FINITE STATE DESCRIPTION LANGUAGE: A NEW TOOL FOR WRITING STIMULATING CONTROLLERS

Phillips, G.F.

Institute of Orthopaedics, University College, London, ENGLAND

ABSTRACT

A new computer programming language is presented for writing Finite State controllers. It is also do drive neuromuscular stimulators connected to a PC compatible microcomputer, or it may be cross-compiled and run on an embedded microcontroller in a portable stimulator.

The body of an Finite State Description Language (FSDL) program comprises descriptions of the states through which the controller may pass. The writer specifies the actions to be performed within each state, and the rules for the transitions between states.

The two primary aims of the language are reliability and usability. Reliability is achieved through maximum checking at compile time, and through innovative run-time error checking. Usability is achieved through closely matching the elements of the language with the elements of the specifications of actual controllers.

The experience of using this language in the setting of a clinical FES research unit will be presented. It is found that FSDL programs are concise, intelligible and convenient.

KEY WORDS: Finite state control, Programming language, FES, Computer control, Gait

INTRODUCTION

Finite state control

The prevailing method for specifying controllers for neuromuscular stimulators is the "finite state" model. The designer divides the control problem into a number of distinct states, and specifies the rules which govern the choice of state at each moment in time. Within each state, the designer specifies what local control actions, either open loop or closed loop are to be performed. The resulting controller is sometimes known as a rule-based controller or as a finite state automation (FSA).

While it is possible ti implement FSAs directly in hardware, they are more usually undertaken by programmable systems. The designer of the controller must then convert the rules and actions into an appropriate program in a language such as C, Pascal or assembly language. This presents a number of problems:

- participation in the controller design process is restricted to those with an understanding of the chosen programming language, thereby undervaluing the contributions of clinical staff.
- thinking about *what* the controller should do becomes confused with thinking about *how* the controller is to do it,
- small changes in the controller specification may entail large changes in the control program, and are likely to result in unforeseen side effects,
- errors arising at run time (such as unending loops) are liable to cause incorrect or potentially dangerous stimulation output.

Design tools for finite state controllers

There are ways of facilitating the design process. The most commonly used is the state diagram, in which circles or ovals indicate the states, the arcs and arrows represent the transitions. Despite its appealing simplicity, the state diagram may be of guaranteeing a correspondence between the diagram and the program. If a program fails to function correctly, it is likely that the program is failing to implement the diagram as that the diagram is failing to reflect the designer's intentions.

It might be possible to automate the process of creating programs from state diagrams. But this approach would still have some serious shortcomings:

- diagrams generally indicate only the transitions between the states, not the actions within the states,
- if they were to include the actions they would become clutters and unreadable,
- there is no satisfactory method of representing on a diagram the "global" information which in programs is declared in symbolic constants, procedures, functions etc.

This point about global information is important in considering the maintenance of programs. If for example a complex rule governs five different state transitions, there must be a method of declaring the rule once and not five times, and thus of subsequently modifying it without unnecessary repetition.

There have been a number of attempts to simplify the coding of finite state controllers. Abbas [1] presented a database within which one could specify control actions and transition rules for driving a 48 channel percutaneous stimulator. The database was hierarchical rather than relational, resulting in gross redundancy in the data and a lack of normalization. The term normalization is used to express one-to-one correspondence between items of data in the database and items of data in the controller specification. This leads to problems in maintaining a controller. Further problems were:

- it was necessary to use messages to synchronize the FSA with an open-loop controller running on the same processor at the same time,
- editing facilities were poor,

- local controllers were chosen from a fixed number of available options: control strategies not built and could not be implemented.

Another attempt to simplify the coding of finite state controllers has been made by Perkins at the Medical Research Council Neurological Prostheses Unit in London (personal communication). This consists of a language "StatCon" implemented as a set of macros within the context of an assembler for 1806 code. This language has been used for a number of controllers for implanted stimulators including an intercostal breathing program [3]. The drawbacks of this language are:

- it is closely tied to the instruction of one particular microcontroller,
- nobody but Perkins has ever mastered the language.

I attempted to combine the benefits of relational database technology with those of a programming language into a system known as Finite State DataBase (FSDB). Separate tables contained information on states, transitions, output channels etc. One table contained 'functions' which were program fragments in a postfix language similar to Forth. FSDB has not been pursued beyond initial trials because:

- nobody liked the postfix language,
- editing facilities were poor,
- compilation speed was slow.

This paper, now, presents a working language generated in the light of the above experience.

AIMS

Functional aims

Finite State Description Language (FSDL) is designed to perform a specific function: to minimize the distance between *specifying* a finite state controller and *implementing* it as a computer program.

Specifying a controller means describing:

- the states through which the controller may pass,
- the actions top be performed within each state,
- the rules for moving between states.

The actions to be performed within a state comprise such things as switching on or off the stimulator output of a channel, or implementing as feed back controller based on sensor data. These actions fall into four distinct groups:

- those which are to be performed only once on *entry* to a given state,
- those which are to be performed only once after a *delay* while waiting for the occurrence of some predetermined condition,
- those which are to be performed only once on *exit* from a state,
- those which are to be performed *continually* throughout the tenure from a state.

It can be shown that only the first and last are strictly necessary. However, the addition of the two further classes of action greatly simplifies the working. The ability

to specify actions to be done after the *delay* helps to reduce the necessary number of states and thereby eases maintenance.

The designer of the finite state controller next has to specify the rules for changing state. These naturally fall into two classes: "local" rules and "global" rules:

- Those transitions which taker into account the current state are term "local", since they are defined within the description of the state within which they will be applied. These rules constitute the "normal" route around a controller,
- Those transitions which do not take into account the current state (that is, which consider only external or sensor information) are termed "global" since they are defined with the description of any state and are applied throughout the whole duration of the controller. These rules are generally applied to "exceptions" such as transitions top an emergency state on detecting that a sensor has failed.

A language does not need to provide a construct for global transition rules, since it would be possible to declare these within each separate state. However, such an approach would tend to clutter a program with unnecessary detail. It would go against the aims of normalization already referred to, with consequent repercussions on maintainability.

FSDL aims to be machine independent. A program written on a PC may be run on the same machine or on a quite different machine such as an embedded microcontroller in a portable stimulator. To date, target machines are the IBM-compatible PC and the Motorola 68HC11 microcontroller. The list can be extended without great difficulty.

Machine independence is achieved through the use of an intermediate language, termed "F-code". This is described in a letter section. A directive can inform the compiler that the output code is to be run an a certain class of machine: the compiler may impose limitations appropriate to the target configuration.

Language aims

Reliability is the primary aim of the language. Readability and usability come next. Efficiency is also a consideration.

Reliability firstly means that any errors in the program will be detected and reported at the first time the program is compiled. While the program cannot of course know by telepathy what the programmer intends, it can detect most inadvertent errors. There are no automatic declarations, no default assignments, no implicit type coercions. The present version of the compiler does coerce integer to Booleans, however; this is due to be corrected in the next edition. Formal and actual parameters must match exactly. The compiler can check that there exists a route around all states.

The language has been designed to discourage "short cuts" and "fancy" techniques. In this respect it is modified more closely on Pascal than on C. Statements do not yield return values and cannot be used in expressions. Also, procedures and functions are not recursive, since recursion is inappropriate in a real-time system.

Correct programs constitute only a minority of those submitted to a compiler: the error case is the normal case [4]. An explicit aim of the language is that the compiler give helpful and localized error messages.

FSDL is equipped with a further feature for maximizing reliability, which is not found in general purpose languages. This is known as the "watchdog" system, and is discusses in a later section.

A language is usable if it provides in an understandable form the constructions which the programmer needs. In FSDL, the keyword of the language are the terms used for specifying the controller. On reading a program, one should understand how the controller will behave, as illustrated by the example at the end of this paper.

Usability implies maintainability. The ease of maintenance of a program is enhanced by its modular nature, by the use of names for constant items, by the elimination of side effects (such as actions embedded in expressions), and by a syntax which makes the use of "goto" unnecessary.

The ease of use of the language is improved by an integrated environment in which the compiler runs as a task an editor. Compilation is performed with a single press of a function key. Errors in the source program are reported within the editor, with the cursor placed at the very near the error. Debugging facilities are also being developed: an annotated object code ,listing can be produce for manual checking, and we also have plans for run-time debugging aids.

From the user's point of view, speed of compilation also contributes to ease of use. The language has been designed to allow single-pass compilation without back-tracking. A moderate sized program can be compiled in a few seconds.

Efficiency has not been given high priority in this project. It is known that some inefficiency is introduced as a result of the stack-based nature of F-code, and steps have been taken to minimize these inefficiencies. However, two considerations dictate against excessive attention to efficiency:

- If it happens that our hardware becomes unable to cope with a complex controller in real time, it will be safer and easier to buy more powerful processors than to try to maximize efficiency through software optimization.
- The important aims of reliability and usability must never be sacrificed to efficiency. No matter how fast a control program may run, it will be no use if it might crash if it cannot be safely modified.

LANGUAGE DESCRIPTION

Program output to the stimulator

The method of specifying the action of the stimulator generally follows the principles adopted in the Strathclyde Research Stimulator [11]. The following routines are intrinsic procedures of FSDL; other routines are also available.

```
StimStart;        {switch on the high voltage supply and enable timed interrupts}
StimStop;                                   {interrupts disabled; high voltage off}
```

StimOff*(channel);* *(no output on The specified channel}*

StimOn *(channel, PulseWidth,IPI);* *{start stimulation of given parameters}*

StimSetPW *(channel.PulseWidth);* *{vary the pulse width only}*

StimRamp *(channel,PW1,IPI,PW2,deltaPW);* *{ramp pulse width}*

StimBurst *(channel,PW1,IPI1,PW2,IPI2,delay)* *(automatic changeover}*

General language considerations

Symbols (names) are used for constants, variables, procedures, functions and states. A symbol may be virtually unlimited in length. Except in the context of the destination of state transitions, symbols must be defined before they are referenced.

The layout of the language is free-form. In any place where a space may be put, any number of spaces, tabs or new lines may also be put. The recommended layout is illustrated in the example given latter.

Comments may be put into the code at any place, enclosed within braces { }.

Built-in data types in the first edition of the compiler were integer only. Floating point types will be included in a later version, although these may not be appropriate for 8-bit microcontrollers.

Program structure

An FSDL program comprises the following parts:

- Constant declarations;
- Variable declarations;
- Function and procedure declarations, which comprise
  - local constants,
  - local variables,
  - the function or procedure body;
- Global transitions definitions;

header (optional)

- State descriptions. which in turn comprise:
  - Entry actions,
  - Delayed actions,
  - Continual actions,
  - Exit actions,
  - Local transitions.

optional

body (required)

Program components

Statements are actions to be performed. They are used in the body of a function definition, and in the **entry, delay, continual** and **exit** components of a state description.

A simple statement is either an assignment , eg *x:=5;* or a call to an intrinsic os user defined procedure.

A structure statement is one of the following types:

- a **while ... do** statement,
- an **if ... then** statement,
- an **if ... then ... else** statement,
- a **for ... do** statement.

Expressions are language morsels which may be evaluated. Expressions are used on the right hand side of a assignment statements, in actual parameters to functions or procedures, and in the condition specification of transition definitions.

Expressions ,may include any of the following elements:

- immediate or symbolic constants,
- variables,
- intrinsic or user-defined functions,
- mathematical or logical operators,
- parentheses for controlling the precedence of operations; these may be nested.

Constant and variables are defined similarly to those of Pascal. The current version of the compiler allows variables of type integer only, and constants of type integer, character or string. Future compilers will extend the range of data types, will allow constant expressions to be evaluates at compile time and will allow automatic initialization of variables.

const, *HipChannel = 3;*

var, *HipAngle : integer*

var, *StepCount : integer : = 0;*

Functions are declared similarly to Pascal functions except that at present the return type is always integer. Other type of return will be added later. A function may contain local constants and variables. Local variables are dynamic, that is, they do not retain their value from one call to the next. A function may also contain local static variables, declared with the keyword **StaticVar**, which retain their values from one call to the next.

Every function must contain a function body, which is a list of one or more statements enclosed within the keywords **begin** and **end**.

A function body must include an assignment to the pseudovariable *ReturnVal*, which passes a value back to the caller. It is the responsibility of the programmer to ensure that the *ReturnVal* assignment is executed once and only once during the

execution of a function body. If it is not executed, garbage will be passed back to the caller.

Note that the *ReturnVal* assignment does not cause an immediate return from the function body: it only causes to be passed back: the program returns to the calling part only when the **end** keyword is reached.

Functions are not recursive, in other words, a function may not call itself, not may two or more functions call each other. Functions are not hierarchical, that is, a function may not enclose another function. The scope of an identifier is thus always either global to the whole controller or local to one function.

Procedures are functions which do not return the value. Apart from the absence of type declaration and of the *ReturnVal* assignment statement, procedure declarations resemble function declarations. Procedures may call functions and *vice versa*.

```
procedure  greeting;
           const   line = 1;
                   column = 5;
           begin
               GotoXY (column,line);
               write ("Hello,World");
           end;
function   smooth (a,b,c: integer) : integer;
           begin
               ReturnVal : = ((a + (2*b) + c) div 4);
           end;
function   difference (x : integer) : integer;
           StaticVar previous : integer;
           begin
               ReturnVal := (x - previous);
               previous := x;
           end;
```

Transitions between are defined with the keyword **transit**. This is followed by a list of transitions, each comprising the name of the destination state and a Boolean expression. The destination state will be jumped to as soon as the Boolean expression becomes *true*. The name of destination state may be that of a state already described, or it may be an implicit "forward reference". In this case, the state must subsequently be described.

A transition defined a state description is local to that state: its Boolean expression will be evaluated only during the tenure of that state. A transition defined before the start of the state descriptions is global, and its Boolean expression will be evaluated continually.

```
transit        EndSwing : HipAngle (left) > FlexLimit;
```

State descriptions are indicated by the keyword **state** followed by the name of the state, and are finished with the keyword **end**. Every control program must contain a description of at least one state.

A state description comprises statements lists for **entry, delay, continual** and **exit** actions, and a list of local transitions. Each of these items is optional, except that if no global transitions have been defined then every state other than *Last* must have at least one local transition.

There are predefined states which have special roles. These are named *First, Last* and *Error*. *First* is the state in which the controllers begins. *Last* is the state at which the controller ends, and the computer will return to the operating system or monitor. *Last* may contain **entry** actions but no **continual, delay** or **exit** statements or **transit** definitions. *Errors* is jumped to on run-time errors.

First must be described. *Last* and *Error* may be left undescribed and empty, the program simply terminates on a jump to *Last*. If state *Error* is empty, a run-time error causes the program to terminate.

Syntax

Syntax diagrams and grammatical rules are given in the user manual; space restrictions preclude their being given here.

Keyword of the language (which may not be used for any other purpose) are:

const var function procedure begin end group

state entry delay continual exit transit

Operators are

1 Unary operators **not BitNot** and unary negation -
2 Multiplicative operators * **div mod**
3 Additive operators + -
4 Relational operators > < >= <= = <>
5 Boolean logical operators **and nand or xor**
6 Bitwise logical operators **BitAnd BitNand BitOr BitXor**
7 Assignment operator :=

Functions intrinsic to the language include

time which gives the time since the start of the current state,

FullTime which gives the time since the start of the whole program,

DigitalIn and *AnalogIn (channel)* which read the input ports,

KeyPressed which enables the PC keyboard to be inspected.

Procedures intrinsic to the language include the stimulator output procedures *(StimOn, StimOff, etc)* already mentioned, as well as

inc () dec () ClearScreen ClearLine GoToXY () Write () NextLine Bell.

Pre-defined symbols include

First Last Error True False ReturnVal TickCount.

COMPILATION

Grammatically, FSDL is a type II context-free language with context-sensitive restraints [5]. The appropriate compilation algorithm is a push-down automation, that is, a finite state automation plus a stack [7].

The first version of the compiler was based on the "recursive descent" algorithm [12]. This compiler is difficult to maintain and extend, since the syntactic analysis (checking the *form* of the input program) is intertwined with the semantic analysis (checking the *meaning* of the input program).

For the second version of the compiler, a table-driven parsing method is being considered. Here, the grammatical productions of the language are kept separately from the compilation program. Questions of *what* the compiler does are considered separately from questions of *how* it does it. A similar distinction is one of the primary aims of FSDL. The grammar is int the form known as LL(1) [8].

F - CODE

Intermediate language

The FSDL compiler yields a sequence of numbers each of which is either a numeric constant or a "token", that is, a number representing an intrinsic operation of function. This forms an intermediate language, that is, a simple language much closer to machine code than it is to its sores code. We call it "F - code".

Intermediate languages have a number of advantageous properties: they are compact and they are portable. However, they are also comparatively slow because the "virtual machine" for which they are written has a stack-based architecture, whereas the actual hardware does not. A "user stack" must be implemented using an index register and indirect addressing.

At present, F-code is executed through an interpreter which scans the list of tokens, continually jumping to appropriate subroutines. Naturally this is slow. For improved speed we are developing second stage compiler which reads in an F-code program and translates it into the machine code of the target processor

F-code components and structure

For the most part the intrinsic operations of F-code are the same as those of the FSDL. Operations include operators as well as calls to intrinsic procedures and functions. The following are the major differences:

- each operation is identified by a number,
- parameters and operands are always pulled from the stack, and results are pushed back onto the stack,
- symbolic constants are replaced by their actual value,
- variables are replaced by their address,
- formal parameters are replaced by their stack offset,
- structured statements are replaced by their expanded equivalents using conditional and unconditional jump instructions.

Furthermore, F-code contains no information about states. All the actions associated with states and transitions are "deconvoluted" into a linear stream of operations with jump instructions inserted as appropriate.

Order of execution

An F-code controller is run synchronously. A hardware timer generates interrupts art regular intervals, normally once every 10 ms. The controller actions are performed once only for each "tick" of this timer. The advantages of this over an asynchronous or free-running system is that differentiation and integration of a signal are more readily performed.

This is the sequence of actions performed by the run-time system:

- first, it runs through all the *entry* actions fore a state,
- then it deals with all the global transition rules,
- then it deals with all the relevant local transition rules,
- then it performs all the *continual* actions pertaining to the current state, including checking whether any *delayed* actions are pending and due,
- it waits for the timer interrupt,
- then it branches to the global transition, followed again by the local transitions and then by the local continuals, and so on.

The process of dealing with transitions comprises evaluating Boolean expression of each in turn: on obtaining a *true* value, the processor performs the *exit* actions for the current state, then the *entry* actions for the new state etc as above.

These sequences are made explicit since they have certain repercussions for the designer of a controller. If a transition expression is *true* at the time of entry to a state then the state's *continual* and *delay* actions will not be performed. In contrast, the *entry* and *exit* actions will be performed. Passing fleetingly through a state in this way does not take up a time slot, since waiting for timer interrupts is done only the end of the *continual* section.

The watchdog system

A common programming error is the accidental creation of a loop which take inordinately long to completes or which never completes such as

while *x > 0* **do** *decrement (y);*

Such errors can not be found at compiled time, and normally "freeze" the machine at run time. FSDL provides a mechanism for trapping this kind of error.

The "watchdog" system relies on a counter used by both the control program and the interrupt service routine (ISR). At every interrupt the ISR increments this counter. When the control program is running correctly it will normally complete all necessary actions within the time available between successive interrupts. The program will then set the counter to zero before "marking time" waiting for the next interrupt. If the counter value ever exceeds one, this indicates that the control program has failed to complete all its actions before the next timer tick. This situation is known an "overrun".

Mild overruns are allowed to happen. They are liable to occur around state transitions when there are many *exit* and *entry* actions to do. In this case, the control program will find that the counter has gone up to (say) two or three. This value is available to the FSDL programmer in a pseudovariable called *TickCount*. It may be ignored or acted upon as the designer chooses, perhaps with the transition

transit *emergency : TickCount > 5;*

Severe overruns are detected by the ISR rather than by the control program, since they are likely to indicate that the control program is stack. Severe overruns are defined as occurring when the *TickCount* goes above 50 which is normally half a second. In this case the ISR cases the main program to abandon its current activity and jump immediately to the *entry* section of the special state *Error*.

Optimization

There is a danger that a machine running F-code will spend more of its time dealing with the stack then doing useful work. Every operation involves pulling operands from the stack and pushing results back on. At every stack access there would conventionally be some checking for overflow or underflow of available stack space, since such errors are liable to be fatal if not trapped.

There are three (and doubtless more) techniques we can employ to minimize time wasting.

First, we can eliminate redundant checking of stack limits. Stack underflow will never happen since *pull* instructions are only generated by the compiler when sufficient data is known to have been *pushed*. Furthermore, since recursion is not allowed , the maximum stack use can be computed at compiled time. It is found, roughly speaking, in the evaluation of the most convoluted expression in the most deeply nested procedure or function call. Provided that the known memory requirement is available, stack overflow will not happen.

Secondly, we can eliminate redundant pushes and pulls within operations by using the processors index register to access data beneath the top of the stack. No operations ever needs to perform more than one pointer adjustment regardless of the number of operands.

Thirdly, we can eliminate redundant pushes and pulls between operations. Frequently the result of one operation is pushed onto the stack only to be immediately pulled off again as an operand of the next operation. A quick and simple method can scan the object code and find and remove all push-pull pairs. This is known as "peep-hole optimization" [9]. Since the object code only has to be examined a small section at a time.

EXAMPLE PROGRAM

The example program "Hemi" is included in the paper to demonstrate feature of FSDL. This example program is much shorter than an equivalent Pascal program. This reduction in size is brought about not by using a compact syntax (such as in C, with its inherent problems of unreadability and side effects) but by eliminating the details of the implementation of the controller such as maintenance of flags for actions following a **delay**. The program is concise but not terse.

Also there are no structured statements, such as if ... then or while ... do. These structures are available but seldom need to be used, since branching and looping are implicit.

```
const     Extensors = 0;                          {stimulator output channel}
          Flexors = 1;
          LeftButton = 1;                         {mask for digital input}
          RightButton = 2;
          KneeAngle = 2;                  {A-to-D channel for knee goniometer}
          KneeAngleMin = 63;               {knee starting to flex (A-to-D units)}
          MaxPW = 300;                                {pulse width in μs}
          NormalIPI = 5;                  {5 * 10 ms for 20 Hz stimulation}

var       StepCount : integer;

function  ButtonPressed (mask : integer) : Boolean;
          begin
             ReturnVal := ((DigitalIn BitAnd mask) = 0);
          end;

function  HipFlexed : Boolean;
          const HipChannel = 0;                {A-to-D channel for goniometer}
                HipFlexLimit = 148;           {hip flexion angle in A-to-D units}
          begin
             ReturnVal := (AnalogIn (HipChannel) > HipFlexLimit);
          end;

procedure KneeControl;
          const KneeMoment = 1;       {A-to-D channel knee extension moment}
                proportion = 5;
                offset = 96;                {constants for proportional controller}
          var   level : integer;
          begin
             level := (AnalogIn (KneeMoment) - offset) * proportion);
             StimSetPW (Extensors, MaxPW * level);   {adjust pulse width only}
          end;

{ ---------------------------- end of header : start of states ---------------------------- }

state     first;
entry     ClearScreen;
          write ('Unilateral gait controller');
          StimOff (Extensors);
          StimOff (Flexors);
          StimStart;
          StepCount := 0;
```

```
transit     sit : true;                                    {go immediately to Sit}
end;

state       sit;
transit     RampUp : ButtonPressed (LeftButton);
            last : UpCase (KeyPressed) = 'Q';
end;

state       RampUp;                                        {from sit to stand}
entry       StimRamp (Extensors, 0, Normal1P1, MaxPW, 2);
transit     stand : time > 200
end;

state       RampDown;                                      {from stand to sit}
entry       StimRamp (Extensors, MaxPW, Normal1P1, 0, -2);
transit     sit : time > 200;
end;

state       stand;                          {stand with proportional knee control}
entry       StimOff (Flexors);
continual   KneeControl;
transit     QuadsOn : AnalogIn (KneeAngle) < KneeAngleMin;
            swing : ButtonPressed (LeftButton);
            RampDown : ButtonPressed (RightButton);
end;

state       QuadsOn        {emergency state when knee is collapsing into flexion}
entry       StimBurst (Extensors, MaxPW, 1, MaxPW, Normal1P1, 20);
transit     stand : (AnalogIn (KneeAngle) > KneeAngleMin) and (time > 50);
end;

state       swing;
entry       StimBurst (Flexors, MaxPW, 1, MaxPW, Normal1P1, 10);
delay       time > 25 : StimOff (Extensors);
exit        inc (StepCount);
transit     stand : HipFlexed;
            stand : time > 200;                {two second maximum swing time}
end;

state       last;
entry       StimStop;
            Write ('number of steps = ', StepCount);
end;
```

Example Program

By Graham Phillips, March 1990.

"HEMI" - a controller for a reciprocal gait for a hemiplegic person giving stimulation of the flexors and extensors of one limb only. By Graham Phillips, March 1990.

The user wears a floor reaction orthosis with sensors for knee extension moment and knee angle [2, 10]. These sensor data are used for control of quadriceps. Termination of swing phase is controlled by a hip goniometer.

The user presses buttons to go from sit to stand and from stand to swing etc.

This program is for illustrative purposes only: it is not a tested control program.

FURTHER DEVELOPMENTS

FSDL is still in the early stages in development. There are clearly some deficiencies which need to be addressed. First, we must implement the Boolean data type so as to be able to enforce the strong type checking which is essential for reliability.

The current restriction to integer data stems from the original objective of providing facilities only for those controllers which would be able to run on a portable stimulator with limited processing capabilities. As the use of the language spreads to more complex controllers on powerful microcomputers, it will be necessary to extend the range of data types to include real numbers.

A useful enhancement which we plan to include in the next version of the compiler is the ability to group several stimulator output channels onto one "virtual" channel. Any control action applied to the virtual channel will be expanded to apply to all the actual channel intended. This will be useful with implanted stimulators which may have several electrodes controlling the group of muscles which are always used together. Syntactically, a **group** declaration would be made available in the program header, such as

group *Vasti = VastusMed + VastusLat + VastusIntermed;*

In the first version of the compiler pulse widths are always given in μs. In a later version they me be given relative to two levels (threshold and saturation) which will be specified in μs in the program header:

MinMax *ErectorSpinae, 50, 240;*

It is frequently helpful to think of a finite state controller in terms of hierarchy. Lower limbs stimulator for example might have "higher" states *stand, step, stairclimb, sidestep* etc; each of these will contain "lower" states such as *swing, support* etc, which in turn may contain still lower levels. The syntax of FSDL can be extended to encompass this by allowing a state description to contain other state descriptions. Transitions which are *local* to the higher state become *global* to all contained states. Transition to the *Last* state at a low level causes return to the next higher level. We have no immediate plans to implement this feature in the compiler.

Acknowledgements: This work has been funded by the Milly Apthorp Charitable Trust as a stage in the development of upper limb FNS systems. The implementation of F-code on microcontrollers is being undertaken by the Medical Research Council Neurological Prostheses Unit. I am indebted to Nick Donaldson and to Tim Perkins for their balance of criticism and encouragement.

REFERENCES

1. Abbas,J.J., H.J.Chizeck, G.Borges, P.Chow, P.Lambert and M.Moynihan, (1988), A software structure for implementing multistate feedback controllers in FNS systems, Prof of the 10th IEEE Intern. Conf. in EMBS, New Orleans, 1653-4

2. Andrews, B.J., R.M.Baxendale, R.W.Barnett, G.F.Phillips, J.P.Paul and P.A. Freeman, (1987), A hybrid orthosis for paraplegics incorporating feedback control, in **Advances in External Control of Human Extremities IX**, Dubrovnik, 297-311

3. Brindley, G.S., N.deN.Donaldson and T.Perkins, (1989), A breathing implant that stimulates intercostal nerves, Electrical Stimulation of Muscle: Biol.Eng. Soc. Conf., Hexham

4. Brown, P.J., (1979), **Writing Interactive Compilers and INterpreters**, Wiley Chichester

5. Chomsky, N., (1959), On certain formal properties of grammar, Information and Control 2(2):137-167

6. Fisher, C.N. and R.J.LeBlanc, (1988), **Crafting a Compiler**, Benjamin/Cummings, Menlo Park, California

7. Hunter, R., (1985), **Compilers: Their design and construction using Pascal**, Wiley, Chichester

8. Knuth, D.E., (1971), Topdown syntax analysis, Acta Informatica, 1:79-110

9. McKeeman, W.M., (1965), Peep-hole optimization, Comm Assoc Computing Machinery 8(7):443-444

10. Phillips, G.F., B.J.Andrews, H.Chizeck and K.Barnicle, (1988), Finite state control of paraplegic gait using a hybrid FNS orthosis, Prof of the 10th IEEE Intern. Conf. in EMBS, New Orleans, 1651

11. Phillips, G.G., L.Da Zhang, R.W.Barnett, R.Mayagoita and B.J.Andrews, (1989), The Strathclyde research stimulator for surface FES, 3rd Vienna INtern. Workshop on Functional Electrostimulation, Baden, Vienna, 341

12. Wirth, N., (1976), **Algorithms + Data Structures = Programs**, Prentice Hall, Englewood Cliffs, NJ

REFERENCES

1. Abbas,J.J., H.J.Chizeck, G.Borges, P.Chow, P.Lambert and M.Moynihan, (1988), A software structure for implementing multistate feedback controllers in FNS systems, Prof of the 10th IEEE Intern. Conf. in EMBS, New Orleans, 1653-4

2. Andrews, B.J., R.M.Baxendale, R.W.Barnett, G.F.Phillips, J.P.Paul and P.A. Freeman, (1987), A hybrid orthosis for paraplegics incorporating feedback control, in **Advances in External Control of Human Extremities IX**, Dubrovnik, 297-311

3. Brindley, G.S., N.deN.Donaldson and T.Perkins, (1989), A breathing implant that stimulates intercostal nerves, Electrical Stimulation of Muscle: Biol.Eng. Soc. Conf., Hexham

4. Brown, P.J., (1979), **Writing Interactive Compilers and INterpreters**, Wiley Chichester

5. Chomsky, N., (1959), On certain formal properties of grammar, Information and Control 2(2):137-167

6. Fisher, C.N. and R.J.LeBlanc, (1988), **Crafting a Compiler**, Benjamin/Cummings, Menlo Park, California

7. Hunter, R., (1985), **Compilers: Their design and construction using Pascal**, Wiley, Chichester

8. Knuth, D.E., (1971), Topdown syntax analysis, Acta Informatica, 1:79-110

9. McKeeman, W.M., (1965), Peep-hole optimization, Comm Assoc Computing Machinery 8(7):443-444

10. Phillips, G.F., B.J.Andrews, H.Chizeck and K.Barnicle, (1988), Finite state control of paraplegic gait using a hybrid FNS orthosis, Prof of the 10th IEEE Intern. Conf. in EMBS, New Orleans, 1651

11. Phillips, G.G., L.Da Zhang, R.W.Barnett, R.Mayagoita and B.J.Andrews, (1989), The Strathclyde research stimulator for surface FES, 3rd Vienna INtern. Workshop on Functional Electrostimulation, Baden, Vienna, 341

12. Wirth, N., (1976), **Algorithms + Data Structures = Programs**, Prentice Hall, Englewood Cliffs, NJ