GATLAS: GOOGLE EARTH VISUALIZATION FOR ATLAS

Ken Edwards **Gabriel Wainer**

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive, Ottawa, ON. K1S 5B6. Canada.
ken.edwards@cmcelectronics.ca gwainer@sce.carleton.ca

ABSTRACT
ATLAS is a modeling language that allows one to define a static view of a city section for simulating traffic in an area. By using ATLAS TSC, an intermediary compiler, and CD++, a Cell-DEVS system, traffic simulations may be run. The outputs of the simulation are a collection of individual cell-space simulation results that are difficult to analyze as a whole. This problem is solved by using GATLAS ((ATLAS in Google Earth) to generate KML files from the CD++ outputs so that the simulation results may be examined as a whole in Google Earth.

1 INTRODUCTION

ATLAS (Advanced Traffic Language Specifications) is a high-level specification language defined to represent city sections as cell spaces [1, 2]. The models are formally specified, avoiding a high number of errors in the application, thus reducing the problem solving time. ATLAS specifications were used as a basis to define the TSC compiler, which can be used to convert a city plan file (used as input) into a DEVS formal model [3], producing a coupled model file that can be simulated using the CD++ environment. [4].

Although a VRML-based visualization tool was defined in [2], new technologies for visualization are available based on Web Services and Service Oriented Architecture. The emergence of recent XML-based technologies paved the way for new types of architectures and message exchanges on the Internet. This eXtensible Markup Language has provided interoperability between partners and enabled companies to deploy a myriad of machine consumable Web-based services, which can be later integrated in many different ways to produce multiple sets of services. A system reusing existing distributed services and combining them to provide added value through a web application is called a Mash-Up. The idea of using existing web-based products as a visual aid in displaying other information is often referred to as a "mash-up". Although we have shown that mashups for modeling and simulation can be created [5] (building a web mash-up that uses Google Maps and web-based system for interacting with a CD++ model and simulation), Web Services technologies are still complex to mashup. In this work we show a mechanism to deal with a larger system, which focuses on the visualization of the results of a simulation in Google Earth.

2 BACKGROUND

In recent years, a variety of simulation languages and tools have been created, using different formal methods: queuing networks [6], DEVS [7], Cellular Automata [8], software agents, etc. Our research has focused on the construction of traffic microsimulations that describe precisely the local behavior of traffic, using DEVS and Cell-DEVS [3,4].

Cell-DEVS is an extension of DEVS, especially devoted to define cell spaces. Each cell is defined as an atomic DEVS, and a procedure to couple cells is depicted. Timing delay constructions let the modeler to define the cell timing behavior. Each cell, built as an atomic model, can be described as:

$$TDC = < X, Y, \theta, N, delay, d, \delta int, \delta ext, \tau, \lambda, ta >$$

X defines the external inputs, Y the external outputs. θ is the cell state definition, and N is the set of inputs. Delay defines the kind of delay for the cell, and d its duration. Finally, there are several functions: dint for internal transitions, dext for external transitions, τ, for local computations, λ for outputs and ta for the state's duration. Each cell uses the set of inputs to compute the cell's next state using the τ function. The delay allows to defer the transmission of the results. This behavior is defined by the dint, dext, λ and ta functions. A modeler only focuses in defining the local computing function, the kind of delay and its length.

ATLAS [1] is a specification language built on top of DEVS and Cell-DEVS formalisms. DEVS formalism permits to specify discrete events systems using a modular description. A model is seen as composed by atomic submodels than can be combined into coupled models. The behavior for each of the constructions presented in this language was validated in terms of their correctness when built as Cell-DEVS models. Then, a compiler was built following the specifications [9]. The compiler, called ATLAS TSC (Traffic Simulator Compiler), generates code by using a set of templates that can be redefined by the user. In this way, ATLAS specifications can be translated into different tools with facilities to define cellular models. It also avoids version problems if the underlying tools are modified.

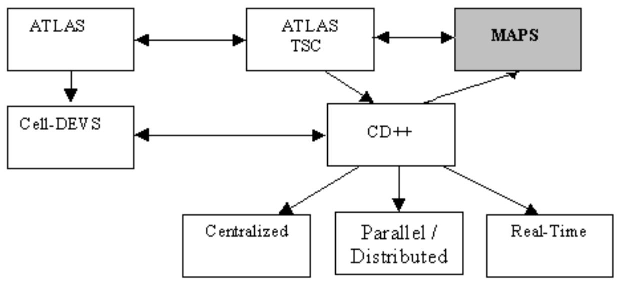

Figure 1: ATLAS Software Architecture

ATLAS allows representing the structure of a city section defined by a set of streets connected by crossings. The language constructions define a static view of the model. ATLAS formal specifications were used to build the ATLAS TSC compiler and the syntax for its language sentences. Following, we present the main constructions of ATLAS and its syntax in TSC.

a) Segments: represent sections of a street between two crossings. Every lane in a given segment has the same direction (one way) and a maximum speed. They are specified as: Segments = { (p1, p2, n, a, dir, max) / p1, p2 ∈ City ∧ n, max ∈ N ∧ a, dir ∈ {0,1} }, where **p1** and **p2** represent the boundaries of the segment (City = { (x,y) / x, y ∈ ***R*** }), **n** is the number of lanes, and **dir** represents the vehicle direction. The parameter **a** defines the shape of the segment (straight or curve, allowing to define the city shape more precisely, including the exact number of cells), and **max** is the maximum speed allowed in the segment. This constraint was included in ATLAS TSC. The compiler permits defining the segments by delimiting them using the sentences **`begin segments`** and **`end segments`**. At least one segment must be defined, using the following syntax:

```
id = p1,p2,lanes,shape,direction, speed, parkType
```

These values map the parameters mentioned previously, with **`shape`**: `[`**`curve`**`|`**`straight`**`]` and **`direction`**: `[`**`go`**`|`**`back`**`]`. Finally, **`parkType`** is used to define parking constructions, formally specified in the following paragraphs.

b) Parking: border cells in a segment can be used for parking. If we review the construction used for Segments in ATLAS TSC also includes information for the parking segments. In this case,

```
parkType: [parkNone|parkLeft|parkRight|parkBoth]
```

defines an area where vehicles can park.

c) Crossings: these constructions are used to represent the places where more than one segment intersects. They are specified as: Crossings = { (c, max) / c ∈ City ∧ max ∈ N ∧ ∃ s, s' ∈ Segments ∧ s = (p1, p2, n, a, dir, max) ∧ s' = (p1', p2', n', a', dir', max') ∧ s ≠ s' ∧ (p1 = c ∨ p2 = c) ∧ (p1' = c ∨ p2' = c) }. In ATLAS TSC, the definitions for crossings are delimited by the separators **`begin crossings`** and **`end crossings`**. Each sentence defines a crossing as:

```
id = p, speed, tLight, crossHole, pout
```

Parameters ***p*** and ***speed*** represent *(p1,p2)* and *max* of the formal specification. ***Pout*** defines the probability that a vehicle leaves the crossing, used to simulate random routing

d) Traffic lights: crossings with traffic lights define a set of models representing the traffic lights in a corner and the corresponding controller. The model sends a value representing the color of the traffic light to a cell in the intersection corresponding to the input segment affected by the traffic light. The following qualifier is added to a standard crossing definition in ATLAS TSC for crossings with traffic lights: **`tLight`**: `[`**`withTL`**`|`**`withoutTL`**`].`

e) Railways: they are built as a sequence of level crossings overlapped with the city segments. In ATLAS TSC, the **`begin railnets`** and **`end railnets`** act as separators. Each railnet is defined using the following syntax:

```
id = (s1, d1) {,(si, di)}
```

where $\mathbf{s_i}$ is the identifier of a segment crossed by the railway, and $\mathbf{d_i}$ is the distance between the beginning of the segment $\mathbf{s_i}$ and the railway. The compiler automatically generates the sequence number.

f) Men at work: In ATLAS TSC, the **`begin jobsites`** and **`end jobsites`** separators define an area with accidents or men at work. Each sit is defined as:

```
in t : firstlane, distance, lanes
```

Here, **firstlane** defines the first lane affected by the jobsite, **distance** is the distance between the center of the jobsite and the beginning of the segment, and **lanes** is the number of lanes occupied.

g) Traffic signs identify the segment where the traffic sign is used, the type of sign, and the distance from the beginning of the segment up to the sign. In ATLAS TSC, the **`begin ctrElements`** and **`end ctrElements`** delimiters define all the signs, with:

```
in t : ctrType, distance
```

being the definition for each sign. Here, **`ctrType`**: `[`**`bump`** `|` **`depression`** `|` **`intersection`** `|` **`saw`** `|` **`stop`** `|` **`school`**`].`

The **`distance`** parameter defines the distance to the beginning of the segment. An extension of this construction allows us to define potholes, whose size is one cell. The definition of these elements is done using the **`begin holes`** and **`end holes`** separators. Each hole is defined as:

```
    in t : lane, distance
```

A pothole can also be included in a crossing. Previously defined in the Crossings paragraphs, **`crossHole`**: `[`**`withHole`**`|`**`withoutHole`**`]` defines if a crossing contains a pothole or not.

h) Experimental frameworks: experimental framework constructions permit build experiments on a city section by providing inputs and outputs to the area to be studied. They are associated with segments receiving inputs, or those used as outputs

DEVS is a discrete event paradigm. It uses a continuous time base, which allows accurate timing representation. Precision of the conceptual models can be improved, and CPU time requirements reduced. The TSC compiler for the ATLAS specification language implements the ATLAS constructions as DEVS and Cell-DEVS models, using a generic rule generation mechanism for describing the traffic behavior. The compiler generates rules based on macro templates, entitling changes in the model implementation in a flexible way. The formal specification avoids a high number of errors in the developed application, and the problem solving time is highly reduced..

Google Maps is one of the leading online consumer mapping technologies. One of the features of Google Maps if viewing current traffic conditions, see Figure 2.

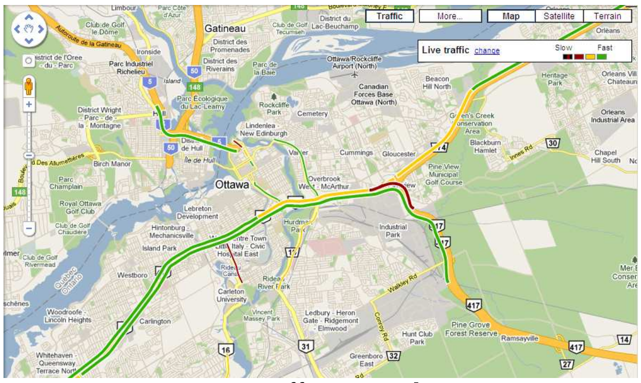

Figure 2 - Traffic in Google Maps

Although the presentation in Figure 2 is compelling, Google Maps are not ideal for simulation projects, because the Google Maps API has no sense of time. Therefore, visualizing simulation results over time would require an extension of the Google Maps API that took time-based events into account. Instead, Google Earth allows users to view satellite imagery, maps, terrain, and user-defined data on a model of the Earth and may be downloaded via `<http://earth.google.com/>.` Google Earth (originally named EarthViewer 3D) explicitly allows for time-based events though its use of KML, formerly Keyhole Markup Language. KML is an open standard officially named the OpenGIS® KML Encoding Standard (OGC KML). It is maintained by the Open Geospatial Consortium, Inc. (OGC), and its complete specification can be found at `<http://www.opengeospatial.org/standards/kml/>.`

Different KML entities can be used for modeling within a simulation, in particular *Style*, *GroundOverlay*, *Path* (LineString), *Point*, and *TimeSpan*.

Styles are defined so that they may be applied to objects in the KML file. A *GroundOverlay* is used to drape a user-defined image over the surface of the earth. This can be useful to use custom imagery (in our case, it is useful to block out the underlying GoogleEarth imagery because of alignment issues between the simulation and GoogleEarth).

We used various objects of type *LineString* to draw the traffic segments (sections of traffic between two corners).

A *Point* is used to define a singular location in KML. We have used a point to define the position of a car, and referenced the *carPlacemark* Style to draw an icon of a car at this point. Within this definition, we used the *TimeSpan* markup to define the start and the end time between which this object exists. The TimeSpan markup is used to animate the cars.

3 GATLAS IMPLEMENTATION

GATLAS (ATLAS in Google Earth) uses a number of Perl scripts in order to be able to process the text files containing the simulation results, the TSC files and the interaction with Google Earth. Three main data objects were created to create the visualization, one hash each for segments, crossings, and events.

The segments and crossings hashes each have as their first dimension key the name of the segment or crossing. The second dimension's key is a property name, one for each item in the description of the segment or crossing in the .plan file. One last key for each crossing or segment (*model*) contains a 2D array to model the cell space for that object. This is used to generate the visualization, as it will be discussed in detail later.

Initially, we build the necessary data structures, and then we pass them back and forth to various helper functions, in the following order:

1. Parse the TSC Plan File: this file contains the TSC definition for the simulation area.
2. Parse the DEVS Coupled Model File: this file contains the coupled model specification generated by ATLAS TSC.
3. Parse the Log File generated by CD++.
4. Build the corresponding KML file.

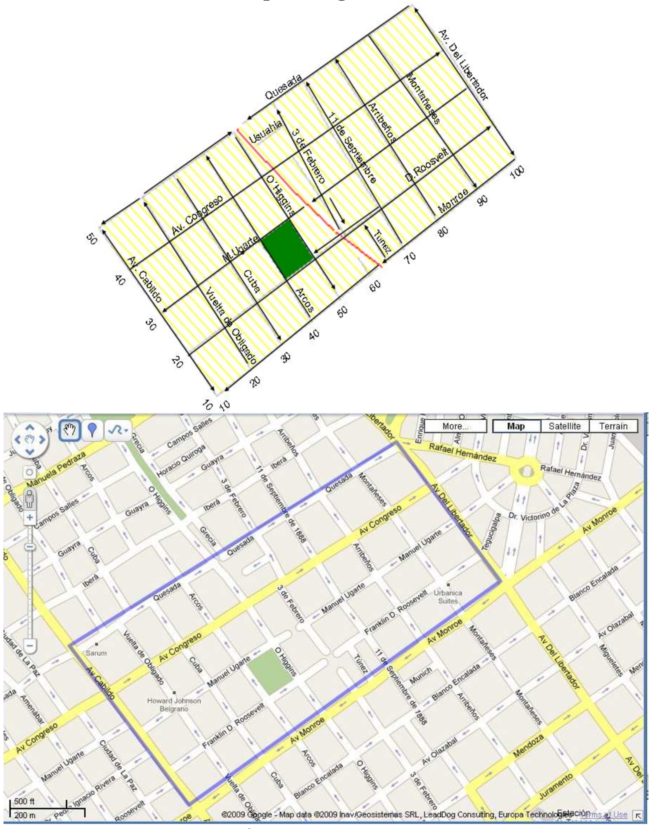

Figure 3 – A section of Buenos Aires and its corresponding Google Map.

Figure 3 shows the topology of an ATLAS model that represents a section of the city of Buenos Aires, Argentina. In Figure 4 we can see the definition of this model in ATLAS/TSC.

As we can see in the figure, each road segment includes the name of the street based on the city map (*Monroe*, *Roosevelt*, *Usuahia*, *Tunez*, etc.). Each of the road segments include the various parameters discussed earlier: the start/end points of the segment, the number of lanes on each segment, the shape of the segment (straight/curve), the direction of the vehicles (go/back), the maximum speed allowed on the segment, and parking information. We can also see the information about one of the crossings in the map (as it can be seen in the position array, this crossing ends at position 10 in the Y axis in the 2D plane). Each of the crossing constructions shows the connection to a different segment, and the kind of connection, which can include Traffic Lights or potholes).

```
begin segments
  Monroe_Exit = (0,10),(10,10),1, straight, back,
                             20,300,parkNone
  Monroe_In = (100,10),(110,10),1,straight,back,
                             20,300,parkNone
  Roosevelt_In= (0,20),(10,20),1,straight,go,20,
                             300,parkNone
  Roosevelt_Exit = (100,20),(110,20), 1,straight,
                             go,20,300,parkNone
...
 Libertador_In1=(100,0),(100,10),4,straight,go,20,
                             300,parkNone
 Libertador_Exit1=(100,50),(100,60),4,straight,go,
                             20,300,parkNone
 Libertador_Exit2 = (100,0),(100,10),4,straight,
                             back,20,300,parkNone
 Libertador_In2=(100,50),(100,60),4,straight,back,
                             20,300,parkNone
 Usuahia1 =(52,45),(60,45),1,straight,back,20,
                             300,parkNone
 Tunez = (65,10),(65,20),1,straight,go,20,300,
                             parkNone
end segments

begin crossings
  c010_10_ = (10,10),20,withoutTL,withoutHole,300
  c020_10_ = (20,10),20, withoutTL,withoutHole,300
  c030_10_ = (30,10),20,withoutTL,withoutHole,300
         ...
  c090_10_ = (90,10),20,withoutTL,withoutHole,300
  c100_10_ = (100,10),20,withoutTL,withoutHole,300
...
end crossings
```

Figure 4. ATLAS/TSC definition of the maps in Figure 3.

Based on this notation, the TSC compiler builds a DEVS coupled model with CD++ notation as follows.

```
components : BigCounter@TSCCounter Monroe_Exit
components : Monroe_ExitCons@TSCConsumer
components : Monroe_InGen@TSCGenerator Monroe_In
components : Roosevelt_InGen@TSCGenerator
components : Roosevelt_In
components : Roosevelt_ExitCons@TSCConsumer
...
components : c010_10_ c010_10_Counter@TSCCounter
...
out : arrived_BigCounter qty_Cons_Monroe_Exit
...
link : y_co_car09@Monroe_Exit
                     x_t_car0@Monroe_ExitCons
link : quantity@Monroe_ExitCons
                     qty_Cons_Monroe_Exit
link : quantityAcum@Monroe_ExitCons
                     qty_Cons_Acum_Monroe_Exit
[Monroe_Exit]
type : cell   width : 10    height : 1
delay : transport border : nowrapped
neighbors : (0,-1) (0,0) (0,1)
in : x_c_car00  x_c_canEnter00
out : y_c_space00  y_co_car09
link : x_c_car00 x_c_car@Monroe_Exit(0,0)
link : x_c_canEnter00
               x_c_canEnter@Monroe_Exit(0,0)
```

```
link : y_c_space@Monroe_Exit(0,0) y_c_space00
link : y_co_car@Monroe_Exit(0,9) y_co_car09
portInTransition : x_c_canEnter@Monroe_Exit(0,0)
              segment1-canEnter-startcross-rule
portInTransition : x_c_car@Monroe_Exit(0,0)
                       segment1-startcross-rule
localtransition : segment1-lane-rule
zone : segment1-cons-rule { (0,9) }

[c010_10_]
type : cell     width : 18       height : 1
delay : transport     border : wrapped
neighbors : (0,-1) (0,0) (0,1)
in : x_t_car0 x_t_car1 x_t_car2 ... x_t_car17
out : y_t_space0 y_t_space1 y_t_space2 ...
link : x_t_car0 x_t_car@c010_10_(0,0)
link : x_t_car1 x_t_car@c010_10_(0,1)
localtransition : cellIn-rule
portInTransition : x_t_car@c010_10_(0,0) car-rule
...
zone : c010_10_-cellOut-rule { (0,5) }
...
```

Figure 5. Translation of the maps in Figure 3 into CD++

This model contains all the rules for the model to execute; each segment and crossing is translated into a Cell-DEVS model, and they are interconnected through input/output ports. It first defines all of the components generated by TSC: a *TSCcounter* model used as an experimental framework connected to the the model exits, a *TSCGenerator* to generate traffic in the zone, and one Cell-DEVS model for each of the segments and crossing. For instance, we show the coupled model definition of *Monroe_Exit* , a 10x1 Cell-DEVS model that (is used to receive and transmite vehicles in the zone (using the links defined at the topmost level). We also show the definition of the crossing *c010_10_*, which is connected to each of the input/output segments in position (10,10) on the plane.

Based on the complete model specification, the following simulation results were obtained:

```
00:01:00:000 arrived_bigcounter 435
00:01:00:000 solved_bigcounter 220
00:01:00:000 qty_cons_monroe_out 1
00:01:00:000 qty_cons_acum_monroe_out 1
00:01:00:000 qty_cons_roosevelt_out 56
00:01:00:000 qty_cons_acum_roosevelt_out 56
00:01:00:000 qty_cons_ugarte_out 0
00:01:00:000 qty_cons_acum_ugarte_out 0
00:01:00:000 qty_cons_congreso_out 19
00:01:00:000 qty_cons_acum_congreso_out 19
00:01:00:000 qty_cons_usuahia1 0
00:01:00:000 qty_cons_acum_usuahia1 0
```

As we can see, in 1 simulated hour, 435 vehicles arrived in the area and 220 left the region (using different streets). Although this cumulative information is useful for statistical purposes, it does not provide any information on the microsimulation for each of the vehicles. This information, instead, can be found in the detailed simulation log showed in the following figure.

```
...
X / 00:00:00:010 / Root / x_t_car0 / 1 to top(01)
X / 00:00:00:010 / top(01) / x_t_car0 / 1 to
MonroeExit(02)
D / 00:00:00:010 / MonroeExit(02) / 00:00:59:990
to top(01)
D / 00:00:00:010 / top(01) / 00:00:59:990 to
Root(00)
X / 00:00:00:020 / Root(00) / x_t_car1 /      1 to
top(01)
X / 00:00:00:020 / top(01) / x_t_car1 /      1 to
MonroeExit(02)
D / 00:00:00:020 / MonroeExit(02) / 00:00:59:980
to top(01)
D / 00:00:00:020 / top(01) / 00:00:59:980 to
Root(00)
X / 00:00:00:030 / Root(00) / x_t_car2 /      1 to
top(01)
X / 00:00:00:030 / top(01) / x_t_car2 /      1 to
MonroeExit(02)
D / 00:00:00:030 / MonroeExit(02) / 00:00:59:970
to top(01)
…
```

Figure 6. Execution of the models using CD++ (.log files).

This figure shows two different outputs provided by the simulator: the first part includes a summary of the simulation results on the different streets; the second part shows a detailed log file showing the simulation execution at every single submodel and at every timestam.

These results were mashed up into a Google Map, using an advanced program (written in Perl due to the ease with which text may be processed). Three main data objects were created to create the visualization, one hash each for segments, crossings, and events. The segments and crossings hashes each have as their first dimension key the name of the segment or crossing. The second dimension's key is a property name, one for each item in the description of the segment or crossing in the .plan file. One last key for each crossing or segment is 'model' that contains a 2D array to model the cell space for that object. This is used to generate the visualization and is explained in detail later in this paper.

The main function of this program is to create the data structures and then passes them back and forth to various helper functions, which, in turn parse the Plan File, parse the corresponding MA and Log File, and then create a
KML file for visualization.

TSC takes a .plan file as input, containing a list of segments and crossings contained in the model. An example of a segment and a crossing definition from mapa.plan file is shown here:

```
Monroe_Exit = (0,10),(10,10),1,straight,back,20,
                                          300, parkNone
c060_10_ = (60,10),20,withoutTL,withoutHole,300
```
Figure 7 - Plan File Sample

The idea was to build a function of the same name that looks for the segment and crossing definitions and fills in the hashes accordingly. Regular expressions were used to parse the file, and care was taken to disregard changes in whitespace. The segments are parsed as follows:

```
/^\s*(\S+)\s*=\s*\(\s*(\S+)\s*,\s*(\S+)\s*\),\s*\(
\s*(\S+)\s*,\s*(\S+)\s*\)\s*,\s*(\S+)\s*,\s*(\S+)\
s*,\s*(\S+)\s*,\s*(\S+)\s*,\s*(\S+)\s*,\s*(\S+)\s*
/
```

Likewise, the regular expression for crossings is defined as follows:

```
/^\s*(\S+)\s*=\s*\(\s*(\S+)\s*,\s*(\S+)\s*\)\s*,\s
*(\S+)\s*,\s*(\S+)\s*,\s*(\S+)\s*,\s*(\S+)\s*,\s*(
\S+)\s*/
```

The need for the .ma file to be parsed came about because the size of each crossing is determined by TSC. This crossing size is included in the .ma file definition for the crossing, and it is based on the number of lanes of each of the roads leading into the crossing. Thus, we created a program that parses the .ma file and looks for the length of crossings that were detected in the .plan file. A new key in the crossing hash was added to hold this value.

3.1 Simulation Outputs

Once the GATLAS data structures are filled up with the segment and crossing definitions, we parse the log file for events detailing the movement of cars around the segments and crossings, as discussed earlier. For instance,

```
#Message      Y      /      00:00:38:400      /
libertador_b1(0,6)(1478)  /  out  /      1  to
libertador_b1(1471)
```

This message indicates that a car is being placed in lane 0, cell 6 of the libertador_b1 segment at time 00:00:38:400 (1 indicates the presence of a vehicle; if that value were 0.0000, it would indicate that the car is being removed from that cell). The regular expression used to parse these events as follows:

```
/Message          Y           \/\s*(\S*)\s*\/
(.*)\((.*),(\d*)\).*\/.*\/\D*([0-9\.]*).*/
```

In DEVS, output messages are part of the set Y, so this regular expression detects only the output messages of each cell. The %events hash has as its first key the time, and then successive keys are the element (segment or crossing), lane, and cell. The final value recorded is whether a car is being placed in or removed from the cell. Once the log file is parsed, all output messages matching the above regular expression are stored in the %events hash for later processing.

3.2 ATLAS/TSC Data Processing

Once the .plan, .ma, and .log file have been processed, all of the data required to visualize the traffic simulation is known. It is at this point that the last two steps in our process occur, create the traffic models and create the KML file.

We first loop through each segment and crossing to create a 2-dimensional array. Each of these two dimensional arrays is a representation of the cell space used by Cell-DEVS to model the segment or crossing. This model is used in the *createKML* function described later. The width of each array is simply the number of lanes of the segment, 1 for a crossing, which is taken from the .plan file. The length of a segment is the number of cells that are used to model the distance between the start point and end point. Note that these points are defined in terms of the cell space, so the distance between the start point and end point is determined and the length is taken to be the floor of that.

As noted previously the length of a crossing is determined by the value read in the .ma file. The length of the segment could have also been read from the .ma file, but the segment length calculation was implemented prior to the time that it was determined that the crossing length would need to be read from the .ma file.

Then, we need to create the KML file, and for doing so, we loop through each discrete time in the %event hash. Then for each element (crossing or segment) the model for that element has its elements set or cleared based on the data in the event hash. Once the element's models are filled for a given time step, the time value is reformatted from CD++ format to ISO format, and then each element and timestamp pair is sent to a function that returns the KML markup for that element which is then written to a file.

Segments are modeled by looking through each cell in the 2d model array and seeing if there is a car present. If there is, a call is made to CDppSegmentToKMLPosition, the prototype of which is seen in Figure 8.

```
my ($kml_latitude,$kml_longitude) =
      &CDppSegmentToKMLPosition      (
                  $segment_ref->{"start_x"} ,
                  $segment_ref->{"start_y"} ,
                  $segment_ref->{"end_x"} ,
                  $segment_ref->{"end_y"} ,
                  $segment_ref->{"direction"} ,
                  $segment_ref->{"num_cells"} ,
                  $lane_i,
                  $cell_j
              ) ;
```
Figure 8 - Call to CDppSegmentToKMLPosition

CDppSegmentToKMLPosition translates the position of the car in the model to a latitude/longitude position which is then inserted in the KML file. The CDppSegmentToKMLPosition function uses the start and end position of the model, the direction, the number of cells in the model, and the position of the given car in the model to calculate the latitude-longitude pair for that car. If the direction is 'back', the start and end points are swapped.

The car's position in terms of cells between the start and end point is then calculated using the fact that the car should be positioned in the middle of the cell that it is occupying, shown in Figure 9.

```
my    $car_pos_x    =    $start_x    +    ($end_x-
$start_x)/(2*$num_cells) * (2*$cell+1);
my    $car_pos_y    =    $start_y    +    ($end_y-
$start_y)/(2*$num_cells) * (2*$cell+1);
```

Figure 9 - Car's Model Position Calculation

Once the car's position between the start and end position is calculated, it's position offset from the line between start and end is calculated, to take the number of lanes in the street into account. A unit vector in the direction of start to end is derived, rotated 90 degrees, and then the car is displaced in that direction by a scale factor. At the time this report is written, the scale factor being used is 1, so the car is moved over a distance of one cell.

Once the car's position is calculated within the model, the distance between this position and the origin of the model is determined – this distance is still measured in units of cells. This cell distance is then transformed into a distance in kilometers based on the fact that a cell is 7.5 meters on one side. The angle this vector position of the car makes with the x-axis is then derived. This angle is then added to an model-specific angle that is hard coded – the angle the model's x-axis makes with the equator of the earth.

Finally this new total angle, the distance from the origin in km, and the origin's position as a latitude-longitude pair are used as inputs into an equation from <http://www.movable-type.co.uk/scripts/latlong.html>, which equation produces a new latitude-longitude pair given an initial position, a bearing, and a distance. The resulting latitude-longitude pair is then returned up the call stack and inserted into the KML template for the car using the time of the event and the next event time.

Crossings are modeled using the same methodology as segments, the only difference being how the position of the car is determined. A crossing has a static x-y coordinate, so if all cars in the crossing were drawn at that point, they would overlap and it would look as though cars disappeared in the crossing if there was more than one car in the crossing. Thus, the position of the car is determined to be the position of the crossing, plus an offset that is calculated by drawing a circle of radius R around the crossing position and then placing the car on that circle based on its position within the crossing model. This has the effect of having the cars look as though they are driving in a roundabout while they are in the crossing.

Once the first KML files were being produced by the system, it became evident that the combination of the models (the .plan files) and the transformation functions being used were not exact enough so that the cars followed the streets they were supposed to when visualized in Google Earth. The lack of traffic lining up with roads can be seen in Figure 10. It was not within the scope of this project to remodel the area, so it was decided that the cars previously modeled would be drawn on top of a white image covering the Google Earth imagery. Thus, it was then necessary to draw the segments so that the cars followed some sort of road system.

The segments were position in KML using the same algorithm as was used to place the cars, this time the start and end points were transformed to latitude-longitude pairs and then drawn as lines in KML, as shown below.

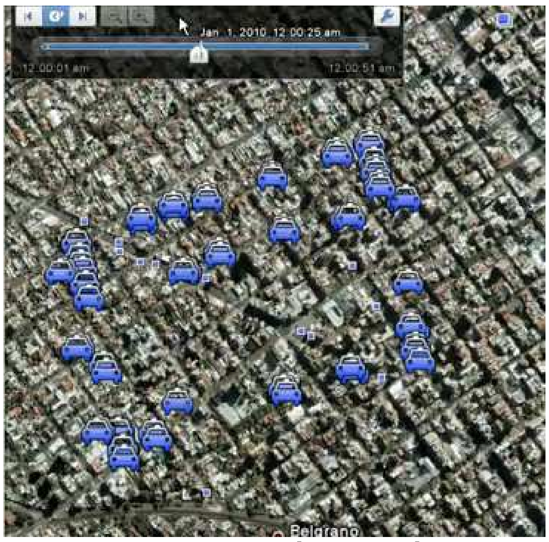

Figure 10 - Simulation Result

It was at this time that the approach of drawing a white box and then drawing the segments as lines in KML was taken, whose results can be seen in Figure 11. Vehicles were seen to follow the roads and enter and exit the system.

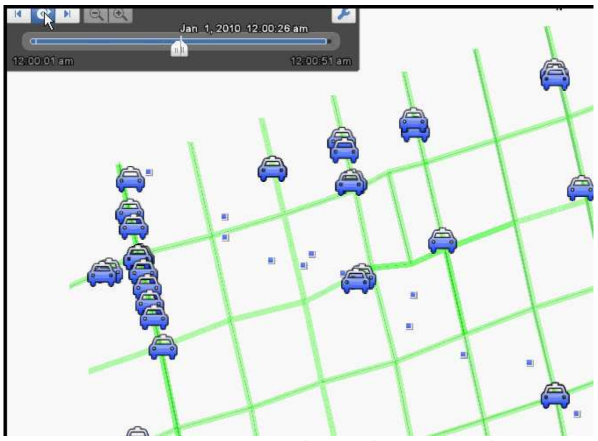

Figure 11 - Final Simulation Result

4 CONCLUSIONS

ATLAS-based traffic simulations were successfully visualized in Google Earth. Future work could look into working with a ATLAS/TSC .plan file that more realistically models a true roadway system, so that the transformation equations may be verified. If a system was produced that allowed a user to specify a ATLAS/TSC model using Google Maps – this time Google Maps would be the better choice because its API is well suited for user interaction – then the positions of the cars could be calculated using the start and end latitude-longitude pairs rather than the cell space positions, which would lead to much better accuracy in the visualizations.

An alternative approach for mapping of the real world to the ATLAS model would be to use a GIS data set to determine the geometry of the model based only on element names.

ACKNOWLEDGMENTS

This work has been partially funded by NSERC and GRAND NCE. Ken Edwards would like to acknowledge the support of Esterline CMC Electronics in this work.

REFERENCES

1. G. WAINER, A. DAVIDSON. "Defining a Traffic Modeling language Using Cellular Discrete-Event abstractions". Journal of Cellular Automata. Volume 2, Number 4, 2007. pp. 291-343
2. G. WAINER "ATLAS: a specification language for traffic modelling and simulation".. *Simulation Modeling, Practice and Theory*. Elsevier. Volume 14, No. 3, pp. 317-337. April 2006.
3. ZEIGLER,B.; KIM,T.; PRAEHOFER,H. *Theory of Modeling and Simulation*. Academic Press. 2000.
4. G. WAINER "Discrete-Event Modeling and Simulation: a Practitioner's approach". G. Wainer. CRC Press. Taylor and Francis. 2009.
5. Y. HARZALLAH, V. MICHEL, Q. LIU, G. WAINER. "Distributed Simulation and Web Map Mash-Up for Forest Fire Spread". Proceedings of IEEE International Conference on Web Services. Honolulu, HI. IEEE Press. 2008.
6. CAMERON, G.; WYLIE, B.; MC. ARTHUR, D. "TOMICS: Moving Vehicles on the Connection Machine". Proceedings of IEEE Supercomputing '95. 1995.
7. CHI, S.; LEE, J.; KIM, Y. "Using the SES/MB framework to analyze traffic flow". Transactions of the SCS. Vol. 14, No. 4, pp. 211-221. 1997.
8. CHOPARD, B.; DUPUIS, A.; LUTHI, P. "A Cellular Automata Model for Urban Traffic and its applications to the city of Genoa". Proceedings of Traffic and Granular Flow. 1997.
9. ERL,T. *Service-Oriented Architecture, Concepts, Technology, and Design*. Pearson. 2005.
10. ALONSO,G. *Web Services : Concepts, Architectures and Applications*. Springer-Verlag. 2003.
11. FOX,G.; PIERCE,M.; MUSTACOGLY,A.F.; and TOPCU,A.E. "Web 2.0 for E-Science Environments". *International Conference on Semantics, Knowledge and Grid*, IEEE. 2007.