A Computer Square Dancer

by

John Dahl Sybalsky

Submitted in partial fulfillment

of the Requirements for the

Degree of Bachelor of Science

at the

Massachusetts Institute of Technology

May 1978

Signature of Author _____

Department of Electrical Engineering
and Computer Science
11 May 1978

Certified by Carl Hewitt

Thesis Supervisor

Accepted by _____

Chairman, Departmental Committee on Theses

A Computer Square Dancer

by

John Dahl Sybalsky

Submitted to the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of technology on 15 May 1978 in partial fulfillment of the requirements for the Degree of Bachelor of Science.

Abstract

The mechanism for defining square dance calls is investigated. The basic dancer motions that can be required are isolated, and the manner of their combination into calls is studied. Based on that research, a language has been developed for defining square dance calls in a machine-comprehensible form. In addition, an interpreter for the call-definition language has been written, which can move images of dancers on a display screen. Indications of useful future research are given.

Thesis Supervisor:

Carl Hewitt
Associate Professor of Computer
Science and Engineering

CONTENTS

FIGURES

1. Acknowledgements

First, to Carl Hewitt for agreeing to supervise this work on rather short notice, and within absurd time constraints.

To Al Vezza, for making available computer time to get the program running, and for putting up with me generally.

To Clark Baker and John DeTreville for much useful advice, and for the benefit of their experience with the problem. Clark also graciously provided the font for displaying dancers in the thesis.

Finally, to Gail for much encouragement and proofreading.

2. Introduction

2.1 Overview

Western-style square dancing is a form of folk dance popular in the United States. Since its upswing in popularity in about 1950, square dancing has evolved rapidly; frequently without much advance thought on the part of the evolvers. Recently, an effort has been made to better understand what square dancing is, so that future changes can best benefit the activity.

Two questions are central to understanding square dancing: How does a caller call, and how do dancers dance? Clark Baker [1] and others he cites have studied the question of calling, with an eye toward modelling the process by computer. The result of this is a program which simulates a caller, and produces sequences of square-dance calls which can actually be danced, and which return the dancers to their starting positions.

This thesis presents the results of an investigation of how dancers perform calls. The understanding gained from this study is embodied in a program which is capable of "dancing" calls it is given. The program moves images of dancers about on a CRT screen, in response to commands from the keyboard.

2.2 Previous Dancer-Modelling Efforts

There have been only two kinds of models used to simulate dancer action in the past. One is that used by Baker in his calling program, the starting-position/ending-position model. The other is not really a model, but consists of having a human move dancers around on a screen in time to recorded calling. The dancer motions are then

recorded in synchrony with the calling, and a replay of the two together gives a very life-like simulation. Of these two, only the former is of interest.

Part of Baker's calling program is a model for keeping track of where dancers are, so that the program can know what calls it can call, and can get people back to their partners. This model is driven by a database which contains starting and ending dancer positions for every place every call [known to the program] can be called from. Naturally, the database must contain separate entries for each different configuration of dancers.

For example, for the call *star thru* done from facing couples, the database must contain two entries. One is for the case in which the couples are normal (*i.e.*, the man is on the left, and the woman is on the right), and one is for when the couples are sashayed (man on the right, lady on the left).

The calling program does not keep track of how the dancers get to their final locations. Thus, it is impossible for such a system to model dancers doing part of a call, or to have it display dancers in the midst of a call. In short, there is no information about how the calls are actually performed, only about their net effects.

The idea of having a computer actually "dance" is not new. Starting in 1971, John DeTreville[2], a graduate student at MIT, studied the problem. He concluded that the only way to do it was to create an instant-by-instant simulation of what occurs, so that dancer interaction could be seen. This effort never got very far, primarily due to DeTreville's insistence that the project had to be instant-by-instant from the start.

2.3 The Present Approach

The intent of this effort has been to hew a middle line between the two extreme approaches. The research has been aimed at understanding how people perform calls on an abstract level. Rather than try to look at physical factors--speed, length of stride, etc.--an attempt has been made to isolate the terms used to define calls, to examine how these terms are combined to create whole call definitions, and to base a dancer model on actions needed to accomplish what the simplest terms require.

A program which incorporates the model was written, using the MDL[3] language. The program accepts commands from the terminal, and displays on a CRT screen the current positions for all dancers. Dancers may be moved individually or as groups by the user.

3. Keeping Track of Dancers

3.1 Notational and Naming Conventions

In square dancing, dancers work in groups of eight called *squares*. When a square is first formed preparatory to dancing, the dancers are grouped into four couples, each of which stands on one side of an imaginary square. Such a setup is shown in figure 1. In each couple, the man stands on the lady's left-hand side. The men are shown as the lightly-shaded figures.

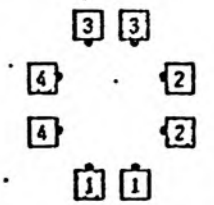

Figure 1. Static Square

The couples are numbered from 1 through 4, starting with the couple closest (and with its back to) the caller, and moving clockwise. The numbering is shown in the figure. All the dancers in the diagram are facing toward the center of the set; the dots represent noses.

For purposes of the dancing program, the dancers are superimposed on a set of axes, as shown in figure 2. The directions that dancers face are read clockwise, with 0^{o} being facing *away from* the caller.

The measure used on the dancer axes is "dancer-wide" units. Dancers are assumed

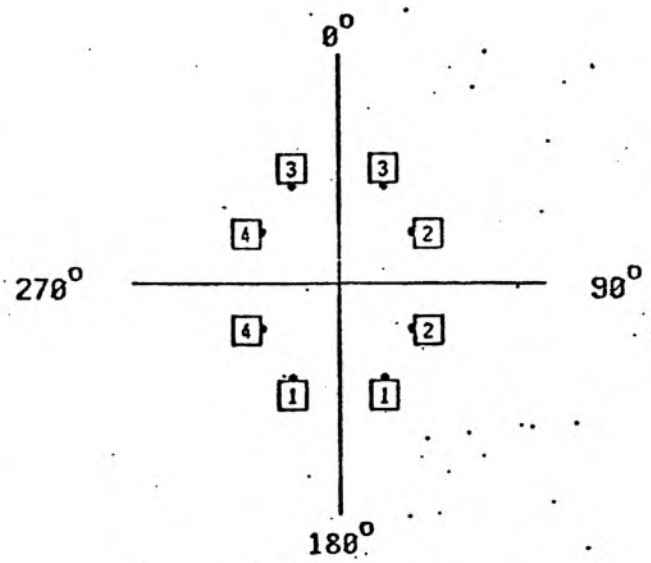

Figure 2. Dancer-position axes

to be 1 unit wide, and 1 unit deep. A dancer's position is the location of his geometric center. The man in couple number 2 in figure 1 is at (-.5 -1.5).

3.2 What a Dancer needs to Know

While dancing, a dancer keeps track of a number of things. For a computer model to succeed, it must keep much the same information. A dancer needs to know:

- Identity
- Sex
- Location
- Facing Direction
- Last turning direction, and
- Last traveling direction.

For the computer model, this information is kept in a special data type called DANCER. Figure 3 shows a sample object of type DANCER. The person it describes is the number 1 man, standing in his home spot.

```
#DANCER [ MALE              ;"Dancer's sex:  male"
          1                 ;"Couple # the dancer is in:  1"
          0.0               ;"Facing direction:  away from the caller"
          #FALSE()          ;"Is he active? No."
          0.0               ;"Last turning direction:  none"
          0.0               ;"Last travelling direction:  none"
          ![ -.5 -1.5 !]    ;"Location:  his home spot"
          0 0 0 0           ;"Filler for compatibility with FORMATIONS"
          []                ;"Slot for the original copy of the data"
        ]
```

Figure 3. Description of Number 1 Man

3.3 Position Information

One particularly interesting thing a dancer keeps track of is his position in relation to the other dancers in the square. It is this last item that tells a dancer whether or not a particular call can be done, and helps determine how it should be performed.

3.4 Describing Positional Relationships Between Dancers

Part of a dancer's knowing what to do involves being told what sorts of setups to look for. To convey this information, a shorthand of setup names has evolved through the years. Usually, the names are self-explanatory, as in the case of *two people facing*, or *a man facing a woman*. At other times, the names are derived from the name of a call which leaves dancers there, like *quarter-tag position*. Yet another type of name comes from a fanciful description of a setup, for instance a *star* or a *galaxy*.

For a dancer to know what to look for, there must be a way of expressing the dancer relationships. This method is called a *setup definition*. It tells a dancer what kinds people or smaller setups to look for, and what the relationships between these components

will be.

In the dancing program, setup definitions are implemented as a special datatype, called SETUP. An object of type SETUP, describing a man-facing-woman situation is shown in figure 4.

```
#SETUP [
        [DANCER DANCER]      ;"There are 2 dancers in the setup"
        #TESTRULE[
          []
          [<N==?             ;"They must be different sexes."
            <Sex <1 .SETUP>>
            <Sex <2 .SETUP>>>
           <INFRONT?         ;"Each .must be in front of the other"
            <1 .SETUP>
            <2 .SETUP>>
           <INFRONT?
            <2 .SETUP>
            <1 .SETUP>>
           <NEXTTO?          ;"And they must be adjacent"
            <1 .SETUP>
            <2 .SETUP>>
          ]
]
```

Figure 4. MfacingW setup definition.

There are two parts to the setup specification as shown. The first element of a SETUP is a vector, which lists the pieces from which the setup is built. In this case, the

[DANCER DANCER]

says that an *MfacingW* setup is made from two dancers.

Some setups are built of smaller groups of dancers. *Quarter-tag* position (shown in figure 5), consists of an *ocean wave* [the dancers marked with W's] between two *couples* [the dancers marked with C's], with the couples facing each other. The component vector in its

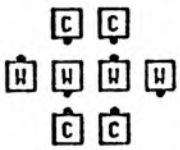

Figure 5. Quarter-tag setup

definition would be

[WAVE COUPLE COUPLE],

which says that there are three parts to the setup, a *wave* and two *couples*. In that case, of course, there would have to be definitions available for the *wave* and *couple* setups.

3.5 Test Rules

The second element of the setup definition shown in figure 4 is called the *test rule* for the setup. These are the rules that determine whether a particular group of components in fact form a given setup.

Whenever the dancing program needs to know if a group of people are arranged in a particular setup, it refers to the setup definition, and attempts to isolate the components of that setup. This may consist of doing nothing, if the desired setup is composed solely of single dancers, or it may require grouping some of the dancers into smaller setups which in turn compose the desired one. In the man-facing-woman example, nothing would be done, since *MfacingW* is made up of single dancers. On the other hand, for *quarter-tag*, the program would look for an instance of *wave* among the dancers, and would then hunt for two *couples*. If the required sub-setups cannot be found, then the dancers are not in the desired setup.

Once possible components for a setup have been isolated, they are placed into a MDL vector, where they are available for reference by the *test rules* in a consistent manner. The *test rules* for the desired setup are then evaluated. If every test succeeds, then the components really do form an instance of the desired setup. Otherwise, they do not.

The isolated components are placed into .SETUP (to which the *test rules* may refer), and are arranged in the same order as the component list for the setup-definition under test. This enables the *test rules* to refer to specific parts of the setup. When the *MfacingW* setup is being tested for, .SETUP would consist of a pair of DANCERs. If the program were looking for a *quarter-tag* setup, .SETUP would consist of three items: a *wave* and two *couples*.

A TESTRULE, the special data type given objects of that use, is made up of a series of vectors. Each vector contains zero or more MDL forms, which are the conditions to be tested for. A schematic representation of a TESTRULE is given in figure 6. The conditions have been structured to simplify backup-and-retry strategies when searching for a setup. The conditions in the first element of the TESTRULE apply only to the first component listed in the component vector for the SETUP. The conditions in the second element apply to both of the first two components, and so forth.

```
#TESTRULE [     ;"This is just a list of sets of conditions"
          [conditions for first setup component]
          [conditions for first 2 components]
          [conditions for first 3 components]
                .
                .
                .
          [conditions on all components]
]
```

Figure 6. A test rule

Returning briefly to the man-facing-woman case, look at the *test rule* in figure 4. For *MfacingW* it doesn't matter which order the dancers appear in, so there are no conditions laid down for the first person.

The second vector of conditions applies jointly to the first two members of the component vector. This is where all the *MfacingW* conditions are laid down, since there are two components to that setup. There are three conditions there:

<N==? <Sex <1 .SETUP>> <Sex <2 .SETUP>>>

is true when the dancers are of different sexes, *i.e.*, one must be male and the other female, although which is which makes no difference.

<INFRONT? <1 .SETUP> <2 .SETUP>>

and

<INFRONT? <2 .SETUP> <1 .SETUP>>

are true when each dancer is in front of the other--when they are facing each other.

<NEXTTO? <1 .SETUP> <2 .SETUP>>

will be true if the two dancers are close to each other, *i.e.*, when there is no room for a third dancer to be between them.

The *test rule* in figure 4 makes references to the first and second elements of .SETUP to get information about the dancers it must examine. To shorten definitions and enhance readability, an abbreviation was developed for "<*n* .SETUP>", which is the MDL expression for "the *n*th element of .SETUP". In the file of call and setup definitions used by the dancing program, the shorthand "$*n*" is used in place of "<*n* .SETUP>". With this notation, the definition for *MfacingW* would appear as in figure 7.

```
#SETUP [
        [DANCER DANCER]          ;"Two dancers in this setup"

        #TESTRULE [
           []                    ;"No conditions on a single dancer"
           [<N==? <Sex $1> <Sex $2>>    ;"Different sexes."
            <INFRONT? $1 $2>            ;"Facing each other"
            <INFRONT? $2 $1>
            <NEXTTO? $2 $1>             ;"Adjacent"
           ]]
]
```

Figure 7. MfacingW setup definition using $ notation

3.6 Finding Instances of a SETUP

Given two dancers, it is fairly simple to tell whether or not they are in the above setup. Given a larger group of dancers, how can one determine which of the group are, by pairs, are in the *MfacingW* setup. To solve this problem, a third piece was added to the setup definition--the *search rule*. A search rule is a short MDL program which directs the dancing program in breaking a large group of dancers down into instances of the setup at hand.

The setup-finder section of the dance program makes several things available to the *search rule*: A vector listing the entire group of dancers being examined, the setup definition being worked on, and an arbitrary person who may be used as a starting point for the search. The *search rule*, when evaluated returns either a vector containing the components of the desired setup; or a MDL FALSE, indicating that no instance of that setup containing the seed dancer could be found.

The setup-finder will present successive members of the group of dancers being examined as the "seed" dancer for the setup, until the *search rule* finds an instance of the setup. The dancers in that instance are removed from further consideration, and are

grouped into a FORMATION for later use. This search process continues until each available dancer has been used as the seed once.

There are two possibilities at this point: Either there are no dancers left, or there are some remaining. Normally, a caller expects everyone to perform the call he gives. As the setup-finder is used to isolate groups of people to do the call, having dancers left over indicates that not everyone can. This will give rise to an error in the dancing program. However, that error may be overridden by having the caller tell only "Those who can" to do a call, by setting the THOSE-WHO-CAN flag. If it is set, no unusual action is taken by the program. Only those dancers in the required setup will perform the call, however.

Figure 8 shows the *MfacingW* definition with a *search rule* added.

```
#SETUP [
        [DANCER DANCER]        ;"Components:  2 dancers"
        #TESTRULE[...]         ;"Test rules (same as before)"
        <VECTOR e1 <TRY-ALL>>          ;"Search rule"
]
```

Figure 8. MfacingW definition with search rules added

This *search rule* starts with the seed dancer ("e1"). It then invokes the TRY-ALL function. TRY-ALL looks at each of the other dancers in the group. Each of these others is paired with the original dancer, and the resulting combination is tested with the *test rules*. If a pair which meets the requirements of the *test rule* is found, the two dancers are placed into a vector, and that vector is returned to the setup-finder.

3.7 Compound Setups

It is possible for a setup to be described in terms of other, smaller setups. A setup consisting of two couples facing each other, whose definition is shown in figure 9, is a good example.

```
#SETUP [
        [COUPLE COUPLE]     ;"It is made up of two couples"

     #TESTRULE[<INFRONT? $1 $2>  ;"Each couple must be in front"
               <INFRONT? $2 $1>  ;"of the other."
              ]

     <PROG                  ;"The search rule.
       ( CPLS CPL1 )        ;"Local variables for the program"
       <SET CPLS                    ;"Break the group into couples."
            <FIND-SETUP COUPLE .SETUP>>
       <SET CPL1 <1 .CPLS>>         ;"Pick a couple to start with"
       <MAPF      ;"Look at the other couples one at a time."
         <>
         #FUNCTION((CPL)            ;"Take this couple, and"
           <SET SETUP               ;"Pair it with the starting couple."
                [ .CPL1 .CPL ]>
           <AND                                 ;"If they meet the"
             <VERIFY-SETUP
                .TEST-RULES                     ;"requirements for the setup,"
                .SETUP>                         ;"then"
              <MAPLEAVE .SETUP>>)               ;"Tell the program about them."
       <REST .CPLS>>>
]
```

Figure 9. Definition of 2-couples-facing setup

This setup is composed of two couples, each of which is in front of the other. The

search rule is a bit more complicated than before, because it now has to break the larger group of dancers down into couples, and then manipulate those couples. The search rule here first couples up all the dancers in the group. It then chooses a single couple, and pairs it up successively with each of the other couples. The first time it finds a pairing which satisfies the test rule, it reports that pairing back to the dancing program.

3.8 Representing a Dancer's Position Knowledge

A dancer thinks of a setup as a concrete grouping of dancers. It is possible for the dancers in a given setup to act as though the group were indivisible, moving from place to place while maintaining the positional relationship the setup requires. For these reasons, it is possible to regard a particular instance of a setup (called a *formation*) in many of the same ways that one regards a dancer. As a result, there is a special data type assigned to the *formation*, called FORMATION. It resembles the DANCER data type closely--it is of the same size, and corresponding information is kept in the same spots. This lets FORMATIONs of dancers be manipulated as easily as single DANCERs can.

There are relatively few differences betweeen a FORMATION and a DANCER, most of them following directly from the fact that a FORMATION has several dancers in it.

- A FORMATION has no sex.
- It does, though, point to the description of the setup it instantiates.
- A FORMATION is not part of any original couple.
- A FORMATION does not necessarily have a facing direction, since the dancers may be facing different ways.
- To compensate, a FORMATION probably does have an orientation. If every dancer in a FORMATION is facing in either a certain direction or

exactly the opposite way, then the orientation of the formation is the "absolute value" of the facing direction; facing direction modulo 180 degrees.

- A FORMATION has a unique number assigned to it, to simplify the task of telling whether two formations are really the same.
- A FORMATION has members. These are the dancers or formations which the setup definition calls for.

A FORMATION which is an instance of the two-people-facing SETUP is shown in figure 10.

```
#FORMATION [ 2PF                    ;"Setup this instantiates"
             #                      ;"Unique identifying number"
             0.0                    ;"Facing direction:  not applicable"
             #FALSE()               ;"unused"
             0.0                    ;"Last turning direction: none."
             0.0                    ;"Last travelling direction:  none."
             ![ -.5 0.0 !]          ;"Center point of formation"
             [ #DANCER [ MALE 1 ... ] ;"List of components"
               #DANCER [ FEMALE 3 ... ]]
             [ #DANCER [ MALE 1 ... ] ;"List of dancers in"
              .#DANCER [ FEMALE 3 ... ]] ;"the formation"
             0.0                    ;"handedness:  not applicable"
             0.0                    ;"Orientation: along 0-180 axis"
             #FORMATION [ 2PF ...] ;"Original copy of this data."
           ]
```

Figure 10. FORMATION of 2-couples-facing

This FORMATION describes the number 1 man and the number 3 woman, who are facing each other. Since they are facing different ways, the formation cannot have a facing direction, but does have an orientation: along the 0^{o}-180^{o} line.

During calls with more than one person moving, it is often necessary to keep track of the state of the acting dancers as of the beginning of the call. This is done by creating a copy of the FORMATION containing those dancers, which in turn makes copies of the DANCERS themselves. Pointers to the original copies are kept in the DANCER and FORMATION structures, so that after calculating some movement based on the copy's data, the original copy can be changed to actually make the move.

4. The Primitive Motions of Square Dancing

4.1 The Motions

There are three basic motions which can be demanded of a dancer. These are turning in place, moving in a straight line, and moving in an arc. Anything else can be expressed as linear combinations of these acts.

Now, a caller doesn't say to a dancer "Turn in place!", he says things like "Turn a quarter turn to your left", or "Turn to face the center of the square." The primitive motions, then, need modification. Each has modifiers appropriate to its characteristics.

- Turning in place can be

(1) to face a given direction, or
(2) to turn a given amount to the left or right.

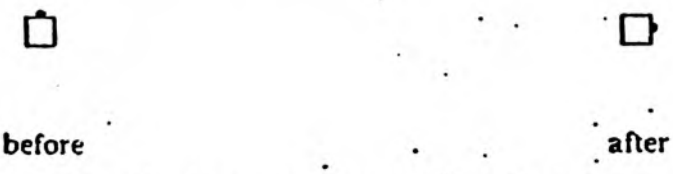

Figure 11. A turn in place of 90° to the right

- Moving in a straight line is moving

(1) to a given spot, or
(2) a given distance in a given direction (*e.g.*,forward one step)

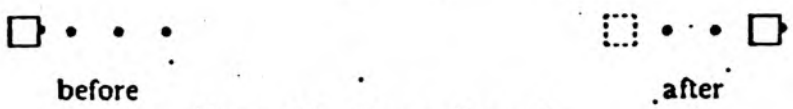

Figure 12. Moving forward

- Curving motion must be

(1) around a given center, and
(2) through a given size of arc, and
(3) either clockwise or counter-clockwise about the center point.

Figure 13. Moving through a 90° arc

4.2 Describing Primitive Motions in the Program

A special data type was created to describe the primitive motions, and to simplify the process of moving dancers within the program. It is called PRIM. There are two kinds of PRIMs. One describes a single motion, *e.g.*, "Turn in place 90° to the left." The other describes a linear conbination of several motions, *e.g.*, "Walk forward one step while turning 90° to your left."

The first sort of PRIM consists of a MDL function (which actually moves the dancer) and a series of modifiers for the requested action. To move a dancer, one APPLYs the PRIM to the DANCER(s) to be moved, much as one applies a function to its arguments. The modifiers tell how far the dancer is to be moved, his ending location, or whatever is appropriate. Figure 14 shows a PRIM which would turn a dancer in place 90° to his right.

```
#PRIM[ ,TURN  90.0 ]
```

Figure 14. 90° spot turn request

The "TURN" function used in it turns a dancer in place the specified amount. A positive turning amount turns the dancer to the right, a negative amount to the left.

Sometimes, one cannot specify particular directions in advance. The primitive motion in figure 15

```
#PRIM[ ,DMOVETO    <Loc <OTHER>> ]
```

Figure 15. PRIM with unevaluated modifier

causes a dancer to be moved to the place where some other person currently is. The modifiers in a PRIM are evaluated each time it is applied to a DANCER. This permits the resulting movement to vary depending on who is being moved.

Certain calls such as *curlique* demand linear motion and body turns at the same time. This combination of two primitive movements at a single time is represented by a PRIM which consists of other PRIMs. When the compound PRIM is applied to a DANCER, it is as though all of its constituents were applied to that same DANCER at the

same time. Thus, figure 16

```
#PRIM[  #PRIM[ ,DMOVE  ,FORWRD  1.0 ]
          #PRIM[ ,TURN  90.0 ]
        ]
```

Figure 16. Compound PRIM

moves a dancer forward one step while having him turn one quarter right. The result of this move can be seen in figure 17.

before after

Figure 17. Compound dancer motion

5. Defining Calls

The primitive motions described above are combined to tell dancers what to do. A dancer must also know what setups of dancers a call can be done from, and how the call is done from each setup. These pieces of information are brought together into a single unit, the definition of a call.

5.1 The Call-Definition Format

A call definition consists of pairings of setups and motion-descriptions, which say in effect, "From here you do this", and "From there you do that." The simplest case of this is a call which can be done by a dancer standing alone. For example, the call *quarter left* has the dancer turn 90° to his left. In the call-definition language developed during this project, *quarter left* would be represented as seen in figure 18.

```
#CALL [ DANCER  ;"Setup to do the call from:  a lone dancer"
        #DEFN [                               ;"Thing to do:"
                *PRIM[ ,TURN  -90.0 ]         ;"Turn 90° leftward"
              ]
      ]
```

Figure 18. Call definition for Quarter Left

This definition consists of the two parts: the legal starting setup, and a description of how a dancer is to execute the call from there. Where he can start is simple--anywhere. What is to be done is also simple--turn 90° leftward, in place.

5.2 Parallelism in Call Definitions

When several people are involved in the execution of a call, there arises the question of who does what. The simplest example of this is the call *pass thru*. A *pass thru* is done from a setup of two people facing each other, and consists of each dancer's moving forward to take the other's original place. Neither dancer turns in the process. Its definition is given in figure 19.

This also consists of two parts, but the what-to-do section is a bit more involved. Each dancer is to perform the action specified by the motion primitive. The DMOVETO function moves a dancer to a specific spot. Because the dancers are supposed to move at the same time, and since the program really only moves one dancer at a time, one can't just specify "The other person's spot" as the place to move to. Figure 20 shows what would happen if that were done. What is really needed is a way of saying, "Move to the other person's original spot", *i.e.*, the place where the other person was at the beginning of the call. The colon in front of the "<Loc <OTHER>>" is read as "the original".

There are calls which have different dancers doing totally different actions

```
#CALL [
        2PF             ;"Setup to start from--two people facing"
        #DEFN [ #PRIM [ .DMOVETO                ;"Move to"
                        :<Loc <OTHER>>]         ;"the other guy's"
                ]                               ;"original spot."
]
```

Figure 19. Call definition for star thru

---

the starting setup

step 1: person 1 moves to person 2's spot

step 2: person 2 moves to person 1's spot

end of call

Figure 20. Pass thru performed with wrong definition

---

simultaneously. Take for example *star thru*, which can be done from a man facing a woman, and which has each dancer take the other's original position, but thas the man turn right while doing so, and has the woman turn left while moving. We now need to specify who does which part of the call, separating the men and the women. This is done with a construct that behaves much like the MDL COND does. The specification:

```
<WHILE (cond1 act1a act1b act1c...)
       (cond2 act2a ...)
       ...>
```

has the people who meet the conditions cond1 perform acts 1a, 1b, etc. while those who meet cond2 perform acts 2a, 2b, and so on. The number of actions for each group to perform need not be equal. Thus, a *star thru* would appear as:

```
#CALL [
        MfacingW        ;" Done from a man facing a woman"

    #DEFN [ <WHILE  ;"People are to work in parallel:"
            (<==?                       ;"If you're male,"
              <Sex .DANCER>
              MALE>
             #PRIM[#PRIM[,DMOVETO...] ;"Move forward"
                  #PRIM[,TURN 90.0]]) ;"While turning right."
            (<==?                       ;"If you're female,"
              <Sex .DANCER>
              FEMALE>
             #PRIM[#PRIM[,DMOVETO...] ;"Move forward"
                  #PRIM[,TURN -90.0]) ;"while turning left."
          >]]
```

Figure 21. Call definition of star thru

5.3 Use of Calls to Define Other Calls

Frequiently, a call is defined as a combination of other calls. The call *pass right* is done from two facing dancers, and consists of a *pass thru* followed by a *quarter right*. Its definition looks like this:

```
#CALL [ 2PF
      #DEFN[ PASS-THRU
             QUARTER-RIGHT]
  ]
```

This tells each dancer to perform the call *pass thru*, and then to do the call *quarter right*.

6. Having the entire square work at once

Many groups of dancers can be performing the same call simultaneously. None of these groups interact while doing a call--each works independently. The dancing program simulates this by having each group do it in succession. Still, the program must know how to isolate these *working groups*, as they are called, and the call and setup definitions must take cognizance of the fact that there are lots of dancers around.

The working-group isolation problem is easily solved. Indeed, the setup finding portion of the dancer was designed specifically for this purpose. When given a group of dancers and a setup, the setup-finder isolates all instances of the given setup within that group of people. When a call is being performed, these groups are made into FORMATIONs, and each is made to execute the call. In this way, the independent *working groups* are isolated, and each performs the call, seemingly in parallel. Figure 22 shows via shading how eight dancers would be divided into working groups to perform a *right and left thru*. Dancers with similar shadings would work together.

Figure 22. Working groups for Right and left thru

The presence of extraneous dancers creates an interesting problem. In theory, it is possible to do certain calls from a setup of two people facing each other, no matter how far apart the people may be. The only condition imposed is that there be nobody between them. If this condition is not imbedded in setup definitions, the following situation arises:

Consider the eight-chain setup shown in figure 23, and the call *right and left thru.*

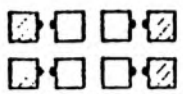

Figure 23. Incorrect working group for right and left thru

If the setup definition for two-couples-facing doesn't test for interference, the program will attempt to have the couples composed of the shaded dancers working together. It will isolate those dancers, and attempt to shoehorn the remaining four into the setup. It of course fails miserably.

The solution to this problem is the INTERFERENCE? predicate. Given a group of dancers, INTERFERENCE? finds the outline of the rectangle containing those dancers, and then looks for other people who impinge on that rectangle. If there is nobody in there, there are no interfering dancers. When this test is included in the definition of the two-couples-facing setup, the program has each adjacent pair of facing couples work with each other, which is the correct thing to do.

7. Proposed Extension to a Full Simulation

The dancing program was designed with John DeTreville's idea of instantaneous simulation in mind. As such, it is structured so that it is easy to change one facet of the design without affecting other parts. There are several areas in which further study and development are desirable.

7.1 The Right-Shoulder Rule

Whenever two dancers are about to collide, convention has it that they pass right shoulders. Simulating this requires instant-by-instant awareness by one dancer of where other dancers are. It further requires some sort of process for making adjustments in a dancers path, while the dancer is in motion.

7.2 Fudging

Because of the imprecise nature of square dancing, one physical arrangement of dancers can be viewed different ways. Usually, the different views require slight adjustments in dancer position to make things work out properly. This adjustment process is called "Fudging." There are two kinds of fudges. These are squared-up set fudges, and initial and terminal fudges.

Squared-up-set fudges are needed to move from a squared-up set into things like eight-chain formation. Dancers are normally expected to return to their exact starting spots on calls like *pass thru* or *square thru*. From a squared-up set however, they are expected to

remain close together rather than returning to their original spots.

Initial and terminal fudges are those adjustments needed to make one call flow into the next properly. These tend to occur when the first call ends in a setup which can be viewed two ways, and the second call takes an unusual view. For instance, from 3/4 tag, one can have the leads quarter right, and all diamond circulate. After the quarter right, the dancers must adjust to form the diamonds needed for the diamond circulate.

7.3 Fractions of Calls

It is possible for a caller to ask dancers to do some portion of a call, *e.g.*, "Swing the fractions 4/5". To do this, dancers must know where the call is divided into parts. Since the definition language developed here contains all the information about how to do a call, it should be fairly easy to insert "part-of-call" dividers into the definitions. The dancing program could then be made to understand fractions of calls in a straight-forward manner.

7.4 Timing

Not all movements require the same amount of time to dance. If a simulation is to be in pseudo-real time, the dancer model must have an understanding of timing built into it. This is a complex mixture of convention and knowing how long a human stride is. Some movements are defined to take specific lengths of time, as in *grand square*, where moving along each side of the square takes four beats. On the other hand, there are situations like the outsides' pat of *spin the windmill*, where the timing is determined by how far the dancers have to walk, and how big an average pace is. There will need to be a way

of saying that an action is to take a pre-determined length of time, within a call definition. For the rest, the dancer model will have to be able to take path lengths and decide how long it should allow.

7.5 Terminal Conditions

Some motions are described this way: "Move forward until you form the end of an ocean wave." This kind of ending condition means that the dancer must look ahead, extrapolating the future positions of himself and others, to tell when he should stop.

7.6 As Couples Movements

Some calls, such as *wheel and deal* are done with groups of dancers acting as fixed units--in this case, as couples. To do these, the dancing program has to be able to move a group of dancers, while translating the group's motion back into motion of the individual dancers. This feature is symptomatic of other kinds of compound-dancer motion, like *tandem* moves, where one dancer is behind another, and the two act as a unit.

7.7 Natural Language

Finally, there is the question of getting a program to dance directly from calling, or at any rate, from calling transcribed verbatim into machine-readable form. Clark Baker has begun a study of this, and has made significant headway using simply a thesaurus of terms and a simple matching program. Whether that is sufficient or not remains to be seen.

Appendix I - The Call Definition Language

1. Specifying Setups

1.1 Setup Definition Syntax

A setup definition (of type SETUP) consists of four parts. They are

(1) A listing of the components of the setup; either single dancers or other, previously-defined SETUPs.

Examples:

[DANCER] marks a setup made up of a lone dancer,

[DANCER DANCER] for a setup composed to 2 dancers,

[COUPLE COUPLE] a setup made of two couples, and

[WAVE COUPLE COUPLE] for a setup made of a wave and 2 couples.

(2) A set of testing rules. Given a MDL VECTOR of possible SETUP components, this set of rules evaluates to T if they meet the criteria, and to *FALSE otherwise.

Examples:

*TESTRULE [[<==? <Fdir $1> ,TO-CALLER>]]

for a single dancer who must be facing the caller.

*TESTRULE [[] [<==? <Fdir $1> <Fdir $2>>]]

for two dancers (or any formations with facing directions), with both of them facing

the same way.

*TESTRULE [[] [<S/O? $1 $2> <==? <Fdir $1> <Fdir $2>>]]

for two dancers who must be facing the same way, and who must be side-by-side. This is in fact the definition of a couple.

(3) A set of seaching rules. This is a MDL FORM which is evaluated by the setup finder. It is passed an arbitrary dancer as the first element of .ACTIVES, and is expected to return either a VECTOR of dancers who are in the specified SETUP, or *FALSE if none can be found.

Example:

<VECTOR el <TRY-ALL>

for a brute-force search for a two-component setup.

(4) A description of how certain fields of a resulting FORMATION are to be filled. A FORMATION has certain attributes, such as handedness, which are not applicable to a dancer. These attributes are not always consistently defined. Hence, each SETUP must specify the rules for determining them.

Example:

The formation-filler for a couple:

```
[ .NAME          ;"The name of the setup (will be COUPLE)"
  <>             ;"Unique id #.  Filled in by program."
  <Fdir $1>      ;"Facing direction:  Either will do."
  <>             ;"Active?  Not applicable to FORMATIONS"
  .0             ;"Last turning direction:  none"
  .0             ;"Last traveling direction:  none"
  <>             ;"Location.  Filled in by program"
  <>             ;"Members.  Filled in by program"
  <>             ;"Dancers in the setup. Filled by program"
  <Fdir $1>      ;"Orientation:  same as facing direction here"
  .0             ;"Handedness:  not applicable"
  []             ;"Original copy of data.  Filled by program"
]
```

Notice that the orientation of the formation will be the same as its facing direction, since both dancers will be facing the same way.

The formation-filler for *two couples facing*

```
[ .NAME          ;"Name of the setup:  2CPLF"
  <>             ;"Unique id.  Fileed by program"
  <>             ;"Facing direction:  none"
  <>             ;"Active? not applicable."
  .0             ;"Last turning direction:  none"
  .0             ;"Last traveling direction:  none"
  <>             ;"Location.  Filled by program."
  <>             ;"Members.  Filled by program."
  <>             ;"Dancers in setup.  Filled by program."
  <Fdir $1>      ;"Orientation:  Facing dir of either dancer"
  .0             ;"Handedness: none"
  []             ;"Original data:  filled by program"
]
```

1.2 Information Available to Test and Search Rules

A number of data are made available by the dancing program for the use of *search* and *test rules*. They are information about the possible components, information about the entire group of dancers being examined, the seed dancer, and data about the setup to be looked for. A complete listing of what is available and where it may be found follows.

(1) A list of the entire group under test (actually, all the people who have not yet been placed into an instance of the setup) can be found in .PEOPLE.

(2) .SETUP contains a vector of potential components for the setup under test. These candidates are arranged in the same order as the component specification in the setup definition, to permit rules to refer to specific portions of a setup.

(3) The seed dancer is kept as the first element of .ACTIVES.

(4) The *test rules* for the setup at hand are kept in .TEST-RULES for reference. The *search rules* can find them there, for use in checking parts of a setup as it goes. An example of this use is shown in the definition for the two-couples-facing setup.

In addition, the source listing in Appendix III shows many utility MDL functions which may be used to determine dancer sex, location, and some of the more common relationships, such as one dancer being in front of another, or two dancers being side-by-side.

2. Defining Calls

2.1 Basic Syntax

A call definition (data type CALL) consists of two parts, a setup specification telling where the call can be done from, and a description (data type DEFN) of how the call is done from there. Conceptually, there could be several such pairs, but in the current implementation, only one is allowed for. The setup specification is either an object of type SETUP, or an ATOM which is the name of some pre-defined SETUP.

The call description, the DEFN, is just a MDL VECTOR of things to be done, either MDL FORMs to be evaluated, PRIMs to be applied to every dancer, or ATOMs which are the names of calls to be performed. The three types can be freely intermixed.

There are two selection constructs, named WHILE and SELECT. SELECT is used when only certain people are to do something, as in "Do a partner trade and just the men roll". This would be specified using the sequence,

```
PARTNER-TRADE          ;"Call name--everyone does it"
<SELECT <MALE? .DANCER> ROLL>      ;"Just the men roll"
```

WHILE is a more powerful construct, allowing different people to be acting independently at the same time. The syntax is

```
<WHILE  (cond1 act act act act...)
        (cond2 act2 act2 act2...)
          . . .
        (condn actn actn actn...)>
```

When viewed from the standpoint of a single dancer, this is an *n*-way branch: If

the dancer meets the criteria set by cond1, he performs whatever acts are specified by the rest of that list, and stops. If he does not meet cond1, he then examines cond2; If he meets cond2, he performs act2a, act2b, *etc.* If he fails cond2, he tries cond3, and so on down to cond*n*. If he doesn't meet that, he does nothing.

Once a dancer has met one set of criteria, he is excluded from consideration for any further tests in that WHILE clause. The effect of this is to make the clause mean, "You do this, while he does that, and they do the other thing". Nobody is doing more than one of those at once.

2.2 Primitive Motion Functions Available

There are several MDL functions which implement the primitive motions, and whic may be invoked in PRIMs:

- DMOVETO moves a dancer to a specific location, moving in a straight line. It requires a uvector containing the location to move to as an x-y pair. Examples of locations may be seen in the dancer definitions in Appendix III.

Example:

```
*PRIM[ ,DMOVETO ![ 0.0 0.0 !] ]
```

would move a dancer to location (0,0), which is the center of the set.

```
*PRIM[ ,DMOVETO ![ -.5 -1.5 !] ]
```

would move a dancer to the number-1 man's home position.

- DMOVE moves a dancer in a straight line some distance in some direction relative to

his current facing direction.

Example:

*PRIM[,DMOVE ,FORWRD 2.0]

moves a dancer forward through a distance of 2.0,

*PRIM[,DMOVE 90.0 1.0]

has a dancer slide to his right (without turning) one spot, and

*PRIM[,DMOVE 180.0 1.0]

has a dancer take a step backward.

- TURN causes a dancer to turn in place. A positive turning amount turns the dancer to the right, while a negative amount causes a leftward turn.

Example:

*PRIM[,TURN 90.0]

causes a dancer to turn 90^{o} to the right, and

*PRIM[,TURN -45.0]

causes the dancer to turn 45^{o} to the left.

- FACE is a function which turns a dancer to face a given direction, in such a way that he turns the smallest amount possible.

Example:

Assume for these that the dancer is facing 0^{o} at the start of each move.

*PRIM[,FACE 90.0]

would turn the dancer 90^{o} to the right.

*PRIM[,FACE 270.0]

would turn him 90° to the left, and

*PRIM[,FACE 180.0]

would turn him about. The direction of the turn in this case is not defined.

Appendix II - Call Definitions Used by the Dancing Program

```
;"-------------------------------------------------------------"
;"              --Call and Setup Definition Dictionary--       "
;"-------------------------------------------------------------"

;"      Definition syntax:

ATOM            ; name of the setup
        #SETUP[ vector of member setups
                testrule
                search rule
                formation-filling instructions
              ]


ATOM            ; name of the call
        #CALL [  #SETUP [...]    ;a setup it can be done from
                 #DEFN[...]      ;and the corresponding definition
                 #SETUP [...]    ;another possible setup
                 #DEFN [...]     ;and its corresponding rules
                   .
                   .
                   .
               ]


COUPLE  ;"A couple, i.e., 2 dancers adjacent, side-by-side, facing the same way"
        #SETUP[ [DANCER DANCER] #TESTRULE[[] [<NEXTTO? $1 $2> <S/O? $1 $2>
                                              <=? <Fdir $1> <Fdir $2>>]]
                                <COND (<LINE8? e1> <TRY <MYHALF e1>>)
                                      (<LINE4? e1>
                                        <TRY <MYHALF e1>>)
                                      (<COL8? e1> <ERROR PUNTEROO!-ERRORS>)
                                      (ELSE [ e1 <TRY-ALL>])>
                        [.NAME
                         <>
                         <Fdir $1>
                         <>
                         .0
                         .0
                         <>
                         <>
                         <>
                         <Fdir $1>
                         .0
                         []]]


2CPLF   ;"Two couples facing"
        #SETUP[ [COUPLE COUPLE]
                #TESTRULE[[] [<INFRONT? $1 $2> <INFRONT? $2 $1>
                                <NOT <INTERFERENCE? .SETUP <OTHERS .SETUP .DANCERS>>>]]
                <PROG (GROUPS SETUP CPLS CPL1)
```

```
                    <SET CPLS <FIND-SETUP COUPLE .PEOPLE>>
                    <SET CPL1 <1 .CPLS>>
                    <MAPF <> #FUNCTION((CPL)
                            <SET SETUP [.CPL1 .CPL]>
                            <AND <VERIFY-SETUP  .TEST-RULES .SETUP>
                                 <MAPLEAVE .SETUP>>)
                            <REST .CPLS>>>
                [.NAME
                 <>
                 <>
                 <>
                 .0
                 .0
                 <>
                 <>
                 <>
                 <Fdir $1>
                 .0
                 []] ]

2PF     ;"Two dancers facing each other."
        #SETUP[ [DANCER DANCER]
                #TESTRULE[[] [<INFRONT? $1 $2> <INFRONT? $2 $1>
                            <NOT <INTERFERENCE? .SETUP <OTHERS .SETUP .DANCERS>>>]]
                <VECTOR e1 <TRY-ALL>>
                [<> <> <> <> .0 .0 <> <> <> <Fdir $1> .0 []]]

MfacingM        ;"Like two pepie facing, but sexes must be different."
        #SETUP[ [DANCER DANCER]
                #TESTRULE[[] [<N=? <Sex $1> <Sex $2>> <INFRONT? $1 $2>
                            <INFRONT? $2 $1>
                            <NOT <INTERFERENCE? .SETUP <OTHERS .SETUP .DANCERS>>>]]
                <VECTOR e1 <TRY-ALL>>   ;"Brute force search technique"
                [<> <> <> <> .0 .0 <> <> <> <Fdir $1> .0 []]]


PASS-THRU       ;"Pass thru -- from two people facing, each moves to the other's spot."
                #CALL[ 2PF
                       #DEFN [ #PRIM[ ,DMOVETO :<Loc <OTHER>>]
                             ]
                     ]


TRADE           ;"Trade--from 2 people side by side, each moves forward in a
                  semi-circle to the other's spot."
        #CALL[  #SETUP [ [DANCER DANCER]
                         #TESTRULE[[] [<S/O? $1 $2>]]
                         <VECTOR e1 <TRY-ALL>>
                         [.NAME <> <> <> .0 .0 <DCENTER .SETUP> <> <> <Fdir $1> .0 []]]
           #DEFN  [  #PRIM[,DCIRC 1.0 180.0 :<DCENTER .SETUP>] ]]


RLT             ;"Right and left thru"
        #CALL[  2CPLF  ;"Done from two couples facing."
                #DEFN[RIGHT-PULL-BY COURTESY-TURN]]


RIGHT-PULL-BY   ;"Right-pull-by"
```

```
        #CALL[  2PF
                #DEFN[ #PRIM[,DMOVETO :<Loc <OTHER>>] ]]


COURTESY-TURN   ;"Courtesy turn for use in RLT--belle forward, beau backs up"
        #CALL[  COUPLE
                #DEFN [<WHILE (:<BEAU? .DANCER>
                                #PRIM[,DCIRC -1.0 180.0 :<DCENTER .SETUP>])
                              (:<BELLE? .DANCER>
                                #PRIM[,DCIRC 1.0 180.0 :<DCENTER .SETUP>])>]]


STAR-THRU       ;"Star thru. like pass thru, but man quarters right, woman left"
        #CALL[  MfacingW        ;"Must be done from a man facing a woman"
                #DEFN[<WHILE (<==? <Sex .DANCER> MALE>
                                #PRIM[ #PRIM[,DMOVE
                                             .0
                                             <+ .5 </ :<DIST .DANCER <OTHER>> 2>>]
                                       #PRIM[,TURN 90.0]])
                             (<==? <Sex .DANCER> FEMALE>
                                #PRIM[ #PRIM[,DMOVE .0
                                             <+ .5 </ :<DIST .DANCER <OTHER>> 2>>]
                                       #PRIM[,TURN -90.0]])>]]


CURLIQUE        ;"Curlique--man and woman facing, join right hands man goes around,
                  turning 1/4 right, woman goes under turning 3/4 right to end on
                  each other's spots."
        #CALL[  MfacingW
                #DEFN[ <WHILE (<MAN?>
                                #PRIM [ #PRIM[,DMOVE
                                             .0
                                             <+ .5 </ :<DIST .DANCER <OTHER>> 2>>]
                                       #PRIM[,TURN 90.0]])
                              (<WOMAN?>
                               #PRIM[   #PRIM[ ,DMOVE
                                             .0
                                             <+ .5 </ :<DIST .DANCER <OTHER>> 2>>]
                                       #PRIM[ ,TURN -270.0]])>]]
```

References

1. Baker, Clark M. A Computer Square Dance Caller, MIT Bachelor's Thesis. 1976

2. DeTreville, John D. Private Communication, 1977

3. Pfister, Greg and Galley, Stu MDL Primer and Manual, MIT Laboratory of Computer Science, Programming Technology Division Document, 1977

4. Schiffman, Robert R. MDL Interactive Graphics System documentation. Available online as the file .INFO.; MIGS ORDER on the MIT-DMS system