Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

LANGUAGE BASED STATE SAVING EXTENSIONS FOR OPTIMISTIC PARALLEL SIMULATION

Fabian Gomes
Brian Unger

Department of Computer Science
The University of Calgary
2500 University Drive N.W.
Calgary, Alberta T2N 1N4
CANADA

John Cleary

Department of Computer Science
University of Waikato
Te Whare Wananga o Waikato
Private Bag 3105
Hamilton
NEW ZEALAND

ABSTRACT

One of the greatest challenges in making optimistic synchronization techniques such as Time Warp practical tools is making state saving efficient and easy to use. State saving is necessary so that when optimistic execution is found to be out of order, rollback can be used to recover an earlier execution state. Previous work has shown that the most robust and efficient technique for saving state is to incrementally save copies of small parts of the state at the point that they are modified. Unfortunately, this requires significant programmer intervention to insert additional code.

In this paper, C^{++} language extensions for transparent incremental state saving are presented. Operator overloading and type parameterization are used to incrementally save basic data types. Building on this, two new type-specifiers, "`recover`" and "`nonrecover`" are described. They allow a single declaration to specify, for example, that all the member variables of a class are to be state saved, and for all the resulting state saving calls to be automatically generated. Issues, including how these specifiers interact with class inheritance and function declarations are examined and solved.

1 INTRODUCTION

Computer simulation is a valuable tool for the design and analysis of complex systems. However, the simulation of many important systems requires massive amounts of processor time and memory. The goal of parallel discrete event simulation is the acceleration of sequential simulation execution through concurrent execution on multiple processors (Fujimoto 1990).

Optimistic synchronization has been proposed to maximally exploit the inherent parallelism within systems. Time Warp, a well known optimistic synchronization technique based on the virtual time paradigm (Jefferson 1985), has been shown to be capable of attaining impressive speedup. Optimistic methods are characterized by logical processes executing aggressively and independently of others. A causality error detection and recovery scheme based on rollback is used to assure causality in the asynchronous parallel execution.

Two serious problems with rollback implementation need to be adequately resolved. The first is the efficient saving of state information during forward execution. The second problem is to solve the first in a way which is transparent to the programmer, i.e. that doesn't substantially complicate model development.

State saving mechanisms can be categorized as copy state saving (CSS) or incremental state saving (ISS) (Bauer and Sporer 1993, Bruce 1995, Cleary et al. 1994, Steinman 1993). In CSS mechanisms, a checkpoint of the entire object's state is taken on each event. CSS mechanisms are processor intensive and consume a large amount of memory. An optimization of the CSS strategy is to reduce the frequency of checkpointing using periodic state saving (PSS) (Fleischmann and Wilsey 1995, Lin et al. 1993, Preiss, Loucks, and MacIntyre 1994, Rönngren and Ayani 1994). Estimation of the optimal frequency of checkpointing, using static or adaptive schemes, is critical, as the object may need to rollback further if there is no state saved at the point of a synchronization error.

One major advantage of CSS is that it can easily be made transparent to the programmer. That is, no extra code is required to ensure that all the appropriate state is saved. Its major disadvantage is that it can easily become very expensive when the state size is large. This is particularly acute when each event has a small computational grain and only a small part of the state is modified.

On the other hand, in ISS mechanisms only information related to changes to the state are saved. Generally, ISS is robust in its performance. Its cost tends

to be a constant fraction of the event execution time, so that it seldom causes unexpected costs. The cost of rollback is proportional to the distance rolled-back, and in principle this can be arbitrarily large. Practical experience however shows that in most Time Warp executions rollbacks are limited to a few events, so that this is not a problem (Fujimoto 1989, Gomes and Unger 1994, Xiao and Unger 1995).

The major difficulty with ISS is the need for programmer intervention, in explicitly saving a state prior to its modification. This code and the effort of inserting it is needed only for Time Warp, thus exposing the underlying synchronization protocol during the model design, development and validation phases. Also mistakes in the ISS code can lead to subtle and difficult to detect bugs. Clearly some easy transparent mechanism for implementing ISS is necessary if Time Warp is to become part of the normal simulation life-cycle.

In this paper, C^{++} language extensions to support transparent incremental state saving are addressed. Incremental state saving is automated without compromising efficiency. A simple ISS backtrail mechanism is presented in section 2. Section 3 details the use of parameterized types to define recoverable basic data types. They have the same semantics as basic data types, but have built-in incremental state saving for write operations. The specification for a recoverable type annotation semantics and its use is detailed in section 4. This uses two new keywords "**recover**" and "**nonrecover**" to enable a type-safe declaration of which state is to be saved. Finally, a summary is presented in section 5.

2 SIMKIT SYSTEM

SimKit is a C^{++} class library that is designed for very fast discrete event simulation (Gomes et al. 1995). The primary goal of SimKit is to provide an event-oriented logical process modeling interface that facilitates building application models for sequential and parallel simulation with high performance execution capabilities.

The parallel SimKit implementation is atop a Time Warp executive interface called *WarpKit*. SimKit specifies three classes called **sk_simulation**, **sk_lp** and **sk_event**. In a simulation execution, there is one instance of **sk_simulation** that controls the various phases of the simulation execution. **sk_lp** is an abstract base class that is used to derive application level logical processes. Logical processes communicate using objects derived from **sk_event**. A **send_and_delete()** primitive in the **sk_lp** class is provided for inter-LP communication.

In SimKit, an ISS backtrail mechanism is implemented. In this mechanism, a *backtrail* of address/value pairs is constructed during forward execution. An address/value pair, representing an image of the state, is referred to as a state log element (SLE). SLEs are recorded prior to a write operation, and are linked to form the backtrail, from which a past system state can be recovered. Recovery entails scanning the backtrail, from the most recent SLE recorded back to the one at the recovery point, and writing the old values back into the associated addresses.

On most architectures two backtrails, one allowing 4-byte data and the other allowing for 8-byte data are instantiated for each **sk_lp**. These two sizes efficiently cover most basic data types, including integer, pointer, double, long long, etc.

3 PARAMETERIZED STATE DATA TYPES

Incremental State saving can be automated in C^{++} using *parameterized types* and *operator overloading*. A parameterized type defines a new type in terms of (or parameterized by) another unspecified type. The following design uses the `template` construct, to define specialized state types from basic data types (Ellis and Stroustrup 1990, Stroustrup 1988). State variables declared using the state type templates are said to be *recoverable*. Each of these specialized recoverable types need to have the same semantics as the base data type but are extended to include state saving. Assignment related operators are redefined to record a backtrail entry prior to the actual write operation, using operator overloading. In C^{++}, write operations on an integer data type include assignment operations like `=`, `*=`, `/=`, `\%=`, `+=`, `-=`, and, prefix and postfix versions of `++` and `--`. Integers also masquerade as bit vectors so assignment operators of the form `\&=`, `|=`, `\^{ }=`, `\~{ }=`, `<<=` and `>>=` also need to be overloaded.

3.1 State Data Type Templates

Two templates `State<T>` and `StatePtr<T>` are used in the declaration of recoverable variables. The former is used in declaring recoverable variables of basic types while the latter in declaring pointer type recoverable variables. Sample source code for the above template class definitions are provided in appendix A.

Basic recoverable types are defined using `class State<T>` as follows:

```
class State<int> SInt1, SInt2( 100 );
```

Recoverable integers `SInt1` and `SInt2` are constructed having the semantics of their base type, i.e. `int`. `SInt2` is initialized to `100` during construction. Such initialization does not cause `SInt2` to be state saved.

An automatic conversion operator allows recoverable objects `SInt1` and `SInt2` to be used in place of an `int` base type, as in the expressions,

```
a_int = SInt2;
if ( SInt2 < 100 ) { ... }
SInt2 = 1 + a_int + SInt1 * 2;
```

The overloaded assignment operator in `class State<int>` records a backtrail entry for `SInt2` prior to the assignment in the last statement.

Recoverable pointer data types are defined using `class StatePtr<T>` as follows:

```
class StatePtr< int > SPtrInt( & a_int );
class StatePtr< State<int> > SPtrSInt;
SPtrSInt = &SInt1;
```

In the first declaration, a recoverable pointer to an `int` data type is defined. `SPtrInt` is initialized to point to the variable `a_int`. No state saving of `SPtrInt` takes place during initialization within the constructor. In the second, a recoverable pointer to a recoverable integer is defined. `SPtrSInt` gets state saved prior to the assignment, in the third statement.

An overloaded de-reference operator in the `class StatePtr` returns the object pointed to by the `StatePtr` type object. Only in the second of the following assignments is a record entered into the backtrail. In the first case, the `int` type data pointed to is not recoverable and so is not state saved. In the second, as the object pointed to is recoverable, the overloaded assignment defined for the `class State` type object `SInt1` is invoked.

```
* SPtrInt = 10;
* SPtrSInt = 10;
```

In the next example, a recoverable pointer `SPtrSPtrSInt`, to another recoverable pointer to a recoverable integer type variable is declared.

```
class StatePtr< StatePtr< State<int> > >
      SPtrSPtrSInt;
```

Now, consider the following assignment statements,

```
SPtrSPtrSInt = &SPtrSInt;
*SPtrSPtrSInt = &SInt2;
**SPtrSPtrSInt = 200;
```

In the first assignment `SPtrSPtrSInt` is saved, in the second `SPtrSInt` is saved, and in the third `SInt2` is saved.

3.2 Template Design Issues

When designing these templates, a number of important design considerations were taken into account.

1. The result should always be safe - all modifications of a recoverable object should be trailed. A recoverable object may be modified directly by write operations or indirectly via a pointer de-reference. All write operations are identified and overloaded to record a backtrail entry prior to the write operation. Recoverable objects modified indirectly by de-referencing a `class StatePtr` are also saved, by overloading the de-reference operator to return a pointer to the recoverable object whose overloaded assignment operator is invoked.

2. As a consequence of the first point, the system should be type safe, so that state accessed via function parameters, or pointers is safe. All recoverable objects are defined using `class State` and `class StatePtr`, that are parameterized by a specified type. Incorrect type usage will generate compile-time error messages, for instance,

```
void fn( int * );
fn( & a_int );
fn( & SInt1 ); // Error
```

A compile time error message is generated in the second function invocation, as the actual parameter is a pointer to a `class State<int>`, where as the formal parameter expects a pointer to an `int`.

3. It should be possible for the user to force the point of state saving and this should be safe and efficient. A type conversion function `Save()` is defined in the templates. It does two things. It forces a state save operation on the object's variable and returns a reference to the variable of the original base type. This reference can be subsequently used to modify the variable without incurring state saving overheads (recall that it is only necessary to record a variable on the backtrail the first time it is modified in each event).

Consider the example:

```
int & temp = SInt1.Save();
for (int i=0; i<100;i++)
    temp++;
```

In the above example, a reference type variable `temp` is initialized to refer to the `int` data type at the location of `SInt1`. `SInt1` is explicitly state saved only once prior to the entry into the `for` loop. The data of the recoverable object is modified within the loop without any records being entered into the backtrail.

4. Temporary variables on the stack must never be trailed as they may no longer exist upon rollback. In C^{++}, variables may be created on the stack due to pass by value parameter or declarations local to a function. Such variables have a temporary lifetime limited to the function execution. A rollback would cause an invalid value to be written into a stack location which once contained the temporary recoverable

variable. In pass by value parameter passing, a compiler invokes an object's copy constructor to create a temporary object that is an exact duplicate of the actual parameter. Pass by value temporary variables are prevented by making the copy constructors private. Local declarations are prevented by using an execution time check in the default constructors. No local state type variables may be declared once simulation event processing begins.

An unfortunate side effect of making the copy constructor private, is that recoverable object construction and initialization using another recoverable object generates a compile-time error message, as in the case of `SInt3` declaration below:

```
class State<int> SInt3( SInt2 ); // Error
```

Alternatively, the programmer can use an assignment to initialize a recoverable object as a copy of another, as shown:

```
class State<int> SInt3;
SInt3 = SInt2;
```

5. It should be possible to dynamicly create recoverable objects. Dynamic state allocation is enabled by providing an overloaded `new` operator in each of the template classes. The `new` invokes a rollback sensitive memory allocator to dynamicly construct a recoverable object.

6. The mechanism must be totally transparent when executed on a non-optimistic system. In a non-optimistic system, the template just returns the base type making all the recovery declarations invisible. So no state saving is performed on write operations.

4 RECOVER TYPE SPECIFIER

One of the problems in using templates parameterized by basic types, is that the programmer must individually declare all state variables. This can be quite cumbersome and inconvenient. To alleviate this, two new keywords "`recover`" and "`nonrecover`" are used as type-specifiers.

In the simplest form, a "`recover int I`" declaration translates to "`State<int> I`". In the context of pointer data types, a declaration "`recover int * IP`" specifies that both the pointer and the object pointed to are recoverable. This declaration translates to "`StatePtr< State<int> > IP`". Declarations of the form "`State<int> * Ip`" and "`StatePtr< int > iP`" need to be explicitly represented as "`recover int * nonrecover Ip`" and "`nonrecover int * recover iP`", respectively.

A preprocessor front end to the C^{++} compiler has been developed for SimKit which uses the new keywords to generate C^{++} code wherein state variables are individually declared using the template classes.

4.1 Semantics of Recover Type Specifier

The `recover` and `nonrecover` keywords may be used in a number of contexts.

1. The first context is in association with a `class-key`. A `class-key` can be one of the `class`, `struct` or `union` keywords (Ellis and Stroustrup 1990). By default all variables do not have a state saving attribute. Consider the example declarations:

```
class z{...};
recover class A{...};
class B : class z{...};
recover class C : class z{...};
recover class D : nonrecover class A{...};
class E : class A{...};
```

Objects instantiated from `class A` will have all their variables recoverable. The `recover` keyword qualifying `class C`, forces all variables including those of the base `class z` to be recoverable in any objects instantiated from `class C`. Objects instantiated from the derived `class D` will have variables declared in `class D` recoverable and, because of the `nonrecover` qualifier, variables declared in its base class non recoverable. Objects of `class E` will only have variables declared within `class A` recoverable. In the case of pointer type member variables in a class, the use of `recover` as a `class` qualifier, specifies that both the pointer and the object pointed to are recoverable, unless explicitly over-ridden by the use of `nonrecover` in the pointer declaration.

2. The second use of the keyword is as an aggregate declaration within a `class`, `struct` or `union`. Members within a class may be grouped together as `recover` and `nonrecover` types, using a similar mechanism to the access specifiers `public`, `protected` and `private`. For instance, in the next example, all of `I`, `J`, `K` and `D` are recoverable, but none of `i`, `j` and `d`.

```
class A {
  recover:
    int I, J, K;
    double D;
  nonrecover:
    int i, j;
    double d;
};
```

The use of such aggregate declarations in classes needs some care. For example in the declaration:

```
class F {
    int x, y, z;
  recover:
    int I, J, K;
```

```
    double D;
  nonrecover:
    int i, j, k;
    double d;
};
class G : recover class F { ... }
```

the variables x, y, z, I, J, K, and D will all be recoverable in the context of class G. However, because i, j, k, d were explicitly declared to be non-recoverable, their type is not modified and they remain non-recoverable.

Note that the variables x, y and z may be recoverable or non-recoverable depending upon the context of their usage. To deal with this correctly, the implementation automatically creates two versions of the class F, one recoverable and the other not. The correct version is used depending upon the context.

3. In specifying recover type functions, the recover type-specifer is applied to the function return type as well as to all the arguments passed to the function. The example declaration

```
int * f1(int i, int *p, int **pp) recover;
```

translates to,

```
State<int> * f1(
 int i,
 State<int> *p,
 StatePtr< State<int> > *pp
 );
```

The translation of types here is rather different to that in the other contexts. Because C^{++} passes parameters by value and because these are allocated as temporaries on the stack, they must not be made recoverable. The rule used is that the "outer-most" type declaration is left non-recoverable and all types within this are made recoverable. So the declaration int i is left non-recoverable. The declaration int *p is changed to a non-recoverable pointer to a recoverable integer. The declaration int **pp is changed to a non-recoverable pointer to a recoverable pointer to a recoverable integer type.

4. A recover keyword may also be used to derive types using the typedef construct. The new type-name may then be used to declare recoverable variables of the new derived type. For instance,

```
typedef recover int SIntType;
SIntType SInt1, SInt2;
```

4.2 Sample State Declaration

An example using recover is shown in Figure 1. class nm_output defines an output buffered logical

```
class nm_output : public sk_lp {
 public:
    nm_output( ... );
    void process(const sk_event *);
    void initialize( );
    void terminate(void);
    afr_link *link(void);
    int id(void);
    int index(void);
    double average_occup(int,sk_time)
    double std_dev_occup(int,sk_time);
    int cells_dropped(int,int);
    int max_occupancy(int);
    int curr_occupancy(int);
    int cells_sent(int, int);
    int cells_received(void);
    double average_ctd(int);
    double std_dev_ctd(int);
private:
    nm_node *My_node;
    afr_link * My_link;
    int My_index, Id;
    sk_time Cell_transm_delay;
    RPT_STYLE Rpt;
    char *Report_1, *Report_2;
    void reset_stats(sk_time);
    void report(sk_time);
    void report_formatted(sk_time);
    void report_wsv(sk_time);
  recover:
    nm_buffer *Buffer;
    int Cells_received;
    int Link_state;
    int Cells_sent[AFR_N_CLASS][2];
    afr_tally State_ctd[AFR_N_CLASS];
};
```

Figure 1: Recoverable State Declaration Example

process in an ATM switch in the ATM-TN simulation model (Unger et al. 1995). It is a typical example of the modifications necessary to ensure correct state saving in this extensive body of software. Note that **recover** is used only once in the example. In 6000 lines of C^{++} header source files, 35 **recover** and 5 **nonrecover** keywords were needed to define the recoverable state in the simulation model.

5 CONCLUSIONS

Two significant barriers to the adoption of PDES techniques such as Time Warp are the execution cost of state saving and the difficulties of writing correctly state saved Time Warp models. Previous work has shown that incremental state saving can be a low cost robust state saving mechanism. This paper has addressed the second problem of ensuring that state is saved safely, correctly and with minimal programmer effort.

ACKNOWLEDGMENTS

This work has been supported by grants from the Canadian Natural Sciences and Engineering Research Council and by the New Zealand Public Good Science Fund. We would also like to thank Steve Franks for many useful discussions.

APPENDIX: TEMPLATE CLASS DEFINITION

```
template <class T> class State {
 private:
  T _datum;
  State(const State<T> &) { }
 public:
  State() : _datum(NULL) { ... }
  State(T t) : _datum(t) { ... }
  ~State() { ... }
  operator T() {  return _datum; }
  T value() { return _datum; }
  State<T>& operator=(T t)
  { ISSMgr.save( &_datum );
    _datum = t;
    return *this;
  }
  State<T>& operator=(const State<T> &t)
  { ISSMgr.save( &_datum );
    _datum = t.value();
    return *this;
  }
   // All other forms of assignments
   // e.g. +=, *=, ++, --, ...
  T& Save()
  { ISSMgr.save( &_datum );
    return _datum;
  }
  State<T>* operator&()
  {  return this; }
};

template <class T> class StatePtr{
 private:
    T* _datum_ptr;
    StatePtr(const StatePtr<T> &)
    { /* NO OP */ }
 public:
  StatePtr() : _datum_ptr(NULL) { ... }
  StatePtr(T* p) : _datum_ptr(p)
  { ... }
  ~StatePtr() { ... }
  operator T*() { return _datum_ptr; }
  T* value() { return _datum_ptr; }
  StatePtr<T>& operator=(T* p)
  { ISSMgr.save( &_datum_ptr );
    _datum_ptr = p;
    _datum_ptr = p; return *this;
  }
  StatePtr<T>&
      operator=(const StatePtr<T>& p)
  { ISSMgr.save( &_datum_ptr );
    _datum_ptr = p.value();
    return *this;
  }
   // All other forms of assignments
   // e.g. +=, *=, ++, --
  T* & Save()
  { ISSMgr.save( &_datum_ptr );
    return _datum_ptr;
  }
  StatePtr<T>* operator&()
  {  return this; }

  T& operator*()
  { return *_datum_ptr; }
};
```

REFERENCES

Bauer, H., and C. Sporer. 1993. Reducing rollback overhead in Time Warp based distributed simulation with optimized incremental state saving. In *Proceedings of the 26th Annual Simulation Symposium*, ed. J. Miller, 12–20. Arlington, Virginia.

Bruce, D. 1995. Treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS95)*, ed. Y-

B. Lin, and M. Bailey, 40–49. Lake Placid, New York.

Cleary, J., F. Gomes, B. Unger, X. Zhonge, and R. Thudt. 1994. Cost of state saving and rollback. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94)*, ed. D. Arvind, R. Bagrodia, and Y-B. Lin, 24(1):94–101. Edinburgh, Scotland, U.K.

Ellis, M., and B. Stroustrup. 1990. *The annotated C++ reference manual.* Addison-Wesley.

Fleischmann, J., and P. Wilsey. 1995. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS95)*, ed. Y-B. Lin, and M. Bailey, 50–58. Lake Placid, New York.

Fujimoto, R. 1989. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239.

Fujimoto, R. 1990. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53.

Gomes, F., S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington. SimKit: A high performance logical process simulation class library in C++. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. Lilegdon, and D. Goldsman, 706–713. Arlington, Virginia.

Gomes, F., and B. Unger. 1994. Benchmarking Shared Memory Time Warp with Signalling System No. 7 Performance Model – 3. Project Report CPSC 601.24, Department of Computer Science, University of Calgary, Canada.

Jefferson, D. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425.

Lin, Y-B., B. Preiss, W. Loucks, and E. Lazowska. 1993. Selecting the checkpoint interval in Time Warp simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93)*, ed. R. Bagrodia, and D. Jefferson, 23(1):3–10. San Diego, California.

Preiss, B., W. Loucks, and I. MacIntyre. 1992. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253.

Rönngren, R. and R. Ayani. 1994. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94)*, ed. D. Arvind, R. Bagrodia, and Y-B. Lin, 24(1):110–117. Edinburgh, Scotland, U.K.

Steinman, J. 1993. Incremental state saving in SPEEDES using C++. In *Proceedings of the 1993 Winter Simulation Conference*, ed. G. Evans, M. Mollaghasemi, E. Russell, and W. Biles, 687–696. Los Angeles, California.

Stroustrup, B. 1988. Parameterized types for C++. In *Proceedings of the 1988 USENIX C++ Conference*, 1–18.

Unger, B., F. Gomes, Z. Xiao, P. Gburzynski, T. Ono-Tesfaye, S. Ramaswamy, C. Williamson, and A. Covington. 1995. A high fidelity ATM traffic and network simulator. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. Lilegdon, and D. Goldsman, 996–1003. Arlington, Virginia.

Xiao, Z. and B. Unger. 1995. Performance of ATM-TN Wnet model. TeleSim Project Report ATM1.1-PR, Department of Computer Science, University of Calgary, Canada: WurcNet Inc.

AUTHOR BIOGRAPHIES

FABIAN GOMES is a post doctorate fellow in the Department of Computer Science at the University of Calgary. He received his Ph.D. degree in Computer Science from the University of Calgary in 1996. His research interests are in parallel simulation, modeling ATM broadband networks, distributed systems and rollback based computing.

BRIAN UNGER is a professor in the Department of Computer Science at the University of Calgary and is the President of WurcNet Inc. His current interests include the modeling and simulation of ATM broadband networks, parallel and distributed simulation, and simulation in Java.

JOHN CLEARY is a professor in the Department of Computer Science at Waikato University. His research interests include parallel and distributed systems, logic programming, complexity theory applied to adaptive and learning systems, and parallel discrete event simulation.