A Sophisticated Environment for Protocol Simulation and Testing

by
Günter Karjoth Peter Sjödin
Steffen Weckner

SICS R86004
Research Report
ISSN 0283-3638

A Sophisticated Environment for Protocol Simulation and Testing

Günter Karjoth
Peter Sjödin
Steffen Weckner

Swedish Institute of Computer Science
P O Box 1263
S-163 13 Spånga, Sweden

ABSTRACT

This paper describes a protocol development environment supporting the development and validation of communication protocols. In the environment, protocols are defined in the specification language BDL, which is based on the process algebra CCS. The behavior of a system is expressed in terms of communicating processes, and there is a set of verification techniques available for analysis of the specification. The BDL language and simulator are outlined in the paper. The interactive user interface, implemented on a graphical workstation, is presented. Further, the paper describes an interface between the operating system and the development environment. It is discussed how this interface can be used for rapid prototyping.

October 20, 1987

A Sophisticated Environment for Protocol Simulation and Testing

Günter Karjoth
Peter Sjödin
Steffen Weckner

Swedish Institute of Computer Science
P O Box 1263
S-163 13 Spånga, Sweden

1. Introduction

The importance of well-suited development environments for protocol design and analysis is widely recognized. A good development environment should support the designer in the design and analysis of the specification of a communication system, in the implementation of the specification, and in its testing. Such an environment should include the following components:

- a formal specification language
- verification techniques
- implementation strategies
- testing methodologies
- intelligent user interface

The specification language is the basis for any development environment. The designer should be able to define a system in a concise and unambiguous manner. It is important that the language supports system descriptions at different levels of abstraction, so that the designer is encouraged to develop the specification by stepwise refinement. Proposed specification languages range from declarative languages (e.g. logic based languages) to operational languages (e.g. state machine languages).

Once the designer has formulated the system in the specification language, the correctness of the specification should be verified. It has been shown that some verifications are possible to do automatically, Several papers on this subject have been published (IFIP6.1)

A proper user interface assists the designer in percepting the semantics of a specification. The constructs of the specification language should have a clear and concise graphical representation. This representation should also be flexible, in the sense that it should be possible for the designer to define how objects in the specification should be visualized on the screen. For example, a protocol specification is easier to understand if Protocol Data Units and Protocol Entities can be displayed as graphical objects.

It is important that a development environment allows for fast prototyping. This means that a specification can be made operational, so that the behavior of the system can be studied by an prototype implementation which is directly obtained from the specification. A prototype implementation is useful in the validation process, since it might be necessary to test that the specification conforms to the required behavior. This is since the designer might have formulated the properties to be verified incorrectly, no matter how carefully the actual verification has been carried out. Further, a development environment might be able to compute the set of possible, or allowed, sequences of actions that can be taken by the specified system. This set of action sequences can then be used to test other implementations of the system, by a prototype implementation.

In this paper the BDL (KAR85) development environment is described, focusing on support for rapid prototyping, tools for machine supported simulation and testing, and on the design of the user interface. The BDL specification language, based on the process algebra CCS (MIL80) and its interpreter are outlined in sections 2 and 3. The fundamentals of rapid prototyping are (1) the possibility to

execute a specification in real-time and (2) the possibility for communication between an executed specification and the operating system. These concepts are explained in section 4, together with a discussion about how the designer should go about to make a formal BDL specification operational. Section 5 describes the user interface, and discusses general problems concerning interaction between the designer and the development environment.

2. The BDL language

BDL originates from Milner's process algebra CCS (calculus of communicating systems). It provides mechanisms to

- represent the behavior by algebraic expressions,
- calculate the combined behavior of communicating processes,
- determine whether two behaviors are within the same equivalence class.

2.1. The process communication model in BDL

A system is specified in BDL as a set of communicating processes. A process can be observed only by its interactions. The interactions are communication events at *gates*, and each gate is identified by its unique name. Communication is synchronous, i.e. simultaneous cooperation of all processes connected to that gate is required. An interaction may involve a value transfer between the communicating processes.

Only one interaction can take place at a time. Processes can be combined in **parallel,** thus allowing the processes to communicate with each other. This concurrency is represented by an arbitrary interleaving of the communication interactions offered by the processes being run in parallel. The accessibility of gates may be restricted by hiding them for the outside world. Internal communication over a hidden gate may still be possible but is not observable. Internal steps are represented by the silent event *tau*.

A process P becomes a new process by making a transition μ. The execution of a process is thus of the form

$$\cdots \quad P_i \xrightarrow{-\mu_i} P_{i+1} \xrightarrow{-\mu_{i+1}} P_{i+2} \quad \cdots$$

where μ_i ,μ_{i+1} $\cdots$ denote either interactions or internal actions, and P_i , P_{i+1} $\cdots$ denote processes.

2.2. Data types

Data expressions in BDL are used for (1) specifying the values to be transferred in interactions between processes (2) binding values to the formal parameters in a process definition (3) in conditional expressions (see below). BDL is a strongly typed language, which means that each data expression must have a *value* of a well-defined *type*. Each data type has a corresponding *domain*, which defines the possible values of that type.

For example, the data type *boolean* (which in fact is the only predefined data type in BDL) has the data domain $\{true, false\}$.

2.3. Language operators

Formally, a system is described by means of behavior expressions. Expressions in the language represents processes. The semantics of a process is its behavior determined by the calculus. See the appendix for a full semantic definition.

2.3.1. Expressions, variables, types and declarations

Data expressions in BDL are specified in a functional language. (Since the BDL interpreter is implemented in LISP, this functional language is actually composed of all non-destructive LISP functions.)

Each data domain has to be declared by defining the name of the type that represents that data domain:

(bdl-type name testf)

This statement declares the data type "name", and the corresponding data domain is defined by the boolean data expression "testf", which should be a predicate that has the value "true" if and only if it is applied to a value of the type "name".

2.3.2. Actions

Let $T_1, \ldots, T_n$ be user defined types, $e_1, \ldots, e_n$ be data expressions, $x_1, \ldots, x_n$ be variables, and alpha be a gate name. Then there are three kinds of actions:

(write alpha $e_1\ T_1\ \cdots\ e_n\ T_n$)	output over gate alpha
(read alpha $x_1\ T_1\ \cdots\ x_n\ T_n$)	input over gate alpha
(tau)	internal action

If a process executes an input action the variables will be bound to the values received over the gate.

2.3.3. Behavior expressions

Let A be an action, B a behavior expression, C be a conditional expression (a data expression of type "boolean"), S a relabeling of gate names, and P be a process identifier. A relabeling S is given explicitly in the form

$(a_1\ b_1\ \cdots\ a_n\ b_n)$

such that each occurrence of a_i is replaced by b_i. A behavior expression may then be of any of the following forms:

(stop)	no further action
(seq A B)	first do A and then behave like B
(choice B1 B2)	behave like B1 or B2
(par B1 B2)	behave like B1 and B2 in parallel
(hide L B)	behave like B, but without external communication over the gates in L
(replace S B)	rename B according to S
(if C B1 B2)	if C has the value "true" then behave like B1, else behave like B2
(P $e_1\ \cdots\ e_n$)	behave like the body of the definition of P, where the values of the expression are bound to the corresponding formal parameters of the behavior definition of P.

A behavior definition is of the form

(beh-id P ($x_1\ T_1\ \cdots\ x_n\ T_n$) B)

where B represents the body, or the behavior, of P, and $x_i\ T_i$ declares the i:th formal parameter of the definition.

It is required that behavior expressions are guarded, i.e. each occurrence of a behavior identifier is preceded by an action.

2.3.4. Example

The following example shows a specification of a timer in BDL:

(bdl-type Integer 'fixp)

This declares a data type called *Integer*. The LISP function *fixp* returns the value true if (and only if) it is called with a parameter that is a (LISP) integer, and can thus be used for defining the *Integer* data domain.

```
(beh-id Clock NIL
  (seq (write tick) (Clock)))
```

Clock is a recursively defined parameterless process. It repeatedly outputs a dataless synchronization signal at the gate "tick".

```
(beh-id Timer NIL
  (choice (seq (read tick) (Timer))
          (seq (read start x Integer) (T x))))
```

The process Timer either interacts over the gate "tick" or the gate "start". If it receives a timeout value at the gate "start", it will behave like process (T x).

```
(beh-id T (x Integer)
  (choice (if (GREATERP x 0)
              (choice (seq (read stop) (Timer))
                      (seq (read tick) (T (SUB1 x)))))
          (if (ZEROP x)
              (seq (write alarm) (Timer)))))
```

T will periodically receive a tick signal causing it to decrease the time value. If a stop signal arrives in time, eventually the timer expires and signals that situation by a communication offer via the gate "alarm".

We compose Clock and Timer in parallel to let them interact with each other. They make matching offers on the gate "tick" and therefore will communicate eventually. Notice that, in contrast to CCS, this communication is still visible to the outside world. This means that another Timer process could be connected to the gate "tick" and will receive the tick signals synchronously with the first process.

3. The BDL Interpreter

The interpreter can be considered as a state generator function, which in each evaluation step computes the set of possible actions for the current state. One of them is chosen to be performed next, and this action determines the next state.

The interpreter can be instructed to record all information necessary to create a finite state model of the course of exploration. This finite state model consists of the reachability graph and a property list for each state. The model description can be used later in the extended model checker (CES83, QUSI82) to verify various properties formulated in temporal logic.

3.1. Next action selection

There are several ways to chose the action to be executed next. In principle, the interpreter can be used in three modes of operation where the the next action is chosen either:

1. by the user
2. by the interpreter itself
3. by reading a log file

Mode 1: Design validation

In this mode the user has full control of the simulation, by selecting the next action from a list of all possible actions presented to the user.

It is possible to switch the simulation back to an earlier state with at least one unexplored action. In this way, all states of a terminating behavior can be visited, going depth-first.

The "old state" recognition facility can be enabled to obtain the state graph of an specification containing loops.

Mode 2: Test Sequence Generation

The task of choice resolution can be assigned gradually to the interpreter. If there is only one possible action the interpreter may take it immediately. Moreover, the user can designate events, to allow them to be automatically chosen by the interpreter. Correspondingly, some events can be excluded from execution. The interpreter can also be instructed to resolve nondeterminism completely by itself, for a given number of steps.

All these facilities together allow for the building of the complete execution tree of finite program descriptions. Hence, the interpreter can be used as an automatic test sequence generator.

Mode 3: Trace analysis

The interpreter may also be used as an (off line) analyzer for traces of a protocol implementation under test (IUT). Connected to the trace file, the interpreter proceeds simulation as long as the current action read from the log file does not cause a deadlock situation, i.e. is a member of the acceptance set of the current state.

It may however be the case that the specification allows some internal actions to occur before the next action actually observed during the execution of the IUT takes place. Therefore the interpreter will explore all branches leading to stable states, i.e. states where no internal action is possible. If there is a state where the current logged action is enabled, the interpreter continues trace checking.

4. The Test Environment Interface

With some extensions to the execution model, the interpreter can be used for real time execution of the specified system. This has mainly two advantages:

- The implementor does not have to rewrite the system in another programming language. Rewriting a system is time consuming and error-prone. It is also difficult to ensure that a rewritten system conforms to the original specification.
- The BDL interpreter has powerful debugging tools. The support for trace recording, and to execute according to a recorded trace, makes the system useful for real time testing of protocol applications.

The main strategy for executing specifications is to extend interactions to involve value transfers to and from the operating system. Processes in the specified system can then exchange information with the world outside the interpreter. From the specification point of view, these information exchanges take place at gates (just as information exchanges between processes during simulation).

4.1. Driver gates and driver processes

The concept of gates in BDL is extended in order to accomplish real time execution of a specification: A gate can have a *driver* property, which allows the gate to be used for communication with entities outside the interpreter (For readability, gates with driver properties are referred to as driver gates). The processes inside the interpreter domain, implemented by BDL statements interpreted by the BDL interpreter, are called *BDL processes*. The entities outside the interpreter domain are referred to as *driver processes*, and can be implemented in practically any manner (even by another instance of the BDL interpreter!).

In other words, driver gates can be regarded as interfaces between BDL processes and driver processes. Driver gates can, for example, be interfaces to management functions (e.g. real time timers), device drivers (Ethernet interfaces) and other programs. An example of a possible configuration, where driver gates are used for communication with an application program and the file system is depicted in fig. 4.1 (The application program and the file system represents driver processes in this picture).

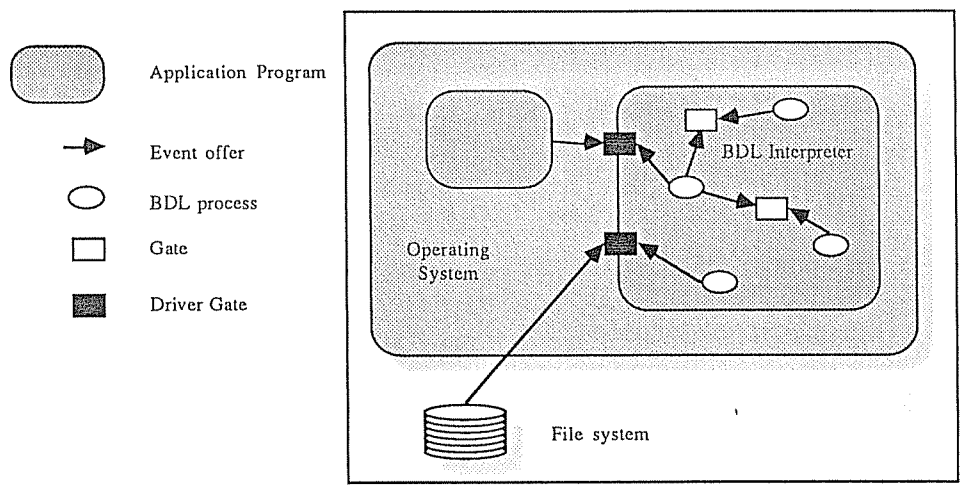

Figure 4.1: Communication between BDL processes and driver processes

Figure 4.2 shows how driver gates can be used to obtain a prototype implementation of a protocol layer from a BDL specification.

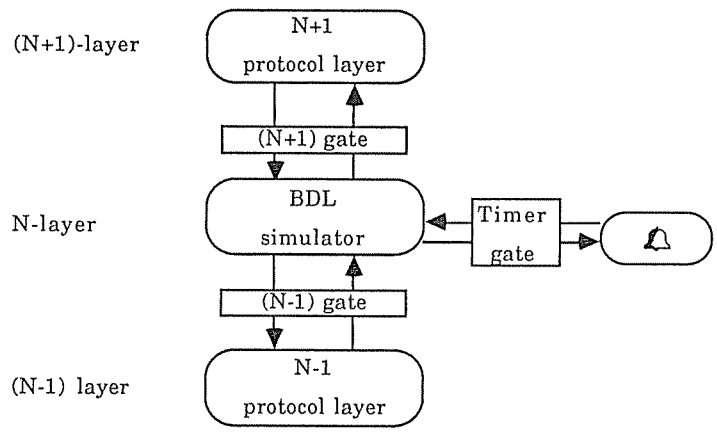

Figure 4.2: Protocol prototype implementation in BDL, using three driver gates

In this example, the N-protocol-layer is implemented by a BDL interpreter executing a BDL specification of the N-layer. The N-protocol-layer-entity communicates with the surrounding layers through the (N+1) and (N-1) driver gates. The timer gate interfaces the N-protocol-layer-entity and real time timers.

In the current version of BDL, driver gates are used for communication with the TCP/IP network and with a real-time timer.

4.2. The use of driver gates in a specification

All gates are treated in the same way in a system specification, irrespective of their driver properties. An exchange of data at a driver gate is specified in exactly the same way as a data exchange at an ordinary gate. Hence, there is no way for a BDL process to determine if it is interacting with a driver process, a BDL process, or a mix of both.

The interface to the driver gate must be designed according to the interaction scheme in BDL. That is, the data that is exchanged at the driver gate has to be structured as a vector of values of certain types.

All interactions are controlled by the interpreter. The interpreter chooses the next action to occur and performs the interaction. However, the BDL interpreter implements only one party when a BDL process and a driver process are interacting. This means that the interpreter can not decide on its own whether it is possible to interact at a driver gates, since the stimuli at driver gates are generated from a source whose behavior is unknown to the interpreter. A situation may occur (not unlikely) where the set of actions, from which the next event is selected, do not contain all actions that are possible according to the specified behavior of the system. This happens when the specified behavior of a driver process allows it to interact, but the driver process has not at that particular moment reached the state where it performs the interaction. This is not a major problem; a driver process that is able to interact will sooner or later reach the point where it actually offers to do the interaction.

4.3. From simulation to real time execution

The interpreter can not make any conclusions about the behavior of a driver process, since the behavior of a driver process is beyond the scope of the BDL interpreter. A driver process, that seems to always offer interaction at a certain state, might behave as

```
(seq (write a x X) (P))
```

as well as

```
(choice (seq (write a x X) (P))
        (seq (tau) (stop)))
```

However, the interpreter can be used to simplify the conversion of a specification into a real-time implementation. A strategy that should be applicable for most situations is described below:

First, the behavior of the complete system, including all driver processes, is specified in BDL. The BDL interpreter is run to ensure that the behavior of the system is correct. When the specification is found to be correct, an event trace is recorded.

The real-time implementation can then be derived from the simulated system. The behavior descriptions of the driver processes are removed, one by one, and replaced by the corresponding driver processes. The resulting system should accept the recorded event trace as a legal sequence of events. If it does not accept the trace, one or more of the driver processes do not behave according to the specification. However, an accepted trace does not guarantee that the system is correct. The trace should be carefully selected to explore significant parts of the execution tree, but will still only give an indication of whether the system behaves correctly or not.

With this strategy, the process to convert a specification of a system to a real time implementation is significantly simplified. The major parts of the system are kept intact, and only a few carefully selected parts have to be rewritten.

4.4. Handling of driver gates in the interpreter

The interpreter uses the same algorithm for calculating the set of possible actions as described in section 3. In the normal case, when no driver gates are involved, the interpreter determines if two (or more) processes are offering matching actions at the same gate. If that is the case, an interaction at that gate is considered a possible action.

Interaction at driver gates require some special treatment. The interpreter must have some way to determine if the driver process has offered an action, since the interpreter only represents a BDL process at a driver gate. When the interpreter has determined that one BDL process can interact at a driver gate, a gate-specific predicate function is called. The value returned by the function indicates whether the driver process is ready to interact at that gate. If the interpreter is running in interactive mode, the user can force the interpreter to wait for interaction at a driver gate without first checking if the driver process really is willing to interact.

4.5. Implementation of driver gates

The interface to a driver gate is composed of a set of gate-specific functions, provided by the designer. These functions are called by the interpreter in different phases of the execution:

- A polling function is called when the interpreter is computing the set of possible actions. This function inquires whether the driver process is offering an action at the driver gate. Of course, it is impossible to interact at a driver gate unless a BDL process is offering interaction at that gate. So the polling function is called only when a BDL process has offered to interact at the driver gate.
- An exchange function is used to carry out the actual transmission of data between the driver process and the BDL process. This function transfers a data value and a data type over the driver gate.
- An initiation function is called when the interpreter is started. This function carries out whatever has to be done to create and initiate that particular gate.

By default, a gate is not a driver gate. Each driver gate has to be explicitly declared to the interpreter as being a driver gate. This is the only visible difference between driver gates and non-driver gates in a BDL specification.

5. An Advanced BDL User Interface.

Since traces of BDL programs tend to be very large (if at all finite), a graphics interface has been developed to simplify comprehension and analysis. An interactive tool has been used (BNW86) to import BDL simulation into a interactive graphics system. This is done in a distributed manner: BDL runs on a UNIX machine while all user interaction takes place in a INTERLISP-D environment on a working station as shown in figure 5.1.

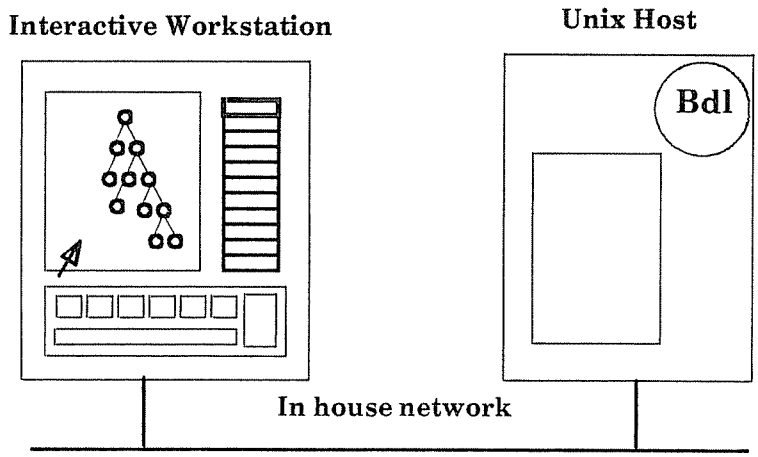

Figure 5.1. The configuration of the BDL interfacing system. The graphics system of the XEROX is used to enhance the BDL interpreters interactive behavior. The communication between the user and the interpreter is realized via the local Ethernet network.

This approach is appealing since it divides the computational work and makes benefit of the different capabilities of the two systems. The workstation provides for a more understandable presentation of a simulation. This *enhanced user interface* also adds functionality to the system.

5.1. Implementation Strategy.

There are two main objectives of an user interface: (1) guide the user through the control states of the program (2) keep the user informed of any results produced by the program. Hence, the user interface describes both what is going on during the simulation, and what commands are available at each moment. The enhanced interface offers an increased level of service in that some interaction is automated.

The graphics system is used to produce a concise display of the current state of the simulation. The interface program continuously collects information from the simulator, and the user may at any time interact with the display to obtain this information. The available commands are presented in such a way that intermediate commands does not interfere with parts of the display that are more stable. This is an advantage compared to conventional character oriented systems, where new output scrolls previous information off the screen. At all points in a simulation the user can use the help facilities, even in the middle of a command to the BDL interpreter. This is natural in a graphics system as it interacts concurrently.

The semantics of a simulation is a trace tree which naturally displays graphicly. The tree is stored in the interface program, and continuously displayed and updated on the screen. This is the most appealing advantage in porting the user interface to a graphics system.

To ensure that a user is not flooded, the interface abstracts away superfluous information by organizing the information in levels. The user can examine the displayed items interactively and thereby stepping down through the abstraction levels.

5.2. The distribution.

The communication between the interpreter and the user interface is layered as shown in figure 5.2.

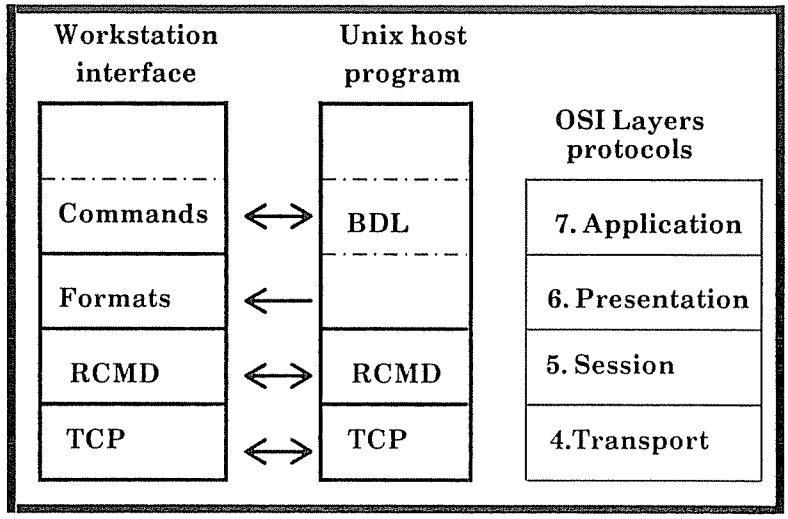

Figure 5.2. Architecture of the remote BDL interface. Interaction is redirected from TTY to the workstation via the RCMD channel to the workstation. Interpretation of application data is regarded as the responsibility of a presentation entity. There is no presentation peer on the UNIX side because the BDL program is viewed from a black box perspective.

The channel is realized by the UNIX "session" protocol RCMD, using a TCP/IP-channel, and an implementation of a corresponding client on the INTERLISP side. The presentation layer contains the

interpretations of BDL commands and information output.

As the presentation entity interprets the output from the interpreter, BDL objects (such as processes, states, behavior and events) are recognized by the interface. These objects are used to build the structures necessary to analyze the history of a simulation. These structures can be saved in memory to be used for automated analysis and for generation of test sequences after the simulation.

5.3. Program Control

The BDL commands, in contrast to the output information, go bidirectionally. The presentation entity identifies program transitions and displays the set of commands that are valid at a given state of the simulation.

All interactions between the user and the interface program are regarded as synchronous. At each point in run time, either the user or the interpreter may introduce interaction primitives. This means, that the interpreter is either computing or waiting for a user interaction following a program prompt. With this view, the interaction between the interpreter and the user can be modeled as a state machine with a main state for each prompt and transitions for all available commands. The interface program is implemented by a state machine depicted in figure 5.3.

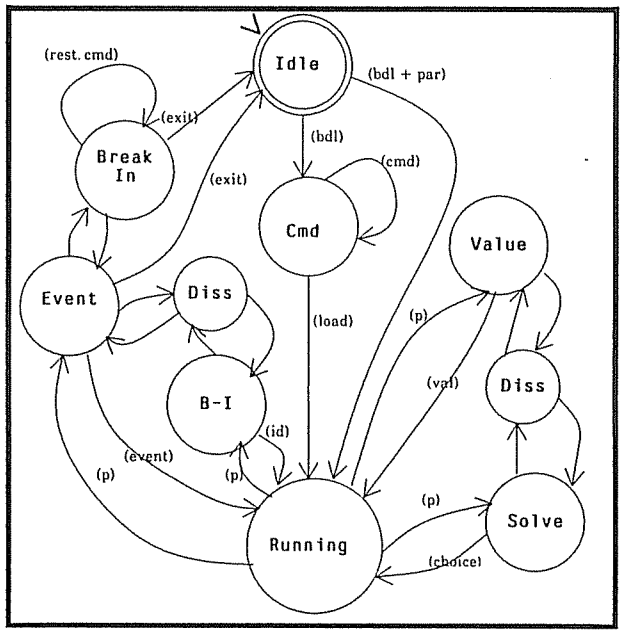

Figure 5.3. State diagram showing the user interface view of the behavior of a BDL simulation. All states except the state *Running* are waiting states where BDL waits for user input. All transitions from state *Running* are initiated by BDL output and all to the *Running* state initiate from user interaction. This state diagram catches all possible interaction transitions although it does not specify the behavior of BDL as it includes impossible sequences of interactions. The unlabeled transitions are, local to the workstation, used to dismiss and resume simulation for local analyze of the current simulation history. Some details of this diagram are explained below but the importance of this picture is to show the bushiness of interaction. It is normally the user manual's responsibility to guide the user through the interaction state space. One intention of the user interface is to simplify the users understanding of what happens, i.e. which state in the state diagram is current.

The state *Idle* is the initial state. The top level command can be issued to start the program. If sufficient parameters for starting a simulation are issued a simulation starts, if not, the *Cmd* state is entered where the user issues commands altering parameters of the simulation strategies. After a specification is loaded, the *Running* state is entered. This means that a simulation is in progress and that the interpreter is either producing information or computing new state spaces. The simulation continues

until the interpreter needs user interaction in order to continue, and then prompts for a *Value*, a behavior identifier *(B-I)*, the next *Event*, or prompts the user to *Solve* a non-deterministic choice between internal state transitions. The ordering of these prompts is part of the BDL simulation algorithm, and the user interface is triggered by primitives and does not reimplement the BDL state generator. Each prompt identifies an unique state transition. The user may choose to answer the prompt or to dismiss *(Diss)* the simulation for the time being, inspecting the simulation tree or return to the Interlisp environment.

5.4. The prototype implementation

The goal of the user interface is *not* to lead the user through the state diagram of figure 5.3, although a session is a traversal of the diagram. The obligation of the user interface is more one of explaining what is possible to do at each moment, and of describing the context of the current state. Two command menus are used to give this information. From the first, "BDL Head", the program can be started and parameters can be set to start the simulation and to initiate the RCMD channel.

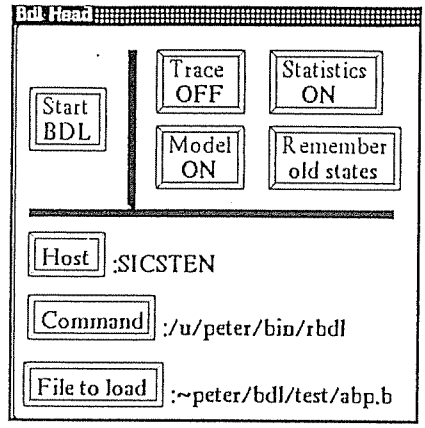

Figure 5.4. The main menu of the BDL interface. A menu with one command button and seven buttons displaying values. The values of the buttons on the low half of the menu are parameters used for the RCMD channel. The values in the top right quadrant correspond to the parameters in the top level command "bdl" in the UNIX environment. The interface takes care to passivate buttons that are not relevant in the current program state.

The menu in figure 5.4 consists of one command button, "Start BDL". Selecting this buttons issues the command which starts the simulation. The other buttons in the menu are all parameters for the start command. There are four two-state value buttons and three string value button. The two-state buttons are used to switch flags, on and off, and are read by the start command. By selecting these buttons the flag is switched which also is displayed by the face of the button. If a string button is selected, a special buffer prompts for the new string which is displayed and used as a parameter by the start command.

The second menu called "BDL Continue" is used to drive the simulation session. This menu describes the same state space as in 5.4 but from a different perspective. The menu describes what can be done in the simulation instead of describing the current program state. The menu is divided into two different parts. The top half of the menu displays the current state of the *behavior specification* and the status of the interpreter. The lower half of the menu displays the different sets of available input commands. If a command set is available, the button is not shaded and a selection of this button will display a new menu from which a specific command can be chosen.

The "Status" button of the menu indicates that the interpreter is waiting for a stimuli. The command "continue" is used to resume a dismissed simulation. When pressing the "Continue" button, a new menu will pop up displaying the set of available events from the current state. The user can choose to press "BDL Cmd" to enter a command mode where parameters directing the BDL simulation can be

altered. As the command button Backup in the figure above shows, the commands that are not available at a certain time are shaded. If the "Browser" is on, the user can follow how the simulation is carried out by a tree display of the simulation trace. Figure 5.6 shows the entire screen of a workstation during a simulation run.

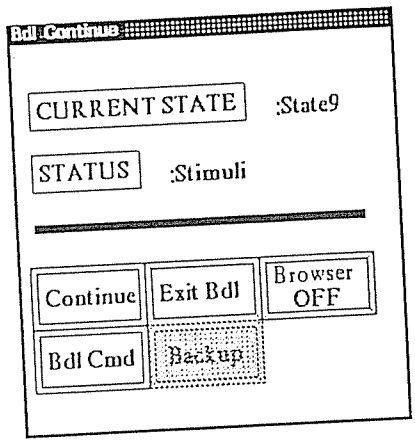

Figure 5.5. The simulation menu displays state and status. The user running this interface is not really concerned with the states of the program, instead he/she is interested in states of the behavior specification under simulation. This menu refers to the ninth state encountered during this simulation. The menu also contains five command buttons. Buttons that are passivated (i.e. illegal commands) are shaded. In this example it is not possible to backup to a previous state.

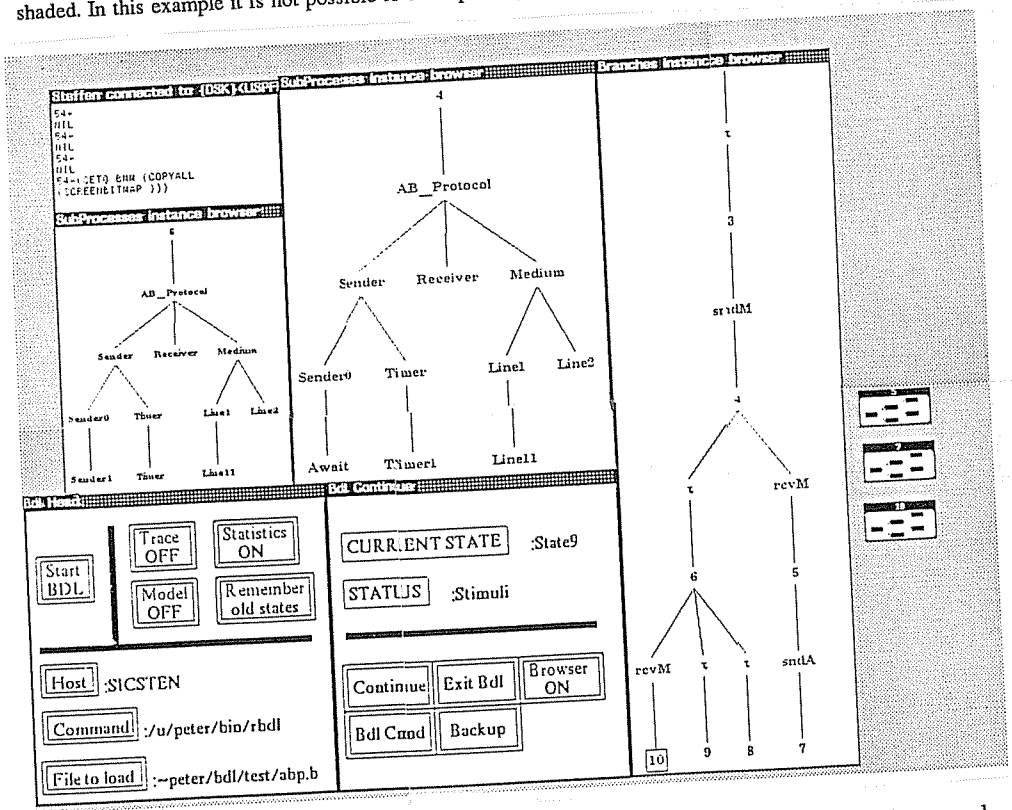

Figure 5.6. An overview of a BDL session, simulating the behavior of the alternating bit protocol. Down to the left the two menus are displayed. The tall browser on the right side is a tree representation of the current simulation trace. On top of the menus, tree representations of the process structure of two different states are displayed. The three symbols to the far right are shrunk browsers.

At each prompt from the interpreter, a menu will be displayed. The user can issue the requested value or dismiss the prompt menu. If dismissed, "BDL Continue" will show the current state of the simulation. In the snap situation of figure 5.7 the user has been prompted for an event from state 9 (i.e. the status value is Stimuli, current state is 9). All simulation commands are active and available. The user can now investigate information collected during the simulation. All the different nodes in the browsers are *interactive* (mouse sensitive and prompts with menus of the commands that can be applied to them) so the user just buttons the states, events and processes he/she is interested in and will then be presented with the chosen information.

The browsers of figure 5.7 are structured displays representing the information collected during the current session. The root of this information tree is the top symbol of the simulation trace browser, to the right in the figure. Each state encountered during the simulation is displayed by its number and the actions are displayed by their labels. Internal events are displayed by τ, and external events by the name of the gate.

More information of states and events can be investigated by selecting any symbol representing the object in the browsers, for example names of synchronizing processes, eventual value transfers, behavior expressions and so on.

Figure 5.7 displays a view into a small partition of a simulation trace browser during a simulation of the alternating bit protocol. In the figure, state 4 is selected causing the command menu, covering the state, to be displayed. The menu displays the different functions that can be applied to the state. Chosing *Inspect* means that all the collected data of this state is shown. By the *Edit* item the user can alter the definition of the state, each slot of the object is displayed in a type sensitive editor. By using the *Inspect* or *Edit* command, the user can investigate and analyze the collected information in a recursive manner.

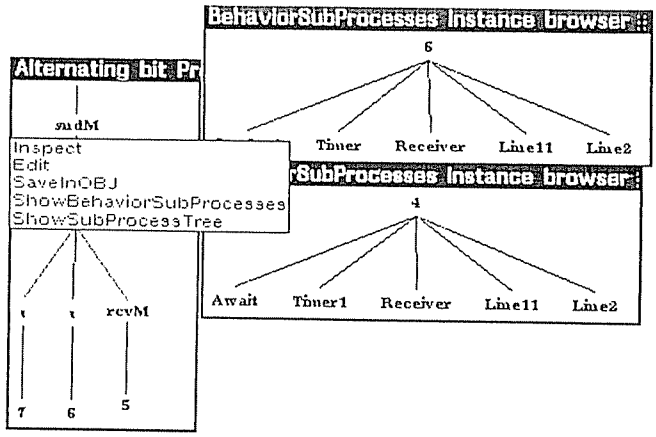

Figure 5.7. A snap shot of four views during a simulation session. A trace of a simulation is shown to the left. The menu which partly covers the trace browser displays the different interactive commands available from a state symbol. To the right, two browsers display the composing subprocesses of state four and six of this session. All symbols are active and will display further information when chosen in an interactive fashion.

The command *ShowBehaviorSubProcesses* is also used to analyze the simulation trace, by displaying a tree of process instances. The investigation can be carried further by selecting a subprocess in the tree, as all symbols in the tree browser are interactive.

6. Conclusion

The prototype implementation has demonstrated some of the advantages of a sophisticated protocol development environment, and the use of the advanced interface has strengthened our opinion that graphics is an essential part of the environment. There are, however, several ways in which a

simulation can be pictured on a screen. In this prototype, we describe a simulation by a tree representation of the traces. There are many other alternatives, e.g. a picture with interconnected boxes, and perhaps tokens jumping around, would, at least for some users, be a more conceptual picture of communicating systems.

The advanced graphical interface has been implemented without modifications of the original BDL interpreter. This demonstrates that it is possible to add an advanced interface to programs with conventional character-oriented interfaces.

The work done on the user interface in this project concentrates on the interaction with thee interpreter. A fully grown development environment should supply a graphical representation of a specification as well, but that requires a graphical syntax for the BDL language to be developed.

The concept of driver gates makes it possible to construct prototype implementations out of formal protocol specifications in a convenient manner, where only the parts of the specification that are directly related to communication with the operating system need to be changed. However, the performance of the current BDL system makes it impossible to test "real" protocol implementations, since BDL can not fast enough respond to stimuli from the environment.

Appendix A. The semantics of BDL behaviour expressions

We assume an infinite set of symbols, called names, and be distinguished as variables and operators. We use the convention that $E\{f_1,..,f_n/x_1,..,x_n\}$ denotes the result of substituting expressions f_i for variables x_i simultaneously in expression E. Variables $x \in X$ occur free in E, and the sort of the replacing expression is the same as the sort of the variable. The function L: B $\rightarrow 2^{Lab}$ gives the set of labels of the behaviour expression B.

Let g be a label, and $\bar{v}$ be a vector of data values. An experiment or stimulus μ=g$\bar{v}$ may cause a process to progress. This fact is defined by the relation R, indexed by Act = (Lab×$\bar{v}$) $\cup$ {*S}, over P;

R[p,μ] = p'

means that p may react upon stimulus μ and becomes p' in doing so. We axiomatize the various relations over each syntactic category and define thereby a family of simulation rules.

Sequencing

R [(*seq* (*write* g d_1 t_1 ... d_n t_n) B), g$\bar{v}$] = B where (*eval* d_i) is $v_i \in t_i$

R [(*seq* (*read* g x_1 t_1 ... x_n t_n) B), g$\bar{v}$] = B{v_i/x_i} if $v_i \in t_i$

R [(*seq* (*tau*) B), *S] = B

Summation

R [(*choice* B_1 ...B_n), μ] = B_i' iff there is a B_i = (a_i B_i') and R [B_i,μ]=B_i'

Parallel composition

$$R\,[(par\ B_1 \ldots B_n),\mu] = \begin{cases} (par\ B_1'\ B_2) & \text{iff } R[B_1,\mu]=B_1' \,\&\, g \notin L(B_2) \\ (par\ B_1\ B_2') & \text{iff } R[B_2,\mu]=B_2' \,\&\, g \notin L(B_1) \\ (par\ B_1'\ B_2') & \text{iff } R[B_1,\mu]=B_1' \,\&\, R[B_2,\mu]=B_2' \,\&\, g \in (L(B_1) \cap L(B_2)) \end{cases}$$

Restriction

R[(*hide* A B), μ] = (*hide* A B') if g$\notin$A & R[B,μ]=B'

R[(*hide* A B), *S] = (*hide* A B') if g$\in$A & R[B,μ]=B'

Relabeling

R [(*replace* (a b) B), μ] = R [B {a/b}, μ]

Conditional choice

R [(*if* d B), μ] = R [B , μ] iff not (*eval* d) equal *nil*

Behaviour definition

R [(b $d_1 \cdots d_n$), μ] = R [B_b {d_1,..,d_n/x_1, . . . ,x_n}, μ]

where B_b is the behaviour expression bound to b.

In contrast to the binary communication found in CCS, the parallel composition operator defined here allows more than two processes to synchronize on the same label.

References

BNW86.

B. Backlund, M. Nordström, and S. Weckner, "Dialogie (overview) - An interactive tool for creating graphic Man/Machine interfaces," T87005, Swedish Institute of Computer Science, Stockholm, Oct. 1986.

CES83.

E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Pratical Approach," in *10th Annual ACM Symposium on Principles of Programming Languages*, Austin, Tx, Jan. 24-26, 1983.

IFIP6.1.

IFIP WG 6.1, *International Conference on Protocol Specification, Testing and Verification*, North-Holland, 1981-1987.

KAR85.

G. Karjoth, "An Interactive System for the Analysis of Communicating Processes," in *5th IFIP Workshop on Protocol Specification, Testing, and Verification*, ed. M. Diaz, pp. 91-100, Moissac, France, June 10-13, 1985.

MIL80.

R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

QUSI82.

J.P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR," in *International Symposium on Programming*, Springer Verlag, LNiCS #137, 1982.