PREV
C. Concurrent Programmi

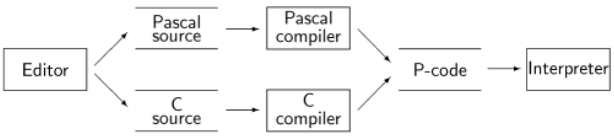
NEXT
E. Further Reading

Appendix D. Software Tools

In this appendix we give a brief overview of software tools that can facilitate studying concurrent and distributed programming. For further information, see the documentation associated with each tool.

BACI and jBACI

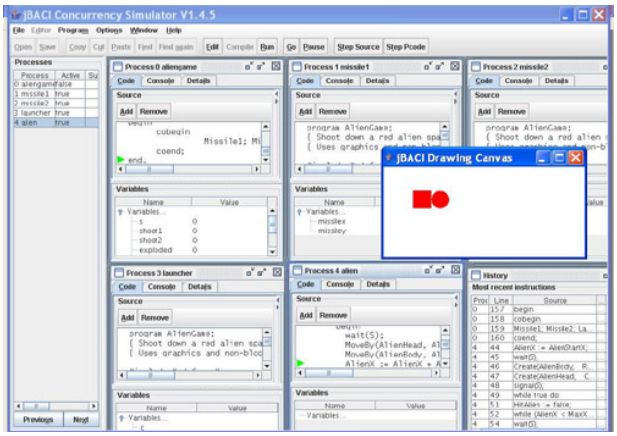
BACI, the *Ben-Ari Concurrency Interpreter*, was developed by Tracy Camp and Bill Bynum [20]. The following diagram shows the architecture of BACI.



The compilers compile the source program into P-Code, an intermediate code for a virtual stack machine. The P-Code is then interpreted by the interpreter. The compilers for Pascal and C and the interpreter are written in C; the compilers are built using the generators `lex` and `yacc` (`flex` and `bison` can also be used). The software has been carefully designed so as to minimize system-dependent aspects, and prebuilt versions for many systems are available on the BACI website. The compilers and the interpreter are run as command-line tools; you can interactively run a concurrent program step-by-step or set breakpoints to examine the values of the variables at any step. There is also a graphical user interface, but this is available for UNIX systems only.

David Strite has developed a new interpreter for P-Code and a GUI

interface, are configurable at either compile time or upon initialization.



After you open a source file, it can be edited in a large panel that appears below the toolbar. (This mode is not shown in the above screenshot.) Then you can select `Compile` and the appropriate BACI compiler is invoked as a subprocess. When the program has been compiled successfully, select `Run` to invoke the interpreter. The program can be interpreted by selecting `Go`. `Pause` will halt the execution, which can be restarted by `Go` or restarted from the beginning by selecting `Run`.

The atomic statements of BACI are the individual P-Code instructions, but in Strite's interpreter and in jBACI you can interleave source code statements as if they were atomic statements. `Step Source` considers a source code*line* as an atomic statement, while `Step PCode` considers a P-Code instruction to be an atomic statement.

> **NOTE**
>
> To run the programs in this textbook in the intended manner, make sure that each assignment statement and each condition in a control statement are written on a single line.

The process table on the left side of the screen enables you to select the process from which to execute the next statement.

There is a process window for each process, displaying the source code and the P-Code for the process, as well as console output from the process and the values of its variables. Breakpoints can be set by clicking on a source code or P-Code line in a process window and then selecting the `Add` button above the display of the code. A red dot will appear in front of the line. To remove a breakpoint, click on the line and select `Remove`.

Other optional windows can be displayed which show the global variables, the Linda tuple space, a history of the last statements executed, and a console containing the output from all processes.

As part of the jBACI package, I have modified the compilers and interpreters to support a simplified version of the Linda model of synchronization. In addition, some graphics commands from the Java Swing library are made available at the source code level, so

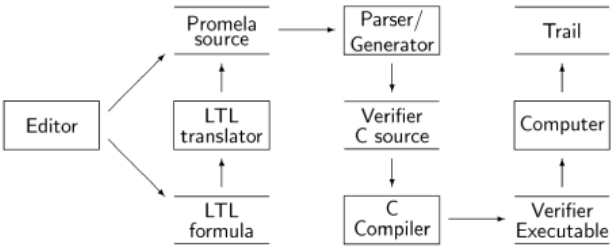
that you can study synchronization issues within the framework of simple game-like displays.

Spin and jSpin

The Spin Model Checker was developed by Gerard J. Holzmann [33]. It is written in C and the source code is available together with prebuilt executable files for many computers and systems. Spin has a command-line interface using options to configure the software. Two GUIs exist: XSpin written by Holzmann in Tcl/Tk, and jSpin written by this author in Java.

Spin can be used as a concurrency simulator. In *random simulation* mode the interleaving as well as the resolution of nondeterminism in `if` and `do` statements is performed randomly, while in *interactive simulation* mode the user is prompted to guide the selection of the next statement to execute.

Verification is performed as shown in the following diagram:



First, Spin is executed to analyze the Promela source code and to generate source code in C for a verifier that is optimized for the specific program and the options that you have chosen. The Promela source code may be augmented with a **never** claim produced by translating a formula in linear temporal logic; Spin includes such a translator, although other translators can be used.

When the verifier program has been generated, it is compiled with a C compiler and linked as usual to produce an executable verifier. Running the verifier will result in a notice that no errors were found or an indication of the error. In the latter case, the verifier writes a coded file called the *trail*, which can be used to guide Spin to reproduce the computation that led to the error.

There are three modes of verification in Spin (safety, acceptance and non-progress), and one of these must be selected. Safety mode is used to verify safety properties like freedom from deadlock, assertions and invariants, such as those expressed by □-formulas. Acceptance mode is used to verify liveness properties such as freedom from starvation which can be expressed with ◇ -formulas. Non-progress mode is an alternate method of verifying liveness properties without explicitly writing LTL formulas.

You can select if (weak) fairness is to be assumed during verification.

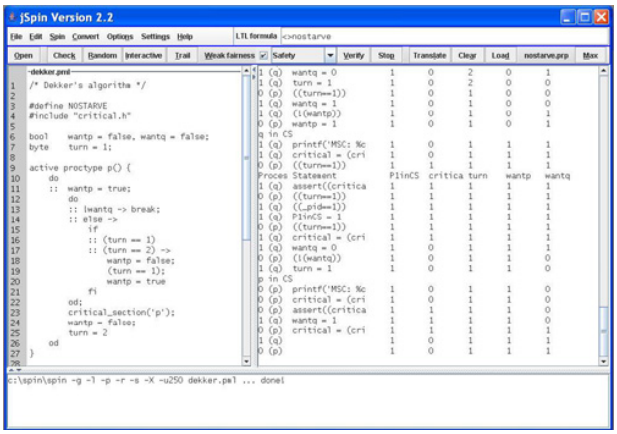
The jSpin User Interface

The user interface of jSpin consists of a single window with menus, a toolbar and three adjustable text areas. The left area is an editor for Promela source code; the right area is used for the output of the Spin simulation (random, guided or obtained from a trail file); the

bottom area displays messages from Spin. Spin option strings are automatically supplied.

When running a random simulation, the output appears in the right text area; the degree of detail can be set through menu selections. For guided simulation, the alternatives for the next statement to execute are displayed in the lower text area and a popup window enables selection of one of the alternatives. The Spin output is filtered so as not to overwhelm the user; the filtering algorithm works on output lines of Java type `String` and can be easily changed by modifying the class `Filter` of jSpin. A special feature enables the user to interactively select which variables will be displayed.

The menu bar contains a text field for entering LTL formulas. Invoking `Translate` negates the formula and translates it to a **never** claim. When `Verify` is selected, the claim together with the Promela program is passed to Spin.



jSpin is written in Java for portability and is distributed under the GNU General Public License. All aspects of jSpin are configurable: some at compile time, some at initialization through a configuration file and some at runtime.

How Spin Verifies Properties

The description of verification by model checking in Section 4.6 is over-simplified, because Spin does not check a property $\Box F$ by constructing all states and checking F on each state. Instead, states are constructed incrementally and the property is checked as each state is constructed. If $\neg F$ is true in a state, then the formula $\Box F$ is falsified and there is no reason to construct the rest of the states.

A correctness property expressed as a formula in temporal logic is negated and added to the Promela program in a form that enables it to be executed in parallel with the program. So, to prove $\Box F$, we add $\neg\Box F$ to the program. This formula can be written as $\Diamond\neg F$, which can be considered as a "program":

```
loop forever
   if ¬F is true in this state then break
```

One step of this "program" is run after every step of the Promela program; if the condition is ever true, the "program" terminates and Spin has found a counterexample to the correctness claim $\Box F$.

It may help to understand this process if we repeat the explanation on a specific example, Dekker's algorithm described in Section 4.7. Consider the correctness claim given by the formula$F = criticalp + criticalq \leq 1$.$\Box F$ means that something good (at most one process in its critical section) is always true; that is, true in every state. What does $\neg\Box F$ mean? It means that it is not true that in every state$F$ is true, so, eventually, there must be a state in which$F$ is false, that is, a state in which$criticalp + criticalq > 1$ is true, meaning that more than one process is in its critical section, violating mutual exclusion. The corresponding "program" is:

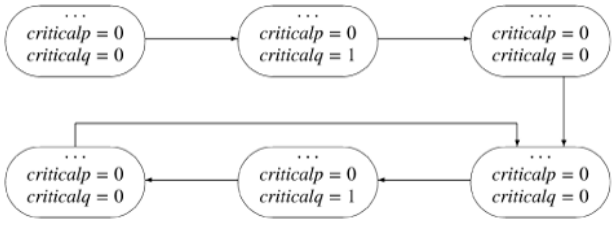
```
loop forever
   if criticalp +criticalq > 1 in this state then break
```

If Spin successfully finds a state in which$criticalp + criticalq > 1$ is true, then we have failed to prove $\Box(criticalp + criticalq \leq 1)$; conversely, if Spin fails to find a state in which$criticalp + criticalq > 1$ is true, then we have successfully proved $\Box(criticalp + criticalq \leq 1)$.

For liveness properties, the situation is somewhat more complex. Consider the correctness claim of freedom from starvation $\Diamond(criticalp > 0)$. Negating this gives $\Box(criticalp \leq 0)$. Now it is not necessary for$criticalp \leq 0$ to be true in*all* states for a counterexample to exist. All that is necessary is that there exist *some* (potentially infinite) computation in which$criticalp \leq 0$ is true in all states of that computation. A potentially infinite computation in a finite program is one that contains a cycle of states that can be repeated indefinitely. So Spin looks for*acceptance cycles*:

```
loop forever
   if criticalp > 0 then abandon this computation
   else if this state has been encountered then break
```

A computation in which starvation does not occur is uninteresting as we are trying to see if there exists*some* computation that can starve process `p`. However, we find a counterexample if a state is reached that has been previously encountered and $criticalp \leq 0$ has remained true throughout. This is demonstrated in the following diagram; the values of the variables `criticalp` and `criticalq` are explicitly given, while the ellipses represent all other elements of the state.



This computation goes through three states followed by a cycle representing a potentially infinite repetition of three other states. While the critical section of process `q` is repeatedly entered, the critical section of process `p` is never entered. Such a cycle is called an acceptance cycle and proves that there exists a computation that starves process `p`.

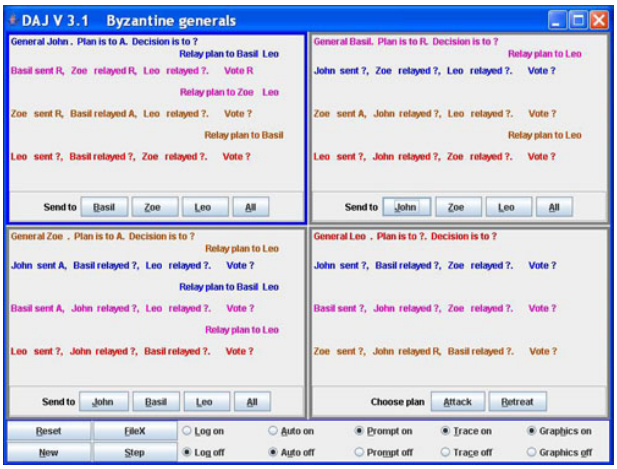
The "programs" constructed from correctness claims are actually nondeterministic finite automata which can be executed in parallel with the Promela programs, a process called*model checking*. For a brief introduction to model checking see [9 , Section 12.5]; more

advanced treatments are given in [51, Chapter 5] and [24]. The details of the construction of the automata in Spin and an explanation of the concept of acceptance cycles (and the related non-progress cycles) can be found in the Spin reference manual [33].

DAJ

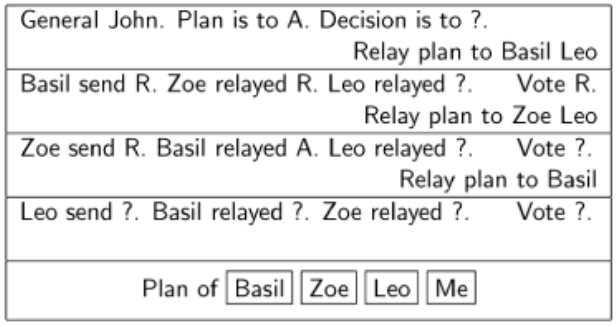
DAJ is an *interactive*, *visual study aid* for learning distributed algorithms [8].*Interactive*, because you must explicitly specify every step of the interleaved execution sequence.*Visual*, because the state of the nodes is continuously displayed.*Study aid*, because DAJ solves one of the most difficult problems encountered by students of these algorithms by automatically doing the necessary "book-keeping."

The program is an interactive state machine that displays the state of execution of an algorithm as you construct a scenario. The structure of the display for each algorithm is identical. Most of the screen is devoted to a grid of two to six panels, each one displaying the local state of a node in the system. The data for each node is color-coded, with a separate color for each node. Each panel contains a pair of lines for data relating to the node itself, followed by a pair of lines for data relating to each of the other nodes. The first line of a pair contains the data, followed by a second line which lists the outstanding messages that must be sent.



At the bottom of the screen is a line of buttons that globally affect all the nodes, while at the bottom of each node panel is a line of buttons for choosing a step of the algorithm. The contents of these buttons change according to the state of the node. To perform a step of an algorithm, select a button in one of the nodes. A sequence of such steps results in sending a message to another node, and the data structures of both the sending and receiving nodes are updated. Some buttons may not be active, although this is not denoted by any visual change in the buttons. It is a test of your understanding of the algorithm that you *not* click on a non-active node, though if you do so, the data structure is not changed.

For example, in the Byzantine Generals algorithm, a possible panel for General John is structured as follows:



The first line displays John's plan `A` and eventually his decision; the second line shows that he has yet to relay his initial choice to Basil and Leo. Since he has already relayed it to Zoe, the button for sending to Zoe will be inactive. The second pair of lines shows that Basil has sent his plan `R` to John, and that Zoe has relayed that she thinks that Basil's plan is `R`. This information is sufficient to enable John to carry out the vote concerning Basil's plan, as shown by the notation `Vote R`. The second line of prompts shows that John still has to relay the plan he received from Basil to Zoe and Leo. The line of buttons would be the second in a sequence for relaying message: `Send to`, `Plan of`, `Which is to`.

The `Reset` button returns all the nodes to their initial state. The `New` button will terminate the execution of the algorithm and return to the algorithm selection menu. The buttons `FileX`, `Step`, `Log on/off` and `Auto on/off` are used for reading and writing the log file, so that you can save a scenario and automatically rerun it. `Prompt on/off` selects whether to display the prompt lines, and `Trace on/off` selects whether to display the action trace window. Normally, prompts and traces will be turned on, but they can be turned off for assessment. `Graphics on/off` selects whether to display the graphics window. Currently, the visualizations that have been implemented are: the global queue of the Ricart–Agrawala algorithm (Section 10.3), the knowledge trees of the Byzantine Generals algorithm (Section 12.6) and the spanning tree of the Dijkstra–Scholten algorithm (Section 11.1).

A trace of all actions is displayed so that you and your instructor can discuss the sequence of actions leading to a particular state. The log file can also be used to analyze scenarios offline.

Ten distributed algorithms are currently implemented, including the following algorithms described in this book: the Ricart–Agrawala algorithm, the Dijkstra–Scholten algorithm, the Chandy–Lamport algorithm, the Byzantine Generals algorithm, the Berman–Garay King algorithm and the Huang–Mattern algorithm. The source code is structured so that you can implement other algorithms with only an elementary knowledge of Java: you need to implement a state machine for the algorithm and to construct the strings to display in the panels. Knowledge of the Swing libraries for constructing a user interface is not needed, because the uniform interface is constructed in an abstract superclass common to all the algorithms.

PREV
C. Concurrent Programming Problems

NEXT
E. Further Reading

Recommended / Queue / History / Topics / Tutorials / Settings / Blog / Get the App / Sign Out