The LGDF2 Language and Preprocessor

David C. DiNucci

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-026

October 1990

1. Introduction

This document explains the syntax of the Large-Grain Data Flow 2 (LGDF2) language, as based on the F-Net model of Portable Parallel Software Engineering. This section will give a brief description of F-Nets and their semantics. A much more detailed account can be found through the bibliography at the end of this document.

The F-Net formal model was devised as a basis for architecture-independent parallel software engineering. An F-Net consists of a set of variables[1], a set of operations ("processes"), and a set of instructions ("process calls") which reference both operations and variables.

An F-Net variable can be likened to the combination of a traditional imperative-language variable and a finite-state machine. As such, it possesses both a data value, called its data state, and the current state of the FSA, called its control state. It is the generalization of a single-assignment variable in a dataflow language, which can be regarded as having a two-state FSA with states "defined" and "undefined".

An F-Net operation consists of a signature and an implementation. The signature declares a set of arguments (formal variables), and identifies whether the instruction will use each for reading, writing, both reading and writing, or neither. In addition, it declares some transitions (formal control states) for each argument. The implementation expresses a functional mapping from the data states of the read arguments (when the operation begins execution) to both a new data state for each write argument and a transition for each argument. In the LGDF2 language discussed here, the implementation of an operation is expressed via the C language, augmented with a means of performing (declaring) transitions for arguments. Arguments that are used for both reading and writing can be updated in place. For efficiency, after a transition is performed for an argument, the data state of the argument is no longer accessible by the C program. Thus, performing a transition can be considered as "returning" that argument, while leaving the rest of the arguments for further computation. When all of the arguments have had a transition declared for them, execution of the operation terminates. If an operation never performs a transition for some argument (because, say, it goes into an infinite loop before performing the transition), it has the effect of performing a $\perp$ transition to the argument. This $\perp$ transition has the effect of leaving the control state of the argument permanently undefined.

An F-Net instruction provides a means of instantiating an operation, with bindings relating each of its arguments to an actual F-Net variable and each transition to an actual control state of that variable. In addition, it dictates when the operation can be invoked by naming a control state of each of the variables to which arguments are bound (pole bindings). The instruction will only "fire" (execute) when each of the variables has the named control state. Thus, the order in which instructions appear in an F-Net has no relation to the order in which they will execute. Any number of instructions can instantiate the same operation, with different variables and/or control states.

An F-Net is often represented graphically (see examples near the end of this document). Each variable is shown as a polygon, with one side for each of its control states. An instruction is represented graphically as a circle labeled for its operation, with one arc for each of its arguments between it and the variable to which that argument is bound. Read/write usages of the arguments are shown with arrow-heads, pole bindings are shown by the side of the polygon to which the arc is bound, and transitions are shown with pointers inside of the polygon from the end of the arc to the appropriate sides of the polygon.

The abstract F-Net model has two characteristics which greatly diminish the pitfalls normally associated with parallel programming. Since the LGDF2 language is a valid representa-

[1] In other literature on LGDF2 or F-Nets, LGDF2 variables are referred to as states, datapaths, data switches, or switches.

tion of the model[2], the LGDF2 programmer can also rely on these characteristics, even if they may seem unintuitive in the framework of operation implementations in C.

(1) Operations (and therefore instructions) execute atomically—i.e. as though each execution took no time, or identically, as though the executions were serialized in time with each finishing within a finite amount of time. In LGDF2, this characteristic ensures that no instruction can detect or effect the execution of other concurrently-executing instructions, and that the order in which an operation declares transitions for its arguments is not important in determining the effects of the operation's execution.[3]

(2) If an argument has but one declared transition in its signature, and does not have write usage, then the argument is called non-volatile. An F-Net operation will never perform a transition of $\perp$ to a non-volatile argument. In LGDF2, this means that a transition will be performed to a non-volatile argument even if the C program implementing the operation never does so explicitly. This allows the possibility for the scheduler to invoke a subsequent instruction to access the variable even before the current one has finished. In a shared-memory environment such as the Sequent Symmetry, if that subsequent instruction does not have write usage, both instructions will access the data state of the variable concurrently. If the subsequent instruction has write usage, the scheduler will supply it with a new version of the data state, and the old data state being discarded when the reader is finished with it. Note that all of this occurs behind the programmer's back—there is no need or ability to compensate for or explicitly request buffering or shared reading.

An LGDF2 program consists of 4 sections: (1) Included text, (2) Variable declarations, (3) Operation declarations, and (4) Instruction declarations. Six optional pragmas, to be discussed later, are scattered among these other sections, each of the form

// hints //

where ***hints*** is a space-separated list of hints. Pragmas never affect the formal semantics of the execution, but are used to increase the parallelism or efficiency of the execution and/or turn tracing and debugging options on or off. C-style comments are allowed anywhere within any of the sections.

Some examples of LGDF2 programs are given in section near the end of this document which may provide grounding for the reader as the syntax is discussed.

2. Included text

The included text section is optional. It's function is to provide a means for the user to include un-interpreted material directly into the C program produced by the `lgdf2` preprocessor. If the included text section appears, it is of the form:

`Include {` ***included_text*** `} endinclude`

where ***included_text*** is any text, and may include newline characters. Newline characters cannot appear between the braces and their adjoining keywords (though white-space can).

The included text section is provided primarily to compensate for a "shortcoming" in the LGDF2 syntax. LGDF2 does not know C, and specifically, does not know the syntax of a C data type or constant expression. In LGDF2, all data types and constant expressions must be in the form of an identifier. The included text section provides a means to associate identifiers with complex data types and constants through the use of `#include`, `#define`, and `typedef` statements. In the rest of this document, ***type*** fields will refer to either an identifier

[2] A model for the model?

[3] Other than possibly altering the efficiency of execution.

defined within the included text section in this way, a simple C type (`int`, `float`, `char`, etc.), or a special LGDF2 type (`stream` or `file`).

3. Variable Declarations

All LGDF2 variables used within any of the instructions must be declared in the variable declarations section. Any value which must be communicated between LGDF2 instructions or which must persists across executions of a single instruction must reside on an LGDF2 variable.

The variable declaration section consists of one or more variable declarations, each of the form

type var dim (*control_domain*) *initial* ; *var_pragma*

where *type* is as defined above, *var* is the name of the variable being declared, *dim* is an optional dimension of the form

[*constant*]

(to be described in a later section), *control_domain* is a space-separated list of identifiers, *initial* is an optional field of the form

= "*string*"

and *var_pragma* is an optional pragma. The *type* field describes the domain of the data state of the variable, and the *control_domain* field describes the domain of the control state of the variable[4] (when interpreted as a set containing the named identifiers). The first element of the control domain corresponds to the initial value given to the control state of the variable. If the *initial* field is present, its *string* subfield must be a valid C initializer, complete with braces, for a variable of the type denoted by *type*. If the *type* field is `file` or `stream`, the *string* subfield should not contain braces, but instead the name of a file, as described in a later section.

When the *control_domain* has but a single element, it is often clumsy to try to find a name for it, so the following syntactic sugar is provided: if the *control_domain* field and its surrounding parens are omitted, a control state domain containing a single element, null, is declared for that variable.

4. Operation Declarations

The Operation Declaration section consists of the keyword `Ops` followed by one or more operation declarations, each of the form

opname [*sig*] *op_pragma* { *opbody* } `endop`

where *opname* is the name of the operation being defined, *sig* is the signature of the operation, and *opbody* is the "mainline" C program which implements the operation. No newline characters may appear between the right brace and the `endop` keyword. The signature *sig* consists of one or more argument signatures, each of the form

usages type argname (*transs*) *arg_pragma*

where *usages* is either `in` (signifying read usage), `out` (write usage), `inout` (read and write usage), or `nodata` (neither read nor write usage), *type* is as described above (and is omitted the *usages* field is `nodata`), and *transs* is a space-separated list of identifiers representing the possible transitions for the argument. Following the same syntactic sugaring guidelines expressed above for variables, if the *transs* field and its surrounding parens are omitted, a single transition with a null name will be declared for the argument.

The *opbody* consists of a C block—i.e. local variable declarations followed by C statements, which may include other nested blocks) with the following exceptions and additions:

[4]In the literature, the elements of the control state domain are sometimes called poles

(1) No variables should be declared as `static`. (This is unfortunately not enforced by the `lgdf2` preprocessor.) The proper way to handle static variables is to make them LGDF2 variables.

(2) I/O should be performed only as suggested by the guidelines in a following section. (Again, not enforced.)

(3) Arguments having other than `nodata` usage can be used as variables of the same type anywhere within the opbody. However, `in` arguments cannot be assigned to. (Not enforced, but could have disastrous consequences if violated.) Any reference to an argument for which a transition has been performed will produce a run-time error.[5]

(4) Transition statements, each of the form

    $halt trans arg

or

    $$halt arg

may appear. The second form is used when the argument has only a single null transition declared for it. The ***halt*** field is either null, an exclamation point (`!`), or a question mark (`?`), and no white space may appear between it and the dollar sign to its left. ***arg*** is the name of an argument declared in the signature, and ***trans*** is the name of a transition for that argument.

The execution of a transition statement has several effects: (1) the control state of the switch to which the argument is bound is assigned the control state corresponding to the transition binding in the instruction, (2) the argument is made unavailable to further data state accesses or transitions, and (3) if transitions have already been executed for all volatile arguments, the operation execution halts. It is this final effect to which the ***halt*** field refers. If ***halt*** is null, it signifies that the programmer does not expect the operation to halt with this transition. If it does halt, a warning will be issued. If halt is `!`, it signifies that a halt is expected on this transition. If there are still some volatile arguments for which transitions have not been performed, the operation will halt nonetheless, issuing a warning and performing the first-declared transition to each of the remaining arguments. If halt is `?`, it signifies that the transition may or may not cause the operation to halt: no warning will be issued either way.

Finally, there is a third form of transition,

$$$!

which takes no parameters. It is provided to halt the operation as described above, but without performing a transition first—i.e. it has the effect of branching past the last line of the opbody.

5. Instruction Declarations

The instruction declaration section consists of the keyword `Instrs` followed by instruction declarations, each of the form

opcode `[` ***bindings*** `]` ***instr_pragma***

where ***opcode*** is the name of a declared operation and ***bindings*** is one or more argument bindings, each of the form

arg `:` ***pole_bdg var*** `(` ***trans_bdgs*** `)` ***bdg_pragma***

where ***arg*** is an argument of operation ***opcode, var*** is a declared LGDF2 variable, ***pole_bdg*** is a control state of that variable, and ***trans_bdgs*** is a comma-separated list of transition bindings, each of the form

[5] In fact, this will de-reference a null pointer, which may or may not yield an error. On the Sequent, this will simply produce incorrect results unless the `-h` option is used on the loader, in which case it *might* produce a run-time error.

trans **:** *c_state*

where *trans* is a transition of argument *arg* and *c_state* is a control state of *var*. A null control state in the *pole_bdg*, *c_state*, or *trans* field is signified by omitting the field. Note that even if the *trans_bdgs* field consists of a null transition bound to a null control state, it must still be present—as `(:)`.

Unlike the formal model of F-Nets, this implementation provides the ability to bind multiple transitions to the same control state. This introduces a point of confusion: if an argument does not have write usage but has multiple transitions, and all of those transitions are bound to the same control state, is the argument non-volatile or not? If it were not, then the resulting instruction would have no representation in the formal model, so in such a case, the argument will be taken to be non-volatile.[6]

A further bit of syntactic sugar is also permitted. The *pole_bdg* field can actually consist of any number of control states, separated by the word `or`. In this case, the textual instruction declaration declares many different instructions, one for each permutation of pole bindings. (To illustrate, the last instruction declared in the example below actually becomes 8 instructions which are all the same but for their pole binding.) The logical result is as if there were a single instruction that could fire when the var has any of the named control states.

6. I/O

I/O is not explicitly addressed in the abstract F-Net model, so the following extensions have been made to facilitate a useful binding between the LGDF vars and Unix files and streams.

An LGDF var can be declared as type `file`, in which case it corresponds to a file descriptor, or `stream`, in which case it corresponds to a stream descriptor. The initializer (*initial*) field for vars of type `file` or `stream` plays a slightly different purpose than for other variables: it contains the name of the Unix file to which the var should correspond.

When execution begins, the file or stream named in the initializer is opened automatically. If there is no initializer for the variable, a temporary file will be opened (except for LGDF vars `stdin` and `stdout`, described below). The LGDF var can then be used as an argument to the standard Unix I/O routines (`fprintf`, `fscanf`, `fputc`, `fgetc`, etc., for streams, and `read`, `write`, etc., for files) by any operation bound to that var. Since the var represents both the content of the Unix file and a pointer into that file, all arguments of type `file` or `stream` should have `inout` usages. A `file` or `stream` should not be closed, and since these streams are unbuffered, it is unnecessary to flush their buffers (`fflush`).

Standard I/O (`printf`, `scanf`, the vars `stdin` and `stdout`) should not be used. However, if an LGDF2 var is named `stdin` or `stdout` and has a type of `file` or `stream` and has no initializer string, any I/O performed to the var will correspond to the appropriate standard I/O port.

7. Array Vars

If the optional *dim* field is included on a var declaration, it signifies that the switch is to be replicated *dim* times, where *dim* is either an integer or an identifier `#defined` in the included text section. If an initializer is present for an array var, each element of the var is initialized with its value. There is no difference between a var with dimension 1 and a var with no declared dimension. Any instruction bound to such an array var is replicated, with one instruction bound to each element. If an instruction is bound to multiple array vars, the instruction is

[6] This requires that there be a possible one-to-many relationship between the operations in this implementation and the operations in the formal model.

9. Pragmas

As described earlier, each optional pragma section is of the form

//hints//

and each hint is an identifier or an integer, and may be followed by a list of "subhints" within parentheses. Each subhint has the same form as a hint, allowing nesting to any level. The meaning (and thus effect) of a hint may depend on the pragma section in which it appears as well as the target architecture of the LGDF2 program. All hints defined here are valid for the Sequent Symmetry, and may not apply to other implementations. For the most part, hints which are not valid within some particular pragma are ignored.

An optional global pragma can appear before or after the included text section. The hints currently understood in the global pragma are as follows:

```
nactive(n)
wait
nowait
tslice(n)
trace(aspects)
```

The `nactive` hint tells the number of Unix (heavyweight) processes that should be utilized to execute the program. In general, if system load is light and the number of processes is fewer than the number of available processors, this will correspond to the number of processors being utilized in parallel. (In fact, there may be more processes than this forked at any one time, but the only `nactive` will be scheduled for execution under Unix.) If a parameter of 0 is given (the default), the number of available processors is checked at runtime, and 1/2 of that number of processes is used. If the parameter is greater than the number of available processors, only the number of available processors will be used.

The `wait` and `nowait` hints set the global defaults for whether non-volatile arguments should attempt to act like volatile arguments as long as possible. Recall that an argument is defined as non-volatile if and only if it has exactly one transition and does not have write usage, and that a non-volatile argument is guaranteed to perform its transition in a finite time, whether or not a transition is explicitly performed for it. If the `wait` hint is given, non-volatile arguments will not perform a transition until (a) the instruction explicitly performs a transition for the argument, (b) the instruction finishes, or (c) the instruction is descheduled (see `tslice` below). If the `nowait` hint is given, non-volatile arguments perform their transitions when the instruction is initiated, even if it requires that a copy of the data state of the corresponding switch be made to facilitate this, and any explicit request to perform a transition is ignored. If neither `wait` or `nowait` is present, `nowait` is assumed.

The `tslice` hint tells the time slice in cpu seconds. In the general case, no time-sharing is performed, and an instruction executes until it has performed a transition for each of its arguments. If, however, all of the available processors (dictated by `nactive`) become "plugged" with instructions which execute longer than a time slice, and there are other instructions waiting to execute, then one of the processors will suspend the currently executing instruction to begin another. The overhead for this operation is quite high, and its only purpose is to prevent a few divergent instructions from clogging the system and violating the liveness of the implementation, so the time slice should be set to some large value which is assumed to be larger than any instruction execution, unless that instruction has entered an infinite loop. The default value is 10 seconds.

The `trace` hint tells which aspects of the execution to trace. If not present, the global default is for no tracing. If present, the *aspects* can be `fire`, to trace instruction firings, `trans`, to trace each execution of a transition statement, and/or `var`, to print the contents of LGDF vars after each transition. The format for a trace will be described later.

replicated to create one instruction for each combination of array indices—i.e. the total number of instructions created is the product of the dimensions of all vars to which the instruction is bound. The effective result is as if there is a single instruction which can fire whenever each of the array vars it is bound to has at least one element with the "proper" control state.

It is sometimes useful for an operation to determine the element of an array var which it is accessing. The construct

$&argname

within an operation body will evaluate to the index of the var element to which argument *arg-name* is bound, with the first element having an index of 0. Thus, this construct will always return 0 for arguments bound to non-dimensioned vars. The use of this construct is illustrated later in the examples section (doctor's office).

Vars with type `file` or `stream` can have dimension other than one if and only if (1) the name of the var is not `stdin` or `stdout` and (2) there is no initializer for the switch. In this case, an array of temporary files or streams is created. The utility of this feature is limited by Unix, since a maximum of 20 files may be open at one time.

8. Subroutines and Separate Compilation

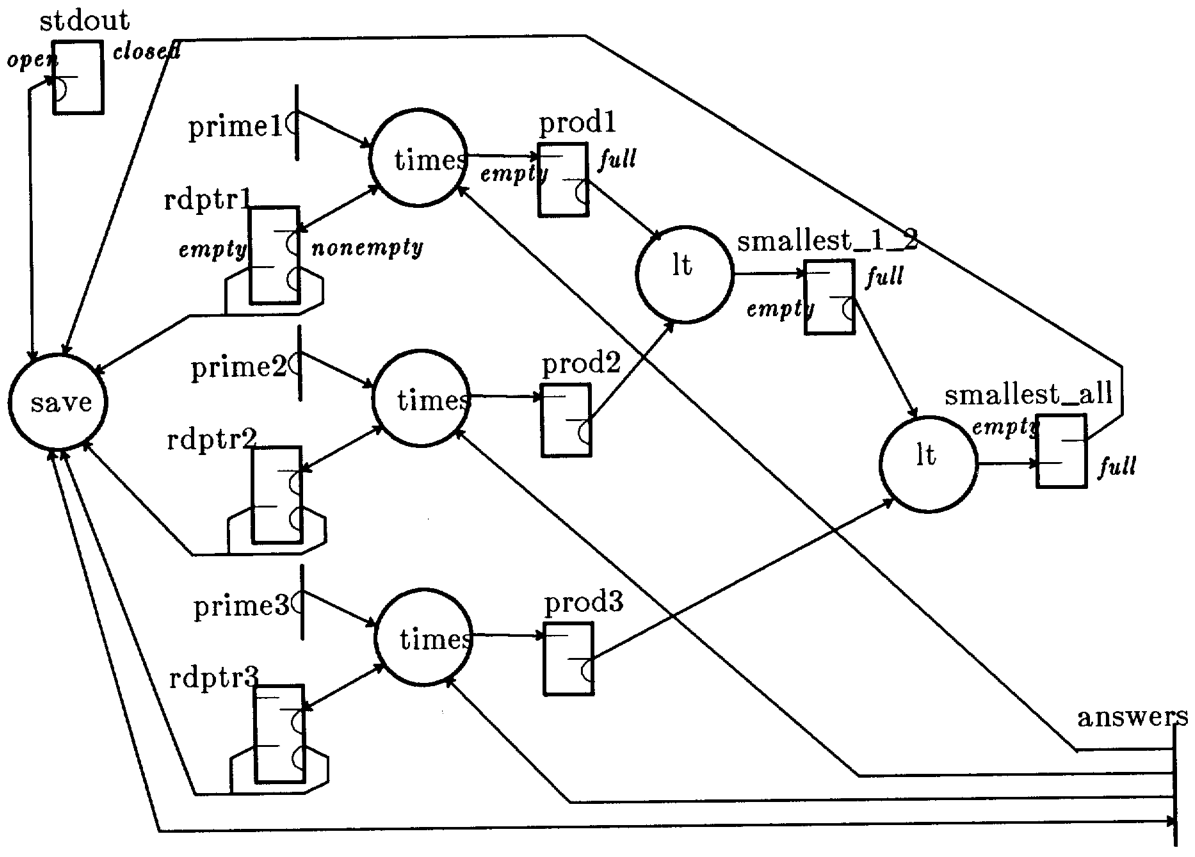
In an operation declaration, the *opbody* is restricted to being one C block—i.e. no functions may be declared within the *opbody*. Functions that may be ultimately called from the *opbody* fall into two categories: "normal" functions, which do not contain any of the special constructs offered by the preprocessor (transition statements, data state references, or index determination), and "LGDF2" functions which do contain one or more of these constructs.

Normal functions are compiled separately in the normal way. They are still bound to the LGDF2 rules that they may not harbor or access static or global variables (including `malloc`'d or `shmalloc`'d memory), and should not perform I/O. Data state of LGDF2 vars may be passed to them as arguments in the standard way.

An LGDF2 function is identical in form to a C function except that it should may contain the above constructs, must be preceded by the keyword `lgdf2`, and the terminal } must be followed by the keyword `endlgdf2` on the same line. LGDF2 vars should not be passed to an LGDF2 function: it is not only unnecessary, since the function body has direct access to the data states of its LGDF2 arguments, but is also dangerous, since it may allow access to the data state (through the alias created by C argument passing) after a transition has been performed for the LGDF2 argument.

In order to be correctly preprocessed, LGDF2 functions can either appear between the endop of the *opbody* from which they will be called and the following operation declaration, or can be compiled separately. A file for separate compilation is identical to an LGDF2 program minus the variable declarations section, instruction declarations section, pragmas, and *opbodys*—i.e. it consists only of the Ops keyword and operation headers followed by LGDF2 functions. The operation header in a separate compilation file must exactly match that for the same operation in the main LGDF2 program, though there is currently no automatic means for verifying this.

For large LGDF2 programs, it may be desirable to make all *opbodys* consist of a single function call, then define that function in a separate compilation file. (The function can even have the same name as the operation calling it, since LGDF2 operation names do not find their way into the resulting C program, while LGDF2 function names are passed along undisturbed.) In this case, the main LGDF2 program becomes no more than a skeleton, or "wirelist", for how the modules will interact. On the other hand, for small LGDF2 programs, normal functions may be bracketed by the `lgdf2`-`endlgdf2` keywords to allow their inclusion after the *opbody* of their associated operations.



Graphical representation of Hamming's Problem

```
stream  stdout       ( open closed )          = "hammingout";
int    prime1                                 = "{3}";
int    prime2                                 = "{5}";
int    prime3                                 = "{7}";
int    rdptr1        ( nonempty empty )       = "{0}";
int    rdptr2        ( nonempty empty )       = "{0}";
int    rdptr3        ( nonempty empty )       = "{0}";
int    prod1         ( empty full );
int    prod2         ( empty full );
int    prod3         ( empty full );
int    smallest_1_2  ( empty full );
int    smallest_all  ( empty full );

Ops
  times [ inout   int   rdptr      ( getl getlast )
          in      HLIST prev_ans
          in      int   prime
          out     int   product
          ]
```

There is but one possible hint for the *var_pragma*:
`onwrite(`*routine*`)`
where *routine* is the name of a user-supplied function to be called after each transition to the variable by an instruction which has write (`out` or `inout`) usage to the variable. The assumption is that *routine* will analyze and/or print the data state of the variable for debugging purposes. *routine* will be called only if variable tracing has been turned on in the global pragma. The function should be declared as follows:

```
routine(idx, lgdfvar)
int   idx;
type *lgdfvar;
```

where *type* is the type of the variable being traced. `idx` will contain the index of the variable. `lgdfvar` is the value of the variable. (Only one element of an LGDF array var, namely `idx`, is passed to the routine.) The routine should not alter the value of `lgdfvar`. Any output performed by *routine* should occur to `stderr`.

Only `wait` and `nowait` hints are allowed for the *arg_pragma* and *bdg_pragma* pragmas. The argument hints over-ride the effects of the associated hints in the global pragma for all instructions in which the operation is used, and the binding hints over-ride both the global pragma and the argument hints for a single instruction.

10. Examples

10.1. Hamming's Problem

The goal of Hamming's Problem is to take three prime numbers, i, j, and k, and to produce an ordered list of the first p numbers of the form $i^l j^m k^n$, where l, m, and n are non-negative integers. The strategy is to multiply the list of answers formed so far by each of the three primes and merge the resulting lists. As the list of answers grows, so does the list of answers!

In this case, the list of answers is kept in var `answers`, the three primes are kept in `prime1`, `prime2`, and `prime3`, the read pointers into the list of answers for each of the three multiplies kept in `rdptr1`, `rdptr2`, and `rdptr3`, and two temporary variables `smallest_1_2` and `smallest_all` are used to aid in the three-way merge. The control states of the read pointers reflect whether or not the portion of the answers which have not been traversed by it is empty or not. As new answers are added to the list, the value of these control states is therefore always set back to `nonempty`.

It is tempting to consider using an array of primes rather than three distinct vars so that any number of primes can be accomodated, but the binary tree for finding the smallest product only works for exactly three primes, and cannot dynamically grow or shrink to accomodate different sizes of prime arrays. This is being remedied in future versions of LGDF2.

```
// nactive(4) tslice(10) //

Include {

#define HLIST_SIZE      500
typedef struct {
        int     length;
        int     contents[HLIST_SIZE];
    } HLIST;

} endinclude

  HLIST answers                                    = "{1, {1}}";
```

```
        smallest: empty smallest_1_2  (: full)]
lt    [ opd1    : full prod3          (leave: full, take: empty)
        opd2    : full smallest_1_2   (leave: full, take: empty)
        smallest: empty smallest_all  (: full)]
save  [ new_ans : full smallest_all   (: empty )
        prev_ans: answers             (:)
        output  : open stdout         (putl: open, putlast: closed)
        rdptr1  : empty or nonempty rdptr1 ( : nonempty)
        rdptr2  : empty or nonempty rdptr2 ( : nonempty)
        rdptr3  : empty or nonempty rdptr3 ( : nonempty) ]
```

10.2. A Doctor's Office

The following problem simulates a doctor's office. Idle doctors line up waiting for patients. A flu-bug will non-deterministically descend upon unsuspecting people, at which time they go to the doctor's office and line up. When there is at least one idle doctor and one patient in line, a receptionist (`recept1`) takes the first doctor and first patient and puts them into a treatment room. Here they wait for a non-deterministic amount of time until mercy descends upon them and the patient is cured. At this time, a second receptionist notices and takes them from the room, putting the doctor back in line and sending the cured patient back into the world.

The `mercy` instruction is superfluous, and has been shown with an empty body.[7] Even if a body was present, it would not be executed: the run-time system knows that it has only non-volatile arguments, so the execution of the body can have no effect on the result. If the user desired to make `mercy` execute for some period of time, the argument to mercy would need to be declared as volatile (by altering the usages or number of transitions).

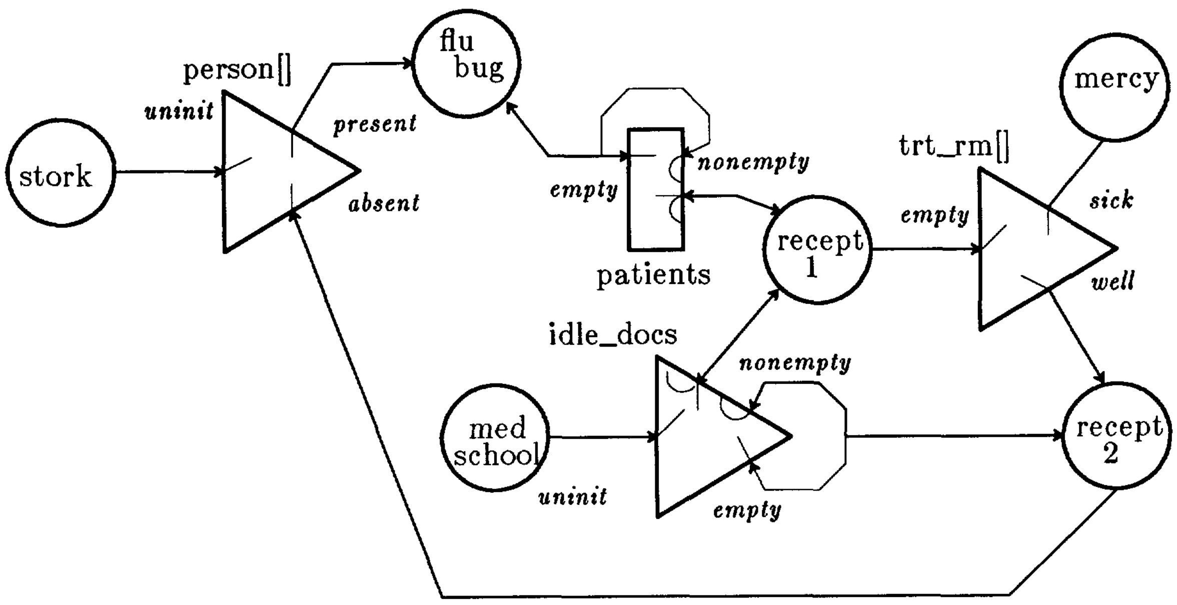
```
//trace(var fire trans) nowait//
Include {
#define NROOMS  50                        /* Number of rooms */
#define NDOCS   50                        /* Number of doctors */
#define NPEOPLE 50                        /* Number of people */
#define QSIZE   100

typedef struct room {
        int     treating_doc;
        int     patient;
   } ROOM;

typedef struct queue {
        int     name[QSIZE];
        int     last_in;
        int     next_out;
   } QUEUE;

} endinclude

        QUEUE   idle_docs       (uninit empty nonempty) = "{{0},-1,0}";
        QUEUE   patients        (empty nonempty)        = "{{0},-1,0}";
        ROOM    trt_rm[NROOMS]  (empty sick well);  //onwrite(p_room)//
        int     person[NPEOPLE] (uninit absent present);
        stream  stdout;
```

[7] Have you no mercy?

```
  {
      product = prev_ans.contents[rdptr] * prime;
      $$product
      if (++rdptr == prev_ans.length)
        $!getlast rdptr
      else
        $!getl rdptr
  } endop

  lt      [ in   int opdl    ( leave take )
            in   int opd2    ( leave take )
            out  int smallest]
  {
        if (opdl < opd2) {
            smallest = opdl;
            $take opdl
        } else if (opdl == opd2) {
            smallest = opdl;
            $take opdl
            $take opd2
        } else {
            smallest = opd2;
            $take opd2
        }
  } endop

  save  [ in    int     new_ans
          inout HLIST   prev_ans
          inout stream  output ( putl putlast)
          nodata        rdptr1
          nodata        rdptr2
          nodata        rdptr3]
  {
      prev_ans.contents[prev_ans.length++] = new_ans;
      fprintf(output, "%d0, new_ans);
      if (prev_ans.length == HLIST_SIZE) $putlast output
  } endop

Instrs
  times [ rdptr   : nonempty rdptr1     (getl: nonempty, getlast: empty)
          prev_ans: answers             (:)
          prime   : prime1              (:)
          product : empty prod1         (: full )]
  times [ rdptr   : nonempty rdptr2     (getl: nonempty, getlast: empty)
          prev_ans: answers             (:)
          prime   : prime2              (:)
          product : empty prod2         (: full )]
  times [ rdptr   : nonempty rdptr3     (getl: nonempty, getlast: empty)
          prev_ans: answers             (:)
          prime   : prime3              (:)
          product : empty prod3         (: full )]
  lt    [ opdl    : full prodl          (leave: full, take: empty)
          opd2    : full prod2          (leave: full, take: empty)
        if (patients.next_out++ == patients.last_in) $takelast patients
    } endop

    recept2    [ in    ROOM  room
                 inout QUEUE docs
                 out   int   well_person]
    {
        docs.name[++docs.last_in % QSIZE] = room.treating_doc;
        well_person = room.patient;
    } endop

    flu_bug    [ in    int   well_person
                 inout QUEUE patients]
    {
        patients.name[++patients.last_in % QSIZE] = well_person;
    } endop

    mercy      [ nodata      room]
    {    } endop

Include {
    void p_room(idx, room)
    int         idx;
    ROOM        *room;
    {
        fprintf(stderr, "Room %d, Patient %d, Treating doc %d0,
                        idx, room -> patient, room -> treating_doc);
    }
} endinclude

Instrs
    med_school [ docs:    uninit idle_docs            (: nonempty) ]
    stork [ person: uninit person                     (: present)  ]
    recept1    [ docs:    nonempty idle_docs
                                (take1: nonempty, takelast: empty)
                 patients: nonempty patients
                                (take1: nonempty, takelast: empty)
                 new_rm:   empty trt_rm               (: sick)
                 outfile:  stdout(:)]
    mercy      [ room:    sick trt_rm                 (: well)     ]
    recept2    [ room:    well trt_rm                 (: empty)
                 docs:    empty or nonempty idle_docs (: nonempty)
                 well_person: absent person           (: present)  ]
    flu_bug    [ well_person: present person          (: absent)
                 patients: empty or nonempty patients (: nonempty) ]
```

11. Tracing

Tracing is divided into transition tracing, listing each transition as it is performed, and firing tracing, listing each instruction as it fires. If both are requested, a firing will always be the result of a transition, and this cause-effect relationship is shown by listing the firing to the right of the transition which enabled it. Tracing is always printed to stderr.



Graphical representation of Doctor's Office

```
Ops
    med_school [ out   QUEUE docs ]
    {
        int docname;

        docs.next_out = 0;
        for (docname = 0; docname < NDOCS; docname++)
            docs.name[docname] = docname;
        docs.last_in = NDOCS - 1;
    } endop

    stork [ out int person]
    {
        person = $&person;
    } endop

    recept1    [ inout QUEUE docs         (take1 takelast)
                 inout QUEUE patients     (take1 takelast)
                 out   ROOM  new_rm
                 inout stream outfile]
    {
        new_rm.treating_doc = docs.name[docs.next_out % QSIZE];
        if (docs.next_out++ == docs.last_in) $takelast docs
        new_rm.patient = patients.name[patients.next_out % QSIZE];
```

```
LGDF: +  stork@83 $         person ( person&41 present)
LGDF: +  stork@83 $         person ( person&39 present)
LGDF: +  stork@83 $         person ( person&42 present)
LGDF: +  stork@83 $         person ( person&44 present)
LGDF: +  stork@83 $         person ( person&43 present)
LGDF: +  stork@83 $         person ( person&45 present)
LGDF: +  stork@83 $         person ( person&40 present)
LGDF: +  stork@83 $         person ( person&48 present)
LGDF: +  stork@83 $         person ( person&47 present)
LGDF: +  stork@83 $         person ( person&46 present)
LGDF: -flu_bug@94 $        well_person ( person&0  absent)
LGDF: +  stork@83 $         person ( person&49 present)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&49 0
LGDF: -flu_bug@94 $        well_person ( person&49  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&48 0
LGDF: -flu_bug@94 $        well_person ( person&48  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&47 0
LGDF: -flu_bug@94 $        well_person ( person&47  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&46 0
          ...
LGDF: -flu_bug@94 $        well_person ( person&6  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&4 0
LGDF: -flu_bug@94 $        well_person ( person&4  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&5 0
LGDF: -flu_bug@94 $        well_person ( person&5  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&3 0
LGDF: -flu_bug@94 $        well_person ( person&3  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&2 0
LGDF: -flu_bug@94 $        well_person ( person&2  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> flu_bug@94 $&1 0
LGDF: -flu_bug@94 $        well_person ( person&1  absent)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty) -> recept1@84 $&0 0 49 0
LGDF: +recept1@84 $  take1    docs (idle_docs&0 nonempty)
LGDF: +recept1@84 $  take1 patients (patients&0 nonempty)
Room 49, Patient 0, Treating doc 0
LGDF: +recept1@84 $         new_rm ( trt_rm&49    sick) ->   mercy@90 $&49
LGDF: -  mercy@90 $           room ( trt_rm&49    well) -> recept2@91 $&49 0 1
LGDF: +recept1@84 $        outfile ( stdout&0         )
LGDF: -recept2@91 $           room ( trt_rm&49   empty)
LGDF: +recept2@91 $           docs (idle_docs&0 nonempty) -> recept1@84 $&0 0 49 0
LGDF: +recept1@84 $  take1    docs (idle_docs&0 nonempty)
LGDF: +recept2@91 $        well_person ( person&1 present)
LGDF: +recept1@84 $  take1 patients (patients&0 nonempty) -> flu_bug@94 $&1 0
LGDF: -flu_bug@94 $        well_person ( person&1  absent)
Room 49, Patient 49, Treating doc 1
LGDF: +flu_bug@94 $        patients (patients&0 nonempty)
LGDF: -  mercy@90 $           room ( trt_rm&49    well) -> recept2@91 $&49 0 1
LGDF: -recept2@91 $           room ( trt_rm&49   empty)
LGDF: +recept1@84 $         new_rm ( trt_rm&49    sick) ->   mercy@90 $&49
LGDF: +recept2@91 $           docs (idle_docs&0 nonempty)
LGDF: +recept1@84 $        outfile ( stdout&0         ) -> recept1@84 $&0 0 49 0
LGDF: +recept1@84 $  take1    docs (idle_docs&0 nonempty)
LGDF: +recept2@91 $        well_person ( person&1 present)
```

The format for a transition trace is:

opn@line $trans arg (var&index cstate)

The first two fields identify the instruction in which the transition is being performed by its opcode and the line of the source file it is declared on. The next two fields are the transition name and argument for the transition. The last three fields (in parens) are the variable to which the arg is bound, expressed as its name and index, and the new control state assigned to the variable as a result of the transition. The transition trace may be preceded by a +, signifying that the transition was performed automatically after the opbody finished, or a - signifying that the transition was performed automatically by the runtime system because the argument was non-volatile.

The format for a firing trace is:

-> *opn@line $& index1 index2 ...*

The first two fields describe the instruction which fired, again by its operation and line in the source. The rest of the fields are the indices for each of the arguments. These are listed in the order in which the arguments are declared for the operation, not in the order of the bindings for the instruction.

What follows is a partial trace from the doctors office example:

```
% doc
LGDF: ------------------init------------------ -> med_school@82 $&0
LGDF: ------------------init------------------ ->   stork@83 $&0
LGDF: ------------------init------------------ ->   stork@83 $&1
LGDF: ------------------init------------------ ->   stork@83 $&2
LGDF: ------------------init------------------ ->   stork@83 $&3
LGDF: ------------------init------------------ ->   stork@83 $&4
LGDF: ------------------init------------------ ->   stork@83 $&5
           ...
LGDF: ------------------init------------------ ->   stork@83 $&46
LGDF: ------------------init------------------ ->   stork@83 $&47
LGDF: ------------------init------------------ ->   stork@83 $&48
LGDF: ------------------init------------------ ->   stork@83 $&49
LGDF: +med_school@82 $           docs (idle_docs&0 nonempty)
LGDF: +   stork@83 $        person ( person&0 present) -> flu_bug@94 $&0 0
LGDF: +   stork@83 $        person ( person&1 present)
LGDF: +   stork@83 $        person ( person&2 present)
LGDF: +   stork@83 $        person ( person&3 present)
LGDF: +   stork@83 $        person ( person&5 present)
LGDF: +   stork@83 $        person ( person&4 present)
LGDF: +   stork@83 $        person ( person&6 present)
LGDF: +   stork@83 $        person ( person&7 present)
LGDF: +   stork@83 $        person ( person&8 present)
LGDF: +   stork@83 $        person ( person&9 present)
LGDF: +   stork@83 $        person ( person&10 present)
LGDF: +   stork@83 $        person ( person&12 present)
LGDF: +   stork@83 $        person ( person&14 present)
LGDF: +   stork@83 $        person ( person&11 present)
LGDF: +   stork@83 $        person ( person&15 present)
LGDF: +   stork@83 $        person ( person&16 present)
LGDF: +   stork@83 $        person ( person&13 present)
LGDF: +   stork@83 $        person ( person&18 present)
LGDF: +   stork@83 $        person ( person&17 present)
          ...
```

```
LGDF: +recept1@84 $  take1 patients (patients&0 nonempty) -> flu_bug@94 $&1 0
LGDF: -flu_bug@94 $        well_person ( person&1  absent)
Room 49, Patient 48, Treating doc 2
LGDF: +flu_bug@94 $        patients (patients&0 nonempty)
LGDF: +recept1@84 $         new_rm ( trt_rm&49    sick) ->   mercy@90 $&49
LGDF: -recept2@91 $           room ( trt_rm&49   empty)
LGDF: -  mercy@90 $           room ( trt_rm&49    well) -> recept2@91 $&49 0 1
LGDF: +recept1@84 $  take1    docs (idle_docs&0 nonempty)
LGDF: +recept1@84 $        outfile ( stdout&0        )
LGDF: +recept1@84 $  take1 patients (patients&0 nonempty)
LGDF: +recept2@91 $           docs (idle_docs&0 nonempty) -> recept1@84 $&0 0 49 0
Room 49, Patient 47, Treating doc 3
LGDF: +recept2@91 $        well_person ( person&1 present) -> flu_bug@94 $&1 0
LGDF: -flu_bug@94 $        well_person ( person&1  absent)
LGDF: +recept1@84 $         new_rm ( trt_rm&49    sick) ->   mercy@90 $&49
LGDF: -recept2@91 $           room ( trt_rm&49   empty)
LGDF: +flu_bug@94 $        patients (patients&0 nonempty)
LGDF: -  mercy@90 $           room ( trt_rm&49    well) -> recept2@91 $&49 0 1
LGDF: +recept1@84 $  take1    docs (idle_docs&0 nonempty)
LGDF: +recept2@91 $           docs (idle_docs&0 nonempty) -> recept1@84 $&0 0 49 0
LGDF: +recept1@84 $        outfile ( stdout&0        )
LGDF: +recept2@91 $        well_person ( person&1 present) -> flu_bug@94 $&1 0
LGDF: +recept1@84 $  take1 patients (patients&0 nonempty)
LGDF: -flu_bug@94 $        well_person ( person&1  absent)
Room 49, Patient 46, Treating doc 4
LGDF: +flu_bug@94 $        patients (patients&0 nonempty)
LGDF: +recept1@84 $         new_rm ( trt_rm&49    sick) ->   mercy@90 $&49
LGDF: -  mercy@90 $           room ( trt_rm&49    well) -> recept2@91 $&49 0 1
LGDF: +recept1@84 $        outfile ( stdout&0        )
LGDF: -recept2@91 $           room ( trt_rm&49   empty)
LGDF: +recept2@91 $           docs (idle_docs&0 nonempty) -> recept1@84 $&0 0 49 0
LGDF: +recept1@84 $  take1    docs (idle_docs&0 nonempty)
LGDF: +recept2@91 $        well_person ( person&1 present)
LGDF: +recept1@84 $  take1 patients (patients&0 nonempty) -> flu_bug@94 $&1 0
LGDF: -flu_bug@94 $        well_person ( person&1  absent)
Room 49, Patient 45, Treating doc 5
LGDF: +flu_bug@94 $        patients (patients&0 nonempty)
LGDF: +recept1@84 $         new_rm ( trt_rm&49    sick) ->   mercy@90 $&49
LGDF: -recept2@91 $           room ( trt_rm&49   empty)
LGDF: +recept1@84 $        outfile ( stdout&0        )
LGDF: -  mercy@90 $           room ( trt_rm&49    well) -> recept2@91 $&49 0 1
LGDF: +recept2@91 $           docs (idle_docs&0 nonempty) -> recept1@84 $&0 0 49 0
LGDF: +recept1@84 $  take1    docs (idle_docs&0 nonempty)
LGDF: +recept2@91 $        well_person ( person&1 present)
LGDF: +recept1@84 $  take1 patients (patients&0 nonempty) -> flu_bug@94 $&1 0
                ...
```

It is interesting to note that only room 49 gets used for treating patients, since the next patient is only taken when the previous patient is already cured and leaves the room vacant. This is at least partially due to the time required for tracing. If only var tracing is used, room 48 is also used sometimes.

12. Preparing an LGDF2 Program for Execution

The LGDF2 program is processed in three steps: (1) The `lgdf2` processor,

`/ogc/projects/parallel/bin/lgdf2`

converts the program to a `c` language program, (2) that `c` language program is compiled with the standard `c` compiler, and (3) the resulting object file is linked with the LGDF2 runtime object files,

`/ogc/projects/parallel/lgdf/v1.0/sequent/runtime/sched.o`
`/ogc/projects/parallel/lgdf/v1.0/sequent/runtime/tslice.o`

to yield an executable file. Here is a sample make file from directory `/ogc/projects/parallel/lgdf/v1.0/samples`.

```
hammings:       hammings.o
        cc -g -h -o hammings hammings.o ../sequent/runtime/sched.o
hammings.o:     hammings.c
        cc -g -c hammings.c
hammings.c:     hammings.l
        ../bin/lgdf2 < hammings.l > hammings.c
```

C syntax errors within the operation definitions or initializers will not be caught by the `lgdf2` preprocessor, since the preprocessor does not know C. These will be caught during compile time, but the line numbers issued by the compiler will conform to those in the initial LGDF2 (`.l`) file.