On the Construction of Complex Models Using Reusable Components

Peter MacSween
Defence R&D Canada Ottawa
3701 Carling Avenue
Ottawa, ON. K1A 0Z4. Canada.
pattyandpete@ieee.org

Gabriel Wainer
Dept. of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.
gwainer@sce.carleton.ca

Keywords: DEVS, Cell-DEVS, reusable components.

ABSTRACT: We show how to apply a development technique to build complex simulation models using a systematic method in which a model evolves incrementally. The method relies on a modular, hierarchical view, in which a higher-level model consists of a set of lower-level interactions. This view enables the reuse of simulations and components, where the integration. The approach relies on the use of DEVS methodology and it is supported by the use of CD++, a DEVS tool that has been built following the formal definitions of DEVS and Cell-DEVS. We will show how to use CD++ for common understanding, sharing and interoperability of model implementations, focusing in different model examples in the area of Defence. We will introduce means of developing independent models that can be integrated at the level of DEVS interactions, and will discuss how CD++ models can be composed into simulations that can execute in distributed environments.

1. Introduction

In recent years, we have witnessed tremendous advances in model building and simulation execution thanks to the improvements in software and hardware technology. The definition of the High Level Architecture (HLA) standard [1] put into discussion fundamental issues, such as model credibility and interoperation.

The HLA focuses the on interoperation of existing geographically dispersed simulation assets. However, the HLA is not addressing how to solve the problem on creating the models to be executed in the simulation environment. Current practices in development still use ad-hoc techniques, trying to encapsulate models, simulators and experimental frames into tightly coupled packages. As a result, testing, maintenance and software reuse become a hard task.

At present, there is a need to solve these problems, enabling interoperability (including computer-based and analog-based simulations), model reuse (using centralized or distributed repositories), while keeping high performance in the model execution. There are different efforts addressing these issues, for instance, the Base Object Model specifications, C4ISR, Extensible Modeling and Simulation Framework, Simulation Conceptual Modeling, etc. [2]. Other efforts consider the use of widely used standards like the UML, or simulation languages including support for execution on the RTI. Our proposal, instead, is based on the use of the DEVS formalism [3].

DEVS (Discrete Event systems Specifications) allows modular description of models that can be integrated using a hierarchical approach. DEVS has been successfully used in previous efforts in model interoperability (see, for instance, [4, 5]) providing ease for reuse of simulation models. Another advantage of using DEVS is that different existing techniques (Bond Graphs, Cellular Automata, State Charts, Partial Differential Equations, Petri Nets, Queuing models, Timed Automata, etc.) have been mapped to DEVS. This permits sharing information at the level of the model, and different submodels can be specified using different techniques, while keeping independence at the level of the simulation engine. Existing DEVS tools have showed their ability to execute such wide variety of models with high performance in standalone or distributed environments.

We will show how to integrate different models defined independently within a DEVS framework. We will incrementally build such a model with support for

interoperability at the level of the model. A first model shows synchronization effects between radar receivers and transmitters. A second example describes the behavior of a simple vehicle, which seeks a target. This cellular model was defined using Cell-DEVS [6], a technique created to describe cell spaces as a DEVS models including explicit timing delays.

Our experiences were carried out on the CD++ environment [7, 8], which implements the DEVS and Cell-DEVS concepts. CD++ has been recently extended, and a CD++ wrapper enables executing these models on top of the HLA [9], enabling distributed simulation of DEVS models using the HLA as the enabling platform for distribution and coordination. We will show how two models built independently can be easily integrated, improving the facilities for reuse and interoperation, using CD++ as the development environment for the experiences.

2. Background

DEVS is an increasingly accepted framework for understanding and supporting the activities of modeling and simulation. DEVS is a sound formal framework based on generic dynamic systems, including well defined coupling of components, hierarchical, modular construction, support for discrete event approximation of continuous systems and support for repository reuse. DEVS theory provides a rigorous methodology for representing models, and it does present an abstract way of thinking about the world with independence of the simulation mechanisms, underlying hardware and middleware.

A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral (atomic) or structural (coupled). A DEVS atomic model is can be informally described as in Figure 1.

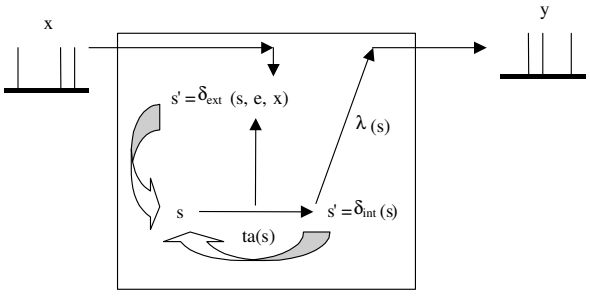


Figure 1. Informal description of an atomic model.

Each atomic model can be seen as having an interface consisting of *input* (x) and *output* (y) *ports* to communicate with other models. Every *state* (**s**) in the model is associated with a *time advance* (**ta**) function, which determines the duration of the state. Once the time assigned to the state is consumed, an internal transition is triggered. At that moment, the model execution results are spread through the model's output ports by activating an *output function* ($\boldsymbol{\lambda}$). Then, an *internal transition function* ($\boldsymbol{\delta_{int}}$) is fired, producing a local state change. Input external events (those events received from other models) are collected in the input ports. An external transition function ($\boldsymbol{\delta_{ext}}$) specifies how to react to those inputs.

A DEVS coupled model is composed by several atomic or coupled submodels, as seen in Figure 2.

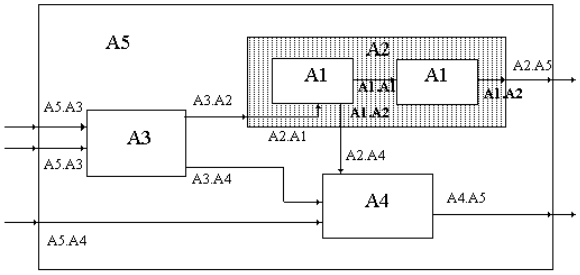


Figure 2. Informal description of a coupled model.

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model's interfaces. The model's coupling defines how to convert the outputs of a model into inputs for the others, and to inputs/outputs to the exterior of the model.

Cell-DEVS [6] has extended the DEVS formalism, allowing the implementation of cellular models with timing delays. A cellular model is a lattice of cells holding state variables and a computing apparatus, which is in charge of update the cell state according to a local rule. This is done using the present cell state and those of a finite set of nearby cells (called its neighborhood). Cell-DEVS improves execution performance of cellular models by using a discrete-event approach. It also enhances the cell's timing definition by making it more expressive [10]. Each cell is defined as a DEVS atomic model, and it can be later integrated to a coupled model representing the cell space. Cell-DEVS atomic models are informally defined as in Figure 3.

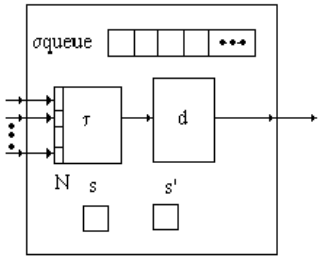


Figure 3. Description of a Cell-DEVS atomic model.

Each cell uses N inputs to compute its next state. These inputs, which are received through the model's interface,

activate a local computing function (**τ**). A delay (**d**) can be associated with each cell. The state (**s**) changes can be transmitted to other models, but only after the consumption of this delay. Two kinds of delays can be defined: *transport* delays model a variable commuting time (using a **queue** to keep every cell change), and *inertial* delays, which have preemptive semantics (scheduled events can be discarded).

Once the cell behavior is defined, a coupled Cell-DEVS can be created by putting together a number of cells interconnected by a neighborhood relationship. A Cell-DEVS coupled model is informally presented in Figure 4.

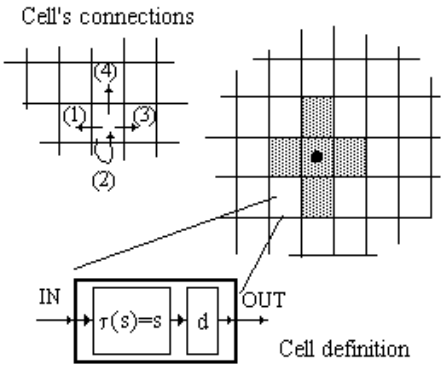


Figure 4. Description of a Cell-DEVS atomic model.

A coupled Cell-DEVS is composed of an array of atomic cells, with given size and dimensions. Each cell is connected to its neighborhood through standard DEVS input/output ports. Border cells have a different behavior due to their particular locations, which result in a non-uniform neighborhood. Finally, the model's couplings can be defined.

3. The CD++ toolkit

CD++ [7, 8] is a modeling tool that was defined using the specifications presented in the previous section, and the basic simulation techniques introduced in [3, 6]. The toolkit includes facilities to build DEVS and Cell-DEVS models. DEVS Atomic models can be programmed and incorporated onto a class hierarchy programmed in C++. Coupled models can be defined using a built-in specification language. Cell-DEVS models are built following the formal specifications for DEVS models (informally presented in the previous section), and a built-in language is provided to describe them. CD++ makes use of the independence between modeling and simulation provided by DEVS, and different simulation engines have been defined for the platform: a stand-alone version [7], a Real-Time simulator [11], and a Parallel simulator [12]. At present, a CD++ wrapper has been built, enabling CD++ simulations to run as HLA federates [9, 13], and the simulation engine is being extended to support distributed simulation of atomic models using the HLA. Another current effort is focused in providing support for development of real-time simulation in embedded platforms [14].

CD++ is built as a class hierarchy of models related with simulation processing entities. DEVS Atomic models can be programmed and incorporated onto the *Model* basic class hierarchy using C++. A new atomic model is created as a new class that inherits from the *Atomic* base class. The state of a model is defined in the *AtomicState* class. When creating a new atomic model, a new class derived from *Atomic* has to be created.

```
class Atomic : public Model  {
public:
virtual ~Atomic();     // Destructor

protected:
//Kernel services
Time nextChange();
Time lastChange();
holdIn(AtomicState::State &, Time &);
passivate();
ModelState* getCurrentState() ;
sendOutput(Time &time, Port &port, Value value);


//User defined functions.
initFunction();
externalFunction(ExternalMessage & );
internalFunction(InternalMessage & );
outputFunction(CollectMessage & );
string className() const
};      // class Atomic
```

Figure 5. The Atomic Class

Atomic is an abstract class that declares a model's API and defines some service functions the user can use to write the model. The *Atomic* class provides a set of services and requires a set of functions to be redefined:

- **`nextChange()/lastChange()`**: return the time until the next internal transition/since the last state change.

- **`holdIn(state, Time)`**: tells the simulator that the model remains in a *state* during a given *Time*. It corresponds to the ta(s) function of DEVS.

- **`passivate()`**: sets the next internal transition time to infinity. The model will only be activated again if an external event is received.

- **`getCurrentState()`**: returns the current model's phase.

- **`sendOutput(Time, port, value)`**: sends an output message through the specified *port*.

The new class should override the following functions:

- initFunction(): method invoked by the simulator at the beginning the simulation.

- externalFunction(ExternalMessage &): method invoked when an external event arrives to a port. It corresponds to the δ_{ext} function of the DEVS formalism.

- internalFunction(InternalMessage &): method defining the δ_{int} function of the DEVS formalism.

- outputFunction(const CollectMessage &): in charge of transmitting the output events of the model. It corresponds to the λ function of the DEVS formalism.

Once an atomic model is defined, it can be combined with others into a multicomponent model using a specification language specially defined with this purpose. The coupled model at the higher level is always named `[top]`. Four properties must be configured: components, output ports, input ports and links between models. The following syntax is used:

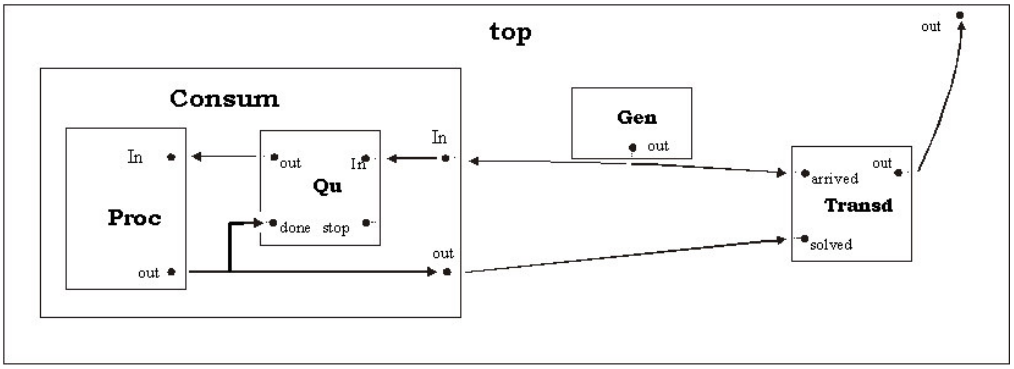
Components: `name1[@atomicClass1] name2 ...`
Lists the components of the coupled model (atomic or coupled). For atomic models, an instance and a class name must be specified, allowing a coupled model to use more than one instance of a given atomic class. For coupled models, only the model name must be given, and it must be defined as another group in the same file.

Out: `portname1 portname2 ...`
Enumerates the model's output ports (optional clause).

In: `portname1 portname2 ...`
Enumerates the input ports (optional clause).

Link: `source[@model] destination[@model].`
It describes the internal and external coupling scheme. If the name of the model is not included, the default will be the coupled model currently being defined.

The following figure shows a sample coupled model and its specification in CD++. It defines a simple queuing system consisting of three models: a generator, in charge of creating requests for service, a consumer servicing them, and a transducer, in charge of computer metrics for the system. The consumer is also a coupled model, composed by an entity processing them, and a queue to keep waiting jobs.




```
[top]
components : Transd@Transducer Gen@Generator Consum
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out
```

```
[Consum]
components : Qu@Queue Proc@CPU
in : in
out : out
Link : in in@qu
Link : out@qu in@Proc
Link : out@Proc done@qu
Link : out@Proc out
```

Figure 6. Definition of a DEVS coupled model in CD++.

In the top level of this example, the *Generator* outputs will be converted into inputs for the *Transducer* and the *Consumer*. The *Consumer* also generates outputs for the *Transducer*. Likewise, the *Consumer* and the *Processor* influences the *Queue*, which influences the *Processor*. Finally, the *Transducer* influences the top model.

CD++ also includes an interpreter for Cell-DEVS models. The language is based on the formal specifications of Cell-DEVS [6]. The model specification includes the definition of the size and dimension of the cell space, the shape of the neighborhood and borders, as presented in figure 4. The cell's local computing function is defined using a set of rules with the form:

POSTCONDITION DELAY { PRECONDITION }

These indicate that when the *PRECONDITION* is satisfied, the state of the cell will change to the designated *POSTCONDITION*, whose computed value will be transmitted to other components after consuming the *DELAY*. If the precondition is *false*, the next rule in the list is evaluated until a rule is satisfied or there are no more rules. The following figure shows the definition of a very simple example implementing the "Life" game [15].

```
[life]
type : cell
width : 20              height : 20
delay : transport
border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1)
neighbors : (0,-1)  (0,0)  (0,1)
neighbors : (1,-1)  (1,0)  (1,1)
localtransition : new-life-rule

[new-life-rule]
Rule: 1 10 { (0,0) = 1 and ( truecount = 3
                  or truecount = 4 ) }
Rule: 1 10 { (0,0) = 0 and truecount = 3 }
Rule: 0 10 { t }
```

Figure 7. Definition of the Life game.

The rules in this example say that a cell remains active when the number of active neighbors is 3 or 4 (*truecount* indicates the number of active neighbors) using a transport delay of 10 ms.. If the cell is inactive (`(0,0) = 0`) and the neighborhood has 3 active cells, the cell activated (represented by a value of 1 in the cell). In every other case, the cell remains inactive (*t* indicates that whenever the rule is evaluated, a *True* value is returned).

Complex cellular models can be defined with simple rules (see, for instance, [16, 17, 18, 19, 20]), using the various operations included in the language.

4. Incremental development of a DEVS simulation model

We have built a simulation model to integrate components of a Radar system model. The first stage in the definition of this example consisted on building a model to examine the synchronization effects between radar receivers and transmitters. When using a scanning radar receiver, the interception of radar signals can be severely limited if the scan rate of the receiver becomes synchronized with a radar transmitter. Every effort must be made to generate a receiver scan pattern that limits this effect, as it seriously degrades the Probability of Intercept (POI) for the receiver.

Synchronization occurs when a particular transmitter sends out radar pulses periodically, with the receiver scheduled to scan periodically in such a manner that the receiver is never 'listening' when the transmitter is transmitting. This can lead to the transmitter *not* being detected by the receiver, even though it may be transmitting. Radar transmitters transmit on a particular frequency, with a particular pulse rate, azimuth, and beam width. Scanning Radar Receivers receive on a tuned frequency (for a specified duration), with a particular azimuth and beam width, and have a 'tuning time' associated with the change from one listening frequency to another. The sequential operation of the receiver that defines the tuned-frequency, listening-time, azimuth, and beam width are specified by a 'scan pattern'.

Receivers can communicate with each other, with each receiver notifying the other receivers about radar transmitters that have been detected. Each receiver is connected to a simple communications bus, and it maintains a tracking table containing all the information about the currently known transmitters.

In order to analyze the behavior or this system, we built a DEVS model, whose structure is the one presented in the following figure.

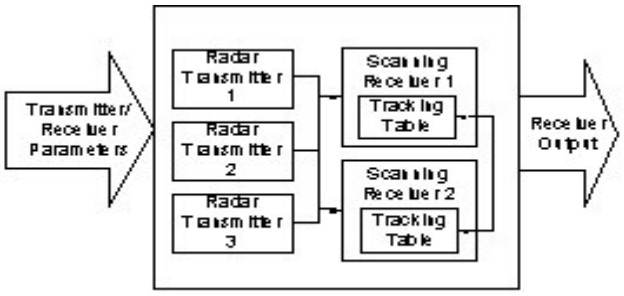


Figure 8. Structure of the Radar Tx/Rx model.

The first step was to identify and define each one of the model components. Once identified, a DEVS atomic model was built for each subcomponent. Following, we exemplify the definition of one of these models by showing the Tracking Table atomic model. The Tracking Table model is responsible for maintaining the list of transmitters that are 'known' to the local receiver.

Tracking Table = < S, X, Y, δ_{int}, δ_{ext}, ta, λ >

S = { Receive_Update_From_Bus,
Wait,
New_Signal_Detected,
Send_Update_To_Bus,
Notify_New_Freq }

X = { signal_props,
bus_receive_freq,
bus_receive_id }

Y = { bus_send_freq,
bus_send_id,

new_freq }

δ_{int} = { δ_{int}(Receive_Update_From_Bus) = Notify_New_Freq,
δ_{int}(Notify_New_Freq) = Wait,
δ_{int}(New_Signal_Detected) = Send_Update_To_Bus,
δ_{int}(Send_Update_To_Bus) = Wait }

δ_{ext} = { δ_{ext}(Wait, signal_props) = New_Signal_Detected,
δ_{ext}(Wait,bus_receive_freq) = Receive_Update_From_Bus }

ta = { ta(Receive_Update_From_Bus) = UPDATE_TIME,
ta(New_Signal_Detected) = PROCESS_TIME,
ta(Send_Update_To_Bus) = BUS_TIME,
ta(Notify_New_Freq) = NOTIFY_TIME }

λ(S) = { λ(New_Signal_Detected) = (bus_send_freq,bus_send_id),
λ(Receive_Update_From_Bus) = new_freq }

This model evolves through different states (S): Receive an update from the bus, wait, detection of a new signal, transmission of an update to the bus, or notification of a new frequency. The model changes from one state to the other by executing the transition functions. As seen in the external transition (δ_{ext}), from a *wait* state, the tracking table receives information from either the local receiver (*signal props,* one of the external input events) or the communication bus (*receive freq*). If the local receiver detects a new signal, the signal is appended to the local tracking table, and an update is sent over the bus for use by any remote tracking tables. If the local tracking table receives an update from the bus, it appends the information to the local tracking table and notifies the local receiver. The Tracking Table then returns to a wait state.

The model was subsequently built in CD++ using the state machine specification presented in the following figure. CD++ shows two views of the state machine: the left side of the GUI contains a sorted tree diagram, and the right side contains a visual representation of the model. The five states of the tracking table are immediately apparent. External transitions are displayed as dashed lines, with internal transitions as solid lines. The input and output ports are visible in the tree diagram.

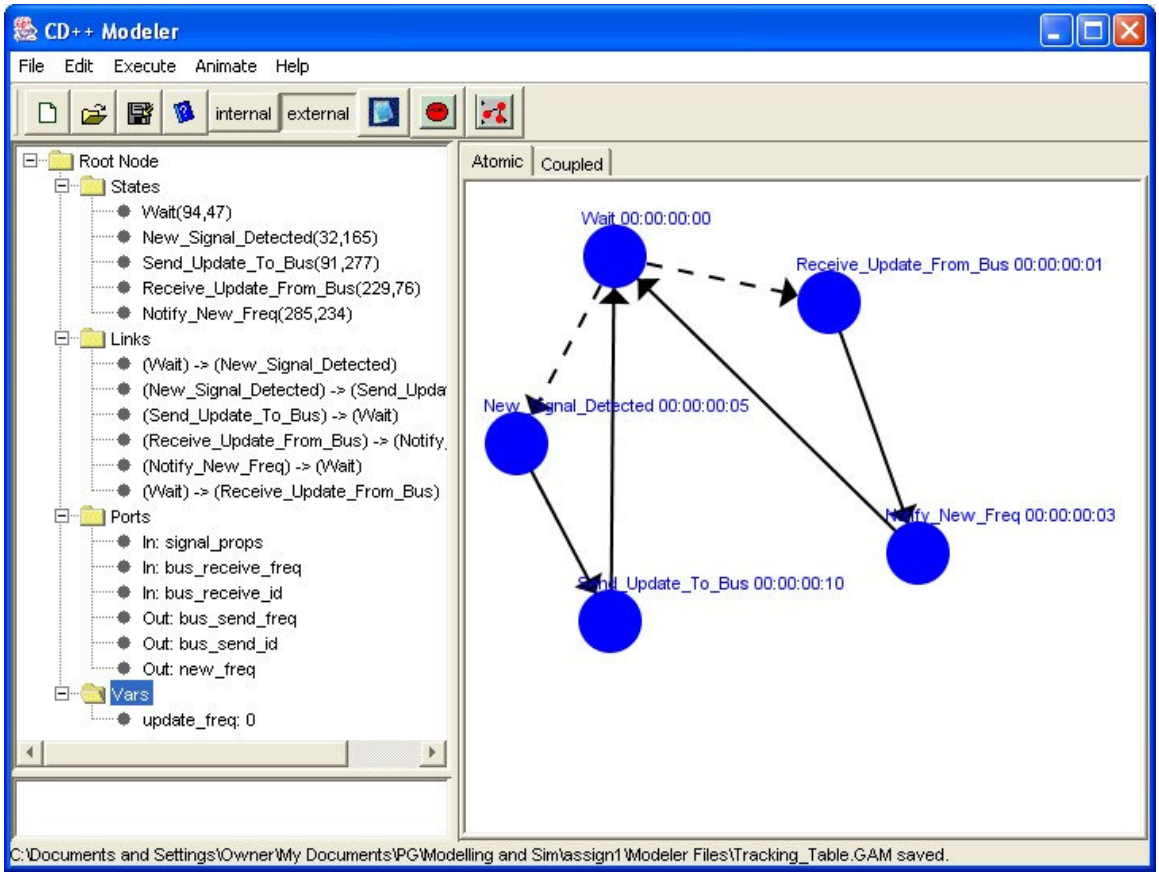


Figure 9. Specification of the Tracking Table model.

Each of the models presented in Figure 8 were thoroughly tested, and they performed as described in their conceptual model specifications (further details about the definition of each of the models can be found in [21]). A problem with the specification of the network receiver was revealed while testing (the tracing of the signals that are received by the network receivers became very difficult when numerous signals are transmitted, and the receivers start to share information).

```
[top]
components: tr1@Transmitter tr2@Transmitter
            tr3@Transmitter netrx1 netrx2
out: notify1 notify2 notify3
Link: pulse_out@tr1 ext_signal@netrx1
Link: pulse_out@tr1 ext_signal@netrx2
...
Link: notify@netrx1 notify1
Link: notify@netrx2 notify2
...

[netrx1]
components:tt1@Tracking_Table
rx1@Scanning_Receiver
in: ext_signal brf brid
out: notify bs_id bs_freq
Link: ext_signal ext_signal@rx1
Link: brf bus_receive_freq@tt1
Link: brid bus_receive_id@tt1
...
Link: bus_send_freq@tt1 bs_freq

[rx1]
freq_lower_bound: 18000
freq_upper_bound: 20000

[tt1]
table_id: 1

[netrx2]
components:tt2@Tracking_Table
rx2@Scanning_Receiver
in: ext_signal brf brid
out: notify bs_id bs_freq
Link: ext_signal ext_signal@rx2
Link: brf bus_receive_freq@tt2
Link: brid bus_receive_id@tt2
Link:detected_signal_properties@rx2
signal_props@tt2
Link: new_freq@tt2 ext_signal@rx2
Link: notify@rx2 notify
Link: bus_send_id@tt2 bs_id
Link: bus_send_freq@tt2 bs_freq

[rx2]
freq_lower_bound : 21000
freq_upper_bound : 22000

[tt2]
table_id : 2

[tr1]
frequency : 19000
pulseDuration : 00:00:00:5
pulsePeriod : 00:00:00:40
...
```

Figure 10. Coupled model definition: Radar Tx/Rx.

The use of the formal specification defining the atomic and coupled model behavior was very useful in debugging the models when they were implemented. The iterative procedure of updating the formal specification, then updating the implementation was quite efficient. Following these iterations resulted in the models matching the specifications.

Once this stage was completed, a Coupled Model was built, integrating all of the systems' components. The description of this model can be found in Figure 10. This coupled model defines the situation as described in Figure 8. Three Transmitter atomic models are configured with different frequencies and pulse characteristics (tr1, tr2, and tr3). Two network receivers (another coupled model containing a scanning receiver and tracking table) are configured (netrx1, netrx2), each listening to different frequency bands.

The various atomic models contained in the previously defined coupled model were tested using different scenarios, including the following:

- Transmitter: pulse sent at 22kHz, Pulse Width = 3 ms, Pulse Interval = 30 ms.
- Scanning Receiver set to listen for pulses between 18kHz and 25kHz.
- Tracking table: test that a signal is recorded and a bus message is sent, then test that bus messages are received correctly.
- Network Receiver: Test the reaction to a signal and to bus messages.
- Net_Of_ Network_Receiver_With_Transmitter:
  - Transmitters send on frequencies which are not being scanned:
  - Transmitters send on frequencies which are scanned by Network Receiver #1 only
  - Transmitters send on frequencies which are scanned by Network Receiver #2 only
  - Transmitters send on frequencies which are scanned by both Network Receivers

The following figure shows the result of the testing scenario for the network with a transmitter. In this case, the transmitter sends out pulses at 24kHz, Pulse width of 5 ms, Pulse interval of 40 ms. Bus Message at t=20 ms. Receiver listening between 22kHz and 25kHz.

As we can see in the figure, the receiver gets a signal from the transmitter every 40 ms, and a bus message at t=20ms. The bus message is ignored because it is not within the listening range of the receiver (19 kHz and the receiver is listening from 22 to 25 kHz). Note that the model does not queue received pulses or bus messages. For each pulse received by the local transmitter, a bus message is generated after a delay of 15 ms.. The bus message stays active for 40ms.

Events	Outputs
00:00:20 brf 19000 00:00:20 brid 3	00:00:001 notify 1 00:00:016 bs_id 1 00:00:016 bs_freq 24000 00:00:026 bs_id 0 00:00:026 bs_freq 0 00:00:041 notify 0 00:00:081 notify 1 00:00:096 bs_id 1 00:00:096 bs_freq 24000 00:00:106 bs_id 0 00:00:106 bs_freq 0 00:00:121 notify 0 00:00:161 notify 1 00:00:176 bs_id 1 00:00:176 bs_freq 24000 00:00:186 bs_id 0 00:00:186 bs_freq 0 00:00:201 notify 0 00:00:241 notify 1 00:00:256 bs_id 1 00:00:256 bs_freq 24000 00:00:266 bs_id 0 00:00:266 bs_freq 0 00:00:281 notify 0

Figure 11. Testing scenario: network with transmitter.

During this phase, we were able to detect a problem with the specification of the network receiver: the signal information received by the bus was sent to the scanning receiver, which treated it like an external signal (thus causing a second bus transmission). The specification was corrected so that signal information is not re-sent over the bus.

A different model, built using Cell-DEVS, describes the behavior of a simple vehicle, which seeks a target [22]. As showed in the following figure, the seeker acts to steer a vehicle towards a specified position in global space. This behavior adjusts the vehicle so that its velocity is radially aligned towards the target [22].

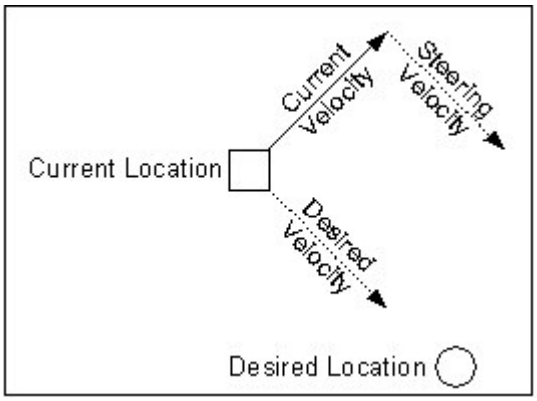


Figure 12. Informal behavior of the Seek model.

Using the hierarchy of motion behaviors defined in [22], the "Action Selection" of the seek behavior is specified by dictating the destination location. The Simple Vehicle Model has the following attributes:

{mass scalar, position vector, velocity vector, max_force scalar, max_speed scalar, orientation, N basis vectors}, where N=2

The motion of the model is defined by:
steering_force = truncate (steering_direction, max_force)
acceleration = steering_force / mass
velocity = truncate (velocity + acceleration, max_speed)
position = position + velocity

and the new basis vectors by:
new_forward = normalize (velocity)
approximate_up=normalize (approximate_up) // if needed
new_side = cross (new_forward, approximate_up)
new_up = cross (new_forward, new_side)

The seek behavior motion is defined by:
desired_velocity=normalize (position-target)*max_speed
steering = desired_velocity - velocity

To model the seek behavior using Cell-DEVS, it was necessary to create discrete states to represent the 'current' behavior of the simple vehicle. The following state variable was used:

State	Description
Current Velocity	A state indicating a vehicle with no velocity, or motion in one of 8 directions: moving diagonally up and left (value=1), up (2), diagonally up and right (3), left (4), stationary (5), right (6), diagonally down and left (7), down(8), diagonally down and right(9)

The model uses the following neighborhood definition:

N = { (-2,-2) (-2,-1) (-2, 0) (-2,1) (-2,2) (-1,-2) (-1,-1) (-1,0) (-1,1) (-1,2) (0,-2) (0,-1) (0, 0) (0,1) (0,2) (1,-2) (1,-1) (1, 0) (1,1) (1,2) (2,-2) (2,-1) (2, 0) (2,1) (2,2) }

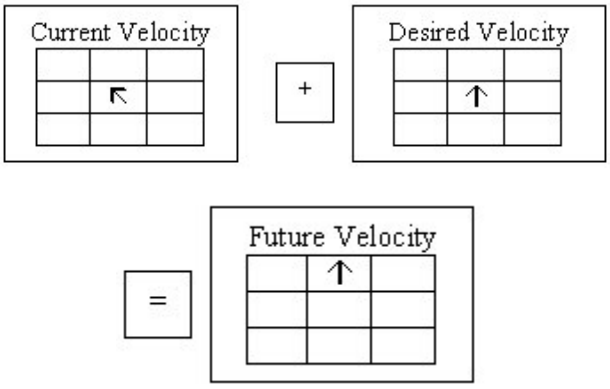


Figure 13. Definition of update rules.

An input was provided to each cell to specify the Desired Velocity of the vehicle. The model rules detail the discrete motion that was implemented to simulate the effect of a desired velocity on a vehicle. Multiple combinations of actual and desired velocity could result in the same destination cell for a vehicle.

With the many combinations of velocities, the possibility for collisions is great. The neighborhood for each cell is dependant on its velocity. A simple priority is used when multiple cells want move into the same cell. Stationary vehicles have the highest priority , "Up and left" have the lowest, and "Down and right" have the second highest.

The model was completely implemented in CD++ following the previous rule specifications, and it and first tested using a single vehicle, with different initial velocities and different desired velocities. After all the rules were implemented, all possible velocities were tested in all possible desired velocities. Following that, collisions were tested using multiple vehicles.

The following figures display the two state variables employed in the definition of the Cell-DEVS model (displayed side-by-side). The left-hand plane (mostly white) displays the current location and velocity of the three vehicles. The right-hand plane describes the 'desired velocity vector field' of the vehicles. The 'desired location' for all three vehicles is the center of the plane, and the 'desired velocity vectors' steer them to that point.

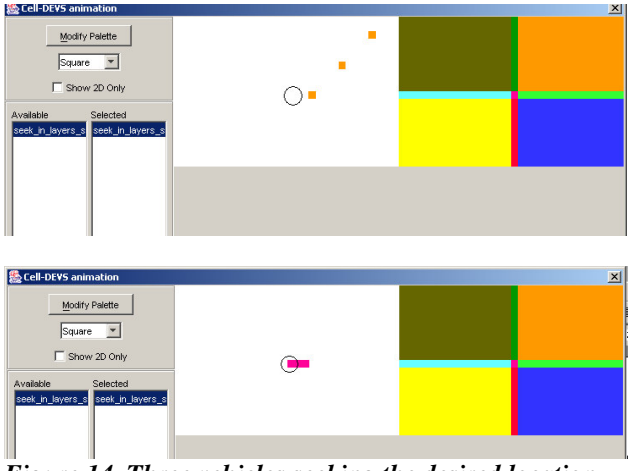


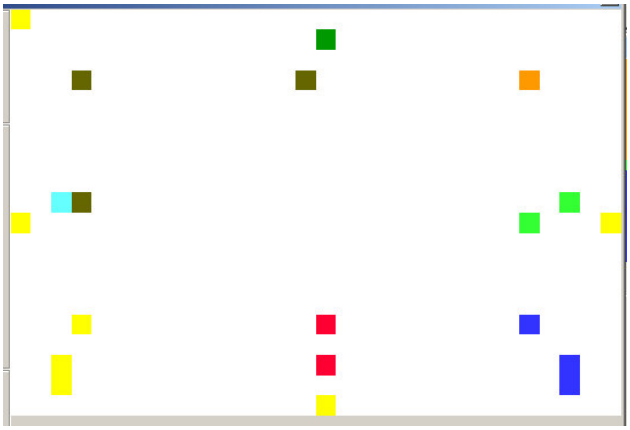
Figure 14. Three vehicles seeking the desired location.

As we can see, the three vehicles enter from the top-right corner of the plane, and they stop when they cannot move any closer to the 'desired location'.

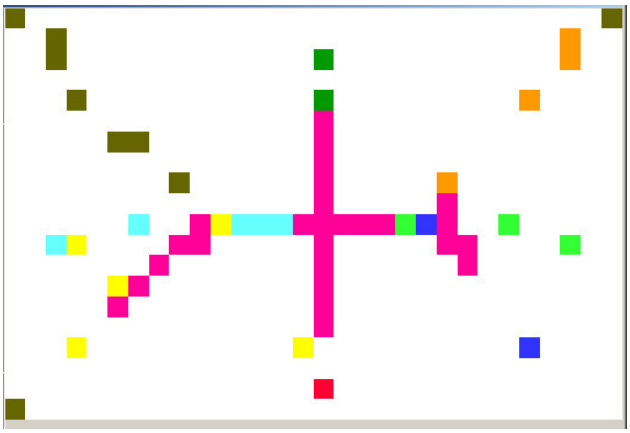
The vehicles enter (at time 0, 500, and 900 ms) with a velocity different from the desired velocity, and each acts in accordance with the state transitions to 'turn' to the desired velocity. At 1.2 seconds, the first vehicle enters a region with a different desired velocity. Note that the vehicle (and each subsequent vehicle) 'turns' to the desired velocity.

The first vehicle reaches the target cell at 1.5 seconds, and stops. The other vehicles follow the same path, and once they cannot move any closer to the target cell, they stop moving.

The following figures display a snapshot of the test where vehicles enter the plane from various locations and with different velocities (each must respond to the desired velocity in accordance with each current velocity).



(a)



(b)

Figure 15. Seekers with collision avoidance.

Initially (Figure 15.a), the vehicles moving towards the center. Then, a second set of vehicles (with different initial velocity) enters the plane. They are space closely together to generate collisions. The final steps present a vehicle 'jam' as the vehicles move towards the desired location.

The final stage of development consisted in showing how to provide interoperation of these models by allowing interaction of the components. This interaction is done at

the level of the model, and no contact with the simulation engine is needed, as the models only communicate at the level of their interfaces. Let us consider, for instance, the existence of a new model, *Radar*. The radar model is prepared to scan a cell space according to a given frequency. The following figure shows how to integrate this new model with the two other models defined earlier in this section. These three models were built independently, but they can be easily integrated thanks to the definition of DEVS interfaces.

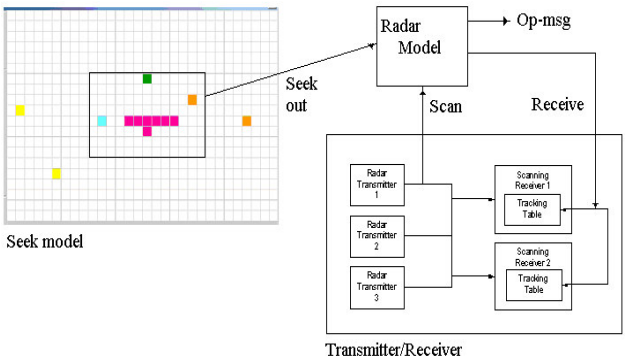


Figure 16. Multimodel integration.

The Transmitter/Receiver model is used to start radar scanning activities. Upon activation, the Radar will scan the field defined by the Seek Cell-DEVS model, and will generate two outputs: a reception signal for the Transmitter/Receiver, and a number of Operator Messages according to the values received in the field. The Seek model advances independently of the execution of the radar, because these models are built as discrete-event specifications, and each subcomponent progresses according its own internal time base. In CD++, the coupled model defining the composition of the submodels can be defined as in Figure 17.

As we can see, our *top* model is now integrated by the three original components. We initially define the model's coupling using the description presented in Figure 16. Then, we show the definition of the *seek* model (which is the one previously used to produce the simulation results presented in figures 14 and 15). The model produces outputs that can be used by the Radar model.

We have defined a *zone* in which the cells will generate outputs (by using the *out-rule* definition). Finally, the *Tx-Rx* model, defined earlier in Figure 10, includes two new input/output ports in order to provide interaction with the Radar model. This model is not defined in the file, as it has been defined as a DEVS atomic models, and we just need to define the coupling between this model and the remaining components.

```
[top]
components : seek Tx-Rx radar@Radar
out : op-msg
link : seek-out@seek seek-out@radar
link : op-msg@radar op-msg
link : scan@tx-rx scan@radar
link : receive@radar receive@tx-rx

[seek]
type : cell
dim : (20,30)
delay : transport
border : wrapped
out : seek-out
neighbors : (-2,-2) (-2,-1) (-2,0) (-2,1) (-2,2)
neighbors : (-1,-2) (-1,-1) (-1,0) (-1,1) (-1,2)
neighbors : (0,-2)  (0,-1)  (0,0)  (0,1)  (0,2)
neighbors : (1,-2)  (1,-1)  (1,0)  (1,1)  (1,2)
neighbors : (2,-2)  (2,-1)  (2,0)  (2,1)  (2,2)

% Cells producing outputs
zone : out-rule { (10,0)..(19,19) }
link : outTRP@seek(10,0)..(10,19) seek-out
localtransition : out-rule


[out-rule]
rule : {(send(1, y-t-room)} 0 { t }

localtransition : move-rule

[move-rule]
% Rules which do not allow a move to occur
(collision avoidance)
rule : 5 100 {
  % Moving up and left
  (
    ( ((0)=1) and ((0,1)=1) ) or
       ( ((0,0)=5) and ((0,1)=1) ) or
       ( ((0,0)=2) and ((0,1)=1) ) or
       ( ((0,0)=2) and ((0,1)=4) ) or
       ( ((0,0)=2) and ((0,1)=7) ) or
       ( ((0,0)=4) and ((0,1)=1) ) or
       ( ((0,0)=4) and ((0,1)=2) ) or
       ( ((0,0)=4) and ((0,1)=3) )
  )
... % Remaining moving rules

 [Tx-Rx]
components: tr1@Transmitter tr2@Transmitter
            tr3@Transmitter netrx1 netrx2
out : notify1 notify2 notify3 scan
in : receive
...
% Same as the previously defined model
```

Figure 17. Defining a multimodel in CD++.

This model can be executed in parallel Cell-DEVS by just defining a partition file as follows:

```
0 : Tx-Rx
1 : radar
2 : seek(0,0)..(9, 19)
3 : seek(10,0)..(19,19)
4 : seek(20,0)..(29,19)
```

Figure 18. Partition file for distributed execution.

This model will be split in 5 processors, which will execute different parts of the model driven by the CD++ simulator [6]. We can also split the three submodels and execute them using a CD++ HLA wrapper [13]. The wrapper is in change of creating a federate for each component, launch the execution of the federation, and synchronize the timing of the subcomponents using the RTI timing services (further information on how to use this wrapper can be found in [14]).

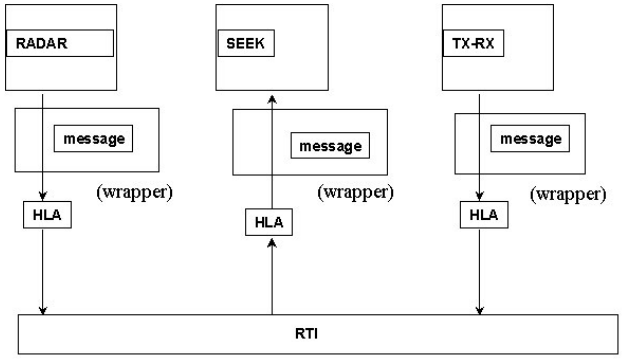


Figure 19. A multimodel in CD++/HLA wrapper [14].

5. Conclusion

We have showed a development technique to build complex simulation evolving incrementally from simple subcomponents to complex simulations. The method relies on the use of DEVS, which provides an incremental, hierarchical view. This view enables the reuse of simulations and components, where the integration of simulations and components is seamless.

The experiments were carried out using CD++, a DEVS tool that has been built following the formal definitions of DEVS and Cell-DEVS. The models were developed independently, and were later integrated at the modeling level. These models can be integrated into simulations that can execute in distributed environments. At present, we are working on an extension of CD++ that will be able to run distributed models on top of the RTI.

The use of DEVS can improve the security and cost in the development of the simulations. The main gains are in the testing and maintenance phases, the more expensive for these systems. The use of a formal approach like DEVS made easy the development of the applications.

We are currently working on a standardized version of DEVS models within a DEVS Study Group [23], whose goal is to provide a standardized version of DEVS models, enabling the multiple existing DEVS environments to interact, and to include non-DEVS models in larger simulations. In this way, we will be able to shorten the gap existing between academic versions of DEVS, and industrial/government needs. Having standardized means of defining models will enable defining standard libraries that can be integrated in user-friendly modeling and simulation environments.

References

[1] "IEEE Standard for Modeling and Simulation High Level Architecture (HLA)", IEEE Std. 1516-2000.

[2] SISO Product Development Activity. URL: `http://www.sisostds.org/stdsdev/index.cfm`. Checked Feb. 1, 2004.

[3] ZEIGLER, B.; KIM, T.; PRAEHOFER, H. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press. 2000.

[4] SARJOUGHIAN, H.S., ZEIGLER, B.P. "DEVS and HLA: Complimentary Paradigms for M&S?" Transactions of the SCS, (17), 4, pp. 187-197, 2000.

[5] KIM, Y. J., KIM, and T.G. " A Heterogeneous Simulation Framework Based on the DEVS BUS and the High Level Architecture". In *Proceedings of the Winter Simulation Conference*. Washington, DC. 1998.

[6] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modeling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", Springer-Verlag. 2001.

[7] WAINER, G. "CD++: a toolkit to define discrete-event models". G. Wainer. *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[8] CHRISTEN, G.; DOBNIEWSKI, A.; WAINER, G. "Defining DEVS models with the CD++ toolkit". In *Proceedings of European Simulation Symposium*. Marseilles, France. 2001.

[9] PEARCE, T. "Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems". Real-Time Systems Symposium. Work-in-Progress Session. Cancun, Mexico. 2003.

[10] WAINER, G.; GIAMBIASI, N. "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation". G. Wainer, N. Giambiasi. *Simulation*, Vol. 71, No. 1. January 2001. pp. 22-39.

[11] Glinsky, E.; Wainer, G. 2002. "Performance Analysis of Real-Time DEVS models". In

Proceedings of 2002 Winter Simulation Conference. San Diego, U.S.A.

[12] TROCCOLI, A.; WAINER, G. “Implementing Parallel Cell-DEVS”. In *Proceedings of Annual Simulation Symposium.* Orlando, FL. U.S.A. 2003.

[13] PEARCE, T.; ZHANG, C. “Federate Wrappers for the HLA”. Carleton University Technical Report SCE-02-04. 2004.

[14] YU, H.; WAINER, G. “Embedded CD++”. Carleton University Technical Report SCE-03-04. 2004.

[15] GARDNER, M. "The fantastic combinations of John Conway's New Solitaire Game 'Life'.". *Scientific American*. 23 (4). pp. 120-123. April 1970.

[16] AMEGHINO, J.; WAINER, G. "Application of the Cell-DEVS paradigm using CD++". In *Proceedings of the 32nd SCS Summer Computer Simulation Conference*. Vancouver, Canada. 2000.

[17] AMEGHINO, J.; TROCCOLI, A.; WAINER, G. "Modeling and simulation of complex physical systems using Cell-DEVS". In *Proceedings of the 33rd SCS Summer Computer Simulation Conference*. Seattle, WA. USA. 2001.

[18] AMEGHINO, J.; GLINSKY, E.; WAINER, G. “Applying Cell-DEVS in Models of Complex Systems”. In *Proceedings of Summer Simulation Multiconference*. Montreal, QC. Canada. 2003.

[19] MUZY, A.; WAINER, G.; INNOCENTI, E.; AIELLO, A.; SANTUCCI, J.F. "Cell-DEVS quantization techniques in a Fire Spreading application". In *Proceedings of 2002 Winter Simulation Conference*. San Diego, CA. USA. 2002.

[20] LO TARTARO, M.; TORRES, C.; WAINER, G. “Defining models of urban traffic using the TSC tool". In *Proceedings of 2001 Winter Simulation Conference*. Arlington, VA. USA. 2001.

[21] MacSWEEN, P.; WAINER, G. "Definition of DEVS and Cell-DEVS models for defence applications". Carleton University Technical Report SCE-06-04. 2004.

[22] REYNOLDS, C. W. “Steering Behaviors for Autonomous Characters”. `http://www.red.com/cwr/steer/gdc99`. Checked on December 2, 2003.

[23] The DEVS Standardization Study Group. `http://www.sce.carleton.ca/faculty/wainer/standard`. Checked February 1, 2004.

Peter MacSween is a Captain in the Canadian Forces (Defence R&D Canada) and is a M.A.Sc. student in the School of Information Technology and Engineering at the University of Ottawa. His research interests include the modelling and simulation of electronic signals. His email address is <pattyandpete@hotmail.com>.

Gabriel Wainer received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the Universidad de Buenos Aires, Argentina, and Université d’Aix-Marseille III, France. He is Assistant Professor in the Dept. of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada). He was Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires, and a visiting research scholar at the University of Arizona and LSIS, CNRS, France. He is author of a book on real-time systems and another on Discrete-Event simulation and more than 70 research articles. He is Associate Editor of the Transactions of the SCS. He is Associate Director of the Ottawa Center of The McLeod Institute of Simulation Sciences and a coordinator of an international group on DEVS standardization. His email and web addresses are gwainer@sce.carleton.ca and <www.sce.carleton.ca/faculty/wainer>.