CLONING PARALLEL SIMULATIONS:

Clone-Sim Version 0.9

A Programmer's Manual and Specifications

Maria Hybinette Richard Fujimoto

{ingrid,fujimoto}@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, U.S.A.

August 20, 1999

Contents

1 Introduction **1**

2 Design Goals **3**

3 Assumptions **3**

4 The Clone-Sim Application Programmer Interface **4**

5 The Simulation Application **5**
5.1 The Initialization Phase 6
5.2 The Simulation Phase 7
5.3 The Wrap-up Phase 8

6 Command Line Parameters **8**

7 Compiling and using Clone-Sim **9**

8 An Example **9**

9 Reference Manual: Simulation Application **18**

1 Introduction

This document describes the Clone-Sim application programmer interface (API), and how to use it for simulation cloning. Clone-Sim enables on-demand "cloning" of parallel and distributed discrete event simulations. The package can be used in interactive as well as non-interactive environments. Both optimistic [Jefferson and Sowizral 1982] and conservative simulators [Bryant 1977; Chandy and Misra 1979] can be supported. Currently, Clone-Sim has been implemented with Georgia Tech's Time Warp simulation executive [Das et al. 1994] called GTW, an optimistic simulator.

Cloning is a mechanism that enables the concurrent evaluation of multiple simulated futures. The approach has been developed for parallel discrete event simulators, where the simulation consists of a collection of logical processes (LPs) potentially executing on different processors [Fujimoto 1990a]. These types of simulators traditionally have two types of primitives: (1) *send and schedule* which schedules an event on some logical process (ScheduleEvent) and (2) *receive* which processes a scheduled event (ProcessEvent).

A running parallel discrete event simulation is dynamically cloned at *decision points* to explore different execution paths concurrently. A decision point is where the states of different versions of a simulation begin to diverge. A decision point is defined or inserted *on* a specified logical process or processes and in this manner enables the exploration of different scenarios. The *user views* the whole simulation domain replicated into many different planes where each plane is an independent version of the simulation executing in parallel with the other cloned versions (See Figure 1).

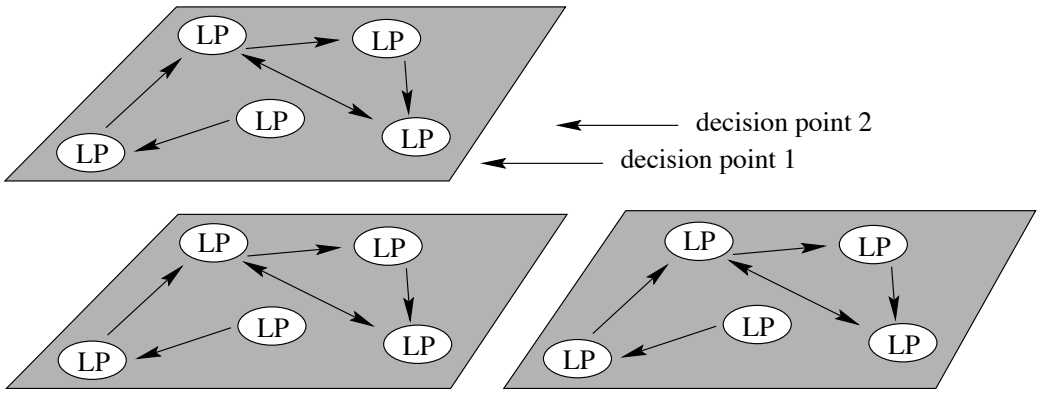

Figure 1: A Parallel Discrete Event Simulation Represented by Logical Processes (LPs) Replicated Twice; the Upper Left Plane Shows the Original Simulation

The Clone-Sim implementation avoids the cost of brute-force methods that replicate an entire simulation. Cloning in Clone-Sim uses an incremental update scheme. In this paradigm each plane or *version* of the simulation contains a collection of virtual logical processes. A collection of virtual logical processes is created each time a simulation is cloned. The difference between the cloned simulations is in the mapping of virtual logical processes to physical logical processes. Here, a physical logical process refers to the run-time environment of virtual logical processes. Each virtual logical process (V) is assigned or *mapped* to a physical logical process (P). The idea is that several virtual logical processes can share the same physical logical process thereby

avoiding replication of common computations. But two physical processes cannot be mapped to the same virtual process. An analog is virtual memory where the same main memory address can shared by several virtual addresses, but two addresses in main memory cannot be mapped to the same virtual address. The mapping between virtual and physical processes is updated as the clones diverge. Resources are re-used as long as possible and only the smaller portions which cannot be shared are replicated.

To illustrate, the bottom image in Figure 2 shows the mapping between virtual processes and

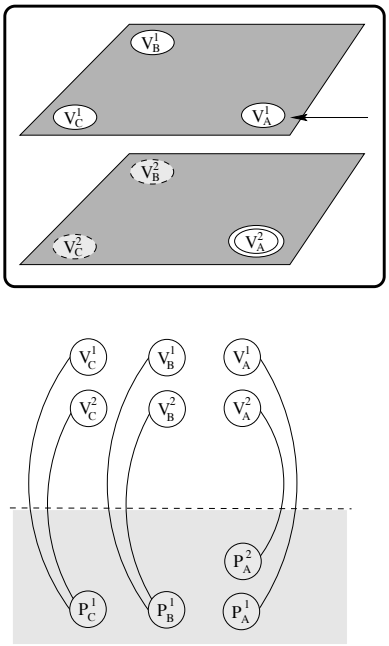

Figure 2: A Snapshot of a Simulation That Has Been Cloned; the Top Image Shows the Two Virtual Versions of the Simulation, the Bottom Image Shows the Mapping of the Virtual Processes to Physical Processes A, B and C

physical logical processes after the simulation is cloned on physical process A. Here the mapping of the original version of virtual processes stays the same and the computation between clones is shared. Virtual processes B and C, version one and version two share the same corresponding physical process; while virtual process A version one and version two maps to different physical processes.

As the simulation progresses the mapping of virtual processes to physical processes changes, as new physical processes are created. Message sends and receives are carried out in the physical layer. In this manner a physical send corresponds to a *set* of sends in the virtual process layer. For more details on the incremental update scheme and its performance see [Hybinette and Fujimoto 1998].

Clone-Sim achieves efficient cloning by intercepting the communication primitives of a simulator executive. By monitoring the send and receive primitives Clone-Sim can avoid unnecessary cloning of logical processes (LPs). Likewise, it can determine which logical processes need copies of messages. The key idea is that the receiving LP can determine whether to clone or forward by inspecting bits that are piggy-backed by the cloning mechanism on messages.

This manual is organized as follows. The following section describes the design goals of Clone-Sim. The underlying assumption are discussed in Section 3. Section 4 discusses the software

architecture and the interactions between the software modules. The application programmer interface of Clone-Sim is described in Sections 5 and 6. The compilation of Clone-Sim and header file requirements are discussed in Section 7. Section 8 describes illustrates the use of the cloning primitives by describing an simulation that utilizes cloning written for the Georgia Tech Time Warp executive. Section 9 contains a reference manual.

2 Design Goals

Goals for the Clone-Sim implementation are:

- efficiency,
- transparency and
- simulator independence

Efficiency is in terms of number of alternatives evaluated in a time-constrained period and memory resource usage. Efficiency is achieved by enabling multiple scenario analysis and allowing different versions to share computations between themselves. Transparency is with respect to the simulation application and is accomplished by monitoring pre-existing primitives (*send* and *receive*). Simulator independence refers to the choice of optimistic or conservative synchronization. Here, Clone-Sim provides simulator independence with respect to this framework.

3 Assumptions

Clone-Sim is based on the assumption that the simulator executive provides *send and schedule* and *receive* primitives to the simulation application. The relationship between the simulation executive, the user application and the ScheduleEvent (send and schedule) and ProcessEvent (receive) primitives are illustrated in Figure 3. Here, the simulation application defines the events and the simulation executive manages the synchronization. For example if the simulation uses GTW, the application defines ProcessEvent, tells GTW when to schedule the event, then GTW actually makes sure that ProcessEvent is called when appropriate.

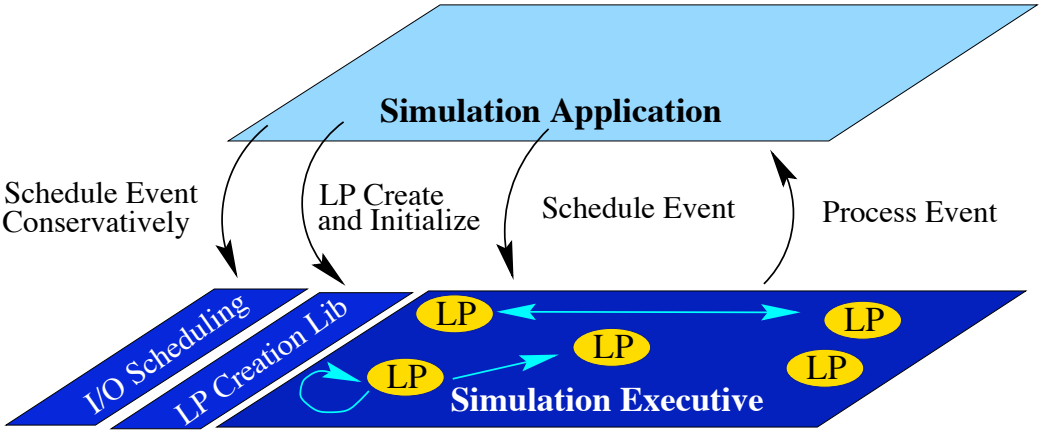

Figure 3: Clone-Sim Assumptions

Clone-Sim also assumes that the simulation executive provides for dynamic LP creation including allocation, initialization and copying LPs. It is assumed that one can schedule events conservatively, i.e. the event can be scheduled with the assumption that it will never roll back, (this is trivial if the simulation executive is conservative). To summarize, Clone-Sim assumes:

- a *send and schedule* primitive (ScheduleEvent)
  - including capability to schedule an event conservatively,
- a *process event* primitive (ProcessEvent) and
- a capability to create/copy logical processes

4 The Clone-Sim Application Programmer Interface

A simulation consists of a simulation application (provided by the user) and a simulation executive that implements the synchronization protocol. The simulation executive provides primitives that allow simulation programmers to define their own applications. This is a layered system, with the operating system at the bottom, the simulator executive in the middle and the simulation application at the top (See Figure 4). Clone-Sim consists of two modules: the **Interactive-**

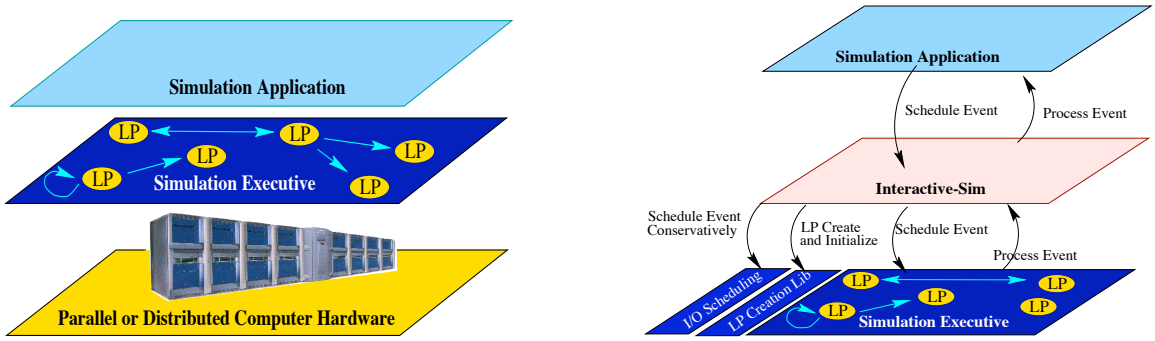

Figure 4: Views of Simulations: Traditional Parallel Discrete Event Simulation Is Shown on the Left; The Monitoring Layer called Interactive-Sim in Relation to the Simulation Executive and the Simulation Application Is Shown on the Right.

Sim module which is layered between the simulation executive and the application simulation program and the **Clone-DB** database that is independent of the synchronization primitive of the simulation executive. From the point of view of the simulation executive, Interactive-Sim is a simulator application. In the context of the layered system Interactive-Sim is between the simulation executive and the user's application simulation program. Interactive-Sim itself is decomposed into sub-modules, where each is implemented for a particular simulation executive. The sub-modules are "pluggable"in that the appropriate submodule is plugged in for a specified simulation executive. New sub-modules can be implemented using a specified application interface. The structure, semantics and the functions that need to be implemented in a sub-module are described in a companion manual [Hybinette and Fujimoto 1999]. The general idea behind Interactive-Sim is that it is transparent to the simulation program, and also to the programmer utilizing the cloning primitives.

The key function of Interactive-Sim is to (1) intercept message sends and (2) to process events. For example, Interactive-Sim needs to know the message send and message receive primitives in

order to intercept the invocation to process events or to forward copies of a message to cloned LPs. After interception, Interactive-Sim queries Clone-DB to determine message or process cloning. Interactive-Sim also intercepts functions that control or inquire about the number of LPs, since cloned simulation has a larger number of LPs than an un-cloned simulation. Interactive-Sim may require a minor adjustment of the interface between the simulator executive and its application to accomplish control of LPs, for example in GTW we protected accessibility to the constant `TWnlp` that returns the number of logical LPs in the simulation with a function that returns the same number (`int TWGetNumLPs()`). Similarly a function is used to set the constant: `TWSetNumLPs()` (See [Hybinette and Fujimoto 1999] for details).

The architecture of Clone-Sim is presented in Figure 5.

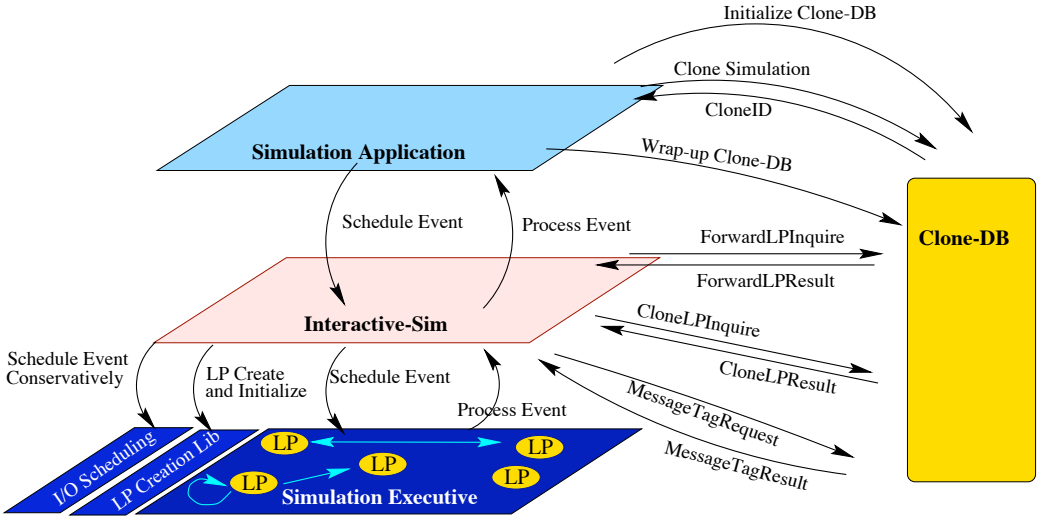

Figure 5: Clone-Sim Architecture and Application Programmer Interface

The next section will describe the simulation application programmer interface to the Cloning functions. These are the primitives that enables a programmer to clone simulations.

5 The Simulation Application

Clone-Sim accounts for the mapping of virtual LPs to physical LPs by assigning identifiers to each physical LP. There are three types of identifiers:

1. unique
2. global and
3. simulation.

The unique identifier (`UID_LP`) distinguishes physical logical processes. In addition each physical logical process is assigned to a global identifier (`GID_LP`), the same global identifier may be shared by multiple physical logical processes. The `GID_LP` corresponds to the original physical ID (before cloning) of the LP at time 0 (if this physical logical process is cloned the `GID_LP` may corresponds to multiple physical processes). Finally each version of a simulation (a version consists of a set of virtual LPs or a *plane* in Figure 1) is associated to a simulation identifier (`Clone_ID`). These identifiers can be accessed by specified functions that are described below.

There are currently six functions available to the simulation application. These are primitive functions, more complex cloning scenarios can be composed by these primitives. Each of these will be discussed below in detail. The API functions are:

```
void CloneSim_InitAppl( int argc, char ** argv )
void CloneSim_CloseAppl( void )
int  CloneSim_Create( int UID_LP, double current_sim_time )
int  CloneSim_Delete( int clone, double start, double end, CSFunc_p trig )
int  CloneSim_GetCloneID( int LP )
int  CloneSim_GID( void )
int  CloneSim_UID( void )
```

The utility of these functions depends on the state of the simulation. The state of the simulator is viewed as moving through three phases: initialization, execution of the simulation, and wrap-up. The initialization phase determines the initial number of logical processes, the mapping between processes and processors, and specifies event handlers. The simulation phase calls and schedulers event handlers specified for LPs. The wrap-up phase is where the simulation phase is complete and before the application terminates. Each of these phases are described in more detail below.

5.1 The Initialization Phase

The implementation assumes that the simulation executive allows the user application to set up the mapping. The mapping is thus exposed and allows Clone-Sim to manipulate the mapping. This is important, because Clone-Sim will exploit this ability and maintain the mapping transparently from the simulation application.

In order to initialize Clone-Sim, `void CloneSim_InitAppl()` is called at *the end* of the initialization phase. The function has two arguments: `argc` and `argv` that specify command line parameters that are passed to Clone-Sim. The main function of command line arguments is to specify the maximum number of clones that can be instantiated simultaneously. Details on these parameters are discussed in Section 6. Initializing Clone-Sim sets up data structures that: (1) control the mapping between logical processes and processors, (2) provide buffer space to cloned physical logical processes via a process buffer pool of LPs and (3) determine message or process cloning or determine child, sibling, parent relationships between cloned simulations.

In the current implementation, the mapping between logical processes and processors is assumed to be static after the initialization phase, however an extension is planned to allow the mapping to be dynamic and allow for automatic load-balancing. Also the maximum number of clones that can be instantiated simultaneously is specified during the initialization phase, currently this is specified via a command-line parameter (command-line parameters are defined in Section 6). An example initialization is shown below:

```
void InitializationPhase( int argc, char **argv )
  {
  /* code initialization is defined here */

  /* call initialization procedure for the cloning library */
  CloneSim_InitAppl( argc, argv );
  }
```

5.2 The Simulation Phase

During the simulation phase, cloning allows for the insertion and deletion of decision points via the cloning primitives `CloneSim_Create()` and `CloneSim_Delete()`. This can be implemented interactively or non-interactively by the simulation programmer. The event that causes cloning must be a **conservative** event (guaranteed to never rollback). If the decision point occurs on a set of LPs then a conservative event must be scheduled at the same simulation time by each of the LPs defined in the set of the same decision point.

The prototype of the function that allows for the insertion of a decision point is defined below:

`int CloneSim_Create(` int UID_LP, double current_sim_time `)`

The call returns an identification number of the newly cloned simulation, so that one can refer to the clone when deleting or pruning it. A negative number is returned upon error. The result is the instantiation of a new simulation. This represents the location in the execution path where the state of the newly created version start to diverge from the version that called it. As the function is called one new physical logical process is created. Any assignment to variables or calls to functions within this conservative event *after* the call to `CloneSim_Create()` *only effect the original clone*. Assignment or calls to functions within the conservative event before `CloneSim_Create()` effect both versions of the simulation: the newly created clone and the original clone.

The argument `UID_LP` is the unique identifier of the LP, and can be accessed via the call `CloneSim_UID()`. The argument current_sim_time is the simulation time of the event that calls the primitive. An example use of this function is included below:

```
void A_Conservative_Event( arguments )
  {
  int  unique_LP_identifier;
  int clone_identifier;

  /* simulator dependent code here */
  /* effects both original LP and instantiated LP below */

  /* access the unique logical process identifier of callee */
  unique_LP_identifier = CloneSim_UID();

  /* instantiates a new clone, a new logical process is created */
  clone_identifier
      = CloneSim_Create( unique_LP_identifier, current_sim_time );

  /* code here only effects caller LP of original simulation   */
  /* the new LP created via the Clone_SimCreate is un-effected */
  }
```

In addition to creating clones, Clone-Sim provides a mechanism to eliminate simulations that are not needed. This is done by installing a "trigger". The primitive which installs the trigger is: `CloneSim_Delete()`. The function can be called interactively or non-interactively. The prototype is:

`int CloneSim_Delete( int clone, double start double end, CSFunc_p trigger )`

The trigger is a condition defined by the argument **trigger** that is sampled within the simulation period specified by the arguments: `start`, `end`. The installation of the trigger only effects the logical process that installs the trigger and only the simulation whose version is given by the first argument: `clone`. So if all versions in the simulation need to be monitored the trigger needs to be installed for each version. If trigger is `NULL` the version that calls `CloneSim_Delete()` is pruned un-conditionally.

Currently, the pruning function only provides un-conditional pruning, conditional pruning is only available in an un-released version of Clone-Sim.

```
void Some_Event( arguments )
  {
  /* possibly some  simulator dependent code here */

  if( some condition )
     {
     /* prune if the simulation time of the callee is within the */
     /* simulation period [0.00, END_TIME] */
     CloneSim_Delete( cloneID, 0.00, END_TIME, NULL );
     }

  /* possibly some  simulator dependent code here */
  }
```

5.3 The Wrap-up Phase

When the simulation completes `CloneSim_Close( void )` should be called to clean up data structures and compute statistics. It should be called after the simulation code has completed and before terminating the program. The prototype is defined below:

`int CloneSim_Close( void )`

6 Command Line Parameters

The main purpose of the command line arguments is to set the maximum number of clones that can be instantiated simultaneously. The command line allows the programmer to set this parameter either implicitly by specifying the `cloning activation time` of each clone or directly by specifying the maximum number of clones. Cloning activation time is the earliest time (in simulated time) a clone can be scheduled.

The switch that sets the activation time is: `-c`. The switch is set for each clone that may be instantiated during the simulation. An example command line to enable the instantiation of a second clone at time 10 and a third at 20 is (the first clone is the initial simulation):

a.out -c 10 -c 20

When this option is used Clone-Sim parses the arguments and stores the times in a user-accessible array. Since clones may only be triggered from user code it is up to the application to monitor the simulation time and trigger clones appropriately.

The switch that sets the the maximum number of clones directly is: `-V` (where V is for versions). An example example command line that enables 10 simultaneously clones is:

a.out -V 2

Interactive-Sim initializes two data structures during command line processing:

- `CloneSim_CLONETIME` an array of doubles, and
- `CloneSim_NumClones` an integer.

Each element in `CloneSim_CLONETIME` corresponds to the activation time for a particular version that is instantiated, here index 0, refers to the first instantiated version, index 1 to the second instantiated version and so on. `CloneSim_NumClones` limits the number of clones that are instantiated, and is set directly by the switch `-V` or indirectly by the number of times the switch `-c` is set on the command line. If activation time is not set on the command line via the `-c` switch, then the activation time of each clone is set to the length of the simulation. For the `a.out` examples above `CloneSim_NumClones` is set to 2. The maximum number of clones in this case is three, where two clones are instantiated during the simulation.

The implementation cloning can set statically or dynamically. One way to implement dynamic cloning is to initially set the cloning time for each clone to the value that corresponds to the length of the simulation. To push the activation time for a particular version to an earlier time, its value can then be manipulated while the simulation is running. In the current release of Clone-Sim both `CloneSim_NumClones` and `CloneSim_CLONETIME` are accessible to the simulation programmer, in a later release however they will only be accessible via specified function primitives.

7 Compiling and using Clone-Sim

In order to utilize cloning the application needs to link with Clone-Sim via the flag `-lCloneSim`. The math library is also required, and is linked with the -lm flag. The Clone-Sim interface is defined in the header file: `cs_api.h` and must be included in each file that uses cloning primitives.

8 An Example

Clone-Sim consists of two modules: Clone-DB and Interactive-Sim. Clone-DB is independent of the synchronization mechanism of the simulator executive. A sub-module in Interactive-Sim in contrast must address the particulars of a simulator executive. There is a separate sub-module for each simulation executive supported. However, the programmer interface between Clone-Sim and the simulation application that is to be cloned is the same between all different simulation executives. Here, in this manual we assume that the sub-module has been implemented.

To illustrate the utility of Clone-Sim we will describe how cloning can be utilized for an application written for GTW using the cloning primitives. The same primitives may be used for other simulation executives, including simulator that utilizes a conservative synchronization mechanism. We assume that the reader has previous experience with writing applications for a simulator such as GTW, and we will provide high-level details as needed. Events in GTW is scheduled by specifying the destination LP and the time stamp increment. Each LP is instantiated by an event.

The GTW application that will be used for illustrative purposes is called P-Hold. P-Hold provides synthetic workloads using a fixed message population. The P-Hold simulations described

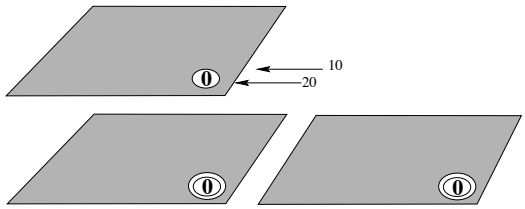

Figure 6: A Parallel Discrete Event Simulation Represented by Logical Processes (LPs) Replicated Twice (Where Each Plane Consists of a Set of LPs); the Upper Left Plane Shows the Original Simulation. (Only LP 0 is Shown In Each Version of the Simulation)

here will use a message population of 256 (**message_population**) and 1024 logical processes (**number_lps**) for the length of the simulation (**END_TIME**) is set to 100 simulated seconds (for more details on P-Hold see [Fujimoto 1990b]). The message population is set by having each LP send **message_population/number_lps** messages during the initialization phase of the simulation (when simulated time is 0 for each of the logical LPs). For this application assume we want to instantiate two clones each from logical process 0: one clone is to be instantiated at simulated time 10 and the other at simulated time 20. A way to view this topology is shown in Figure 6, here LP 0 is physically cloned from the original simulation once at simulated time 10 and a second time at simulated time 20.

A GTW program consists of 3 phases: initialization, simulation and wrap-up. In addition a GTW program must define the structures for the state vectors of its logical processes and the message format to schedule events. In the below structure of the state space and each of the 3 phases of P-Hold is described.

GTW allows the user to specify the data types which give the executive an idea how to maintain the state space. Example data types include read-only, incremental and automatic. Figure 7 defines the state vectors and the message format for P-Hold events. The variables preceded by the prefix `cs_` are used to control cloning. There are two variables defined in the state space used for this purpose: `cs_CloneCount` and `cs_cloned`. `cs_CloneCount` is used to control the number of clones that are instantiated and `cs_cloned` is used to control the LP that instantiates the clone (in this example LP 0 will instantiate all clones). The message format in this case is simple and contains a single integer that counts the number of times a message "bounces" between LPs. An LP is instantiated by an event and a new event is scheduled by sending a message with a specified location LP and time, so in P-Hold each event corresponds to "a flow of messages", and the originator of the message flow can always be found by backtracking from the receivers to senders up to simulated time 0.

The first phase of GTW is initialization. The initialization phase in GTW is defined by two procedures: `TWInitAppl()` and `IProc()`. `TWInitAppl()` defines global initialization and sets a number of things such as number of logical processes, event handlers, allocates the state space for the logical processes and so on. `IProc()` initializes the state of the LP and sends the initial messages to get the simulation started. The `IProc()` procedure always executes at simulated time zero. The specifics of `TWInitAppl()` and `IProc()` for the example are described in turn below.

The code `TWInitAppl()` for P-Hold is shown in Figure 8. In this example the simulation is homogeneous: each logical process defines the same initializing (`IPHoldLP()`), event handling

(PHoldLP()) and finishing procedures FPHoldLP(). The LPs differ in having different random seeds (See the line that sets Seeds[i] in Figure 8). To initialize cloning CloneSim_InitAppl() is called at the *end* of TWInitAppl(). This enable Interactive-Sim to intercept the event handlers, and manipulate the format of messages.

CloneSim_InitAppl() takes two arguments: argv and argc. In the P-Hold example we set the cloning activation time[1] to 10 for the first clone and 20 for the second clone. This can be accomplished by passing: "-c 10 -c 20" via the above arguments. A GTW application sets several simulation parameters on the command line. Typically, number of processors and simulation time is set on the command line. In our example the command line to run P-Hold for 100 simulation seconds (set by the -t switch) using 4 processors (set by the -p switch) is:

```
phold -p 4 -t 100 -A
```

GTW uses the -A switch to allow command lines arguments to be passed to the TWInitAppl() procedure. All arguments following this flag are passed to the initialization procedure via argc and argv. This is shown above for illustrative purposes and the switch is not required if no arguments follow -A. In the example, the same arguments are sent to CloneSim_InitAppl() as in TWInitAppl(). An example P-Hold command line that enables the instantiation of a second clone at time 10 and a third at 20 (the first clone is the original simulation) is:

```
phold -p 4 -t 100 -A -c 10 -c 20
```

In the second part of the initialization phase GTW initializes the state space via the handler IProc() (set to IPHoldLP() in the example). IProc() can be viewed as an event at simulated time zero. PHoldLP() is shown in Figure 8. Here, IPHoldLP() sets the state vectors, specifies the state space that is automatically check-pointed and initializes the state variables. The state variable cs_CloneCount is set to 0, to indicate that currently the LP has not instantiated a clone and the state variable cs_cloned is set to 0 to enable cloning. To start the simulation each LP schedules **MsgPop/number_lps** messages, where the time stamp increment of each message is picked from a exponential distribution between 0 and 1; and the destination is picked from a uniform distribution consisting of all participating LPs.

After the application is initialized the simulation phase starts and the event handlers are called. The event handlers are specified in TWInitAppl(), and here all event handlers are set to PHoldLP(). The code for the event handler is shown in Figure 8. The procedure records the timestamp of the message that was just received and increments a counter indicating the number of messages received. The procedure determines cloning by evaluating (1) the activation time and the (2) identity of the logical process calling the procedure. The simulation clones LP 0 twice: once when LP 0 progresses beyond simulated time 10 and a second time when LP 0 after simulated time 20. The state variable AVars.cs_cloned is used to prevent undesirable clones. Children of logical process 0 are prevented to propagate further clones. A clone is instantiated by a conservative event that is defined by the procedure ClonePHoldLP() and is scheduled at simulated time TWNow() – resulting in a time stamp increment of zero, the first argument of TWGetMsg(). In this example, cloning is always instantiated by logical process 0. However, the primitives allows it to be instantiated by any logical process, even a logical process that has already been cloned.

The code for the event (ClonePHoldLP()) that instantiates cloning is shown in Figure 8. This event is scheduled conservatively from an event handler. In this case the event handler

[1] the cloning activation time is the earliest time a clone can be activated and is set by the switch -c (see Section 6 for more details).

for the original clone of logical process 0, schedules the cloning event (the cloning event must be a conservative event to prevent rollbacks). During the conservative event the state variable `SV->AVars.cloned` serves to control the capability to clone further clones. If it is set to 0 then further cloning is allowed, if it is set to 1, cloning is disabled. The cloning event scheduled from `PHold()` prevents new clones from further cloning and preserves the right to the caller (the parent). The logical process that instantiates a simulation maintains the identification number of the clone it last instantiated, and its own identification number. Similarly, logical process that is cloned (the child) maintains the parent identification number, and its own identification number. This is also where the versions of the cloned simulations differ.

Before the simulation terminates Clone-Sim collects cloning statistics, and clears data structures. This is accomplish by calling `CloneSim_CloseAppl()` from the GTW function: `WrapUp_Appl()`. This code is shown in Figure 12. GTW wraps up applications by calling `FProc` for each logical process and then after finishing calling each finishing procedure for each LP it calls `WrapUp_Appl` once. The code for `FPHoldLP` is shown in Figure 13 and is an empty procedure.

```
#include "gtw.h"
#include <stdio.h>
#include <malloc.h>

/* remember timestamp on last MSZ messages */
#define MSZ 10

/* read only variables in state vector */
/* automatically check-pointed variables for LP */
struct MyAutoVars
        {
        TWSeed  Seeds;          /* seeds for random number generator */

        int     cs_CloneCount;  /* number of clones instantiated */
        int     cs_cloned;      /* controls who instantiated cloning */

        int     cs_parent;
        int     cs_child;
        };

/* incrementally check-pointed variables for LP */
struct MyIncVars
        {
        TWTime  LastTS[MSZ];    /* remember last MSZ timestamps */
        int     Count;          /* number of messages received */
        };

struct MyLPState
        {
        struct MyAutoVars  AVars;
        struct MyIncVars  ISVars;
        };

struct MyMsgData
        { /* Message data */
        int counter;
        };
```

Figure 7: Data structures that define state vectors and messages for P-Hold

```
void TWInitAppl( int argc, char **argv )
  {
  int  number_lps, i, message_population;

  number_lps         = 256;
  message_population = 1024;

  /* specify number of LPs */
  TWSetNumLPs( number_lps );

  for( i = 0; i < TWGetNumLPs(); i++ )
     {
     /* LP to PE mapping:  map round robin */
     TWLP[i].Map = i % TWGetNumGlobalPEs();

     /* set handlers: initialization, event and wrap-up functions */
     TWLP[i].IProc =  (FuncPtr2) IPHoldLP;
     TWLP[i].Proc  =  (FuncPtr)  PHoldLP;
     TWLP[i].FProc =  (FuncPtr2) FPHoldLP;
     TWLP[i].IncrSave = TRUE;

     /* set and allocate LP state */
     TWLP[i].State = TWMalloc( sizeof(struct MyLPState) );
     TWLP[i].LPStateSize = sizeof(struct MyLPState);
     TWLP[i].CopySize = sizeof(struct MyAutoVars);
     }

  /* get initial random number generator seeds */
  for( i = 0; i < TWGetNumLPs(); i++ )
     TWRandInit( &(Seeds[i]), 0 );

  /* set memory mapping */
  for( i = 0; i < TWnpe ; i++ )
        {
        for( j = 0; j < TWnpe; j++ )
           {
           if( i != TWnpe || j != i )
       TWMemMap[i][j] = 1;
           if( i == j )
               TWMemMap[i][j] = 1;
               TWMemMap[i][j] = 1;
           }
        }

  /* set message size */
  TWMsgSize = sizeof( struct MyMsgData );

  /* call initialization procedure for the cloning library */
  CloneSim_InitAppl( argc, argv );
  }
```

Figure 8: `TWInitAppl()` for P-Hold that uses cloning

```
void IPHoldLP( LPState *SV, int MyPE )
  {
  struct MyMsgData     *TWMsg;
  int                  i,
  TWTime               ts;
  TWEachSeed_t         s1, s2;
  int                  rcv_lp, dst_lp;

  struct LPState * CurState;
  CurState = GState[MyPE].CurState;

  myLP = TWMe();
  SV = (struct MyLPState *) TWLP[TWMe()].State;

  /* specify variables to be automatically check-pointed */
  TWAutoCheck( (char*)&(SV->AVars), sizeof(struct MyAutoVars) );

  TWRandGetSeeds( &(Seeds[TWMe()]), &s1, &s2 );
  TWRandSetSeeds( &(SV->AVars.Seeds), s1, s2 );

  /* initialize state variables */
  if( CurState->IncrSave )
      {
      SV->ISVars.Count = 0;
      for( i = 0; i < MSZ; i++ )
         {
         SV->ISVars.LastTS[i] = 0.0;
         }
      }

  SV->AVars.Count = 0;
  SV->AVars.CloneCount = 0;
  SV->AVars.cloned = 0;

  for( rcv_lp = TWMe(); rcv_lp < MsgPop; rcv_lp += TWGetNumLPs() )
     {
     ts = TWRandExponential( &(SV->AVars.Seeds), 1.0 );
     dst_lp = TWRandInteger( &(SV->AVars.Seeds), 0, TWGetNumLPs()-1 );
     TWGetMsg( ts, dst, sizeof(struct MyMsgData) );
     TWMsg->counter = 1;
     TWSend();
     }
  }
```

Figure 9: Procedure that initializes each LP for P-Hold

```
void PHoldLP( struct MyLPState *SV, struct MyMsgData *M, int MyPE )
  {
  struct MyMsgData     *TWMsg;
  struct LPState       * CurState;
  TWTime               ts;
  int                  dst;
  double               time_now;

  MyState  = &GState[MyPE];
  CurState = GState[MyPE].CurState;

  myLP = TWMe();
  time_now = TWNow();

  if( CurState->IncrSave )
      {
      TWCheckTWTime(&(SV->ISVars.LastTS[SV->ISVars.Count % MSZ]));
      SV->ISVars.LastTS[SV->AVars.Count % MSZ] = TWNow();

      TWCheck(&(SV->ISVars.Count));
      SV->ISVars.Count++;
      }

  SV->AVars.Count++;

  /* determine cloning scheduling */
  if( (TWMe() == 0) && (SV->AVars.cloned == 0)  )
      if( CloneSim_CLONETIME[SV->AVars.CloneCount] > 0.0 )
          {
          if( ((SV->AVars.CloneCount) < CloneSim_NumClones )
             && (CloneSim_CLONETIME[SV->AVars.CloneCount] < TWNow()) )
      {
               SV->AVars.CloneCount++;
               TWGetMsg( 0, TWMe(), sizeof(struct MyMsgData) );
               TWBlockingIOSend( (IOFuncPtr) ClonePHoldLP );
               }
          }
      }

  /* schedule a new event */
  ts  = TWRandExponential(&(SV->AVars.Seeds), 1.0);
  dst = TWRandInteger(&(SV->AVars.Seeds), 0, (TWGetNumLPs())-1);

  TWGetMsg( ts, dst, sizeof (struct MyMsgData) );
  TWMsg->counter = M->counter + 1;
  TWSend();
  }
```

Figure 10: Event handler procedure for P-Hold that may instantiate a clone

```
void ClonePHoldLP( struct MyLPState *SV, struct MyMsgData *M, int MyPE )
  {
  struct PEState       *MyState;
  struct LPState       *CurState;

  MyState  = &GState[MyPE];
  CurState = GState[MyPE].CurState;

  /* access the unique logical process identifier of callee */
  uid           = CloneSim_UID();

  /* disable this current clone and child clone to propagate */
  SV->AVars.cloned  = 1;

  /* instantiates a new clone, a new logical process is created */
  clone_identifier           = CloneSim_Create( uid, TWNow() );

  /* code here only effects caller LP of original simulation   */
  /* the new LP created via the Clone_SimCreate is un-effected */

  /* enable clone 0 to propagate more clones */
  SV->AVars.cloned  = 0;
  }
```

Figure 11: Conservative event procedure for P-Hold that instantiates a clone

```
void WrapUp_Appl( void )
  {
  CloneSim_CloseAppl();
  }
```

Figure 12: Wrap-Up procedure for P-Hold

```
void FPHoldLP( struct MyLPState * SV, int MyPE )
  {
  }
```

Figure 13: Finishing procedure for P-Hold

9 Reference Manual: Simulation Application

Details of the cloning functions available to the simulation application are described in below:

void CloneSim_InitAppl(int argc, char **argv **)**

- This procedure initializes the Clone-DB and sets up an LP buffer pool that is later used when a simulation is cloned. The addressing and handling of the LP buffer pool are transparent to the simulation application programmer.
- A list of arguments can be passed to Clone-Sim via the parameter: **argv_appl**. The number of arguments are given by the parameter **argc_appl**. The main-purpose of the command line arguments is to set the maximum number of clones that can be instantiated simultaneously. The command line allows the programmer to set this parameter either implicitly by specifying the `cloning activation time` of each clone (via the `-c` switch) or directly by specifying the maximum number of clones (via the `-V` switch). Cloning activation time is the earliest time (in simulated time) a clone can be scheduled.
- This procedure must be called at the end of the initialization phase before initializing each logical process.

void CloneSim_CloseAppl(void **)**

- This procedure clears data structures and collects cloning statistics.
- This procedure must be called when the simulation phase is complete and before the application terminates.

int CloneSim_Create(int UID_LP, double current_sim_time **)**

- This function creates a new simulation and clones an LP. It returns an identification number of the newly cloned simulation, so that one can refer to the clone when deleting or pruning it. A negative number is returned upon error. The invocation of **Clone_Create** can be viewed as the insertion of a decision point.
- The argument `UID_LP` is the unique identifier of the callee LP, and can be accessed via the call `CloneSim_UID()`. The argument current_sim_time is the simulation time of LP that calls the primitive.
- The event that calls this function must be a **conservative** event (A conservative event is an event that is guaranteed to never rollback). The argument: `current_sim_time` must be the same as the simulation time of the callee (the conservative event).
- If the a the decision points need to effect multiple LPs then each LP must schedule a conservative event that calls **CloneSim_Create()** where the argument: `current_sim_time` is equivalent. In this special case one version of the simulation is created and each of LPs that calls **CloneSim_Create()** is cloned.
- Within this event, the cloned and original simulation are similar up to the invocation of this function, all statements after the invocation are only applied to the original simulation.

int CloneSim_Delete(int cloneID, double start, double end, CSFunc_p func**)**

- This procedure prunes the cloned simulation identified `cloneID`. The pruning can depend on a trigger specified by a condition function `func()` and time period when to sample the condition specified by the argument: `func`, (`func` must return an integer).
- **cloneID** is a unique number specifying the clone that is pruned, the time period when the trigger `func` is affected is specified by the arguments `start` and `end`. The trigger

is specified by the argument: `func`, and (`func` must return an integer).

- **func** is a user defined function determining if a clone should be pruned. The clone is pruned if **func** returns 1. A **trigger** of NULL is equivalent to a return of TRUE and then the version of the simulation that calls `CloneSim_Delete()` is pruned unconditionally.
- Currently, the pruning function only provides un-conditional pruning, conditional pruning is only available in an un-released version of Clone-Sim.

CloneSim_GetCloneID(void)

- This function returns the CloneID of the clone that invokes it.

CloneSim_GID(void **)**

- This function returns the corresponding logical process number of the original simulation (the first single un-cloned simulation). For example, if the simulation originally consisted of 2 logical processes numbered 0 and 1, then later, the simulation is cloned, all logical processes of the *cloned* simulation corresponding to logical process 0 and logical process 0 return a 0 when called from an event processed on logical process 0.

CloneSim_UID(void **)**

- This function returns a unique identification number of the logical process that invokes it.

REFERENCES

BRYANT, R. E. 1977. Simulation of packet communication architecture computer systems. MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge, Massachusetts.

CHANDY, K. M. AND MISRA, J. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering SE-5*, 5 (September), 440–452.

DAS, S., FUJIMOTO, R., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings* (December 1994), 1332–1339.

FUJIMOTO, R. M. 1990a. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (October), 30–53.

FUJIMOTO, R. M. 1990b. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), 23–28. SCS Simulation Series.

HYBINETTE, M. AND FUJIMOTO, R. M. 1998. Dynamic virtual logical processes. In 12^{th} *Workshop on Parallel and Distributed Simulation* (May 1998), 100–107.

HYBINETTE, M. AND FUJIMOTO, R. M. 1999. Cloning parallel simulations: InteractiveSim 0.9 – A programmer's manual and specifications. Technical report (August), College of Computing, Georgia Institute of Technology, Atlanta, GA. in progress.

JEFFERSON, D. R. AND SOWIZRAL, H. 1982. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Technical Report N-1906-AF (December), RAND Corporation.