COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 943052192

Parallel Program Behavior — Specification and Abstraction Using BDL

Jerry C. Yan

Technical Report: CSL-86-298

August 1986

This research was supported by NASA under contract NCA2-109.

Parallel Program Behavior - Specification & Abstraction Using BDL

by

Jerry C Yan

Technical Report No. 86-298

August 1986

Computer Systems Laboratory

Department of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

This paper describes the syntax, semantics, and usage of BDL - a Behavior Description Language for concurrent programs. BDL program models can be used to describe and abstract the behavior of real programs formulated in various computation paradigms (such as CSP, remote procedures, data-flow, actors etc.). BDL models are constructed from abstract computing entities known as "players". The model can behave as closely as possible to the actual program in terms of message passing, player creation and cpu usage. Although behavior abstraction using BDL only involves identifying the "redundant part" of the computation and replacing them with simple **"NO-OP"** statements, proper application of this technique remains difficult and requires a thorough understanding of how the program is architectured. Simulating BDL models is much more economical than instruction level emulation while program behavior is realistically preserved.

Key Words and Phrases: Parallel Processing, Program Behavior Abstraction, Modelling, Concurrent Languages.

Acknowledgements

I wish to acknowledge Professor Stephen Lundstrom for directing this research and debugging the manuscript of this paper. I also acknowledge Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, TX 78759, for granting me access to the simulation package CSIM during the development of "Axe" — the simulator for BDL. I would also like to thank Harold Brown and many others who have helped me developed these ideas during the three years I worked at Heuristic Programming Project, Knowledge System Laboratory, 701 Welch Road, Palo Alto CA 94305. I acknowledge Tin-Fook Ngai, Fung-Fung Lee, Brian Boesch, Victor Mak and other colleagues at Computer System Laboratory who have contributed their insights and comments all along.

Table of Contents

1. Introduction

Many difficult questions remain to be addressed in the field of parallel computing. For example:

> "Consider two distributed/parallel formulations of the same problem, which one will execute faster on a given multiprocessor?"
>
> "Consider a loosely-coupled multiprocessor[1], how important is the speed of the communication links (relative to other factors such as cache sizes or memory sizes) for programs with a lot of communication?"
>
> "Which routing algorithm works best for a grid-connected machine executing data-flow computations?'

Most of the approaches proposed to solve these problems involve building models to represent the computation (and sometimes, the machine as well). In many cases, graph models were used:

i) Each node represents a task[2] and the edges represent *precedence relationships* between tasks [Mohan 84];

ii) Each node represents a *process* and edges join processes which communicate with one another [Bokhari 8 1];

iii) Each node represents a *functional-unit* and the edges represent *paths* of flow for data tokens [Catto 81];

iv) Each node represents a *procedure block,* the edges represent *entering* (calling) and *exiting* (returning from) the procedure.

v) A program graph consisting of *task-nodes* and a machine graph made up of *site-nodes* represents the execution environment. The *weight* on an edge between a *task-node* and a *site-node* indicates the *affinity* of particular tasks to sites [Stone 77];

Other efforts involve queuing models [Chow 79], Bayesian analysis [Stankovic 81] and applying the solution of problems in other domains if an analogy can be drawn [Hanan 70, Lawler 63]. These models enable analytical methods to be applied (e.g. [Shen 85]) and simulations (of the model) to be carried out (e.g. simulated annealing [Steele 85]). Solutions have, in fact, emerged with some claims of "optimality" for a subset of these problems. However, none of these efforts are completely satisfactory. Their results cannot be directly applied to real programs on real machines because these approaches are based on program models that <u>do not adeauatelv represent the actual behavior of computations</u>:

[1]A loosely-coupled multiprocessor consists of a number of processing elements (called sites) connected together via some communication links. Each site possesses a processor and some memory. There are no globally shared memory modules.

[2]In this context, a task can be regarded simply as a "code body".

i) *Dynamic creationltermination of computing entities* - When using graph models, nodes and edges associated with them may be created (and destroyed) during execution since processes3 can be dynamically created/destroyed. Iterative solutions that assume static graph topologies are therefore rendered useless.

ii) *Data-dependent program behavior* - Only a very small number of programs do not contain "branches". In a sequential program, these "branches" determine how the "flow of control" changes in respond to different input data sets. In the realm of parallel processing, activities such as message passing and process creation may be affected. This is particularly true for symbolic computations which represent some of the most compute-intensive and dynamic programs in the world [Nii 86]. Unfortunately, a model that "mimicks" these characteristics exactly probably involves writing the program in another programming language! The dependencies of program behavior on data characteristics have not been properly captured in any of the models proposed.

Although abstract models do aid the evaluation of new computer structures and operating system algorithms, detailed studies on distributed problem formulation, resource management and software architecture still depend on the study of real applications. Unfortunately, the development of complete language and compiler tools together with run-time environments is impractical for short term experimental use. Compounded with the fast pace of changes in concurrent programming languages, programming environment and hardware architectures, research efforts in the past were forced to depend on these abstract models. This paper proposes an approach that allows detailed description of program behavior yet at the same time, permitting potentially fast turn-around simulation.

The BDL model of concurrent programming is stated explicitly in section 2, followed by its syntax and semantics. The "player computation paradigm" is then compared with other existing paradigms in section 4. Section 5 describes how BDL can be used for program behavior description and abstraction. A few examples of BDL models is presented next. The paper then concludes with a summary.

2. The BDL Model for Concurrent Programming

Behavior Description Language (or BDL) defines a set of constructs that describes the behavior of a parallel computation. Like other programming languages, BDL enables (but also constrains) a programmer to express a computation within a specific computation paradigm. This concurrency model is detailed in the following sub-sections.

3or any abstract computing agents defined by the programming paradigm

2.1 Overview

A parallel computation is defined as a collection of computing agents known as players[4]. There are no shared data structures. Players can be dynamically created. Players may execute concurrently with one another. They communicate via message passing.

2.2 A Plaver

A player's attributes is described in terms of its *internal states.* An internal state may contain either a numerical value or a <mail-address>[5] of some player (called *acquaintance).* Each player is an instantiation of a specific "player type" - which defines a player's internal states and how it responds to messages.

2.3 A Message

Players interact via message passing. A <message> consists of two major parts[6]:

i) a header indicating the <message-type>; and

ii) the contents which consists of a variable number of "values". These values are either internal states of the sender player or constants.

2.4 Message Passing: Semantics

Messages are sent and received asynchronously. In other words, the sender player does not have to suspend computation to wait until its message is received. Each player has an implied (infinite) message buffer. Messages are processed on a first-come-first-serve basis. A player has the option to wait for the first arrival of a message of a specific type. A player may also query whether messages of a specific type are already in its message buffer.

2.5 Behavior Specification

A player is *activated* when a message is received. Each player selects the appropriate handling procedure according to the message type. These handling procedures (called *"behavior") are* defined as part of the player type specification,

[4]The name "players" was chosen not only to avoid saying "BDL objects" every time but also because of its similarity to the actor programming paradigm.

[5]A <mail-address> may be viewed as a "pointer to" or the "name of' a player. For anyone to communicate with a player, its <mail-address> must be known. A global <mail-address> may be refered to by any player.

[6]Only the two fields that can be manipulated explicitely by the programmer are listed here.

2.6 Computation

Simple computations can be carried out in BDL[7]. In response to messages received and values associated with them, a player may:

i) create other players

ii) modify its internal states

iii) send messages; or

iv) check and wait for messages of a particular type.

3. BDL Syntax and Semantics

The syntax and semantics of the language is informally introduced here so that it can be used to illustrate the process of program behavior description/abstraction in the section afterwards. The BNF notations of BDL grammar are presented in the Appendix. BDL follows a LISP-like syntax*. A BDL description consists of two types of statements:

i) ***structure specification*** - defines the internal states of a player type and its behavior in response to different messages;

ii) ***player declaration*** - instantiates a "global" player?

3.1 The Dining Philosophers - An Example

The "dining philosophers" problem Peterson 83] should be familiar to readers with a distributed computing background. This problem is commonly used to illustrate dead-lock situations. The situation involves a few (say four) "philosophers" sitting around a (virtual) table. A fork is placed between each philosopher. Each philosopher must grab the fork on both sides before he can start eating. After the philosopher finishes, he releases both forks. Figure 1 on the next page illustrates how a "simple philosopher" may be specified as a player:

[7]In fact, BDL is "not interested" in the exact nature of the user computation since it is designed for behavior abstraction.

[8]c.f. "flavors" in [Weinreb 81]

[9]This notion of a *globally known* player is necessary for two purposes:

i) Initialization - For a computation to take place, at least one player must be declared. It may then proceed to create and set up other players to carry out the actual computation.

ii) "System facilities" - Players with special abilities (such as I/O servers or error handlers) may only reside in certain sites with the appropriate hardware component or system software but need to be assessed by all parts of the system.

```
(DefPlayer Philospher                          Structure specification
    (init start-eat   ack)                     , message names
    (Left-fork  Right-fork)                    acquaintances
    0                                          ; no numerical states
    (init                                      ; initialization
        (record Left-fork Right-fork))         ; get neighbors' addresses
    (start-eat                                 ; execution begins
        (post Left-fork pick self)             ; try to get left fork
        (wait ack)                             ; wait for acknowledgment
        (post Right-fork pick self)            ; try the other fork
        (wait ack)                             ; wait for acknowledgment
        (hold 23)                              ; he eats!
        (post Left_fork release)               ; releases left fork
        (post Right_fork release)))            ; releases right fork

(Global Joe Philosopher)                       ; Player declaraton
```

Figure 1. BDL Specification of a "philosopher" at the dining table

Joe, a philosopher, has *two acquaintances* ("Left-fork" and "Right_fork"). When the INIT message is received (at initialization time), Joe records which forks are positioned next to him. When given the **START-EAT** message, Joe attempts to grab the left fork by sending it a **PICK** message (which a Fork will understand). On receiving the acknowledgement, he proceeds to get the other fork. If successful, he will begin eating. Eating takes 23 time units (at least for Joe!). Afterwards, he releases both forks and waits for the next message.

3.2 Structure Specification

The structure specification statement consists of four parts:

- <header> - consists of the keyword **DefPlayer** and an identifier indicating the player type.
- <message list> - indicates the message types that it understands.
- <internal states> - acquaintances and numeric states are declared separately to facilitate run-time type checking and statistic gathering purposes which will be explained in a forth-coming paper.
- <behavior list> - the last portion of the specification consists of a list of clauses which specify how the player responds to different message types. Each clause begins with the message type followed by executable statements.

Three types of executable statements are responsible for message manipulation, resource utilization and user computation. "Meta" statements which implement special control structures such as *loops* or *conditional branches* are also included.

3.3 Message Manipulation

Message passing is the only means by which players interact with one another. The basic operations offered include message sending and receiving, value transmission and queries on buffer contents:

- **(post** <receiver> <message-type> $<arg_1>$ $<arg_2>$ . ..) - for message sending. The <receiver> may be any acquaintance or **self -** i.e. the sender player itself. There is no limit on the number of arguments to be passed. They can be used to transmit values necessary for computation or <mail-addresses> to *introduce* players to one another. For example **(post** Right_fork pick **self)** is used to send the (acquaintance) *"right fork"* a **PICK** message. The keyword **self** is passed with the message to indicate from whom this request comes from.
- **(wait** <message-type>) **-** blocks the caller (player) until a message of the specified type has arrived. When a philosopher executes **(wait** ack), it "goes *to sleep"* until an **ACK** message arrives (from the fork).
- **(arrived?** <message-type>) **-** returns a (numerical) value indicating the number of messages found with the specified <message-type>. This statement allows the message buffer to be inspected before processing. By suitably combining **wait** and **arrived?,** messages buf - fered can be processed in any way the programmer so desires.
- **(record** $<state_1>$ $<state_2>$ . ..) - store the arguments associated with the last message received into the corresponding states. If the receiver executes record with fewer arguments than the sender does with send, these extra values will be dropped. On the contrary, if record is called with too many arguments, the extra states will be filled with zeros.
- **self -** a keyword indicating the (<mail-address> of) the player currently executing.
- **sender -** a keyword indicating the (<mail-address> of) the sender of the last message received.
- ***exception handling*** - Assume message <m> of type $<m_t>$ is received by player <P> of type $<P_t>$:

  If $<m_t>$ is not declared in the message list of $<P_t>$, <m> is dropped.

  If $<m_t>$ is declared in the message list of $<P_t>$ but its behavior is not specified, <m> remains in the message buffer until the execution of a "wait" statement: **(wait** $<m_t>$).

3.4 User Computation

These statements always return a value. This value may be:

i) stored in an internal state of an player;

ii) transmitted to other players using a message; or

iii) used to select conditional branches.

The forms currently implemented includes:

- **(setq** <state> <value>) - the basic assignment statement follows the LISP convention. It returns <value>.
- (***<comparator>*** <value$_1$> <value$_2$>) - this statement returns a " 1" if the comparison is a success and "0" otherwise. "=" and "≠" may be used to compare numbers as well as two <mail-addresses> whereas "<", ">", "≤", and "≥" only defined for numerical comparisons only.
- ***(<math-operator>*** <value$_1$> . ..) - the current BDL language includes only the following mathematical operators "+", "-", "*", "/", "1+"(increment), "1-" (decrement).

3.5 Resource Utilization

BDL is designed to be executed in a simulation environment (at least for now). There are two executable statements which affect how resources are utilized during the computation:

- **(make** <state> <player_type> [<directive>]) - creates a player - which "occupies memory space" at the site in which it is placed. This is the only means by which a player may be created dynamically. The <mail-address> of the creat<u>ed</u> player is stored in the specifed <state> of the creat<u>or</u> player. The third parameter, <directive>, is optional and implementation dependent. It directs the operating system to place the creat<u>ed</u> player at certain site. In the current implementation, allowed directives include:
  - * LOCAL - at the same site as the creator;
  - * REMOTE - at any <u>other</u> site;
  - * RANDOM - randomly at any site.
  - * <site_address> - at a specific site[10]
- **(run** <no-of-time-units>) - requests of use the processor of the site in which the player resides for a certain time. Although the player may request a certain amount of processing, the actual response time still depends on how crowded the site is and what local scheduling policy the local operating system employs. It should also be noted that these "time units" are meaningful only when discussed in conjunction with the characteristics of the hardware (such as communication link bandwidth, context-switch overhead etc.)

3.6 Meta-Statements

Four simple constructs are provided:

[10]In the current implementation, a <site address> is simply an integer identifying the actual site.

- **(if** <condition> <statement$_1$> <statement$_2$>) - If the <condition> evaluates to non-zero, <statement$_1$ > is executed. Otherwise, <statement$_2$> will be executed.
- **(repeat** <no_of_times> <statement$_1$> . ..) - a simple loop construct.
- **(branch** (<p$_1$> <statement$_{p1}$> . ..) (<p$_2$> <statement$_{p2}$> . ..> . ..> - the branch statement contains a list of clauses, each of which has a defined probability of occuring[11]. For example:

```
(branch (40 (...) (...) . ..)
        (50 (...) (...) . ..)
        (10 ()))
```

  states that the first branch is taken with a probability of 40%, the next one 50%, and there is a 10% probability that nothing will happen. It is the programmer's responsibility to ensure that all probabilities (pi) add up to 100%[12].
- **(progn** <statement$_1$> . ..) - groups a number of statements to be parsed as a single statement.

3.7 Player Instantiation

Finally, a player may be declared "at compile time" using the following statement:

- **(global** <mail-address> <player-type> <site-address>) - instantiates an instance of a particular player type - assigning to at a global <mail-address> at the same time.

4. Comparing BDL with other Concurrent Paradigms

BDL was designed to build models for parallel computations. This section demonstrates that the BDL paradigm can in fact represent computations expressed in a number of proposed concurrent paradigms. The differences between BDL and these established paradigms are also described. It must be emphasized that the issue here is not one of *efficiency* but *expressibility*. In other words, BDL implementations may not be *efficient* but it is certainly capable of expressing a lot of other parallel processing models.

4.1 Parallel "Code-bodies"

One simple way to "parallelize" a sequential program is to introduce constructs which explicitly initiate and terminate parallel execution into an existing language. The constructs proposed

[11] In the current implementation, these probabilities cannot be changed during run-time.

[12] The current BDL compiler [Yan 86] makes sure that these probabilities do add up to 100%, otherwise, an error message is generated.

include “fork/join” [Conway 63, Dennis 66], "parBegin/parEnd" [Dijkstra 68], "ForAll/DoAll" etc. BDL differs in two aspects:

i) players are more than code-bodies because they possess “states”; and

ii) there are no implied shared data-structures in BDL.

If BDL were used to represent such parallelism, it could be achieved as follows:

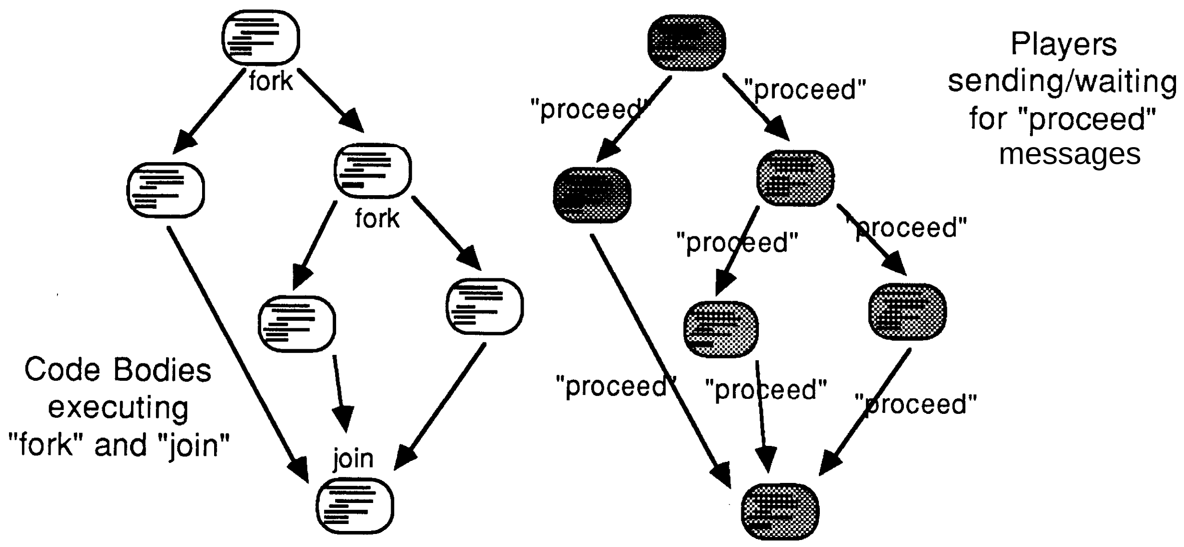
fork
fork
join
Code Bodies executing "fork" and "join"
"proceed"
"proceed"
"proceed"
"proceed"
"proceed"
"proceed"
"proceed"
Players sending/waiting for "proceed" messages

Figure 2a. “code bodies” vs. Players

i) represent code-blocks and data-structures as players;

ii) use messages to synchronize code-bodies (figure 2a);

iii) use messages to access and update data-structures (figure 2b)

```
(Def Player Simple_Data
    (read wriie)
    0                                   ; no acquaintances
    (content)
    (read (reply content))              ;"read"s
    (write (record content)))           ; “write?
```

Figure 2b. “Data” implemented as Players

4.2 Concurrent Processes

Concurrent Pascal [Hansen 75] and CSP [Hoare 78] were among the first programming languages that introduced the notion of “processes” explicitly as “code-bodies plus state”. These processes do not share any common variables. Processes in Concurrent Pascal communicate via two abstract data-types - classes and *monitors* [Hoare 74]. CSP processes, on the other hand, communicate via synchronous message passing. In both cases, the number of processes is *statically* declared (i.e. fixed at compile time). Players can, in fact, be modeled by sequential processes executing the following loop:

(1) receive any message

(2) select message handler depending on message type

(3) process message

(4) go back to (1).

However, CSP and Concurrent Pascal processes only perform a subset of operations permissible in BDL. They do not allow:

i) dynamic process creation;

ii) asynchronous message sending/receiving; and

iii) message type screening;

Constructs for synchronous message sending can be defined in terms of the **send/wait** primitives :

syn_send = (progn (post...) (wait ack[13]))

syn_recv = (progn (wait . ..) (reply ack))

The processing of implementing monitors (and *signaling)* as player (and messages) is left to the reader as an exercise!

4.3 Functional-units and Data-tokens

In data-flow [Davis 82] and systolic [Kung 82] paradigms, the program text defines an "abstract machine" that is made up of functional-units connected in some topology. A functional-unit possesses one or more input ports and (usually) one output-port. When all necessary data-tokens arrive at the input-ports, the functional-unit is activated. The result(s) is(are) then passed out of the output-port(s). Dynamic creation of functional units is necessary to handle certain programming constructs such as recursion. Players may be programmed to behave as functional-units and messages can be used to transmit data-tokens. Pure data-flow paradigms need to use "feed-back" connections to implement "memory" (or state) for functional-units.

[13]Since everybody is doing synchronous message sending, there cannot be more than one outstanding message. Therefore there is no need to check for message id in the acknowledgment.

4.4 Remote Procedures

Remote procedures [Nelson 81] simply involve invoking a procedure located at a "remote" (i.e. non-local) site. The passing of arguments and returning of values can be implemented as message passing. Procedure calls can be easily implemented as follows:

- a "procedure player" receives the "call" message with the appropriate arguments;
- player computes result;
- result is replied to the caller by a "return" message.

Since players can be dynamically created, they can comfortably model the dynamic nature of procedure calls and returns[14].

4.5 "Futures" and "Streams"

The concepts of *"futures"* [Baker 77] and *"streams"* [Weng 75] stem from two completely opposite ideas: *eager* and *lazy* evaluations respectively. A <future> can be thought of as a mail box (or data-structure) to which a computation will deliver a value. This computation was started as a result of the creation of the <future>. The <future> is in fact a *promise* that a value will arrive *"some time in the future"*. A <stream> on the other hand, represents a *producer* from whom a value can be obtained on demand. Implementation of these concepts using players is outlined as follows:

<future> creation - creates a player of the appropriate type and starts computation by sending it the appropriate values; player will deliver result as soon as it is available

requesting value from <future> - use the **arrived?** primitive to check if the value has returned, if not call **wait;**

<stream> creation - creates a player of the appropriate type; computation does not begin until a"value_needed' message is received

requesting value from <stream> - send the <stream> player a "value_needed' message and wait for the reply

4.6 Actors

Players can be thought of as a subset of Actor [Agha 85] types. The Actor programming paradigm may be summarized as follows:

i) Actors (c.f. players) are defined in terms of their *behavior*. "Behavior" in actor terminology is equivalent to "player type"[15].

ii) Actors interact via *communications* (c.f. <messages>).

[14]Players can even implement co-routine calls [Conway 63] with suitable arrangement of message passing

[15]i.e. state plus handling procedures

iii) Actors can only send communications to actors that it knows about (i.e. *acquaintances)* or actors it created.

iv) Actors may be *unserialized* or serialized. Unserialized actors do not change their behavior (c.f. a player with no state). They can arbitrarily duplicate themselves to handle incoming communications. Serialized actors (*serializers*) can change their "behavior". They handle communications in a first-come-first-serve manner[16].

v) Actors respond to communications just like players. They can
- create other actors
- make simple decisions
- send more communications
- become different actors

In fact, BDL was designed to express a subset of the actor paradigm. This subset was chosen as a mid-ground between implementation efficiency and expressibility:
- a large number of concurrent computations can be expressed while
- a compiler/simulator for players can be built quickly!

5. Program Behavior Description/Abstraction

"Program behavior" may be described in terms of abstract computing agents/operations or the utilization of actual computing resources. Either way, it must involve "observable actions" - i.e. requests posed to the machine by the program such as storage and processing demands. A good program model *behaves* similar to the program itself[17] - e.g. the model poses similar requests to the multiprocessor when the real data set is fed into the program model.

5.1 Sequential Programs

In sequential computing, the process of computing follows a somewhat "simple cycle". Assuming a typical von-Neuman architecture?

(1) instructions and data are fetched from some storage device and supplied to the central processing unit (CPU)

[16]The behavior of *serializers* can be modified in a more versatile manner than that allowed for players. Not only may the internal states be modified - an actor can change its into another actor-type with a completely different structure and behavior!

[17]Similarly, a good machine model produces the same responses as a real one when the same requests are made to the real machine.

[18]a processor making references to a memory module

(2) the CPU processes the input data and produces output data
(3) This new data is transfered back to the storage device
(4) The next instruction to be fetched is determined depending on the result obtained in step (2).
(5) Goto step (1).

Sequential program behavior can therefore be described in terms of the basic activities of "processing" and "data reference":

i) **Access locality** is associated with the "reference pattern" of the program. A program with high locality make references frequently to data/code that reside in close proximity.
ii) A sequential program may be **CPU or I/O bound** depending on whether it spends most of its time occupying the CPU or waiting for the response of the (slower) storage device doing data transfer.
iii) Any non-trivial program consists of branches. They are either used to implement control structures (e.g. loops) or to perform data-dependent operations (e.g. conditional branches). **Branch probabilities** may or may not be dependent on different input data sets.

A good sequential program model resembles the real program in terms of reference loccality, cpu/io utilization as well as branch probabilities.

5.2 Concurrent Programs

Similar considerations may be applied to parallel computing in BDL. The basic computing cycle in BDL involves:

(1) Players are created.
(2) Messages are sent from one player to another.
(3) In response to messages received, a player may "compute", send more messages, block to wait for other messages, or create other players
(4) Replies may be sent back to the sender after the message is processed

BDL provides the necessary constructs to describe parallel program behavior in terms of the basic operations: "player creation", "communication" and "processing".

i) **Degree of parallelism -** The amount of parallelism available in a particular formulation depends on the total number of players present. This in turn depends on how players are created during the computation. Although the degree of parallelism is limited by the total number of players declared, the communication and synchronization requirements remain the key factors that govern the maximal available speed up for a given program.

ii) **Communication amount/pattern/protocol -** To obtain speed-up by exploiting concurrent hardware, a single computation is broken into a number of parts[19]. These parts must communicate to facilitate cooperation between them. The amount of communication, size of messages, frequency of communication between any pair of players, and the syn - chronization protocol for delivering messages characterizes the computation.

iii) **Computation amount/pattern -** Since players that reside in the same site will compete for one CPU, the amount of computation each player requires is therefore important.

iv) **Data-dependency -** The three "observables" in BDL computing are "player creation", "computation" and "message delivery". The amount of these activities may vary for different input data-sets for the program.

v) **Parallelism "grain size" -** The trade-off between potential parallelism and communication overhead varies with the "grain size" of a parallel program. The more modules one breaks an application into, the more potential parallelism is expected. However, more communication is also expected between these modules

In summary, BDL provides the capability to describe the three basic operations in a parallel program: player creation, message passing and "computing". The next section will demonstrate how the "run" statement can be used to build realistic program models.

6. Building Program Models with BDL - Three Examples

6.1 Example 1: The N-body Problem

The N-body problem may be defined as follows:

> N heavy bodies are initially suspended in space at some initial coordinates. They are subsequently released simultaneously. They interact via gravitational forces and move along some trajectories. Plot these trajectories.

A possible[20] distributed formulation involves creating n players, each representing one "body". The structure of a "body player" is shown "schematically" in figure 3 on the next page. The corn - putation begins by having a "release" message sent to each player[21]. Each player responds by sending its coordinates to the other (n-l) players. Whenever a (new) coordinate is received, the

[19]Multiprogramming on multiprocessors is a related but separate issue because the programs of different users are independent.

[20]This asynchronous formulation is only one of the many possible. Each "body" does not have to wait for all the other (n-l) new coordinates to arrive before computing its new coordinates. For iterative solutions to this class of differential equations, the accuracy of the solution is acceptable if the iteration step size is small enough.

[21]presumably from some "god" player sitting out there watching these bodies

internal state of the receiver is updated. Its coordinates and velocity after a certain time At is calculated. This new coordinate is recorded locally and then transmitted back to the sender.

```
   (DefPlayer Body                                     ; Structure specification
      (init release update)                            ; message names
      (b1 b2 ... bn-1)                                 ; acquaintances
      (my-coord my-velocity crd1 crd2 . . . crdn-1)    ; numerical states[22]
      (init                                            ; initialization
         (record b1 b2 . . . bn-1                      ; get neighbors' addresses
                 crd1 crd2 . . . crdn-1))              ; & initial coordinates
      (release                                         ; begin execution
         (post b1 update my_coord)                     ; tell everybody where
         ...                                           ;   I am and wait for
         (post bn-1 update my-coord))                  ; their responses
      (update                                          ; new coordinates arrived!
         (if (= sender b1) (record crd1)               ; find the sender and
         ...                                           ; update its
         (if (= sender bn-1) (record crdn-1)           ; coordinates
☞        (setq my-coord (find-new-ones,))              ; compute my coord.
☞        (setq  my-velocity  (find-new-ones,))         ; and new velocity
         (post output device[23] update my-coord)      ; send it to the output_device
         (reply update my-coord)))                     ; and reply the sender
```

Figure 3. Actual Program Text of a "Body"

Two important observations are made here are:

a) The most time consuming lines in the program text are the ones which compute the new coordinates and velocity.

b) The time spent computing here is <u>indenendent</u> of the actual values of the coordinates of all the other players.

If the lines in question were replaced with a NO-OP statement that "simply consumes time" - (run <some_time>[24]), program behavior does not change. In other words, the number, sequence and pattern of messages delivered and the demand posed on the processor by each player remains identical to that of the actual program. This program "model" is shown in figure 4. As far as

[22]Although coordinates and velocities are actually vectors (as opposed to scalars), they are declared as single state variables (e.g. in stead of vx, v_y, v_z) in order to simplify presentation.

[23]The output device is formulated as a "globally known" player. Its definition is not presented here.

[24]The actual time spent can be calculated from the actual machine cycles required by the instructions or from measurements of actual program execution.

simulation is concerned, emulating the actual instructions that solve the differential equations is much more expensive than simply "advancing the clock" using the "run" statement.

```
(DefPlayer Body
    (init release update)
    (bl b2 ... bn-1)
    0                                                ; no numerical states
    (init                                            ; initialization
        (record b1 b2 . . . bn-1))                   ; record neighbors' addresses
    (release                                         ; begin execution
        (post b1 update)                             ; tell everybody where
        ...                                          ;    I am and wait for
        (post bn-1 update))                          ,    their responses
    (update                                          ; new coordinates arrived!
        (run 2)                                      ; time spent recording coord.s
        (run 40)                                     ; time spent computing my coord.s
        (run 50)                                     ,    and new velocity
        (post output-device update)                  ; send coord.s to o/d
        (reply update my_coord)))                    ; and reply the sender
```

Figure 4. Program Model for a "Body"

```
(DefPlayer File
    (init read write reserve . ..)
    (owner writer)
    (content)                                                ; file content
    (init (record owner))                                    ; initialization
    (read (reply data content))                              ; anyone can read
    (reserve                                                 ; writer needs reservation
        (if writer (reply NIL)                               ; already reserved
                (progn (setq writer sender) (reply T))))     ; reservation granted
    (write                                                   , "write" request
        (if (= writer sender)                                ; is file reserved?
            (progn (record content) (reply T))               ; yes...
            (reply NIL)))...)                                ; no!
```

Figure 5. A "File Player"

6.2 Files - An Example of Data-dependent Message Sending

Although the process of replacing parts of the program with NO-OPS may seems to be very straight-forward, that is, in fact, not true. The distinction between computations that "matter" and corn - putations that do not is very difficult to make. Consider the following example shown in figure 5 where a file is implemented as a player. Although a straight-forward option is to replace the line which actually updates the content of the file with a **NO-OP,** an alternative is illustrated in figure 6.

This model is constructed based on two observations for a particular computer system over a long period of time:

i) when a "reserve" message was sent, it was granted 70% of the time; and

ii) only 5% of user processes send "write"s without making the reservation first.

This model performs "statically correct". However, the message pattern that it generates may be completely different from that of a real program because a "write" request is not honored by "reservation" but by mere chance! It may still be sufficient if, for example, it is only used to drive experiments in routing algorithms on a particular connection topology. In general, the data-dependent portion of the computation cannot be simply "abstracted away"!

```
(DefPlayer File
    (init read write reserve . ..)
    (owner writer)
    0                                   ; no content needed
    (init (run 1))                      ; initialization
    (read (reply data ))                ; anyone can read
    (reserve       (run 1)              ; writer needs reservation
        (branch (30 (reply NIL))        ; already reserved
                (70 (reply T)))         ; reservation granted
    (write         (run 1)              ; writer needs reservation
        (branch (5 (reply NIL))         ; write without reservation!
                (95 (reply T)))...)     ; "write" successful
```

Figure 6. Model for a "File Player"

6.3 Servers - An Example of Data-Dependent Plaver Creation

Consider another example in which the computation is organized as a collection of players with "special skills" serving different request types. When "simple" requests are presented to a server (player), they are processed immediately. If a request turns out to be "too difficult" for the receiver to handle[25], it may be forced to create an "expert" to help deal with the situation. Figure 7 il - lustrates such a "server" player. The computation which determines whether a player is needed to be created or not may be very time consuming but, it CANNOT be easily abstracted away. Although a probabilistic approach is always possible, the researcher must examine the overall structure of the computation and before making his/her decision.

[25]For example, an inspection of the data and request type reveals that this request can be better served by servers of another kind or that an error handler needs to be created to handle erroneous requests.

```
(DefPlayer Server
    (.. a-request a-reply...)
    (... expert_1 client_1 . ..)
    (... result . ..)
    ...
    (a-request
        (record client_1 req-type the-request)
☞          (if (expert_needed? req-type the-request)
            (progn (make expert_1 An-Expert) (post expert_1 req-type the-request))
            (progn (setq result (compute-result ) ) (reply done result))))
    ...
    (a-reply (record result client_1) (post client_1 done result))
    ...)
```

Figure 7. A "Server" Player Processing Requests

6.4 Summarv

1. Building program models in BDL involves the following steps:
   a. study and understand the actual program text;
   b. specify (or rewrite) the program using BDL syntax;
   c. select the computation that does not affect program behavior:
      - either abstract it away using the "run" statement; or
      - reformulate it using probabilistic branches;
   d. simulate[26] the model and compare its behavior with the actual program:
      - find out the limitations of the model
      - make modificatons if necessary
2. The data-dependent behavior of programs remains the biggest challenge for model builders. Although the use of probabilistic branches may solve some simple cases, it is, in general, very

[26]The "Axe" simulation facilities [Yan 86] (built on top of CSIM [Schwetman 86]) was specifically designed to model the execution of concurrent programs on multiprocessors using discrete-time simulation. An integrated environment is provided to facilitate specification of software (using BDL) and hardware models, simulation and automatic data collection. The current version of "Axe" models a specific class of multiprocessors known as *ensemble architectures* [Lutz 84, Seitz 82]. This class consists of a collection of homogeneous *autonomous* processing elements (or sites) - each of which is connected to its nearest neighbors in a regular fashion (e.g. the Cosmic Cube [Su 85]). A collection of pre-defined abstract machines is provided in "Axe". Their hardware parameters, connection topology and routing algorithms may be tailored further by the user.

difficult model the variation of communications, player creation, and usage of computing resources in response to changes in input data-sets.

3. It is up to the researcher to decide, recognize and remember what level of abstraction his/her model was constructed for. A model should not be pushed beyond its limitations.

7. Conclusions

BDL expresses parallel programs in terms of abstract computing agents known as “players”. Players send messages to one another, create players dynamically and “compute” - expanding cpu time at the sites in which they reside. This paradigm encompasses a lot of proposed models for concurrent computing. It offers the potential for constructing accurate program models which are also efficient for simulation.

Building program models using BDL is not as straight-forward as it seems. It involves replacing part of the actual computaton with a NO-OP statement “(run <time-interval>)“. Program models may be constructed at different “levels of abstraction”. Some may be statistically correct while others may emulate an exact sequence of events over a range of input data. Although replacement of the program text with NO-OPS is the key to building good program models27 and facilitate its efficient simulation, proper application of this technique still demands a thorough understanding of the architecture of the computation.

[27] which behave exactly like the program in terms of message passing, player creation and cpu usage

References

[Agha 85] Gul A Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", PhD Thesis, Massachusetts Institute of Technology, AI Lab., TR *844.*

[Baker 77] Henry G. Baker, Jr. and Carl Hewitt, "The Incremental Garbage Collection of Processes", Massachusetts Institute of Technology, AI Lab., AI Working Paper 149, July, 1977.

[Bokhari 8 1] S H Bokhari, "On the Mapping Problem", *IEEE Transactions on Computers,* March, 1981

[Catto 81] A.J. Catto and J.R. Gurd, "Resource Mangement in Dataflow", *Proc. ACM Symp. on Functional Programming Languages and Computer Architecture,* 1981.

[Conway 63] M Conway, "A Multiprocessing System Design", *Proceedings of the AFIPS Fall Joint Computer Conference, 1963.*

[Chow 79] Y. C. Chow, W. H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System", *IEEE Transactions on Computers,* May, 1979.

[Dennis 66] J B Dennis and E C Van Horn, "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM,* March 1966.

[Davis 82] A.L. Davis and R.M. Keller, "Data flow program graphs", *IEEE Computer,* February, 1982.

[Dijkstra 68] E W Dijkstra, "Go To Statement Considered Harmful", *Communications of the ACM,* September 1965.

[Hanan 70] M Hanan and J M Kurtzberg, "Force-vector Placement Techniques," International Business Machines Cooperation, IBM Report RC 2843, April 1970.

[Hansen 75] P Brinch Hansen, "The Programming Language Concurrent Pascal", *IEEE Transactions on Software Engineering,* June 1975.

[Hoare 74] C A R Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM,* October, 1974.

[Hoare 78] C A R Hoare, "Communicating Sequential Processes", *Communications of the ACM,* August, 1978.

[Kung 82] H T Kung, "Why Systolic Architectures?", *IEEE Computer Magazine,* January, 1982.

[Lawler 63] E L Lawler, "The Quadratic Assignment Problem", *Management Science,* September, 1963

[Mohan 84] J. Mohan, "Performance of Parallel Programs: Model and Analysis", Ph.D. dissertation, Carnegie-Melon University, July 1984.

[Nelson 8 1] B J Nelson, "Remote Procedure Call", Xerox Palo Alto Research Center, Palo Alto, California, Tech. Report CSL-8 l-9

[Nii 86] H Penny Nii, "Blackboard Systems Part Two: Blackboard Application Systems", *AI Magazine,* August 86.

[Peterson 83] J and A Silbershatz, *Operating System Concepts*, Addison-Wesley Publishing Company, 1983.

[Schwetman 86] Herb Schwetman, "CSIM: A C-Based, Process-Oriented Simulation Language", Microelectronics and Computer Technology Corporation, *Proceedings for Winter Simulation Conference* '86, Washington DC.

[Seitz] Charles L Seitz, "Ensemble Architectures for VLSI - A Survey and Taxonomy", *1982 Conference on Advanced Research in VLSI,* Massachusetts Institute of Technology.

[Shen 85] C Shen and W Tsai, " A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimix Criterion", *IEEE Transactions on Computers,* March 1985.

[Stankovic 8 1] John Stankovic, "The Analysis of a Decentralized Control Algorithm for Job Scheduling Utilizing Bayesian Decision Theory", *IEEE Proceedings of the 1981 International Conference of Parallel Processing,* August 198 1.

[Steele 85] Craig Steele, "Placement of Communicating Processes on Multiprocessor Networks", MS Thesis, Department of Computer Science, California Institute of Technology, Technical Report 5 184:TR:85

[Stone 77] Harold Stone, " Multiprocessor Scheduling with the Aid of Network Flow Algorithms", *IEEE Transaction on Software Engineering,* January 1977.

[Su 85] W-K Su et al, "C Programmers' Guide to the Cosmic Cube", California Institute of Technology, M.S. Thesis (CITCS) 5 129:TR:84.

[Weinreb 8 1] Daniel Weinreb and David Moon, *LISP Machine Manual,* Symbolics Inc., 198 1

[Weng 75] K-S Weng, "Stream-Oriented Computation in Data Flow Schemas", TM 68, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1975.

[Yan86] J.C. Yan and S.F. Lundstrom, " AXE: A simulation environment for actor-like computations on ensemble architectures," *Proc. I986 Winter Simulation Conference,* Washington DC.

Appendix - BDL Syntax

This appendix follows BNF notations. Included are some keys to help interpret these grammatic rules:

- All non-terminal symbols are enclosed in "<>"
- All reserved symbols are printed in **bold**
- <term>* means that the <term> occurs **0** or more times
- <term>+ means that the <term> occurs 1 or more times
- *Identifier* means a string of any ascii literals

<Declarations>:: <Structure-Declaration> | <Identity-Declaration>

<Structure_Declaration>:: **(DefPlayer** <Type_ID> <MessageList> <AcquaintanceList> <StateList> <Behavior>*)

<Type_ID>::*Identifier*

<MessageList>:: (<MessageID>*)

<MessageID>:: *Identifier*

<AcquaintanceList>:: (<Reference>*)

<Reference>:: *Identifier*

<StateList>:: (<State>*)

<State>:: *Identifier*

<Behavior>:: (<MessageID> <St>*)

<St>:: <Empty> | <Meta_St> | <Rsrc_St> | <Message-St> | <Value_St>

<Empty>:: ( )

<Meta_St>:: <Branch-St> | <Conditional-St> | <Loop-St>

<Branch-St>:: **(Branch** <Clause>+)

<Clause>:: (<Probability> <St>+)

<Probability>:: <I-value>

<I-value>:: <State> | *Integer*

<Conditional-St>:: (**If** <Condition> <Success> <Failure>*)

<Condition>:: <I-value> | <Value-St>

<Success>:: <St>

<Failure>:: <St>

<Sequence-St>:: **(Progn** <St>+ **)**

<Loop_St>:: **(Repeat** <LoopCount> <St>+)

<LoopCount>:: <I_value>

<Rsrc_St>:: (**Run** <Duration> **)** | **(Make** <State> <Type_ID> <Directive>)

<Directive>:: **local** | **remote** | **random**

<Message-St>:: **(Record** <Local>+) |

(Wait <MessageID>) |

(Reply <MessageID> <Value>+) |

(**Post** <Receiver> <MessageID> <Value>+) <Receiver>:: <A_value>

<A_value>:: <PlayerReference> | <Player_ID> | **Self** | **Sender**

<Value>:: <A_value> | <I_value>

<Local>:: <Reference> | <State>

<Value_St>:: (**Dec** <State>) | (**Inc** <State>) |

(**Setq** <State> <I-value>) | (**Setq** <PlayerReference> <A_value>) |

(❑ <Value> <Value>) | (≠ <Value> <Value>) | (**<** <I_value> <I-value>) |

(**>** <I_value> <I_value>) | (≤ <I_value> <I_value>) | (≥ <I_value> <I_value>)

<Identity-Declaration>:: **(Global** <Player_ID> <Type_ID> <Site_ID>)

<Player_ID>:: *Identifier*

<Site_ID>:: **(Site** *Integer* **)**