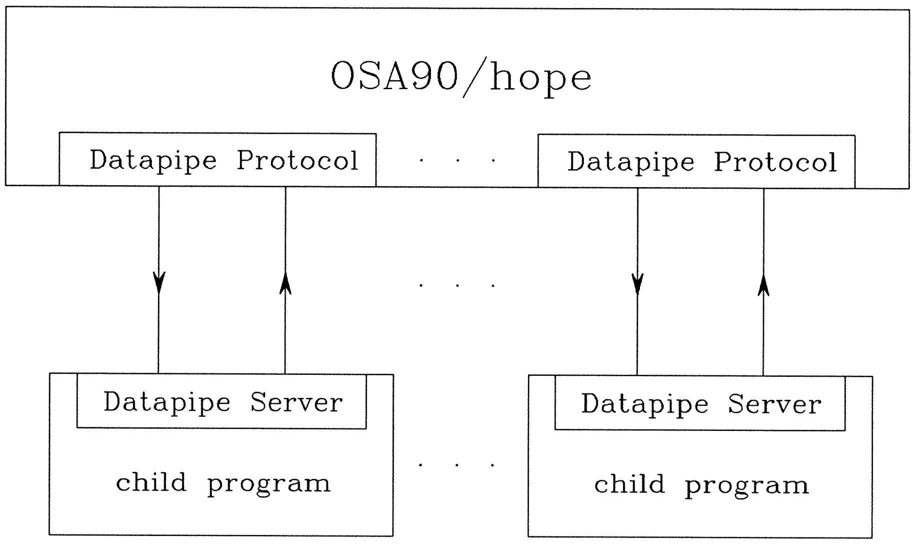


Fig. 1. OSA90/hope Datapipe schematic. Several child programs can be connected to OSA90/hope using Datapipe technology.

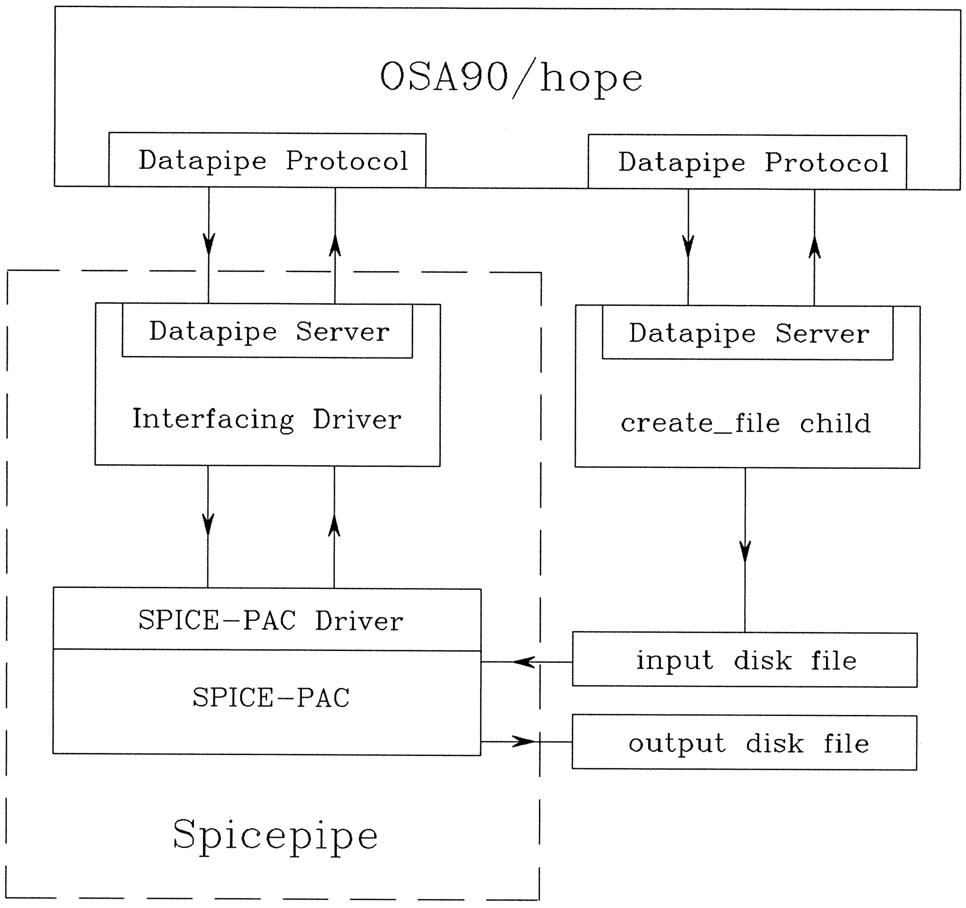


Fig. 2. Integrating SPICE-PAC with OSA90/hope. Spicepipe consists of the interfacing driver, SPICE-PAC driver and the SPICE-PAC library. The *create_file* child creates the SPICE-PAC input disk file.

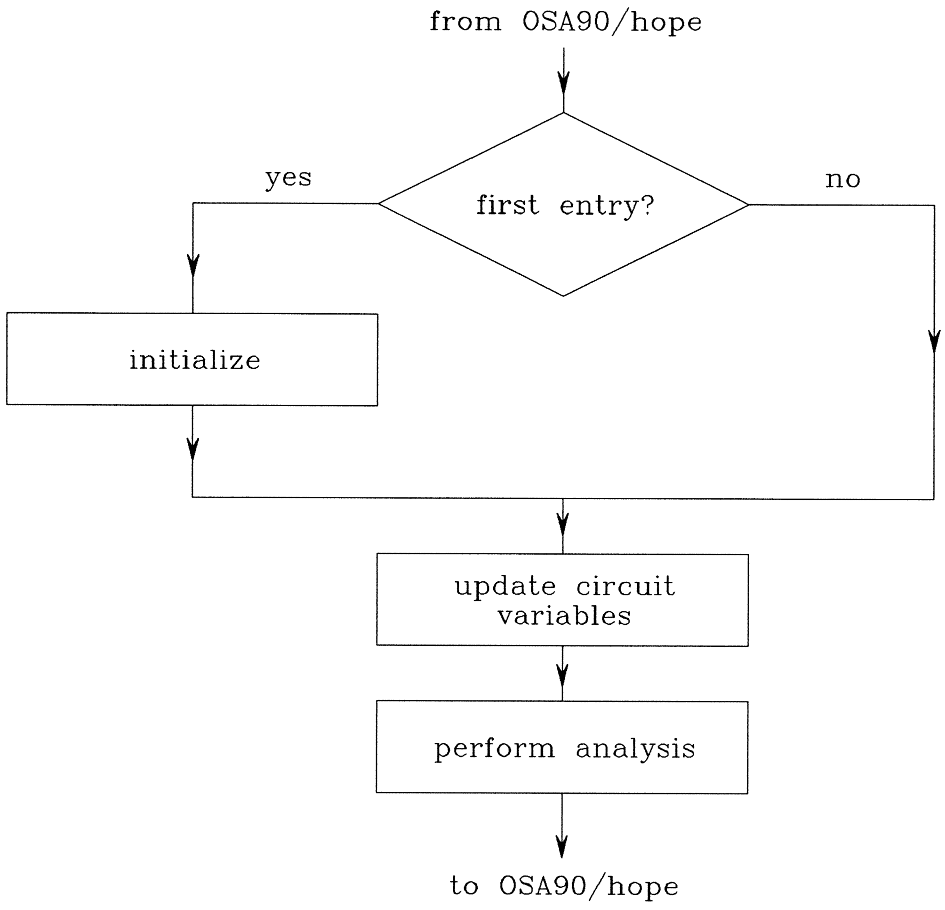


Fig. 3. Block diagram of the SPICE-PAC driver.

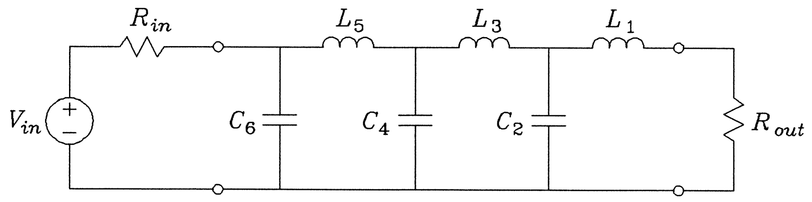
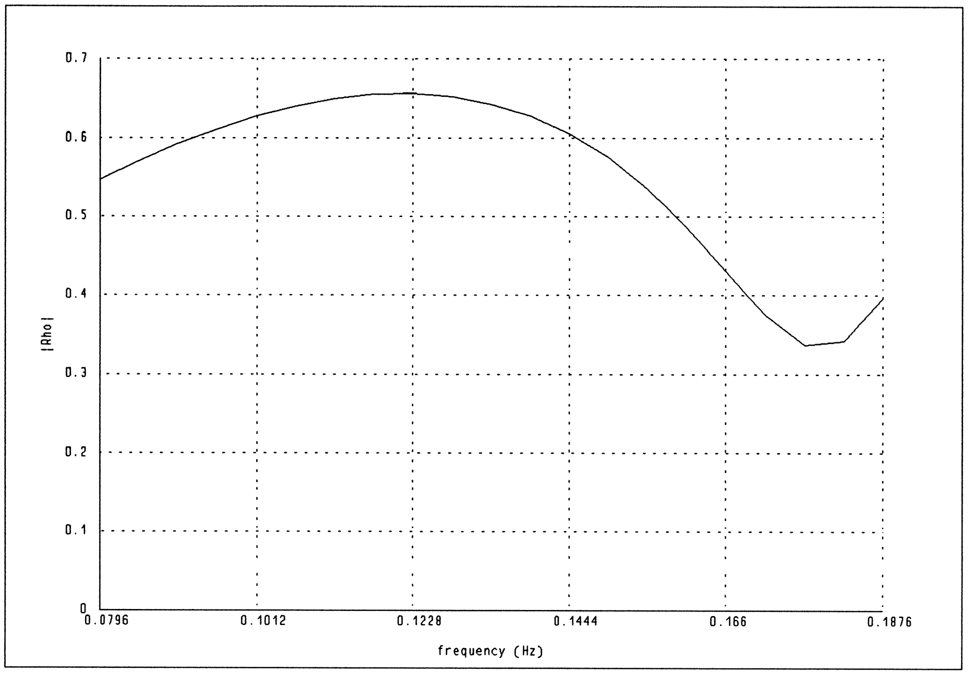
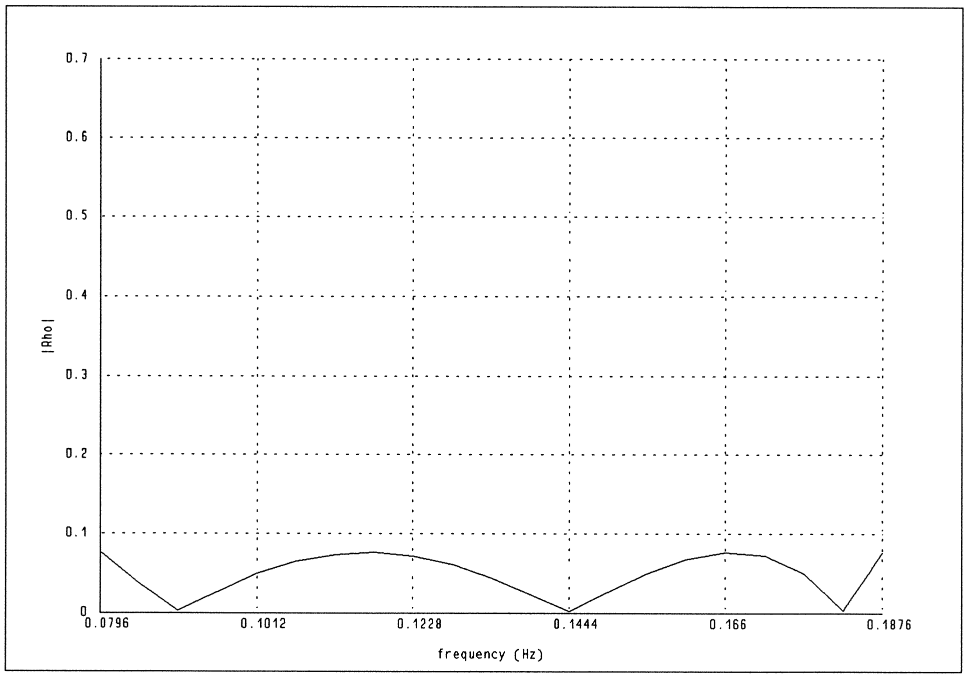


Fig. 4. LC transformer circuit.



(a)



(b)

Fig. 5. Input reflection coefficient of the LC transformer (a) before and (b) after optimization.

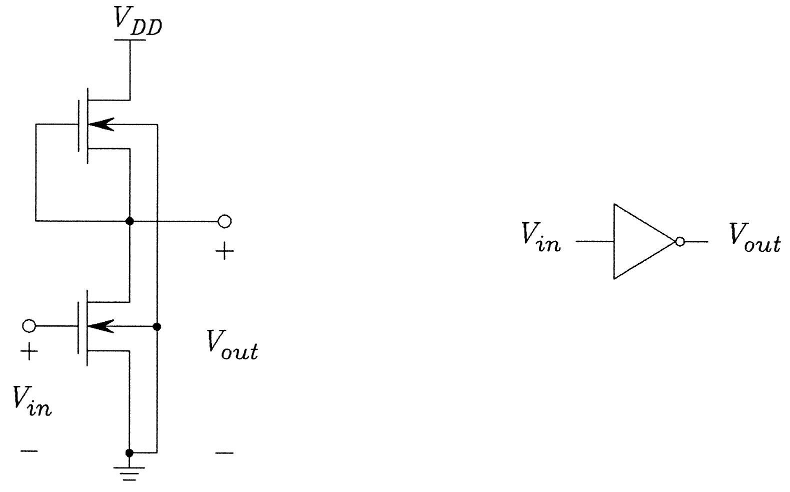
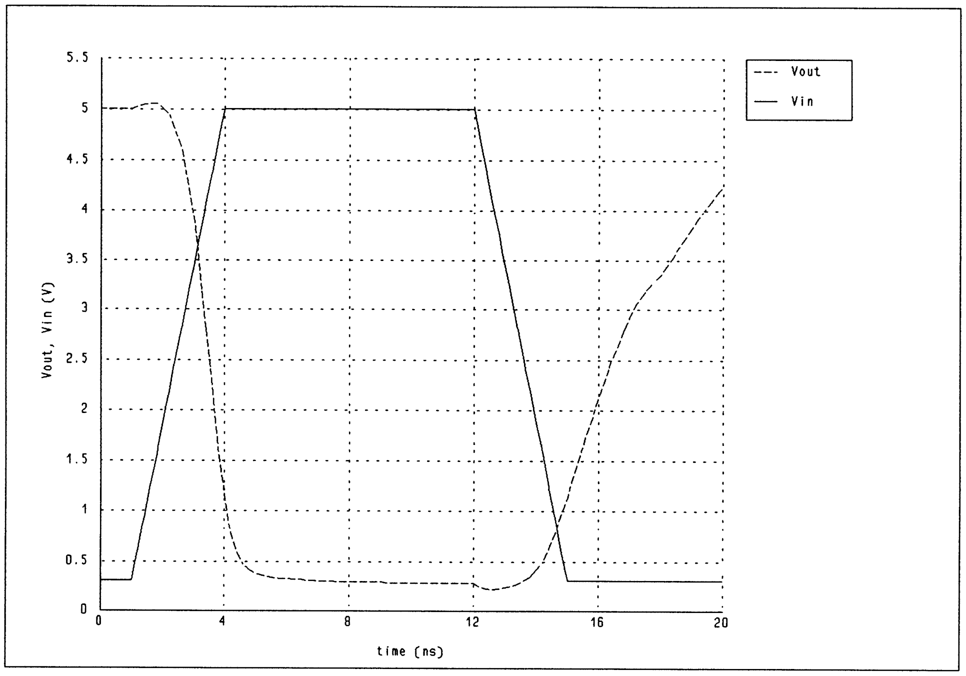
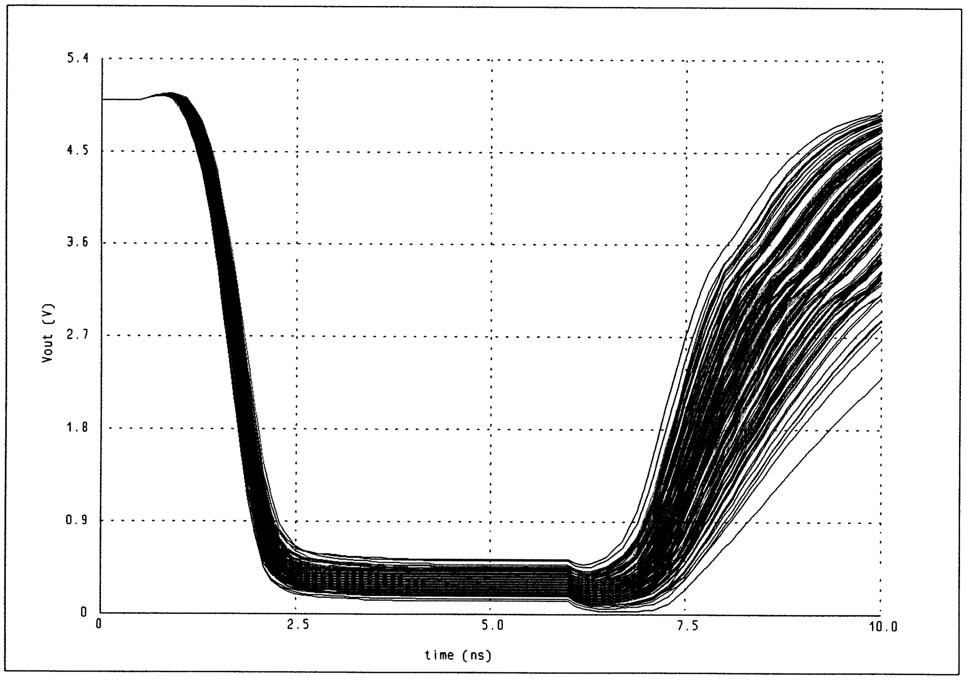


Fig. 6. NMOS inverter, depletion load [7].



(a)



(b)

Fig. 7. Time-domain responses of an NMOS inverter. (a) input and output waveforms and (b) Monte Carlo output waveform sweep.

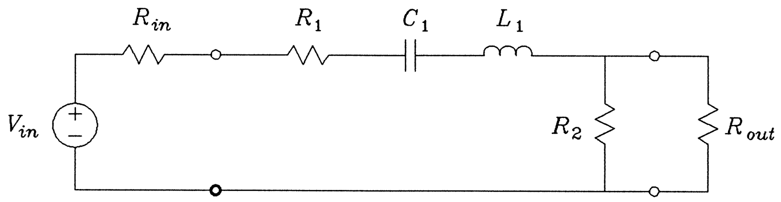


Fig. 8. RLC second-order circuit.

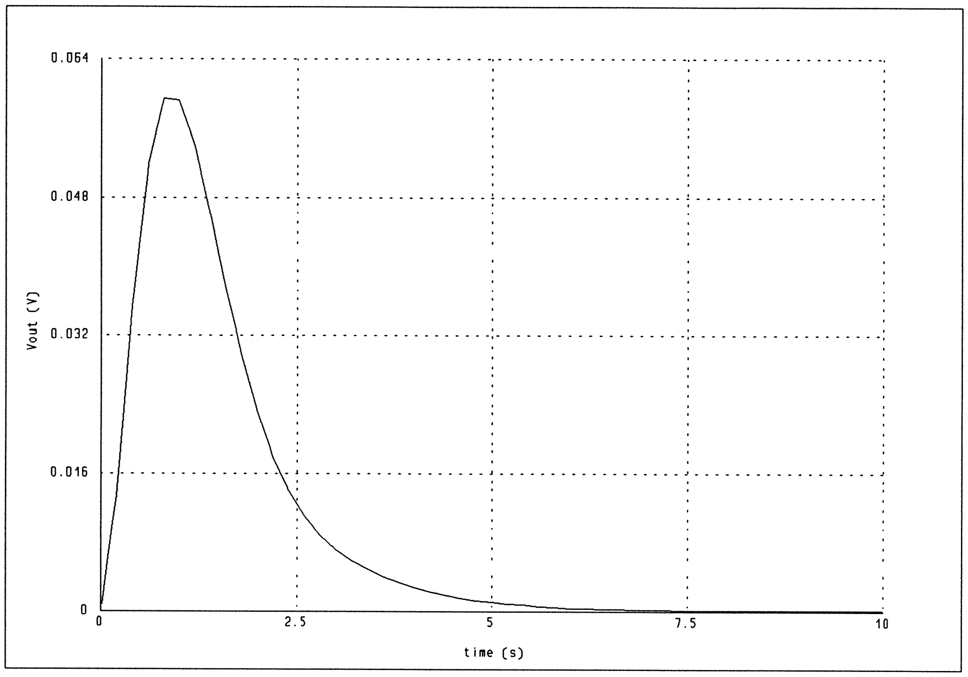
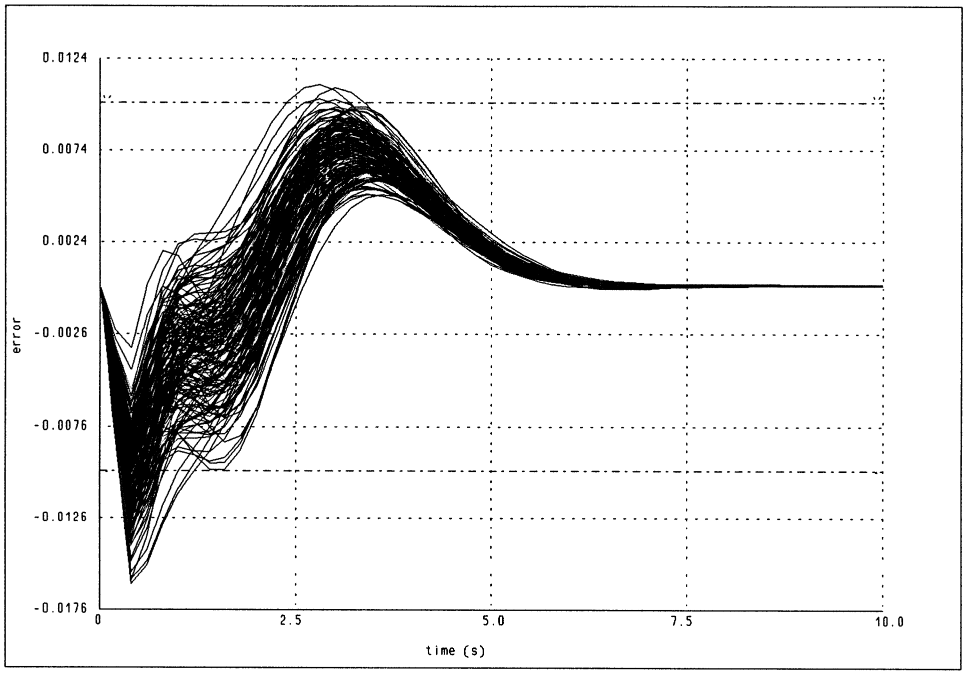
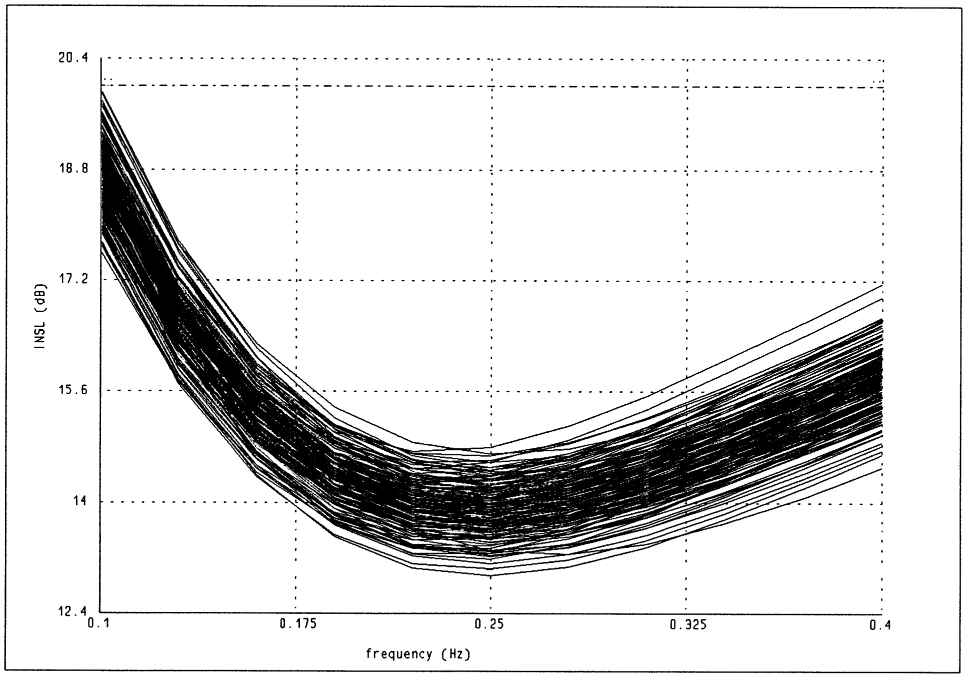


Fig. 9. Impulse response of the fourth-order system, given analytically by (1).

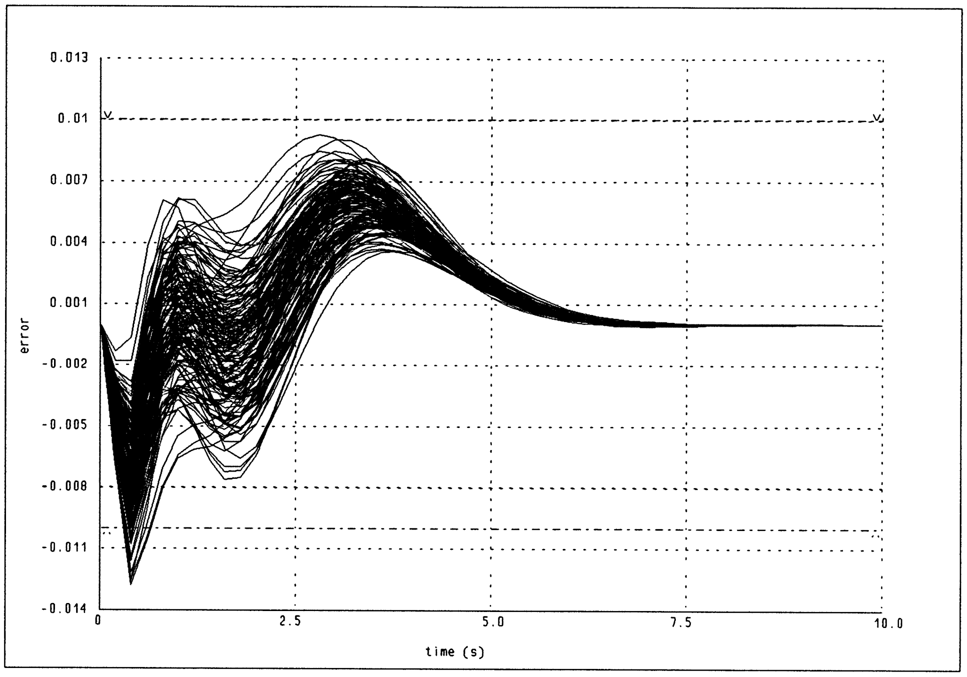


(a)

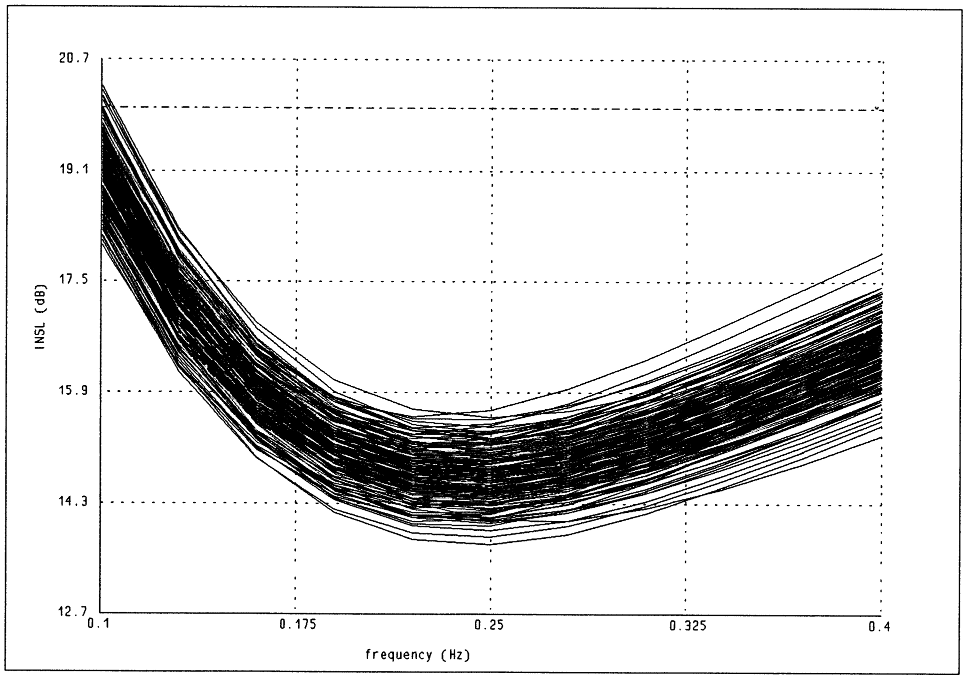


(b)

Fig. 10. Monte Carlo sweep results for the RLC circuit before yield optimization: (a) time-domain match error, and (b) insertion loss.



(a)



(b)

Fig. 11. Monte Carlo sweep results for the RLC circuit after yield optimization: (a) time-domain match error, and (b) insertion loss.

APPENDIX A

HOW TO OBTAIN SPICE-PAC

SPICE-PAC is a version of the public domain software circuit simulator SPICE. Anyone interested can obtain SPICE-PAC free of charge.

To obtain SPICE-PAC you can use the `ftp` utility and connect to the *garfield* computer at the Memorial University of Newfoundland, Canada. The command to invoke `ftp` is

```
ftp garfield.cs.mun.ca
```

or

```
ftp 134.153.1.1
```

When the connection is established type `login` at the `FTP>` prompt. Log into the anonymous account providing your e-mail address as the password:

```
Foreign username: anonymous
Password: ...
```

SPICE-PAC can be found in the *pub/sppac* subdirectory. The *pub/sppac* subdirectory contains the following files: *Makefile*, *Readme*, *sppac.tar.Z*, *tr-8902.tex.Z* and *tr-8903.tex.Z*. *Makefile* installs SPICE-PAC, *Readme* provides brief information about SPICE-PAC and *sppac.tar.Z* is a compressed and "tared" version of SPICE-PAC. *tr-8902.tex.Z* and *tr-8903.tex.Z* are compressed versions of the reports [6] and [2], respectively, written in LaTEX.

Transfer the files to your local machine using the following commands:

```
get Readme
get Makefile
binary
get sppac.tar.Z
get tr-8902.tex.Z
get tr-8903.tex.Z
```

If you do not need the reports [2] and [6], or you do not have an access to LaTEX you do not have to transfer the last two files. This completes the process of obtaining SPICE-PAC.

Directions on how to install SPICE-PAC can be found in the *Readme* file. The package includes an exemplary SPICE-PAC driver. The driver provides an interactive interface between the user and SPICE-PAC simulators. It is also an excellent reference on how to write more specialized SPICE-PAC drivers.

To create Spicepipe change the last line in *Makefile* from

```
f77 -o sppac sppac-drv.f sppac.a
```

into the following sequence of commands:

```
cc  -c -o Sppipe_c.o Sppipe_c.c
f77 -c -o Sppipe_f.o Sppipe_f.f
cc  -o Spicepipe Sppipe_c.o Sppipe_f.o ippcv2.o sppac.a -lF77 -lc -lm
rm Sppipe_c.o Sppipe_f.o
```

Sppipe_c.c and *Sppipe_f.f* are the interfacing driver and SPICE-PAC driver, respectively. *ippcv2.o* contains Datapipe functions used by *Sppipe_c.c*. The source codes of *Sppipe_c.c* and *Sppipe_f.f* are listed in Appendices B and D, respectively.

APPENDIX B

SOURCE CODE OF THE INTERFACING DRIVER

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "ippcv2.h"

#define LR  1001
#define LC    12

void* mymalloc();                    /* used to allocate memory for input data */

char  error_str[128];
char  error_strs[12][128];
FILE* tmpp;

void main()
{
  int     input_no,                  /* number of input variables (=2) */
          output_no,                 /* number of output variables (=1, dummy) */
          group_no,                  /* number of groups (=2) */
          i, j, k, l,                /* loop counters */
          error=0,                   /* stores the error message */
          errorlen,                  /* stores the length of an error message */
          data_type,                 /* stores the type of a group */
          data_size,                 /* stores the size of a group */
          row_no,                    /* number of output rows */
          col_no,                    /* number of output columns */
          bad_group_no=-1,           /* indicates wrong data group */
          exp_data_type,             /* indicates expected data type */
          firstentry=1;              /* just in case of subsequent entries */
  void**   input_data;               /* contains addresses to the data groups */
  float    outputdataf[LR*LC];
  double   outputdatad[LR*LC];
  double*  inputdatad;

                                     /* error messages */
  strcpy(error_strs[0], "");
  strcpy(error_strs[1], "Spicepipe:  Cannot open the input file for SPICE-PAC.");
  strcpy(error_strs[2], "Spicepipe:  Cannot open the output file for SPICE-PAC.");
  strcpy(error_strs[3], "Spicepipe:  An error occurred during initialization.");
  strcpy(error_strs[4], "Spiecpipe:  An error occurred while defining variables .");
  strcpy(error_strs[5], "Spicepipe:  An error occurred during initialization.");
  strcpy(error_strs[6], "Spicepipe:  An error occurred while setting the temperature.");
  strcpy(error_strs[7], "Spicepipe:  Unknown analysis type passed to SPICE-PAC.");
  strcpy(error_strs[8], "Spicepipe:  An error occurred while updating variables.");
  strcpy(error_strs[9], "Spicepipe:  An error occurred while performing the analysis.");
  strcpy(error_strs[10], "Spicepipe:  The number of circuit variables not consistent.");
  for (;;)
  {
    pipe_initialize2();              /* read data header */
    pipe_read2(&input_no, sizeof(int), 1);
    pipe_read2(&output_no, sizeof(int), 1);
    pipe_read2(&group_no, sizeof(int), 1);
    if(firstentry)                   /* allocate memory for data group pointers */
      input_data=mymalloc(group_no*sizeof(void*));
```

```
for(i=0;i<group_no;i++)
{                               /* read data group header */
  pipe_read2(&data_type, sizeof(int), 1);
  pipe_read2(&data_size, sizeof(int), 1);
  if(data_type==IPPC_DATA_CHAR)
  {                             /* read char string data */
    if(firstentry)
    {                                                       /*  groups  0,  2  and  3  must  be  strings  */
      if((i!=0)&&(i!=2)&&(i!=3)&&(bad_group_no==-1))
      {
        bad_group_no=i;
        exp_data_type=IPPC_DATA_FLOAT;
      }                         /* allocate memory for the data */
      input_data[i]=mymalloc(data_size);
    }                           /* read data */
    pipe_read2(input_data[i], 1, data_size);
  }
  else                          /* data type must be float */
  {
    if(firstentry)
    {                           /* groups 1 and 4 must be float (0 - 4) */
      if((i!=1)&&(i!=4)&&(bad_group_no==-1))
      {
        bad_group_no=i;
        exp_data_type=IPPC_DATA_CHAR;
      }                         /* allocate memory for the data */
      input_data[i]=mymalloc(data_size*sizeof(float));
    }                           /* read data */
    pipe_read2(input_data[i], sizeof(float), data_size);
  }
}
if(firstentry)
{                               /* check if right number of groups (4 or 5) */
  if((group_no!=4)&&(group_no!=5))
  {         /* if not report error and terminate (pipe_initialize2()) */
    error=strlen(sprintf(error_str,
              "Spicepipe:  wrong # of input data groups - should be 4 or 5.", group_no))+1;
    pipe_write2(&error, sizeof(int), 1);
    pipe_write2(error_str, 1, error);
    pipe_initialize2();
  }
 if(bad_group_no!=-1)
    if(exp_data_type==IPPC_DATA_CHAR)
      error=strlen(sprintf(error_str,
        "Spicepipe:  wrong input data type - 'char*' type expected in data group: %d.", bad_group_no+1))+1;
    else
      error=strlen(sprintf(error_str,
          "Spicepipe:  wrong data type - 'float' type expected in data group: %d.", bad_group_no+1))+1;
  if(error)
  {
    pipe_write2(&error, sizeof(int), 1);
    pipe_write2(error_str, 1, error);
    pipe_initialize2();
  }
  if(group_no==5)
    inputdatad=mymalloc(data_size*sizeof(double));
  firstentry--;/* no mem alloc. and error check from now, should be O.K. */
}
if(group_no==5)                 /* data conversion from float to double */
  for(i=0;i<data_size;i++)
    inputdatad[i]=*((float*)input_data[4]+i);
else
  data_size=0;
spicepac_(&error, &row_no, &col_no, (float*)input_data[1], inputdatad,
    outputdatad, &data_size, (char*)input_data[0], (char*)input_data[2],
    (char*)input_data[3], strlen(input_data[0]), strlen(input_data[2]),
    strlen(input_data[3]));
    for(i=0,k=0;k<col_no;k++)
    {
      for(j=0;j<row_no;j++,i++)
      {
        outputdataf[i]=outputdatad[LR*k+j];
      }
    }    errorlen=strlen(error_strs[error]);
    pipe_write2(&errorlen, sizeof(int), 1);
    if(error)
      pipe_write2(error_strs[error], 1, errorlen);
    else
    {
      data_type=IPPC_DATA_FLOAT;
      pipe_write2(&data_type, sizeof(int), 1);
      pipe_write2(&output_no, sizeof(int), 1);
      pipe_write2(outputdataf, sizeof(float), output_no);
    }
  }
}


void* mymalloc(malloc_size)
int malloc_size;
{
  void* pointer=NULL;
  int   error=0;

  if((pointer=(void*)malloc(malloc_size))==NULL)
  {                                 /* if error, report it and terminate */
    error=strlen(strcpy(
          error_str, "Spicepipe:  memory allocation error in 'msppacc'"))+1;
    pipe_write2(&error, sizeof(int), 1);
    pipe_write2(error_str, 1, error);
    pipe_initialize2();
  }
  return(pointer);
}
```

APPENDIX C

C - FORTRAN INTERFACE RULES

The interfacing driver, described in Chapter VI, supports data transfer between OSA90/hope and SPICE-PAC. Because the interfacing driver is written in C and SPICE-PAC in FORTRAN the C - FORTRAN interface rules for a SPARCstation 1 had to be applied to maintain the communication. Here we will limit ourselves to a short explanation and simple examples of how to pass float, integer and string variables from C to FORTRAN. A more complete description can be found in [8].

Because FORTRAN refers to a variable through its address a call from C to FORTRAN must provide the addresses of all the parameters in the calling statement. An address in C can be obtained by using the "&" operation. This is sufficient for float and integer variables whose formats are the same in C and FORTRAN. Unfortunately internal formats for character strings are different in C and FORTRAN. Therefore, special rules have to be obeyed to make FORTRAN understand C character strings. The basic reason for this is that C does not need to know a character string's length (due to the NULL terminating character), while FORTRAN does. To let FORTRAN know a character string's length we have to find out the string's length and pass it to FORTRAN as an additional parameter. In the parameter list of a FORTRAN function called from C this additional parameter has to be listed as the last parameter in the list. For example, if we want to pass the *string1* character string from a C program to a FORTRAN routine named *WRITEIT* the line calling *WRITEIT* in the C program should be:

```
...
writeit_(string1, strlen(string1));
...
```

where the "_" underline character is required by the Sun linker, *string1* is the address to the string and `strlen(string1)` is the string length. The corresponding FORTRAN routine may look like

```
SUBROUTINE WRITEIT(STRING1)
CHARACTER STRING1*(*)

WRITE(*,*)STRING1

RETURN
END
```

3 strings and an integer could be passed as follows

C program line:

```
...
writeit_(string1, strign2, &integer, string3, strlen(string1), strlen(string2),
         strlen(string3));
...
```

FORTRAN program:

```
SUBROUTINE WRITEIT(STRING1, STRING2, INTEG, STRING3)
CHARACTER STRING1*(*), STRING2*(*), STRING3*(*)

WRITE(*,*)STRING1
WRITE(*,*)STRING2
WRITE(*,*)STRING3
WRITE(*,*)INTEG

RETURN
END
```

According to [8] the C and FORTRAN programs should be compiled and linked using the following options:

```
cc  c_program.c  f_program.f  -lF77  -lI77  -lU77  -lc -lm
```

APPENDIX D

SOURCE CODE LISTING OF THE SPICE-PAC DRIVER

In this appendix we include a complete listing of the SPICE-PAC driver. It has been adapted from the standard SPICE-PAC driver written by W.M. Zuberek and included in the SPICE-PAC package. Subroutines: *EXTIME*, *ERROR*, *OUTRES* and *SETHDR* were taken directly from the standard SPICE-PAC driver.

```
      SUBROUTINE SPICEPAC(RETERROR, IR, IC, DUMP, INPUTDATA,
     +          YTAB, NRVAR, INPUTFILE, OUTPUTFILE, ANALYSIS)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      PARAMETER (LTAB=9,LVAR=200,LR=1001,LC=12,LZ=12012,LX=5)
C
C     LV          - max number of (static) circuit variables (SPICEB)
C     LR          - max number of results from a single analysis
C     LC          - max number of output variables for some analyses
C     LZ          - max number of numerical results from a single analysis
C     LX          - max number of parameters for a single analysis
C
      CHARACTER OUTPUTFILE*(*), INPUTFILE*(*), ANALYSIS*(*)
      CHARACTER*20 ATAB(10)
      CHARACTER*3  TAA(LTAB), TAB(LTAB)
      DOUBLE PRECISION INPUTDATA(*)
      INTEGER RETERROR,FIRSTENTRY,NRVAR,NRV
      LOGICAL CHECK
      REAL DUMP
      DIMENSION IDVAR(LVAR)
      DIMENSION XTAB(LR),YTAB(LR,LC),IVTAB(LX)
      DATA FIRSTENTRY /1/
      DATA ATAB /'DC TRANSFER CURVE   ','TRANSIENT ANALYSIS  ',
     1          'AC ANALYSIS         ','NOISE ANALYSIS      ',
     2          'DISTORTION ANALYSIS ','FOURIER ANALYSIS    ',
     3          'DC TRANSFER FUNCTION','AC SENSITIVITIES    ',
     4          'TR SENSITIVITIES    ','DC SENSITIVITIES    ' /
      DATA TAA /'.dc', '.tr', '.ac', '.no', '.di', '.fo', '.tf',
     1         '.ds', '.as'/
      DATA TAB /'.DC', '.TR', '.AC', '.NO', '.DI', '.FO', '.TF',
     2         '.DS', '.AS'/
C
C  set default values
C
      LENFPT=2
      LENPRL=75
C
C  check if first entry
C
      IF (FIRSTENTRY.EQ.1) THEN
        FIRSTENTRY=0
C
C  open I/O files
C
        INQUIRE(FILE=INPUTFILE,EXIST=CHECK)
        IF (.NOT.CHECK) THEN
          RETERROR=1
          RETURN
        ENDIF
        NRINPF=9
        OPEN(UNIT=NRINPF,FILE=INPUTFILE,STATUS='OLD')
        INQUIRE(FILE=OUTPUTFILE,EXIST=CHECK)
      NROUTF=8
      IF (CHECK) THEN
        OPEN(UNIT=NROUTF,FILE=OUTPUTFILE,STATUS='OLD',ERR=12)
      ELSE
        OPEN(UNIT=NROUTF,FILE=OUTPUTFILE,STATUS='NEW',ERR=12)
      ENDIF
       ENDFILE NROUTF
      GO TO 15
   12 RETERROR=2
      GO TO 99
C
C  initialize
C
   15 CALL SPICEA(NRINPF,NROUTF,NRV)
      IF (NRV.LT.0) THEN
        RETERROR=3
        CALL ERROR(NROUTF,'A',NRV,*99)
      ENDIF
      IF (NRVAR.NE.NRV) THEN
        RETERROR=11
        GO TO 99
      ENDIF
C
C  retrieve circuit variables
C
      IF (NRV.NE.0) THEN
        CALL SPICEB('*RETRIEVE',IDVAR,LVAR,IEX)
        IF (IEX.NE.NRV) THEN
          RETERROR=4
          CALL ERROR(NROUTF,'B/*RETRIEVE',IEX,*99)
        ENDIF
      ENDIF
C
C  continue
C
   20 CALL SPICEC(IEX)
      IF (IEX.NE.0) THEN
        RETERROR=5
        CALL ERROR(NROUTF,'C',IEX,*99)
      ENDIF
C
C  retrieve the temperature
C
      TEMP=-300D0
      CALL SPICEM(TEMP,IEX)
      IF (IEX.NE.0) THEN
        RETERROR=6
        CALL ERROR(NROUTF,'M',IEX,*99)
      ENDIF
    ENDIF
   25 DO 30 I=1, LTAB
      IF(ANALYSIS.EQ.TAA(I)) GO TO 35
      IF(ANALYSIS.EQ.TAB(I)) GO TO 35
   30 CONTINUE
    RETERROR=7
    RETURN
C
C   .DC, .TR, .AC, .NO, .DI, .FO, .TF, .AS
C
C  update circuit variables
C
   35 IF(NRVAR.NE.0) THEN
      CALL SPICEU(IDVAR,INPUTDATA,NRVAR,IEX)
      IF(IEX.NE.NRVAR) THEN
        RETERROR=8
        CALL ERROR(NROUTF,'U',IEX,*99)
      ENDIF
    ENDIF
```

```
      CALL SPICER(I,XTAB,YTAB,LR,-LZ,IR,IC,IEX)
      IF (IEX.LT.0) THEN
        RETERROR=9
        CALL ERROR(NROUTF,'R',IEX,*99)
      ENDIF
      IF(DUMP.NE.0)THEN
        JR=MIN0(LR,LZ/IC)
        CALL OUTRES(NROUTF,I,ATAB(I),TEMP,IVTAB,0,XTAB,YTAB,JR,
     1            IC,IR,IC,2, LENPRL,LENFPT)
      ENDIF
C
C  print *SPICE-PAC* execution time
C
   90 CALL EXTIME(NROUTF,XTAB,LR)
      RETERROR=0
   99 RETURN
      END

C --- extime ------ 89.03.12 (W.M.Zuberek) ---------------------------*
      SUBROUTINE EXTIME (NRF,XT,LX)
      DOUBLE PRECISION XT(1)
      CHARACTER*3 Y
      CALL SPICEW(XT,LX,IEX)
      IF (IEX.NE.8) CALL ERROR(NRF,'W',IEX,*99)
      Y='sec'
      IF (XT(1).GE.600.0) THEN
        XT(1)=XT(1)/60D0
        Y='min'
        IF (XT(1).GE.600.0) THEN
          XT(1)=XT(1)/60D0
          Y='hrs'
        ENDIF
      ENDIF
      WRITE(NRF,900) XT(1),Y
  900 FORMAT(/' *SPICE-PAC* execution time :',F6.1,1X,A)
   99 RETURN
      END
C --- error ------- 88.06.12 (W.M.Zuberek) ---------------------------*
      SUBROUTINE ERROR (NRF,E,IND,*)
C
C  This subroutine prints error messages stored in a file "sppac.err".
C
      CHARACTER*1 E,X,Y,Q,R
      CHARACTER*9 Z
      CHARACTER*60 S
      LOGICAL CHECK
      DATA Z / 'sppac.err' /
      INQUIRE(FILE=Z,EXIST=CHECK)
      IF (.NOT.CHECK) GO TO 90
      OPEN(UNIT=10,FILE=Z,STATUS='OLD')
   10 READ(10,100,END=80,ERR=10) X,N,Y,Q,S
  100 FORMAT(1X,A,I3,A,A,A)
      IF (E.NE.X) GO TO 10
      IF (Y.EQ.' ' .AND. N.NE.IND) GO TO 10
      IF (Y.EQ.'+' .AND. IND.LE.N) GO TO 10
      IF (Y.EQ.'-' .AND. IND.GE.N) GO TO 10
      WRITE(NRF,200) E,IND,S
  200 FORMAT(' *SPICE',A,'*',I4,' : ',A)
   20 IF (Q.NE.' ') THEN
        READ(10,100,END=90,ERR=10) X,N,R,Q,S
        IF (E.EQ.X .AND. (IND.EQ.N .OR. Y.EQ.R)) THEN
          WRITE(NRF,300) S
  300     FORMAT(13X,' : ',A)
          GO TO 20
        ENDIF
      ENDIF
      CLOSE(UNIT=10)
      RETURN 1
   80 CLOSE(UNIT=10)
   90 WRITE(NRF,900) E,IND
  900 FORMAT(' *SPICE',A,'* return code :',I4)
      RETURN 1
      END
C --- outres ------ 90.04.11 (W.M.Zuberek) ----------------------------*
      SUBROUTINE OUTRES (NRF,IT,TT,TEM,KV,NV,XTAB,YTAB,KR,KC,IR,IC,LB,
     &                   LLG,LFP)
C  This subroutine prints results of different analyses.
C  NRF - INTEGER, the unit number of the output file,
C  IT  - INTEGER, the type of analysis results,
C  TT  - CHARACTER (*), the header,
C  TEM - DOUBLE PRECISION, the temperature,
C  KV  - INTEGER (NV), array of circuit variable identifiers,
C  NV  - INTEGER, the number of declared circuit variables,
C  XTAB - DOUBLE PRECISION, the vector of arguments,
C  YTAB - DOUBLE PRECISION, the matrix of results,
C  KR  - INTEGER, the original number of rows,
C  KC  - INTEGER, the original number of columns,
C  IR  - INTEGER, the actual number of rows,
C  IC  - INTEGER, the actual number of columns,
C  LB  - INTEGER, the margin,
C  LLG - INTEGER, the line length,
C  LFP - INTEGER, the length of fractional part.
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION KV(1),XTAB(KR),YTAB(KR,KC)
      CHARACTER*(*) TT
      CHARACTER*40 VAN
      CHARACTER*27 F1,F2
      CHARACTER*16 HEAD(0:12),B
      CHARACTER*8 X
      EQUIVALENCE (X,Z)
      DATA F1 / '(1X,A,1PD10.2,2X,1P12D10.2)' /
      DATA F2 / '(1X,1A,2X,1A8,2X,1P12D10.2)' /
      LF=LFP
      IF (LF.GT.8) LF=8
      LN=LF+8
      I2=ICHAR('0')+MOD(LN,10)
      I1=ICHAR('0')+LN/10
Cmips:F1(10:13)=CHAR(I1)//CHAR(I2)//'.'//CHAR(ICHAR('0')+LF)
      F1(10:10)=CHAR(I1)
      F1(11:11)=CHAR(I2)
      F1(12:12)='.'
      F1(13:13)=CHAR(ICHAR('0')+LF)
      F1(23:26)=F1(10:13)
      F2(23:26)=F1(10:13)
      IF (IT.NE.10) THEN
        NC=(LLG-LN-LB-2)/LN
      ELSE
        NC=(LLG-LB-12)/LN
      ENDIF
      CALL SETHDR(HEAD(0),0,IT,LN,LH,*90)
      WRITE(NRF,110) TT,TEM
  110 FORMAT(/' ***** ',A,10X,'TEMPERATURE :',F7.2,' DEG C')
      IF (NV.GT.0) THEN
        WRITE(NRF,113)
  113   FORMAT(1X)
        DO 15 I=1,NV
          CALL SPICEY(VAN,0,KV(I),JEX)
          IF (JEX.LE.0) CALL ERROR(NRF,'Y',JEX,*90)
          CALL SPICEV(KV(I),VAL,1,IEX)
          IF (IEX.NE.1) CALL ERROR(NRF,'V',IEX,*90)
          WRITE(NRF,115) VAN(JEX:40),VAL
  115     FORMAT(' parameter : ',A,' =',1PD11.3)
   15   CONTINUE
      ENDIF
      DO 30 K=1,IC,NC
```

```
        M=MIN0(IC,K+NC-1)        DO 20 J=K,M
          IF (NV.GE.0) THEN
            CALL SETHDR(HEAD(J+1-K),J,IT,LN,LH,*90)
          ELSE
            CALL SETHDR(HEAD(J+1-K),-J,IT,LN,LH,*90)
          ENDIF
   20   CONTINUE
        B=' '
        WRITE(NRF,120) B(1:LB),HEAD(0)(1:LH),(HEAD(I)(1:LN),I=1,M+1-K)
  120   FORMAT(/1X,A,A,2X,12A)
        WRITE(NRF,130)
  130   FORMAT(1X)
        DO 25 I=1,IR
          IF (LB.EQ.10) THEN
            Z=YTAB(I,KC)
            B(3:10)=X
          ENDIF
          IF (IT.GE.0) THEN
            WRITE(NRF,F1) B(1:LB),XTAB(I),(YTAB(I,J),J=K,M)
          ELSE
            WRITE(NRF,F2) B(1:LB),XTAB(I),(YTAB(I,J),J=K,M)
          ENDIF
   25   CONTINUE
   30 CONTINUE
   90 RETURN
      END
C --- sethdr ------ 88.08.22 (W.M.Zuberek) ----------------------------*
      SUBROUTINE SETHDR (H,J,IT,LN,LH,*)
      CHARACTER*(*) H
      IF (J.EQ.0) THEN
        IF (IT.LE.0) THEN
          H='  OP-POINT'
          LH=10
          IEX=1
        ELSE
          CALL SPICEY(H(1:LN),IT,J,IEX)
          LL=LN
          LH=LN
        ENDIF
      ELSE IF (J.GT.0) THEN
        CALL SPICEY(H(1:LN),IT,J,IEX)
        LL=LN
      ELSE
        I=-J
        IEX=LN
        H=' '
   10   H(IEX:IEX)=CHAR(48+MOD(I,10))
        IEX=IEX-1
        IF (I.GT.9) THEN
          I=I/10
          GO TO 10
        ENDIF
        H(IEX:IEX)='#'
      ENDIF
      IF (IEX.LT.0) CALL ERROR(NRF,'Y',IEX,*90)
      IF (IEX.GT.12) THEN
        H=H(6:LL)//'     '
      ELSE IF (IEX.GT.10) THEN
        H=H(5:LL)//'    '
      ELSE IF (IEX.GT.8) THEN
        H=H(4:LL)//'   '
      ELSE IF (IEX.GT.6) THEN
        H=H(3:LL)//'  '
      ELSE
        IF (IEX.GT.4) H=H(2:LL)//' '
      ENDIF
      RETURN
   90 RETURN
1      END
```

APPENDIX E

SOURCE CODE LISTING OF THE *create_file* CHILD

Appendix E contains a listing of the *create_file* program which if called through OSA90/hope's `COM` datapipe will create an external disk file.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "ippcv2.h"

void* mymalloc();          /* used to allocate memory for input data */
char  error_str[128];      /* used to store the error string */

void main()
{
  int    input_no, /* number of input variables (=2) */
         output_no,/* number of output variables (=1, dummy) */
         group_no, /* number of groups (=2) */
         i,        /* loop counter */
         error=0,  /* stores the length of an error message (or -1) */
         data_type,/* stores the type of a group */
         data_size;/* stores the size of a group */
  void** input_data;/* contains addresses to the data groups */
  float  output=0; /* at least one output must be present */
  FILE*  sppac_cir;/* sppac circuit file pointer */

  for (;;)
  {
    pipe_initialize2();
    pipe_read2(&input_no, sizeof(int), 1);
    pipe_read2(&output_no, sizeof(int), 1);
    pipe_read2(&group_no, sizeof(int), 1);
    input_data=mymalloc(group_no*sizeof(void*));
    for(i=0;i<group_no;i++)
    {
      pipe_read2(&data_type, sizeof(int), 1);
      pipe_read2(&data_size, sizeof(int), 1);
      if(data_type==IPPC_DATA_CHAR)
      {
        input_data[i]=mymalloc(data_size);
        pipe_read2(input_data[i], 1, data_size);
      }
      else  /* dat_type must be IPPC_DATA_FLOAT */
      {
        input_data[i]=mymalloc(data_size*sizeof(float));
        pipe_read2(input_data[i], sizeof(float), data_size);
      }
    }
    if((sppac_cir=fopen((char*)input_data[0], "w"))!=NULL)
    {
      fprintf(sppac_cir, "%s", (char*)input_data[1]);
      fclose(sppac_cir);
    }
    else            /* opening error occurred */
    {
      strcpy(error_str, "create_file:  Cannot open '");
      error=strlen(strcat(strcat(error_str, (char*)input_data[0]), "'."))+1;
    }
    for(i=0;i<group_no;i++)
      free(input_data[i]);
    free(input_data);
    pipe_write2(&error, sizeof(int), 1);
    if(error)
      pipe_write2(error_str, 1, error);
    else
    {
      data_type=IPPC_DATA_CHAR;
      pipe_write2(&data_type, sizeof(int), 1);
      data_size=strlen((char*)input_data[0])+1;
      pipe_write2(&data_size, sizeof(int), 1);
      pipe_write2((char*)input_data[0], 1, data_size);
    }
  }
}

void* mymalloc(malloc_size)
int malloc_size;
{
  void* pointer=NULL;
  int   error=0;

  if((pointer=(void*)malloc(malloc_size))==NULL)
  {   /* memory alloc. error occurred, return the error and terminate the child */
    error=strlen(strcpy(
          error_str, "create_file:  Memory allocation error in 'create_cir'."))+1;
    pipe_write2(&error, sizeof(int), 1);
    pipe_write2(error_str, 1, error);
    pipe_initialize2();
  }
  return(pointer);
}
```

APPENDIX F

OSA90/hope INPUT FILE FOR THE LC TRANSFORMER EXAMPLE

```
#define DUMPON       1
#define DUMPOFF      0

Expression
  char cir_contents[]=
" ***************************
 * TRANSFORMER SIMULATION *
 ***************************
VIN   1 0 AC 1
RIN   1 2 3
C6    2 0 1
L5    2 3 1
C4    3 0 1
L3    3 4 1
C2    4 0 1
L1    4 5 1
ROUT  5 0 1
.PRINT AC VR(2) VI(2)
.AC LIN 21 0.079578H 0.187644H
.END/EXT
.VAR L1
.VAR L3
.VAR L5
.VAR C2
.VAR C4
.VAR C6
.END
";

   char cir_name[]="lc6.cir";
   char out_name[]="lc6.out";
   char ac[]=".ac";

   Datapipe:     COM                    FILE = "create_file"
                 TIMEOUT=0
                 N_INPUT = 2            INPUT = (cir_name, cir_contents)
                 N_OUTPUT = 1           OUTPUT = (char in_name[8]);

   input[1:6]=[?1?, ?1?, ?1?, ?1?, ?1?, ?1?];

   DataPipe:     COM                    FILE = "Spicepipe"
                 TIMEOUT=0
                 N_INPUT = 10           INPUT = (in_name, DUMPOFF, out_name, ac, input)
                 N_OUTPUT = 42          OUTPUT = (Vinr[1:21], Vini[1:21]);

! Now we will calculate the reflection coefficient |Ref|.
! from Ref=2*Vin/Vg-1 we have
   Refr[1:21]=Vinr+Vinr-1;
   Refi[1:21]=Vini+Vini;
   Refm[1:21]=sqrt(Refr*Refr+Refi*Refi);
   i=0;
   st=(0.187644-0.079578)/20;
   fr=0.079578+(i-1)*st;
End
```

```
Sweep
  Title = "Refm"  i: from 1 to 21 step 1 fr Refm[i]
  {PARAMETRIC  TITLE=""
               X_TITLE="frequency (Hz)"
               Y_TITLE="|S11|" Ymin=0 Ymax=0.7
               X=fr
               Y=Refm[i]
               NXTICKS=5 NYTICKS=7};
End

Specification
   Refm=0;
End
```

APPENDIX G

OSA90/hope INPUT FILE FOR THE MOS INVERTER EXAMPLE

```
#define DUMPON          1
#define DUMPOFF         0
#define UP              0
#define DOWN            1
#define Vmin            0.3
#define Vmax            5
#define TIMESTEP        0.2
Expression
  char cir_contents[] =
" *******************
 *  MOS inverter  *
 *******************
* SUBCIRCUIT DEFINITION, Nodes: Input, Output, VCC
 .SUBCKT NOTGATE 1 2 3
 M_INVER 2 1 0 0   NE W=10U L=7U  AD=100P PD=35U AS=100P PS=40U
 M_LOAD  3 2 2 0   ND W=5U  L=12U AD=100P PD=40U AS=25P  PS=15U
 .MODEL NE NMOS (VTO=1.0 KP=20U GAMMA=0.37 NSUB=5E14 TOX=0.1U XJ=1.0U
 +              LD=1.0U CJ=70U CJSW=220P CGSO=345P CGDO=345P)
 .MODEL ND NMOS (VTO=-3.0 KP=20U GAMMA=0.37 NSUB=5E14 TOX=0.1U XJ=1.0U
 +              LD=1.0U CJ=70U CJSW=220P CGSO=345P CGDO=345P)
 .ENDS NOTGATE
* NOMINAL CIRCUIT DEFINITION
 VDD 4 0 5
 VIN 1 0 PULSE(0.3 5 1N 3N 3N 8N 22N)
 XNOT1  1 2 4 NOTGATE
 XNOT2  2 3 4 NOTGATE
 .PRINT  TRAN V(2) V(1)
 .OPTIONS LIMPTS=5001
 .TRAN 0.2NS 20NS 0N
 .END/EXT
* INVER transistor variables of inverter I
 .VAR XNOT1.M_INVER'W
 .VAR XNOT1.M_INVER'L
 .VAR XNOT1.M_INVER'AD
 .VAR XNOT1.M_INVER'PD
 .VAR XNOT1.M_INVER'AS
 .VAR XNOT1.M_INVER'PS
 .VAR XNOT1.NE'VTO
 .VAR XNOT1.NE'KP
 .VAR XNOT1.NE'GAMMA
 .VAR XNOT1.NE'NSUB
 .VAR XNOT1.NE'TOX
 .VAR XNOT1.NE'XJ
 .VAR XNOT1.NE'LD
 .VAR XNOT1.NE'CJ
 .VAR XNOT1.NE'CJSW
 .VAR XNOT1.NE'CGSO
 .VAR XNOT1.NE'CGDO
* LOAD transistor variables of inverter I
 .VAR XNOT1.M_LOAD'W
 .VAR XNOT1.M_LOAD'L
 .VAR XNOT1.M_LOAD'AS
 .VAR XNOT1.M_LOAD'PS
 .VAR XNOT1.ND'VTO
 .VAR XNOT1.ND'KP
 .VAR XNOT1.ND'GAMMA
 .VAR XNOT1.ND'NSUB
 .VAR XNOT1.ND'TOX
 .VAR XNOT1.ND'XJ
 .VAR XNOT1.ND'LD
 .VAR XNOT1.ND'CJ
 .VAR XNOT1.ND'CJSW
 .VAR XNOT1.ND'CGSO
 .VAR XNOT1.ND'CGDO
 .END
";
   char cir_name[]="mos_inv.cir";
   Datapipe:      COM                        FILE = "create_file"
                  N_INPUT = 2                INPUT = (cir_name, cir_contents)
                  N_OUTPUT = 1               OUTPUT = (char in_name[12]);

   XNOT1_M_INVER_W = 10e-6          {Normal Sigma=2%};
   XNOT1_M_INVER_L = 7e-6           {Normal Sigma=2%};
   XNOT1_NE_VTO    = 1.0            {Normal Sigma=12%};
   XNOT1_NE_KP     = 20e-6          {Normal Sigma=6%};
   XNOT1_NE_NSUB   = 5e14;
   XNOT1_NE_TOX    = 0.1e-6;
   XNOT1_NE_XJ     = 1e-6;
   XNOT1_NE_LD     = 1e-6;

   XNOT1_M_LOAD_W  = ?5e-6?         {Normal Sigma=2%};
   XNOT1_M_LOAD_L  = 12e-6          {Normal Sigma=2%};
   XNOT1_ND_VTO    = -3.0           {Normal Sigma=12%};
   XNOT1_ND_KP     = 20e-6          {Normal Sigma=6%};
   XNOT1_ND_NSUB   = 5e14;
   XNOT1_ND_TOX    = 0.1e-6;
   XNOT1_ND_XJ     = 1e-6;
   XNOT1_ND_LD     = 1e-6;

   XNOT1_NE_PB    = 0.0259*log(XNOT1_NE_NSUB/2.1);
   XNOT1_ND_PB    = 0.0259*log(XNOT1_ND_NSUB/2.1);
   XNOT1_NE_CJ    = sqrt(1.6e-19*11.7*8.85e-6*XNOT1_NE_NSUB/2/XNOT1_NE_PB);
   XNOT1_ND_CJ    = sqrt(1.6e-19*11.7*8.85e-6*XNOT1_ND_NSUB/2/XNOT1_ND_PB);
   XNOT1_NE_CJSW = XNOT1_NE_XJ*sqrt(10)*XNOT1_NE_CJ;
   XNOT1_ND_CJSW = XNOT1_ND_XJ*sqrt(10)*XNOT1_ND_CJ;
   XNOT1_NE_Cox   = 3.97*8.85e-12/XNOT1_NE_TOX;
   XNOT1_ND_Cox   = 3.97*8.85e-12/XNOT1_ND_TOX;
   XNOT1_NE_CGSO = XNOT1_NE_Cox*XNOT1_NE_LD;
   XNOT1_NE_CGDO = XNOT1_NE_CGSO;
   XNOT1_ND_CGSO = XNOT1_ND_Cox*XNOT1_ND_LD;
   XNOT1_ND_CGDO = XNOT1_ND_CGSO;
   XNOT1_NE_GAMMA= sqrt(2*11.7*8.85e-6*1.6e-19*XNOT1_NE_NSUB)/XNOT1_NE_Cox;
   XNOT1_ND_GAMMA= sqrt(2*11.7*8.85e-6*1.6e-19*XNOT1_ND_NSUB)/XNOT1_ND_Cox;

   XNOT1_M_INVER_AD=10e-6*XNOT1_M_INVER_W;
   XNOT1_M_INVER_PD=20e-6+2*XNOT1_M_INVER_W-XNOT1_M_LOAD_W;
   XNOT1_M_INVER_AS=XNOT1_M_INVER_AD;
   XNOT1_M_INVER_PS=20e-6+2*XNOT1_M_INVER_W;
   XNOT1_M_LOAD_AS =5e-6*XNOT1_M_LOAD_W;
   XNOT1_M_LOAD_PS =10e-6+2*XNOT1_M_LOAD_W;

   char tr[]=".tr";
   char spp_out[]="mos_inv.out";
   input[1:32]=[XNOT1_M_INVER_W XNOT1_M_INVER_L XNOT1_M_INVER_AD XNOT1_M_INVER_PD
               XNOT1_M_INVER_AS XNOT1_M_INVER_PS
               XNOT1_NE_VTO XNOT1_NE_KP XNOT1_NE_GAMMA XNOT1_NE_NSUB XNOT1_NE_TOX
               XNOT1_NE_XJ XNOT1_NE_LD XNOT1_NE_CJ XNOT1_NE_CJSW XNOT1_NE_CGSO XNOT1_NE_CGDO
               XNOT1_M_LOAD_W XNOT1_M_LOAD_L XNOT1_M_LOAD_AS XNOT1_M_LOAD_PS
               XNOT1_ND_VTO XNOT1_ND_KP XNOT1_ND_GAMMA XNOT1_ND_NSUB XNOT1_ND_TOX
               XNOT1_ND_XJ XNOT1_ND_LD XNOT1_ND_CJ XNOT1_ND_CJSW XNOT1_ND_CGSO XNOT1_ND_CGDO];
   DataPipe:      COM                             FILE = "Spicepipe"
                  TIMEOUT=0
                  N_INPUT = 36                    INPUT = (in_name, DUMPOFF, spp_out, tr, input)
                  N_OUTPUT = 202          OUTPUT = (VOUT[1:101], VIN[1:101]);
! calculate the propagation time tp
   DataPipe:      SIM                             FILE = "tp"
                  TIMEOUT=0
                  N_INPUT = 104                   INPUT = (Vmin, Vmax, DOWN, VOUT)
                  N_OUTPUT = 2                    OUTPUT = (t21, t22);
   Vmed=(Vmax-Vmin)/2+Vmin;!   2.65, for Vmax=5 and Vmin=0.3;
   t201=t21+1; t202=if(t22>0)(t22+1) else (101); t221=t202-1;
   tp2lh=TIMESTEP*(Vmed-VOUT[t21])/(VOUT[t21]-VOUT[t201])+t21*TIMESTEP - 2.5;
   tp2hl=TIMESTEP*(Vmed-VOUT[t221])/(VOUT[t221]-VOUT[t202])+t221*TIMESTEP - 13.5;
   tp=(tp2hl+tp2lh)/2;
   i=0;
   time=0.2*(i-1);
End

Sweep
  i:  from 1 to 101 step 1 time VOUT[i] VIN[i]
  {PARAMETRIC  TITLE=""
               X_TITLE="time (ns)" Xmin=0 Xmax=20
               Y_TITLE="VOUT, VIN" Ymin=0 Ymax=5.5
               X=time
               Y=VOUT[i].red & VIN[i].green
               NXTICKS=5 NYTICKS=11};
End

MonteCarlo
  TITLE="Monte Carlo Sweep of the VOUT Output Waveform"
  N_outcomes=200 i: from 1 to 101 step 1 time tp<2.5 VOUT[i];
End
```

APPENDIX H

OSA90/hope INPUT FILE FOR THE RLC CIRCUIT EXAMPLE

```
#define DUMPON           1
#define DUMPOFF          0
Expression
  char cir_contents[]=
" ***********************
 * RLC serial circuit *
 ***********************
 VIN  1 0 PULSE(0 1 0 0 0 19S 20S)
 RIN  1 2 1
 R1   2 3 1
 C1   3 4 0.5
 L1   4 5 1
 R2   5 0 1
 ROUT 5 0 1
 .PRINT TRAN V(5)
 .TRAN 0.2S 10S 0S
 .OPTIONS CPTIME=6000
 .END/EXT
 .VAR L1
 .VAR C1
 .VAR R1
 .VAR R2
 .END
";
   L1=1   {Normal Sigma=5%};
   C1=?0.5?      {Normal Sigma=5%};
   R1=?0.5?      {Normal Sigma=5%};
   R2=?2? {Normal Sigma=5%};
   char cir_name[]="rlc.cir";
   char spp_out[]="rlc.out";
   char tr[]=".tr";
   Datapipe:     COM                       FILE = "create_file"
          TIMEOUT=0
          N_INPUT = 2             INPUT = (cir_name, cir_contents)
          N_OUTPUT = 1            OUTPUT = (char in_name[8]);
   input[1:4]=[L1 C1 R1 R2];
   DataPipe:     COM                       FILE = "Spicepipe"
          TIMEOUT=0
          N_INPUT = 8             INPUT = (in_name, DUMPOFF, spp_out, tr, input)
          N_OUTPUT=51             OUTPUT = (F[1:51]);
   t=1;
   time[1:51]=[0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
                   3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0 5.2 5.4 5.6 5.8 6.0
                   6.2 6.4 6.6 6.8 7.0 7.2 7.4 7.6 7.8 8.0 8.2 8.4 8.6 8.8 9.0
                   9.2 9.4 9.6 9.8 10.0];
   S[1:51]=3/20*exp(-time)+1/52*exp(-5*time)-1/65*exp(-2*time)*(3*sin(2*time)+11*cos(2*time));
   E[1:51]=S-F;
   Error=E[t];
End

Model
   RES 1 2 R=R1;
   CAP 2 3 C=C1;
   IND 3 4 L=L1;
   RES 4 0 R=R2;
   PORT 1 0 NAME=Vinput V=1 R=1;
   PORT 4 0 NAME=Voutput R=1;
   CIRCUIT;
End
```

```
Sweep
   Title="Function, Specification, Error"
    t: from 1 to 51 step 1 time[t] S[t] F[t] E[t];
   AC: Title="Gain"      freq: from 0.1 to 0.4 n=10 INSL;
End

Specification
   E< 0.01   E>-0.01;
   AC: freq: from 0.1 to 0.4 n=10 INSL<20 w=10;
End

MonteCarlo
   N_outcomes=200        t: from 1 to 51 step 1  Error<0.01 Error>-0.01 F[t];
                         AC: freq: from 0.1 to 0.4 n=10  INSL<20;
End
```