SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.809

Developing a software toolkit for urban traffic modeling

Gabriel Wainer*,†

Department of Systems and Computer Engineering, Carleton University, 4456 Mackenzie Building, 1125 Colonel By Drive, Ottawa, ON, Canada K1S 5B6

SUMMARY

ATLAS is a modeling language that permits a static view of a city section to be defined for simulating traffic in closed areas. We propose a methodology that is focused on the user while being able to improve the software development activities. The models are formally specified, avoiding a high number of errors in the application, thus reducing the problem solving time. Streets are characterized by their traffic direction, number of lanes, etc. Once the urban section is outlined, the traffic flow is automatically set up. Specialized behavior is included to model traffic lights, trucks, traffic signs, railways, etc. The basic idea is to provide a mapping into DEVS and Cell-DEVS models that can be easily executed with a simulation tool. As the modelers can focus on the problem to solve, development times for the simulators can be dramatically reduced. A front-end system allows the user to draw city sections (and then parse the drawing to create a valid ATLAS file), and an output subsystem permitting cars to be shown with realistic 3D graphics. Copyright © 2007 John Wiley & Sons, Ltd.

Received 20 July 2005; Revised 3 October 2006; Accepted 18 December 2006

KEY WORDS: traffic simulation; DEVS models; cellular models; Cell-DEVS models; modeling methodologies; simulation support systems; environments

INTRODUCTION

The domain of urban traffic is of such complexity that it is impossible to use traditional analytical methods for analysis and control. Modeling and simulation techniques, instead, have shown some success, and they have been gaining popularity in this field, as they allow the analysts to study particular problems using virtual experimentation. Numerous software environments have been developed to solve these problems, using a variety of techniques.

*Correspondence to: Gabriel Wainer, Department of Systems and Computer Engineering, Carleton University, 4456 Mackenzie Building, 1125 Colonel By Drive, Ottawa, ON, Canada K1S 5B6.
†E-mail: gwainer@sce.carleton.ca

Contract/grant sponsor: NSERC
Contract/grant sponsor: Canadian Foundation for Innovation
Contract/grant sponsor: Ontario Innovation Fund

The early efforts in this area used *macroscopic* models to analyze traffic demand and flows on a traffic network using static parameters (i.e. average of daily traffic or average for peak hours). The goal was to find long-term forecasts that could be used for investments or dimensioning of the traffic net. *Microscopic* simulations are more recent. They require higher computing power, as they describe both system entities and their interactions at a high level of detail (i.e. a lane change could consider the nearby cars, as well as detailed driver decisions). Nonetheless, they can reproduce the real dynamics of traffic, enabling a modeler to study detailed phenomena as a function of time. Due to the precision of the results they provide, numerous tools for microsimulation are available, such as HUTSIM [1], Transims [2,3], Traffic Simulator [4], AIMSUN [5], CORSIM [6] or PARAMICS [7,8].

In most cases, these tools were built using standard software development techniques (including advanced GUIs) and, in some cases, software agents [9] and object-oriented programming [10]. Although they are usually well-tailored for by civil planners, the software product itself has the standard problems existing in any complex software application: testability, maintainability, legacy, etc. Introducing changes in the structure of the simulator requires a serious amount of effort, thus impeding the introduction of advanced techniques at a reasonable cost. The result is a very expensive product, difficult to modify and upgrade. In order to avoid these well-known problems in the software development of tools, different research efforts focused on a different approach based on the use of formal modeling methods. The idea is to use a formal modeling technique to define the model's behavior, and to create a software implementation following the specification. Some of them were based on queuing networks [7,11], Markov models, cellular automata (CA) [12–15], Discrete-Event Systems Specifications (DEVS) [16,17] and learning automata [18]. Several other approaches have also been used, from game theory [19], Petri nets [20], up to fluid or electrical flow models. Modeling tools based on formal approaches are easier to manage from the software development point of view. Nevertheless, the users of these tools usually need expertise in the formal techniques used for creating the modeling environment, which in most cases implies that an expert in the modeling technique used must interact closely with the traffic team.

Here, we present the results of an effort focused on developing a new environment for traffic microsimulations, in which the software environment was created using a unique approach, departing from previously existing tools. The proposed solution deals with the issues just introduced, serving as a proof of concept for a new methodology in the field. Our goals include the following.

1. *Usability*: allow the end users (civil planners) to ignore details about the underlying techniques, being able to focus on the problem to solve.
2. *Testability*: use formal techniques that make the creation of test cases for the software tool easier.
3. *Evolvability*: provide a means of changing the mechanisms to create traffic behavior, topology of the terrain or traffic light control, using an evolvable approach that would allow the traffic researcher to learn by doing, allowing one to introduce new methods into the toolkit easily.
4. *Maintainability*: provide facilities that would permit the software tool to be changed and re-tested easily, allowing improved maintenance.

In order to achieve these goals, we have organized the creation of the software as follows (see Figure 1). We first defined (and validated) a high-level specification language representing city sections [21,22]. This language, called ATLAS (Advanced Traffic Language Specifications), focuses on the detailed specification of traffic behavior (1) from the user's point of view. This method allows one elaborate study of traffic flow according to the shape of a city section and its transit attributes.

DOI: 10.1002/spe

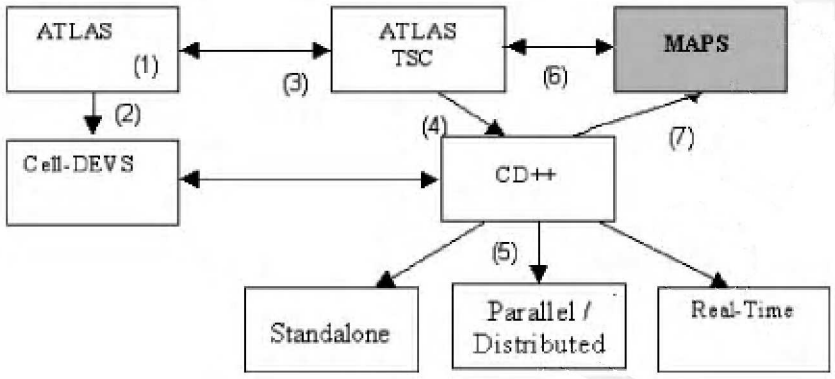

Figure 1. Structure of the proposed method.

A static view of the city section can be easily described, including definitions for traffic signs, traffic lights, etc. A city planner can concentrate on the problem to solve, instead of being in charge of defining a complex simulation, thus allowing us to achieve our first goal. The language constructions were then mapped into formal constructions using, in this case, the Cell-DEVS [23] and DEVS [24] formalisms (2). The behavior for each of the constructions presented in ATLAS was validated in terms of their correctness when built as Cell-DEVS models. Although a Cell-DEVS specification was used, other modeling approaches could be employed (for instance, CA or flow networks for the traffic, and Petri nets for the traffic light controllers). The use of these formal techniques improves maintenance and evolvability, and facilitates testing (this issue was widely discussed in various literature in the field; some of the related work in this area includes [24–30]). One important fact is that the concept of closure under coupling (which guarantees that a DEVS coupled model is equivalent to an atomic model from the point of view of their input/output trajectories), allows coupled models to be integrated into a model hierarchy, thus reducing testing time and improving the quality of the models.

Then, a compiler was built following the formal specifications of ATLAS (3). The ATLAS Traffic Simulator Compiler (ATLAS/TSC or TSC) was also built with our goals in mind [31]. The compiler generates code by using a set of templates that can be redefined by the user, easily adapting the generation of behavior to different modeling and simulation techniques. The extensible software architecture of the TSC compiler serves as the core of a complete system for the input, execution and output of complex traffic microsimulations. This approach also avoids version problems if the underlying tools are modified, improving maintenance and evolvability. TSC was provided with a set of templates that generate code that run on the CD++ toolkit (4) [32]. This approach also helps us to achieve our goals, as DEVS models written in CD++ can execute seamlessly using different engines, including standalone, embedded real-time and parallel versions (5). Thus, a model can be created on a workstation using the standalone simulator, validated in large scale using the parallel engine, and then put to work on the field using specialized embedded hardware. We can also make use of different libraries built for CD++, which allows varied modeling techniques (including Petri nets, finite state machines, Modelica, bond graphs, etc.) to be used. In this way, software developers could use the most

DOI: 10.1002/spe

adequate technique for different model subcomponents. As CD++ follows the formal definitions for DEVS and Cell-DEVS, we can also guarantee that model execution is correct (the CD++ simulator was formally verified for correctness). Hence, our activities were mainly devoted to checking whether the constructions we proposed provided a valid dynamic behavior, which was successfully done in [33]. The separation of modeling and simulation reduced the time spent in testing, as reported in [25–28]. The formal definition permits focusing in the model to develop, whereas the hierarchical modular definition makes it much easier to find related errors.

Defining very advanced models using TSC can be a tedious process that is exacerbated when there are rapid changes to the system input. The outputs of the system also generate text-based log files, which might be complex to analyze. Thus, we built a front-end application (called MAPS), which converts TSC constructions into a graphical representation (6). This representation allows the user to draw a city section with roads, crossings and decorations (potholes, stop signs, etc.), and then (7) parse the drawing to create a valid TSC file [34]. Likewise, the output went from a single segment of road with blocks as cars to a full-blown city section with realistic 3D graphics. In the following sections, we discuss each of the phases in the creation of this environment, focusing on each of the phases, and discussing the characteristics of the software developed to accomplish the goals in each phase.

BACKGROUND

Our goal is to develop microsimulation-based software to describe the local behavior of traffic with high precision. We want to allow modelers to analyze the behavior of traffic in closed sections with complex urban design or in closed traffic conditions (parking, roads in shopping malls, amusement parks or sports stadiums). The CA formalism has proven useful for these applications, and there is a wide variety of success stories of traffic simulation with CA [12–15]. CA defines a model as a grid of cells using discrete variables for time, space and system states [35,36]. The cells are updated according to a local rule function that uses a finite set of nearby cells (called the neighborhood of the cell) and is computed synchronously and in parallel for every cell in the space. It has also been shown that cellular models scale well (models of complete cities were presented in some of the cited work), and cellular models have the advantage of modeling traffic flow on a microscopic scale while expanding to large systems due to a simple type of dynamics. In [3, 12, 14, 37–40] it is also shown in detail that cellular models provide a good means for modeling microsimulation traffic, as they do represent a quite intuitive way of analyzing the traffic flow in detail, and they enable good visualization of the results.

Although cellular computing has shown success in modeling traffic, representing some of the most basic behavior (i.e. vehicle speed) using CA is done in a highly non-intuitive fashion. In addition, most existing models are constrained to representing simple aspects of the traffic flow, such as standard car movement in streets and crossings. In addition to these problems, CA are synchronous, a fact that poses precision constraints and extra computation time. The Cell-DEVS formalism [23] was proposed as a solution to these problems in which CA cells are defined as DEVS models [24]. First, in [41], it was shown that using the discrete event nature of DEVS models combined with parallel simulation techniques can produce speedups in simulations of up to a factor of 1000. In [23], we showed that Cell-DEVS also provides these advantages. Reducing computation time is particularly important for traffic models: although cars move at different speeds, we need to be able represent the fastest. Second, DEVS models permit explicit delays to be set, improving the accuracy of the model's timing properties.

DOI: 10.1002/spe

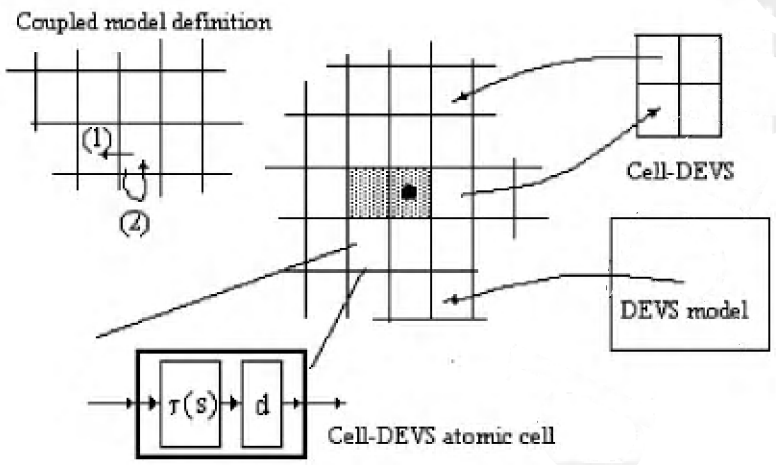

Figure 2. Informal definition of Cell-DEVS.

Finally, Cell-DEVS provides a formal framework that can be used to validate and verify the models, opening the door to re-using the models as well as integrating them with other models based on different formalisms (for instance, using Petri nets or finite state machines to specify the behavior of traffic lights or railway controllers). Q2

DEVS is a formal specification for modeling discrete event systems. A DEVS model is either atomic (behavioral) or it is coupled (structural)—a hierarchical and modular composition of other DEVS sub-models. A DEVS atomic model is a behavioral description of a system into terms of state variables, input and output events, and functions to compute the next state and output events:

$$M = \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, D \rangle$$

Here, S is the state set, X is the set of external input events received from other models and Y is the set of output events provided to other models. X and Y define the model's interface, which is composed of input and output ports. The model's behavior is defined by the remaining four functions. Each state has an associated duration time provided by the lifetime function D. When this duration time elapses, an internal transition is triggered possibly sending outputs out through the model's output ports using the output function λ. The internal transition function δ_{int} is then activated to produce internal state changes. Input events received on the input ports are processed according to the internal transition function δ_{ext}.

A DEVS coupled model is a structural composition of atomic models or other coupled models. DEVS models are closed under coupling, i.e. a DEVS coupled model is equivalent to an atomic model from a higher level of abstraction. A DEVS coupled model is defined as

$$CM = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\} \rangle$$

As with the atomic model, X and Y define the outer interface of this model and the events flowing in and out (allowing modular definition and hierarchical composition of the sub-models).

DOI: 10.1002/spe

The remaining attributes describe the inner composition: the set of component models and the set of interconnections, known as *influencees*, between their output and input ports. D is an index of inner components, and for each $i \in D$, M_i is its (atomic or coupled) DEVS model. I_i is the set of influencees of model i (i.e. which component is connected to which). For each $j \in I_i$, Z_{ij} is the i to j translation function that is in charge of translating outputs of one component's model into inputs for the others. To do so, an index of influences is created for each model (I_i). For every j in this index, outputs of the model M_i are connected to inputs in the model M_j.

The Cell-DEVS formalism is an extension to DEVS to describe complex discrete-event cellular models that can be integrated with other DEVS models. Each cell of a space is defined as an atomic DEVS with explicit timing delays. Cell-DEVS atomic models can be formally specified as

$$TDC = \langle X, Y, S, N, \text{delay}, d, \delta_{\text{int}}, \delta_{\text{ext}}, \tau, \lambda, D \rangle$$

where X represents the external input events, Y the external outputs, S the cell state definition, N the set of input events, delay defines the kind of delay for the cell (i.e. transport versus inertial) and d is its duration. Each cell uses a set of N input values to compute the future state using the function τ. These values come from the neighborhood or other DEVS models. A delay function can be associated with each cell to allow the deferral of outputs. A *transport* delay allows us to model a variable response time for each cell. *Inertial* delays are preemptive: a scheduled event is executed only if the delay is consumed. This behavior is defined by the δ_{int}, δ_{ext}, λ and D functions. Whenever an event arrives, the external transition function is activated. In this case, it takes the set of inputs and computes the cell's future state using the τ function. If the new cell state is different from the previous state, its value should be sent to the neighbors (that is, the cell's influencees). Otherwise, the cell is quiescent and its neighbors must not be activated. In any case, state changes are transmitted only after the consumption of the delay.

A Cell-DEVS coupled model is defined by

$$GCC = \langle X_{\text{list}}, Y_{\text{list}}, X, Y, n, \{t_1, \ldots, t_n\}, N, C, B, Z \rangle$$

Here, X_{list} and Y_{list} are input/output coupling lists. X and Y represent the input/output event sets, which can be used to transfer data from/to other models. The n value defines the dimension of the cell space, $\{t_1, \ldots, t_n\}$ are the numbers of cells in each dimension and N is the neighborhood shape. C, together with B (the set of border cells) and Z (the translation function) define the cells in the space and their interconnection. The cell space defined by this specification is a DEVS coupled model composed of an array of atomic cells, each of them connected to the cells defined by the neighborhood. As the cell space is finite, the borders have a different behavior. The Z function allows one to define the internal and external coupling of cells in the model. This function translates the outputs of mth output port in cell C_{ij} into values for the mth input port of cell C_{kl}. Q3

The formal specifications for DEVS and Cell-DEVS were used to build the CD++ tool [32]. CD++ lets the user define DEVS and Cell-DEVS. DEVS atomic models can be defined as C++ functions (by defining the behavior of the functions δ_{int}, δ_{ext}, λ and D). Alternatively, Cell-DEVS models are built using a specialized built-in language. Cell-DEVS coupled models are defined by their size, neighborhood and borders (as described in the formal specifications). Then, the cell behavior (defined by the local transition function) is described using rules with the following syntax: VALUE DELAY { CONDITION }. If the *CONDITION* of a rule is satisfied, the state of the cell changes to the designated *VALUE* after a *DELAY*. If the condition is not valid, the following rule is evaluated.

DOI: 10.1002/spe

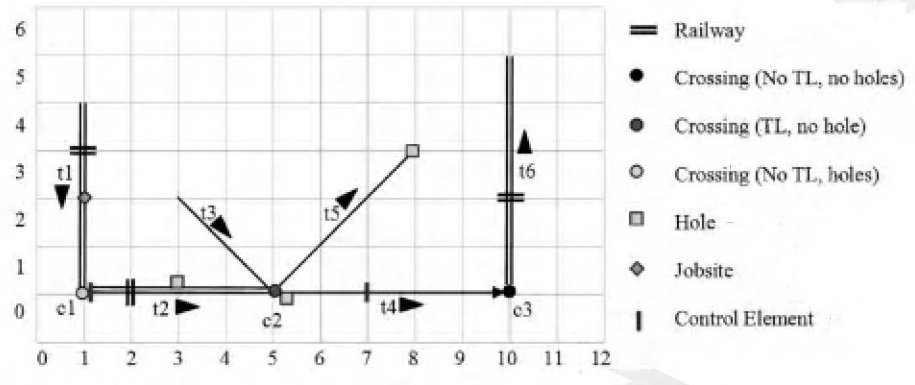

Figure 3. Shape of a city section.

A wide range of functions and operators can be used to define these rules. Input/output ports allow cellular models to be integrated with other DEVS. CD++ also allows n-dimensional *zones* to be defined, in which the behavior is different.

ATLAS

ATLAS is a language for the formal specification of the static structure of a city section defined by a set of streets connected by crossings. The formal specification for ATLAS *constructions* (elements that form a city's landscape) is first presented using an example. A discussion of the mapping of these formal specifications to Cell-DEVS is given to demonstrate the translation of the ATLAS constructions into formal models.

Atlas formal specifications

An ATLAS model is built as a set of different constructions, which are able to represent all standard elements that form our city landscapes. These constructions are presented using the example below in Figure 3, and their generic formal specification (which was used in [21,22] to define the mapping into Cell-DEVS models) is included.

Figure 3 shows a planar representation of a small city section consisting of segments (single and double lines named $t1 \ldots t6$), crossings ($c1 \ldots c3$) and decorations (jobsites, potholes, etc.). The list of constructions includes the following.

- *Segments*: these represent sections of a street between two corners. Every lane in a given segment has the same direction (one-way segments) and a maximum speed:

$$\text{Segments} = \{\ (p1,\ p2,\ n,\ a,\ \text{dir},\ \text{max})/p1,\ p2 \in \text{City} \wedge n,\ \text{max} \in N \wedge a,\ \ \text{dir} \in \{0,\ 1\}\ \}$$

DOI: 10.1002/spe

Here, $p1$ and $p2$ represent the boundaries of the segment (City $= \{(x, y)/x, y \in R\}$), n is the number of lanes and dir represents the vehicle direction. The a parameter (straight $= 0$ or curve $= 1$) allows the city shape to be defined more precisely, max is the maximum speed allowed in the segment. In our example, $t6$ is a straight segment with forward direction (i.e. $(2, 10) \rightarrow (7, 10)$), two lanes and a maximum speed of 60 Km h^{-1}.

- *Crossings*: these represent the places where more than one segments intersect. They are defined as points in the plane, associated to a maximum speed, and with the constraint of having at least one segment connected to them, i.e.

$$\begin{aligned}\text{Crossings} = \{(c, \max)/c \in \text{City} \wedge \max \in N \wedge \exists s, s' \in \text{Segments} \wedge s = (p1, p2, n, a, \text{dir}, \max)\\ \wedge s' = (p1', p2', n', a', \text{dir}', \max')\\ \wedge s \neq s' \wedge (p1 = c \vee p2 = c) \wedge (p1' = c \vee p2' = c)\}\end{aligned}$$

- *Jobsites*: these define an area over the segment where vehicles cannot advance:

$$\begin{aligned}\text{Jobsite} = \{(s, ni, \delta, \#n)/s \in \text{Segments} \wedge s = (c1, c2, n, a, \text{dir}, \max) \wedge ni \in [0, n-1]\\ \wedge \delta \in N \wedge \#n \in [1, n+1-ni] \wedge \#n \equiv 1 \bmod 2\}\end{aligned}$$

  Here, each $(s, ni, \delta, \#n) \in$ Jobsite is tied to a segment where construction works are being done. It includes the first lane affected (ni), the distance between the center of the jobsite and the beginning of the segment (δ) and the number of lanes occupied by the work ($\#n$).
- *Traffic lights*: crossings with traffic lights are formally defined as TLCrossings $= \{c/c \in$ Crossings$\}$. Here, $c \in$ TLCrossings defines a set of models representing the traffic lights in an intersection and its corresponding controller.
- *Railways*: these are built as a sequence of level crossings overlapped with the city segments. The railway network is defined by RailNet $= \{$(Station, Rail)/Station is a model, Rail $\in$ RailTrack $\}$. RailNet represents a set of stations, each generating traffic to a nearby RailTrack, defined as RailTrack $= \{ (s, \delta, \text{seq}) / s \in \text{Segments} \wedge \delta \in N \wedge \text{seq} \in N \}$. Each Railtrack associates a level crossing to a segment, by identifying the segment being crossed (s) and the position of the level crossing from the beginning of the segment (δ). A unique identifier (seq) is assigned to each level crossing, defining its position in the RailTrack.
- *Parking*: border cells in a segment can be used for parking:

$$\text{Parking} = \{(s, n1)/s \in \text{Segments} \wedge n1 \in \{0, 1\} \wedge s = (c1, c2, n, a, \text{dir}, \max) \wedge n > 1\}$$

  Every pair $(s, n1)$ identifies the segment and lane where parking is allowed ($n1 = 0$, on the left; $n1 = 1$, on the right).
- *Traffic signs*: these are defined by Control $= \{(s, t, \delta)/s \in \text{Segments} \wedge \delta \in N \wedge t \in$ {bump, depression, pedestrian crossing, saw, stop, school}$\}$. Each tuple here identifies 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.
- *Experimental frameworks*: these allow experiments to be built on a city section by providing inputs and outputs to the area to be studied. They are associated with segments receiving inputs,

DOI: 10.1002/spe

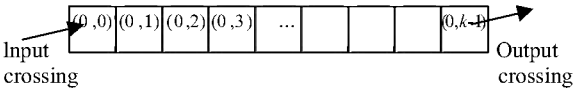

Figure 4. One-lane segment.

or those used as outputs, and are defined as

$$\begin{aligned}\text{InputSegments} = \{s/s \in \text{Segments} \wedge [(\text{dir} = 0 \wedge (\exists v \in N : (p2, v) \in \text{Crossings}))\\ \vee (\text{dir} = 1 \wedge (\exists v \in N : (p1, v) \in \text{Crossings}))]\}\\ \text{OutputSegments} = \{s/s \in \text{Segments} \wedge [(\text{dir} = 0 \wedge (\exists v \in N : (p1, v) \in \text{Crossings}))\\ \vee (\text{dir} = 1 \wedge (\exists v \in N : (p2, v) \in \text{Crossings}))]\}\end{aligned}$$

Translating ATLAS to Cell-DEVS

The ATLAS constructions presented above are formally defined and therefore can be translated into a formal representation. We translated the constructions into verifiable Cell-DEVS models, thus showing that ATLAS models themselves are provable (although a Cell-DEVS specification was defined, other modeling approaches, such as CA or flow networks for the traffic or Petri nets for the traffic light controllers, could be used when doing the mapping).

A detailed mechanism for the translations of each construction can be found in [21,22,33]. Here, we give a few brief examples, in order to illustrate the translation mechanism. As each segment can have a different number of lanes (leading to different border cases in each case), different behavior for the local computing functions must be provided. One-lane segments are the simplest case, and serve as a good starting point for an explanation of the process. A one-lane/one-way street, such as $t3$ in our example in Figure 3, is described as an ATLAS segment as $t3 = ((x1, y1), (x2, p2), n, a, \text{dir}, \text{max}) = ((3, 2), (1, 4), 1, 1, 1, 60)$. The segment is translated into a one-dimensional Cell-DEVS model with transport delays, informally depicted in Figure 4 with k cells and with the two border cells each connected to a crossing. It is a one-lane straight segment with 'go' direction, and a maximum speed of 60 km h^{-1}.

Each cell in the one-lane segment is mapped into a corresponding Cell-DEVS atomic model:

$$S1 = \langle X, S, Y, N, \delta_{\text{int}}, \delta_{\text{ext}}, \text{delay}, d, \tau, \lambda, ta \rangle$$

where

$$X = \{(X_1, \text{binary}), (X_2, \text{binary}), (X_3, \text{binary})\}, \quad Y = \{(Y_1, \text{binary}), (Y_2, \text{binary}), (Y_3, \text{binary})\}$$

$$S = \begin{cases} 1 & \text{if there is a vehicle in the cell} \\ 0 & \text{otherwise} \end{cases}$$

$$N = \{(0, -1), (0, 0), (0, 1)\}, \text{delay} = \text{transport}, d = \text{speed(max)}$$

DOI: 10.1002/spe

λ, δ_{int} and δ_{ext} behave as defined in the Cell-DEVS formalism with transport delays. $\tau : S \times N \rightarrow S$ is defined as follows:

$\tau\ (N)$	N
1	$(0, -1) = 1$ and $(0, 0) = 0$
0	$(0, 0) = 1$ and $(0, 1) = 0$
$(0, 0)$	TRUE /*Otherwise: state unchanged */

As we can see, each cell contains a vehicle or it is empty (S). The cell includes three input (X) and three output ports (Y), each prepared to communicate with the cell's environment: its previous neighbor $(0, -1)$, itself $(0, 0)$ and its next neighbor $(0, 1)$. The local computing function (τ) uses these input events to define the movement of a vehicle. The first rule represents a vehicle arriving at an empty cell from the previous cell. We first check whether the model receives an input event from the previous cell informing that it has a vehicle ready to move (described by $(0, -1) = 1$, i.e. the input event coming into $(0, -1)$ in the N set shows there is a car willing to move from that cell). Then, we check to see whether the current cell is empty (described by $(0, 0) = 0$, i.e. the input event coming from the cell itself shows that there is no car there). If these two conditions are true, then the car arrives into this cell ($t\,(N) = 1$). The second rule represents the car abandoning the present cell towards the front. The final rule, which is not executed unless the first two fail, simply leaves the cell state at its present value, represented by $(0, 0)$. Transport delays are used to model the time a vehicle spends leaving a cell and moving into the next. This time is generated by using a congestion function combined with a speed function related to the maximum speed in the segment (i.e. the max parameter in the segment construction; in our example, 60 km h^{-1}).

The atomic models for each of the individual cells within the segment must then be composed into a Cell-DEVS coupled model for the entire segment. The coupled model is constructed using the remaining information provided by the segment definition: the position (for $t3$, two cells $(3, 2) \rightarrow (1, 4)$) and shape of the segment (straight). A cell represents a physical length of 7.5 meters, computed as the average length of a car plus some extra space, as defined in [14,42]. Thus, the number of cells k is a simple function of the segment's length which is computed using geometry (depending on the value of a) as $\ell = |\sqrt{|x1 - x2|^2 + |y1 - y2|^2}|$ (for a = straight, i.e. the line length in Cartesian 2D space) or $\ell = (2\pi(|\sqrt{|x1 - x2|^2 + |y1 - y2|^2}|)/2)/2$ (for a = curved, i.e. the circle perimeter):

$$CS1(k, \max) = \langle X_{list}, Y_{list}, X, Y, n, \{t_1, \ldots, t_n\}, \eta, N, C, B, Z\rangle$$

is the generic coupled model definition for one-lane, with

$$Y_{list} = X_{list} = \{(0, 0), (0, k-1)\}$$
$$X = \{\langle X_{\eta+1}(0, 0), \text{binary}\rangle, \langle X_{\eta+1}(0, k-1), \text{binary}\rangle\}$$
$$Y = \{\langle Y_{\eta-1}(0, 0), \text{binary}\rangle, \langle Y_{\eta+1}(0, k-1), \text{binary}\rangle\}$$
$$n = 1, \quad t_1 = \ell \text{ computed as explained}, \quad \eta = 3$$
$$N = \{(0, -1), (0, 0), (0, 1)\}, \quad B = \{(0, 0), (0, k-1)\}$$

Z is built using the neighborhood definition, as specified by Cell-DEVS formalism.

The cells $(0, 0)$ and $(0, k-1)$ comprise the external interface of the model, because they must interchange information with the crossings corresponding to the street corners. The coupling scheme for these border cells is described in Figure 5.

DOI: 10.1002/spe

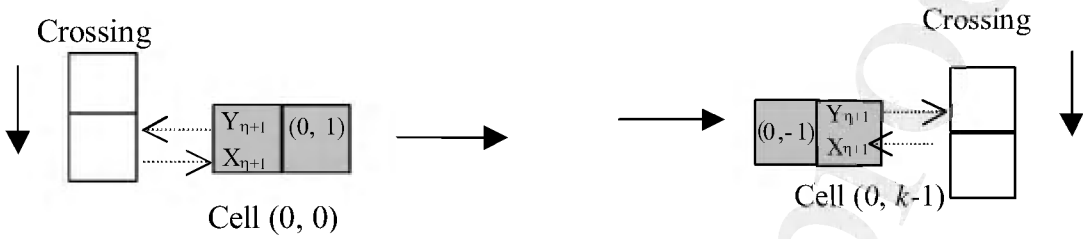

Figure 5. Coupling for the cells (0, 0) and (0, $k-1$).

The port names in the figure are constructed using Cell-DEVS formal specification (parameter i corresponds to the lane number; in our case, as a one-lane segment model is being considered, then $i = 0$). Although we used this formal notation to prove properties about the models, in the following discussion, we will rename the I/O ports in order to make it more readable, as follows:

Port	Name
$X_{\eta+1}(i, 0)$	*x-c-vehicle*
$X_{\eta+1}(i, k-1)$	*x-c-space*
$Y_{\eta+1}(i, 0)$	*y-c-space*
$Y_{\eta+1}(i, k-1)$	*y-c-vehicle*

Notation: x stands for input ports, y for output ports, *space* means this port is used to inform that there is room in the cell and *vehicle* is used to inform that it is occupied (c means that those ports are connected to a crossing).

Remember that the model is for one lane with cars traveling in one direction, from left to right. We can see that the cell (0, 0) receives a new car from the crossing in the *x-c-vehicle* port, and informs the crossing of its state through the *y-c-space* port. On the other end, cell (0, $k-1$) informs the other crossing of the state of the segment through the port *y-c-vehicle*, and receives information from this crossing through port *x-c-space*. Hence, cells (0, 0) and (0, $k-1$) in the segment must be redefined to allow this behavior. Whenever a vehicle wants to leave the crossing (e.g., port *x-c-vehicle* is 1) and the receiving cell is empty, the vehicle moves to the cell. Likewise, if both the cell in the crossing to which the segment is connected to and the previous cell in the crossing are empty, a car can cross. We only allow a vehicle to move to the crossing if the previous cell in the crossing (*x-c-space*) is empty, in order to represent that those cars already in the crossing have higher priority than those trying to cross. We use an inertial delay to represent that a car can only move into the crossing if there is free space during the time needed by a vehicle to move to the cell. Therefore, if there is a fast car arriving to the crossing input cell before the consumption of the delay, the car trying to cross waits.

We defined similar rules for segments of two and more lanes, which are converted into two-dimensional Cell-DEVS with the structure depicted in Figure 6.

The interface of this model is integrated by all of the cells in the first and last columns, which can be used to interchange information with the limiting crossings. Port definitions and rules for the border

DOI: 10.1002/spe

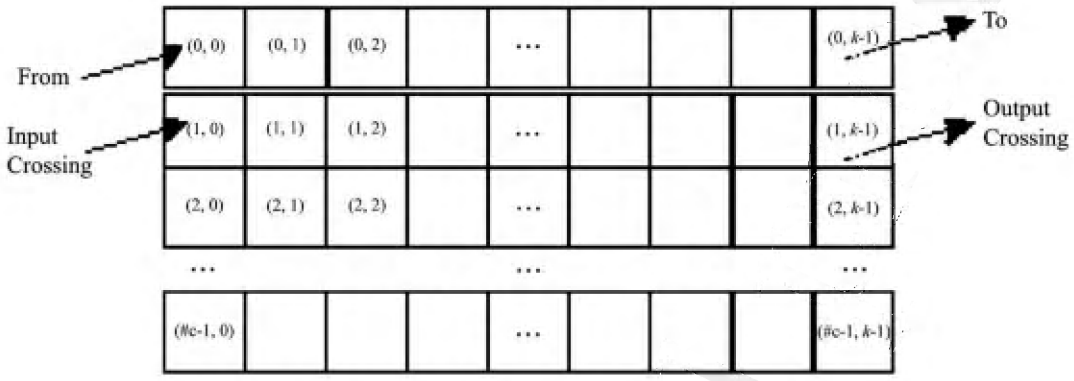

Figure 6. A segment with more than four lanes.

cells are extensions of those defined previously for the one-lane model (further details can be obtained in [21]).

Figure 7 depicts the methods for translating the remaining constructions (Specification details of these models can be found in [21]). As explained in the previous section, the crossings represent places in the city section where the segments join. Crossings are built as rings of cells with moving vehicles following the ideas presented in [43], as depicted in Figure 7(a). Each crossing can connect any number of segments, which are its inputs or outputs. Every crossing in this set is represented as a ring of cells where the vehicles advance (using the parameters $p1$ and $p2$ in the crossing formal definition to define the position of the crossing, and the max parameter to define the speed of the cars in the crossing). In any given instant, they can get to another path, using the proposal defined in [3]. The cars advance continuously in order to avoid deadlocks and, as mentioned before, a car in the crossing has higher priority to obtain a position into the ring than the cars outside the crossing.

The presence of a jobsite causes vehicles to deviate in a segment. The corresponding segment (s in the jobsite specification), the first lane affected (ni), the distance between the center of the jobsite and the beginning of the section (δ) and the number of lanes occupied ($\#n$) are used to define a rhombus over the segment where the vehicles cannot advance, as seen in Figure 7(b). The cars arriving at the jobsite must deviate. The basic behavior is that cells in the rhombus do not receive cars and those before it cause cars to deviate.

Figure 7(c) shows the translation of crossings with traffic lights (which are formally defined as any other crossing: a point in the plane associated to a maximum speed). Each traffic light is a discrete-event model (a DEVS model in our case) that sends a value representing the color of the traffic light to the cell inside the crossing that corresponds to the input segment affected by the traffic light. Another DEVS model is in charge of synchronizing all the lights in the corner. An upper level Cell-DEVS can be built to coordinate all of the controllers in a city section. The number of traffic lights to be created depends only on the number of input segments, and it is automatically defined by the number of elements in the crossing.

DOI: 10.1002/spe

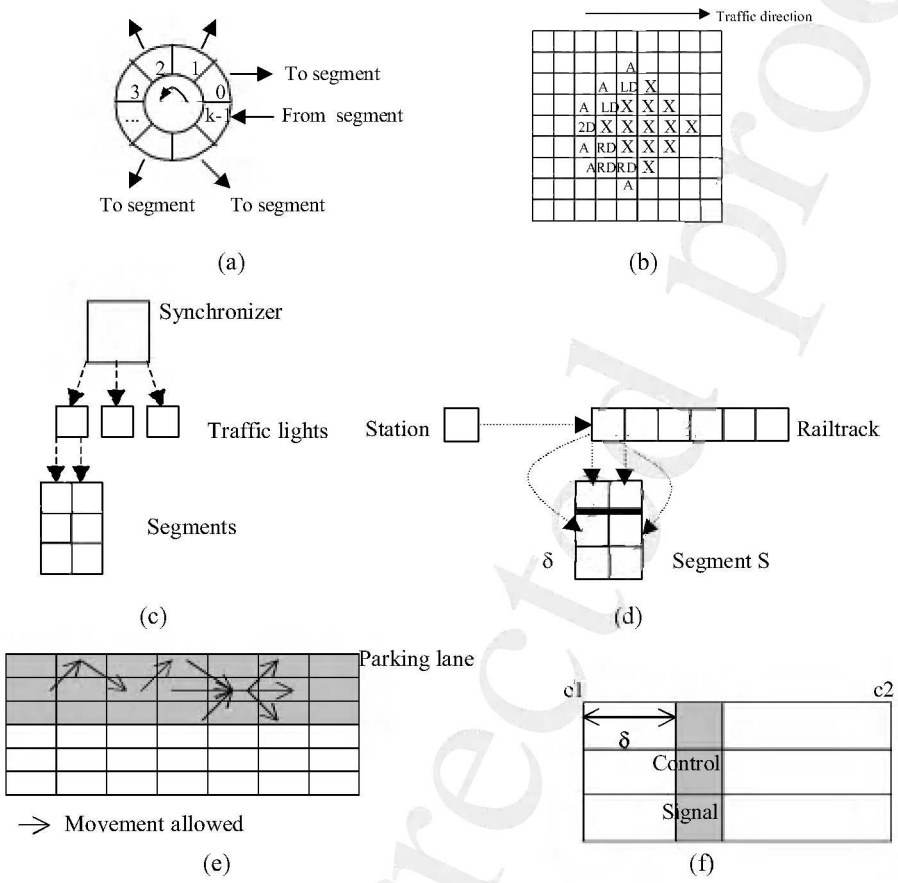

Figure 7. (a) Crossing; (b) jobsite; (c) traffic lights; (d) railways; (e) parking; (f) traffic signs.

For every element in RailNet, two models are defined: a DEVS model representing a station (from where a train departs), and a Cell-DEVS with one cell per level crossing. The station generates trains departing according to the train schedule. The RailTrack model affects the crossed segment as in Figure 7(d).

Figure 7(e) shows the basic behavior of parking areas. Every segment identifies the segment (s) and the lane where car parking is allowed ($n1$). When a car arrives into a parking lane, it stops for a given amount of time. This stopping time is modeled using a long transport delay (several minutes or hours).

Finally, Figure 7(f) shows the basic idea of the traffic signs construction. In this case, according to the type of sign and the distance to the beginning of the segment, the model changes the duration of the delay to slow down the speed of the vehicles passing through the zone. A one-cell extension of this construction allows potholes to be defined in segments and crossings.

DOI: 10.1002/spe

After defining the constructions, and prior to building the compiler, it was necessary to study the validity of the proposed language. A model is considered valid if, for every experiment within an experimental frame, the behavior of the model and the real system under analysis agree within acceptable tolerance. In our case, the validity of the model is guaranteed by the use of rules that are a direct mapping from CA models already proven as valid. The Cell-DEVS specifications were checked to be equivalent to those that were used at the level of input/output trajectories. We also considered structural validity: the structure of coupled models is guaranteed to be correct, as they are automatically built using well-known results in geometry and their coupled behavior is guaranteed by the use of DEVS. Likewise, we check map properties within the specifications written in ATLAS. Different types of conditions can be automatically checked:

- at least one segment exists;
- there are no isolated segments or crossings; specifically, every segment has an associated crossing and every crossing is associated to an existing segment;
- there are no crossing-to-crossing or segment-to-segment couplings;
- street direction is correct in each of the segments;
- a segment does not have the same beginning and end;
- no more than one crossing exists at the same point;
- railways, potholes, control signals and jobsites should be defined within an existing segment, and cannot exceed the segment boundaries;
- railways cannot cross the segments close to their borders;
- segments in which parking is permitted must have at least two lanes (to allow parking in one side) or three lanes (if parking in both sides is allowed).

After carrying out this step, we validated the behavior of city sections consisting of several components. Because of this phase, we were able to find some constructions whose behavior was not valid. For instance, our original definitions for parking and railways (which were defined using rules that were not validated previously) showed incorrect behavior when compared with the real system involved, and they were fixed.

ATLAS/TSC

With a formal specification in place, a compiler was developed. The novel contribution of ATLAS/TSC lies in the methodology proposed for developing the microsimulation tool, which also meets our stated objectives of usability, testability, evolvability and maintainability. The method is based on the use of templates to define the process of code generation. Templates are orthogonal to the translation process, and can thus be easily substituted, effectively re-configuring the code generation of the same simulation model. In this way, the compiler can be adapted to different modeling tools (and, if the underlying tools change, by modifying the template generation we can redefine the whole model). In order to check the feasibility of the approach, the constructions presented in the previous section were built using the CD++ tool, and two different sets of templates were used, each of them adapted to different versions of the CD++ toolkit [25].

The compiler and its language provide an intuitive format for developing ATLAS models, a direct mapping to the ATLAS formal specification and model validation, as discussed in previous sections. Figure 8 is the TSC definition of the sample city section in Figure 3.

DOI: 10.1002/spe

```
begin segments
  t1 = (1,5),(1,1),2,straight,go,21,1100,parkNone
  t2 = (1,1),(5,1),2,straight,go,22,1200,parkRight
  t3 = (3,3),(5,1),1,straight,go,23,1300,parkNone
  t4 = (5,1),(10,1),1,straight,go,24,1400,parkNone
  t5 = (5,1),(8,4),1,curve,go,25,1500,parkNone
  t6 = (10,8),(10,1),2,straight,back,26,1600,parkLeft
end segments

begin crossings
  c1 = (1,1),11, withoutTL, withHole, 221, 111
  c2 = (5,1),12, withTL, withoutHole, 222, 112
  c3 = (10,1),13, withoutTL, withoutHole,223, 113
end crossings

begin railnets
  rn1 = (t1,1),(t2,1),(t6,2),331
end railnets

begin jobsites
  in t1 : 1,2,1,441
end jobsites

begin holes
  in t2 : 1,2,553
  in t4 : 1,0,557
  in t5 : 1,3,559
end holes

begin ctrElements
  in t2 : stop,0,651
  in t4 : saw,2,658
end ctrElements
```

Figure 8. Definition of the city section in TSC.

Figure 8 shows that constructions are grouped into sections according to using `begin/end` blocks. Inside each block are one or more sentences for the respective ATLAS construction. The sentence syntax, summarized in Table I, flows mainly from the formal specification.

The few syntactical differences between TSC and the formal definitions lie in providing an intuitive development environment for the model developer, leaving to the compiler the mechanical translation into the formal specification syntax. For instance, in Figure 8 we show six segments defined, each with an identifier ($t1 \ldots t6$). The sentence syntax for other components then make use of these segment identifiers (e.g. in the jobsite for $t1$) instead of re-stating the specific cell coordinates of the formal specification (which are needed for the validation process). Another difference is the lack of any explicit parking constructions; instead, parking is included as an extra qualifier in the syntax for a segment sentence, because parking is inherently tied to particular segments. Likewise, the crossing sentence includes a qualifier for traffic lights, instead of a separate traffic light sentence. The compiler performs

DOI: 10.1002/spe

Table I. TSC syntax.

Construction	TSC sentence structure	Comments
Segment	`id = p1,p2, lanes, shape, direction, speed, parkType parkType=[parkNone | parkLeft | parkRight | parkBoth]`	parkType translates to a parking construction
Crossing	`id = p, speed, tLight, crossHole, pout` ***p*** and ***speed*** represent ($p1$,$p2$) and max of the formal specification **`tLight: [withTL|withoutTL]`** **`crossHole:[withHole|withoutHole]`** ***`pout:`*** probability of a vehicle abandoning the crossing, used to simulate random routing	A pothole can also be included in a crossing. Defines whether a crossing contains a pothole or a traffic light.
JobSites	`in t : firstlane, distance, lanes` **`firstlane, distance`** and **`lanes`** defined as in the formal specification	
Traffic lights	Included as tLight in crossing	
Railways	`id = (s1, d1) {,(si, di)}` **`si`** segment identifier crossed by the railway **`di`** distance between beginning of the segment s_i and railway	TSC automatically generates the sequence number in railway specification
Parking	Included as parkType in segment	
Traffic signs	`in t : ctrType, distance` **`ctrType: [bump | depression | Crossing | saw | stop | school]. distance`** `to the beginning of the segment`	Potholes: one cell. Each hole is defined as: in t: lane, distance

the multiple validity checks in the city map described in the previous section (at least one segment, only one crossing at the same point, etc.).

We need one template per ATLAS construction. The templates for segments and crossings are used to define how the behavioral models corresponding to them are generated (in our case, using Cell-DEVS). The remaining templates are used to modify the behavior of these basic models, including new behavior according to the construction to be used. For instance, potholes, railways and traffic signs affect the behavior of the segment they are decorating; likewise, potholes and traffic lights can affect crossings.

Every template contains instructions to generate the sentences defining the simulation model. They are organized in different sections using the format shown in Figure 9.

Each part of the template is associated with a set of generation rules for the compiler. The *after_rules* lines must be placed after the rules for segments or crossings if there are other constructions affecting the basic one, for instance traffic lights (lines to generate the traffic light and synchronizer models), railways (to generate rails and *railnet synchronizer* models), input segments (to define traffic generators) or output segments (to define traffic consumers). Q4

DOI: 10.1002/spe

```
|--template identif--|          ; defines a generic identifier to recognize the type of template
|--top components --|           ; these lines are added to the top coupled model
     line1 ... linen
|--top ports--|                 ; these lines define input/output ports for the top model
    line1 ...
|--top links--|                 ; internal/external couplings for the top model
    line1 ...
|--before neighbors--|          ; lines to included before the definition of the neighborhood.
    line1 ...
|--neighbors--|                 ; define the neighborhood shape used for the model
    line1 ...
|--before ports--|              ; to be included before the definition of the ports corresponding
    line1 ...                    ; to the model defining the construction.
|--ports--|                     ; define the input/output ports for the model generated.
    line1 ...
|--before links--|              ; to be included before definition of internal/external couplings
    line1 ...
|--links--|                     ; define the internal and external couplings of the model
    line1 ...                    ; corresponding to the construction.
|--before zones--|              ; to be included when we need zones with special behavior.
    line1 ...
|--zones--|
    line1 ...
|--before rules--|              ; to be included before the rule definition for each cell
    line1 ...
|--rules--|
    block1 ... blockn          ; define the behavior of each cell
|--after rules--|               ; must be placed after the rules for other constructions
|--end template--|              ; end of template definition.
```

Figure 9. Template definition.

The current version of the compiler is provided with a large list of templates. After parsing the file containing the map to simulate, TSC parses the file containing the code generation templates and builds a complete model, as seen in the following figure. For each simulation model, TSC takes the templates defining the transition functions for the construction, and it intercalates them in an ordered way on all of the sections according to the template definition, systematically translating ATLAS specifications into CD++ models without writing any code.

During this process, TSC uses a set of about 60 different macro-variables that can be included for each template, and the user can modify them to generate code accordingly. Macros are used to facilitate the use of the basic information that is repeated in every component: the size of the cell space, basic behavior, position of railways, etc. The following table shows some examples for macro-variables used in generating segments:

IDENTIF	Replaces the identifier
LANE	It is replaced by the lane number, once for each lane
FIRST_CELL	Replaces an identifier used for the first cell
LAST_CELL	Replaces an identifier used for the last cell
WIDTH	LAST_CELL + 1

DOI: 10.1002/spe

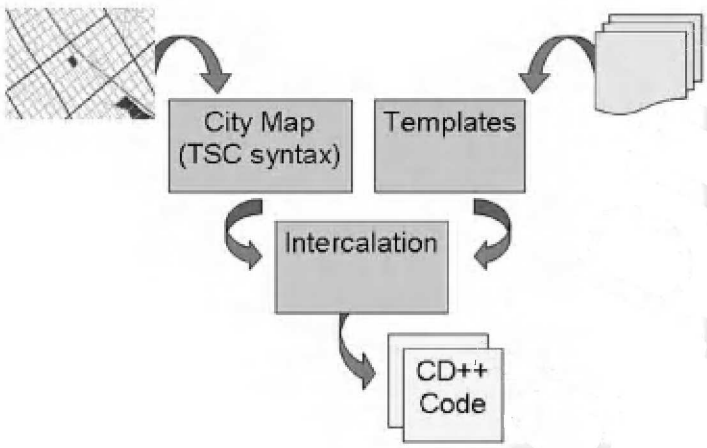

Figure 10. Code generation based on templates.

Macro variable definitions start and finish with '&'; i.e. &IDENTIF&. If, for instance, we consider segment *t*1 in Figure 3 (two lanes and four cells), the translator starts as follows:

[&IDENTIF&]	[**t1**]
width : &WIDTH&	width : **4**
in : x_c_vehicle&LANE&	in : x_c_ vehicle**0** in : x_c_ vehicle**1**
in : x_c_ vehicle&LANE&&FIRST_CELL&	in : x_c_ vehicle**00** in : x_c_ vehicle**10**

The following lines show the sequence of steps to create the input/output links for the same example, using the method explained in Figure 5:

```
1. link : y_c_vehicle@&IDENTIF&(&LANE&,&LAST_CELL&) y_c_vehicle&LANE&&LAST_CELL&
2. link : y_c_vehicle@t1(&LANE&,&LAST_CELL&) y_c_vehicle&LANE&&LAST_CELL&
3. link : y_c_vehicle@t1(&LANE&,3) y_c_vehicle&LANE&3
4. link : y_c_vehicle@t1(0,3) y_c_vehicle03
5. link : y_c_vehicle@t1(1,3) y_c_vehicle13
```

Let us suppose, for instance, that the following city section is defined (one segment and its experimental framework, which is created automatically to test the segment):

```
begin segments
    t1 = (0,0),(10,0),2,straight,go,200,200,parkNone
end segments
```

As this specification defines one segment with two lanes, the two-lane template is used. Figure 11 shows how the intermediate code is generated (using the previous definitions).

DOI: 10.1002/spe

Figure 11. Translation based on templates.

The translation begins by defining the type of template to be used (in this case, a two-lane segment with a generator in the start of the segment). The top components use the specification of the model to get the model identifier ($t1$ in this case) that is used in the following replacements of the identifier macro (&IDENTIF& = 1). Then, the components of the top model are defined. In this case, we have one generator connected to the segment to generate traffic. The *link* statements are used to define the internal and external couplings, according to the ATLAS definitions (using templates for the first and last cell in the segment). In this case, the generators' output ports (whose names are generated using macros) are connected to the input ports of the Cell-DEVS representing the two-lane model. The same procedures are repeated for the couplings in the end model. Finally, Figure 12 shows the complete Cell-DEVS model created using the *Segment-2Lane* template.

We can see that a one-line specification automatically expanded into a CD++ model with more than 40 lines, defining the detailed behavior of this model. The model creation follows the specifications presented in Figures 4 and 6 (following the rules for creating car movement), and the coupled model uses the ideas presented in Figure 5. In this way, the definition time for traffic models can be dramatically reduced. Figure 13 shows an excerpt of the simulation results for Figure 11.

DOI: 10.1002/spe

```
[top]
components : t1Gen@TSCGenerator           // Experimental Frame: Generator+Consumer
components : t1Cons@TSCConsumer
components : t1                           // t1: segment
out : qtyOutSimu_t1Cons
link : y_t_car0@t1Gen x_ge_car00@t1       // External Coupling specification
link : y_t_car1@t1Gen x_ge_car10@t1
link : y_co_car09@t1 x_t_car0@t1Cons
link : y_co_car19@t1 x_t_car1@t1Cons
link : quantity@t1Cons qtyOutSimu_t1Cons

[t1]                                      // t1 segment Coupled Model Specification
type : cell            width : 4      height : 2      // Model size
delay : transport      border : nowrapped
neighbors : (1,-1) (1,0) (1,1) (0,-1) (0,0) (0,1) (-1,-1) (-1,0) (-1,1)
in : x-ge-car00
in : x-ge-car10
out: y-co-car03
out: y-co-car13
link : x-ge-car00 x-ge-car@t1(0,0)        link : x-ge-car10 x-ge-car@t1(1,0)
link : y-co-car@t1(0,3) y-co-car03        link : y-co-car@t1(1,3) y-co-car13
localtransition : t1-segment2-lane0-rule
...

[t1-segment2-lane0-rule]
rule : 1 21 { (0,0) = 0 and (0,-1) = 1 }
rule : 1 21 { (0,0) = 0 and (-1,-1) = 1 and (-1,0) = 1 and (0,-1) = 0 }
rule : 0 21 { (0,0) = 1 and (0,1) = 0 }
rule : 0 21 { (0,0) = 1 and (-1,0) = 0 and (-1,1) = 0 }
rule : {(0,0)} 21 { t }
...
```

Figure 12. Resulting definition for $t1$ in CD++.

```
Message Y / 00:00:00:000 / t1gen(03) / y_t_vehicle0 / 1.00000 to top  (01)
Message D / 00:00:00:000 / t1gen(03) / 00:00:03:000 to top            (02)
Message X / 00:00:00:000 / top(01) / x_ge_vehicle00 / 1.00000 to t1   (03)
Message X / 00:00:00:000 / t1(05) / x_ge_vehicle / 1.00000 to t1(0,0) (04)
Message D / 00:00:00:000 / t1(0,0)(06) / 00:00:00:200 to t1           (05)
Message X / 00:00:00:000 / top(01) / arrived / 0.00000 to bigcounter  (06)
Message D / 00:00:00:000 / bigcounter(02) / 00:01:00:000 to top       (07)
   ...
```

Figure 13. Execution results for $t1$.

DOI: 10.1002/spe

We can see that, initially, model *t*1gen generates a vehicle, which is inserted into the port `y_t_vehicle0` (01). The next activity of this generator (a part of the experimental framework) is scheduled 3 seconds after this (02). The car generated is then transmitted to the segment *t*1 (03), which puts the vehicle on cell (0, 0) of the segment (04). Then, it schedules an internal transition 200 ms after that (05), representing the speed of the vehicle. The information is also transmitted to the output port of the model, so that the experimental framework can record metrics (06)–(07).

As we can see, we have developed the ATLAS/TSC compiler, based on a new methodology for developing microsimulation tools, to meet our stated objectives of usability, testability, evolvability and maintainability. The methodology is based on a set of templates to define the process of code generation. The constructions were translated and tested on the CD++ tool, and two different sets of templates were used, each adapted to different versions of the CD++ toolkit [25].

MAPS

As we can see, although the specification of the model in Figure 3 is simple (and it generates 2400 lines of Cell-DEVS specifications to be simulated), its manual definition is tedious. Likewise, studying the output models using text-based log files is cumbersome. The goal of the MAPS interface is to allow users to draw small city sections, which are then automatically parsed, into ATLAS files and to visualize the results in a meaningful way. Users can change the layout of the city section, as well as ATLAS specific parameters quickly and easily. MAPS eliminates the need to know many details of the TSC syntax, and it dramatically reduces the time it takes to create ATLAS files. MAPS provide an intuitive interface, which allows complete roads (instead of segments) to be generated, automating the generation of crossings and decorations (potholes, stop signs, etc.).

MAPS parses the graphical representation, first removing and storing the crossings information. City level decorations are then stored (e.g. railnets). The parser then loops through each road to see whether they intersect with other roads. If a previously generated crossing exists at the crossing point, then it is used. If not, a new crossing is created. The parser also checks to see whether the road contains a railnet. If it does, the parser checks to which segment the railnet belongs to as the segments are created.

A list of breakpoints determines how to cut up the road into segments (a breakpoint is a simple class storing the location of the cut and its type (e.g. start of the road, end of the road, crossing, parking, etc.)). This list does not contain crossings that do not form segments (e.g. at the start and end of the road being segmented). Breakpoints can also be created by parking, as there can be parking available only on certain parts of the road. The parser loops through the parking decorations for each lane of that road to create breakpoints for that lane. Each segment is given a unique identifier, which is tagged to other decorations that the lane is affected by (e.g. roadwork spanning multiple lanes, potholes, etc.). The lane breakpoints are then sorted and the segments are created, named and decorated. The process is repeated for as many lanes and as many roads as necessary.

The following rules apply to parsing lane decorations:

- there is no parking available at the start and end of a lane;
- parking objects may not overlap;
- parking objects may not be intersected by another road—that is, there is no parking allowed in a crossing;
- segments with parking may not contain a railnet.

DOI: 10.1002/spe

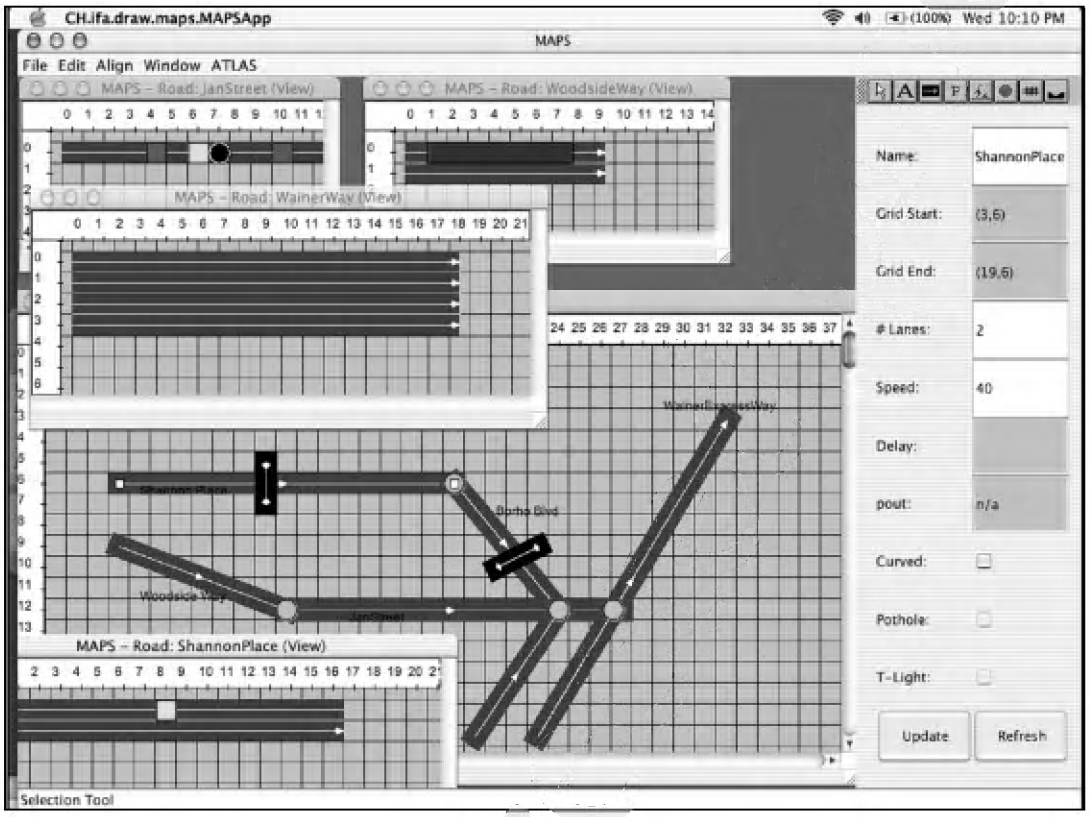

Figure 14. MAPS graphical interface.

The segments are then modified by looping through the decorations for that lane and checking to see whether they lay on that particular segment, adding as many of them as required. Figure 14 presents a screenshot of a city section using MAPS. Note the presence of railnets (black rectangle with white line), crossings (circles, automatically generated), roadwork (light squares), stop signs (dark squares), parking sections (black rectangles), multiple and bidirectional roads. We also include the definition of ATLAS parameters such as speed, curvature of the road, etc. Q5

Figure 15 shows the TSC specifications generated by MAPS for the example in Figure 14. As we can see, the new representation of the model is more intuitive, simpler to modify and faster to understand and run experiments.

MAPS also includes a GUI that shows traffic flowing through a predefined city section based on the results of a simulation. MAPS uses the created TSC definition to determine a static view of the city without cars present, showing the user the various segments and crossings involved in the ATLAS city section. The GUI uses the results file from a previous simulation by the CD++ simulator,

DOI: 10.1002/spe

```
begin segments
BankGOS1=(0,0),(5,0),1,straight,go,60,0,parkNone
BankGOS2=(5,0),(6,0),1,straight,go,60,0,parkNone
BankB1=(0,0),(5,0),1,straight,back,60,0,parkNone
BankB2=(5,0),(6,0),1,straight,back,60,0,parkNone
LibraryG1=(5,0),(5,2),2,straight,go,55,0,parkNone
LibraryGOS2=(5,2),(5,5),2,straight,go,55,0,parkNone
...
AltaVistaBACKS4=(5,5),(6,5),1,straight,back,40,0,parkNone
BronsonGOS1=(2,2),(5,2),1,straight,go,75,0,parkNone
BronsonGOS2=(5,2),(12,2),1,straight,go,75,0,parkNone
end segments

begin crossings
Bank&Library = (5,0),60,withoutTL,withoutHole,0,0.5
Library&AltaVista = (5,5),55,withoutTL,withoutHole,0,0.5
Library&Bronson = (5,2),55,withoutTL,withoutHole,0,0.5
end crossings
```

Figure 15. Resulting specification in TSC.

and determines the location and direction of specific cars at a particular point in time using a log file generated by the simulator. A car shape is displayed on the screen in the appropriate cell on a segment for the time specified in the log file. When that time expires, the car moves to a new cell as per the results file.

The entire city section operates in this manner with cars moving within segments and from segment to segment. The user can navigate around the city section as they wish using any tool capable of displaying VRML files. The time is displayed as it changes according to the log file so that the user has an idea of the time as cars are moving. This allows the user to see the buildup of traffic on different segments graphically as time passes, instead of having to interpret the results using the simulation results.

In order for the system to achieve these goals, it was essential to create VRML objects that represent cars, segments and crossings. The first issue is that a static view of the city should first be drawn from the TSC definition with the segments and crossings displayed to the user. We defined a road shape to be displayed for every cell in the segment, and a crossing shape for crossings. The car shape shown in Figure 16(a) was used to represent traffic (a slightly modified version of a shape found in [44]). Crossings can be defined to have traffic lights or stop signs depending on the decoration defined for the model (Figures 16(b) and (c)).

The different attributes for the segments (including starting and ending points) must be used to reconstruct the city section topology. Let us consider an example of two segments written in TSC, as follows:

```
A = (0,0), (10, 10), 1, straight, go, 40, 300, parkNone
A1 = (0,0), (10, 10), 1, straight, back, 40, 300, parkNone
```

There are two segments, one going from (0, 0) to (10, 10) and the other is going in the opposite direction. Each of the crossings, segment and cars are one-by-one VRML objects, so the mapping

DOI: 10.1002/spe

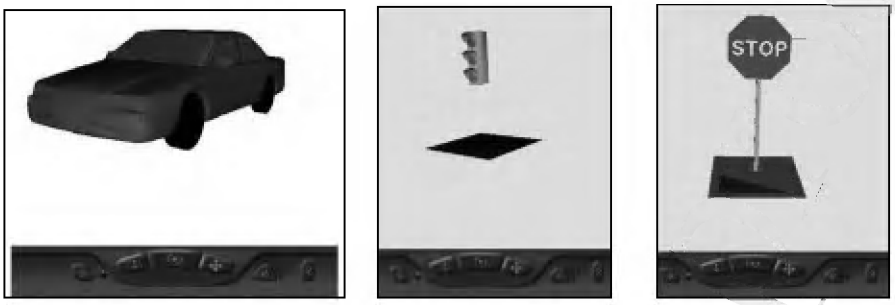

Figure 16. Car and crossing VRML objects.

from the TSC definition to the VRML world is simple: each of the two segments contain 14 consecutive segment objects

$$length = \sqrt{(P_{1y} - P_{2y})^2 + (P_{1x} - P_{2x})^2}$$

If the segments do not run parallel to the x- or y-axis, then the segment objects have to be rotated to make them look consecutive

$$rotation = \tan^{-1}\left(\frac{P_{2y} - P_{1y}}{P_{2x} - P_{1x}}\right)$$

For instance, segments A and A1 are rotated by an angle of 45°. Once the angle and length have been calculated, the segments have to be translated to the appropriate position in the VRML world. The first step is to translate the segment object to the segment's start point, then rotate the objects appropriately as calculated above and, finally, scale the segment object to the calculated length. This is done for every segment in the TSC definition until the static view of the city section is shown in the VRML world. An example of a static view of Carleton University campus is shown in Figure 17.

Once the static view of the section is shown, we verify that each segment and crossing match up with a Cell-DEVS model in the TSC specification. After the model file has been verified, the user can get a log file generated by the CD++ simulator in order to visualize the simulation results. MAPS parses the log file for output messages such as the following:

```
Message Y/00:00:00:200/t1(0,0)/out/1 to t1
```

which indicates that a car (the `1`) has now appeared in cell 0 of lane 0 (the 0, 0) of segment $t1$ at time 200 ms. We also use the logged message saying that a car has disappeared, so that the car can be removed from the now vacant segment.

When MAPS encounters this message, it creates a VRML car object (Figure 16(a)), then rotates it by the same amount and translates it to the same location as the segment object (in this case, on lane 0, cell 0 of segment $t1$) as determined when displaying the static view of the city. Figure 18 shows an example of the execution of the model defined in Figure 14.

DOI: 10.1002/spe

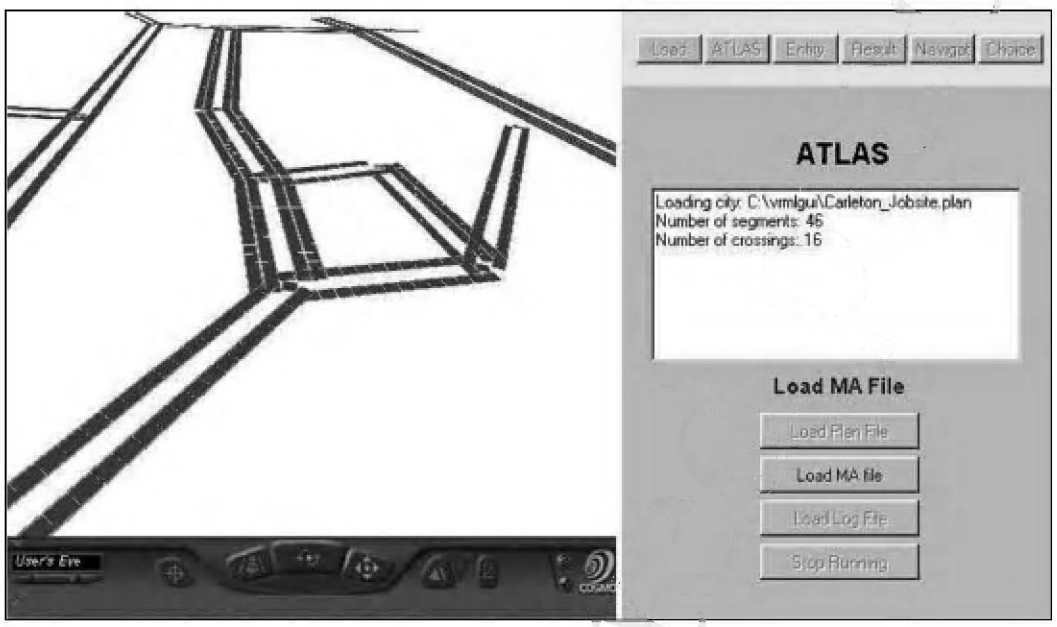

Figure 17. Static view of Carleton University campus with segments and crossings.

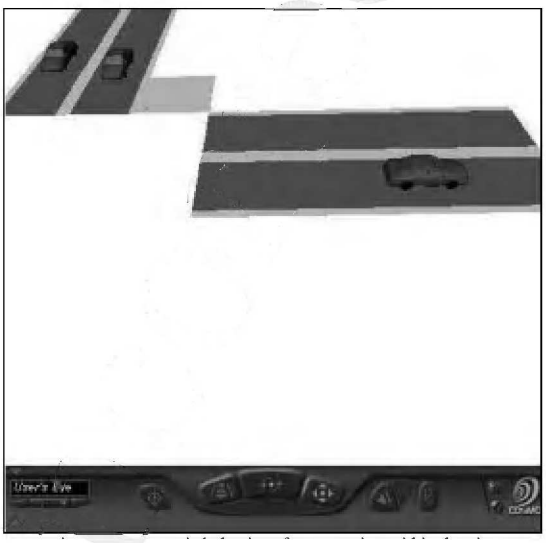

Figure 18. Dynamic behavior of cars moving within the city.

DOI: 10.1002/spe

CONCLUSION

ATLAS provides an application-oriented specification language, which allows the definition of complex traffic behavior using simple rules for a modeler. The models are formally specified, avoiding a high number of errors in the application, thus reducing the problem solving time.

We first defined and validated the ATLAS specification language, representing the city sections as cell spaces, allowing an elaborate study of traffic flow according to the shape of a city section and its transit attributes. The language constructions are mapped into formal constructions using Cell-DEVS. Then, the TSC compiler was built following the formal specifications of the language. The 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, while avoiding version problems if the underlying tools are modified. The models that TSC generates can execute on the CD++ toolkit, allowing us to guarantee that model execution is correct (CD++ simulator was formally verified for correctness).

Although TSC allows very advanced models to be defined, their definition requires manual generation of text files defining city sections using ATLAS constructions. This tedious process does not lend itself for rapid changes to the system input. The outputs of the system also generate text-based log files, which not are user friendly. Thus, we built a front-end application (called ATLAS/MAPS or MAPS), which allows the user to draw a city section with roads, crossings and decorations. Likewise, the output went from a single segment of road with blocks as cars to a full-blown city section with realistic 3D graphics.

Now, a modeler can easily describe a static view of the city section under analysis, including definitions for traffic signs, traffic lights, etc. A modeler can concentrate on the problem to solve, instead of being in charge of defining a complex simulation, and it can be simulated efficiently using a discrete event approach, which can be applied to simulation engines (standalone, parallel, real-time).

We were able to achieve all of the goals discussed in the introduction.

1. ATLAS/TSC is a user-oriented language. The compiler and the visualization tools allow easy interaction with the end user of the system.
2. Testability is improved thanks to the use of a formal methods (which allows system properties to be proved before implementation, including guaranteeing the correctness of the simulation engines, as shown in [23,24]) and a hierarchical modular modeling technique such as DEVS, which enables standalone testing and reuse.
3. Software developers can easily change the environment by modifying the TSC templates. Likewise, the simulation engines are easily modifiable. By following the formal approach we used, it is convenient to formally study the modification of the rules prior modification of the compiler (as done in our previous experiences), which makes finding problems easier (and allows them to be tracked up to the specification level).
4. The issues discussed for items 2 and 3 also help to improve maintainability, as we can easily change the environment according to new needs, or upon finding problems in the simulation environment.

ACKNOWLEDGEMENTS

This work has been partially supported by NSERC, the Canadian Foundation for Innovation, and the Ontario Innovation Fund. Different persons collaborated in different stages, including Professor Cheryl Schramm, Shannon Borho, Jan Pittner (MAPS), Alejandra Davidson, Alejandra Díaz, Verónica Vázquez (ATLAS), Mariana Lo Tartaro and César Torres (TSC).

DOI: 10.1002/spe

REFERENCES

1. Kosonen I, Pursula M. A simulation tool for traffic signal control planning. *Proceedings of the 3rd International Conference on Road Traffic Control* (*IEE Conference Publication*, number 320). IEE: London, 1990.
2. Barret C, Beckman R, Berkbigler K. Transportation analysis simulation systems (TRANSIMS), volume 0—overview. *Technical Report LA-UR 99-1658*, Los Alamos National Laboratory 1999.
3. Nagel K, Stretz P, Pieck M, Leckey S, Donnelly T, Barret C. TRANSIMS traffic flow characteristics. *Proceedings of the Transportation Research Board 77th Annual Meeting*, Washington, DC, 1998.
4. Chopard B, Dupuis A, Luthi P. A CA model for urban traffic and its applications to the city of Genoa. *Proceedings of Traffic and Granular Flow*, 1997.
5. Barceló J, Casas J, Ferrer J, García D. Modeling advanced transport telematic applications with microscopic simulators: The case of AIMSUN. *Proceedings of the 10th SCS European Simulation Symposium*, 1998.
6. Bumble M, Coraor L, Elefteriadou L. Exploring CORSIM runtime characteristics: Profiling a traffic simulator. *Proceedings of the 33rd Annual Simulation Symposium*. IEEE Press: Washington, DC, 2000.
7. Cameron G, Wylie B, McArthur D. PARAMICS: Moving vehicles on the connection machine. *Proceedings of IEEE Supercomputing '95*, 1995.
8. Sahraoui A, Jayakrishnan R. Microscopic–macroscopic models systems integration: A simulation case study for ATMIS. *SIMULATION* 2005; **81**:353–363.
9. Balmer B, Cetin N, Nagel K, Raney B. Towards truly agent-based traffic and mobility simulations. *Proceedings of the Autonomous Agents and Multi-Agent Systems Conference*, New York, 2004.
10. Sadoun B. An efficient simulation methodology for the design of traffic lights at crossings in urban areas. *SIMULATION* 2003; **79**:243–251.
11. Schmidt M. Decomposition of a traffic flow model for a parallel simulation. *Proceedings of AI, Simulation and Planning in High Autonomous Systems (AIS'2000)*, Tucson, AZ, 2000.
12. Chopard B, Queloz PA, Luthi P. Traffic models of a 2D road network. *Proceedings of the 3rd CM Users' Meeting*, Parma, Italy, 1995.
13. Treiber M, Hennecke A, Helbing D. Congested traffic states in empirical observations and microscopic simulations. *Physical Review E* 2000; **62**:1805–1824.
14. Nagel K. Cellular automata models for transportation applications. *Proceedings of the 5th International Conference on Cellular Automata for Research and Industry*, Geneva, Switzerland, 2002 (*Lecture Notes in Computer Science*, vol. 2493). Springer: Berlin, 2002.
15. Rodríguez Zamora R. Using de Bruijn diagrams to analyze 1d CA traffic models. *Proceedings of the 6th International conference on Cellular Automata for Research and Industry*, Amsterdam, The Netherlands, 2004 (*Lecture Notes in Computer Science*, vol. 3305). Springer: Berlin, 2004.
16. Chi S, Lee J, Kim Y. Using the SES/MB framework to analyze traffic flow. *Transactions of the SCS* 1997; **14**(4):211–221.
17. Lee J, Chi S. Using symbolic DEVS simulation to generate optimal traffic signal timings. *SIMULATION* 2005; **81**:153–170.
18. Ünsal C, Kachroo P, Bay J. Simulation study of multiple intelligent vehicle control using stochastic learning automata. *Transactions of the SCS* 1997; **14**(4).
19. Chen O, Ben-Akiva M. Game theoretic formulation of the interaction between dynamic traffic control and dynamic traffic assignment. *Technical Report*, ITS, MIT, Cambridge, MA, 1998.
20. Tolba C, Lefebvre D, Thomas P, El Moudni A. Continuous and timed Petri nets for the macroscopic and microscopic traffic flow modeling. *Simulation Modelling Practice and Theory* 2005; **13**(5):407–436.
21. Davidson A, Wainer G. Specifying control signals in traffic models. *Proceedings of AI, Simulation and Planning in High Autonomous Systems (AIS'2000)*, Tucson, AZ, 2000.
22. Davidson A, Wainer G. Specifying truck movement in traffic models using Cell-DEVS. *Proceedings of the 33rd Annual Simulation Symposium*. IEEE Press: Washington, DC, 2000.
23. Wainer G, Giambiasi N. *N*-dimensional Cell-DEVS. *Discrete Events Systems: Theory and Applications* 2002; **12**(1):135–157.
24. Zeigler B, Kim T, Praehofer H. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press: New York, 2000.
25. Wainer G, Giambiasi N. Application of the Cell-DEVS formalism for cell spaces modeling and simulation. *SIMULATION* 2001.
26. Labiche Y, Wainer G. Towards the verification and validation of DEVS models. *Proceedings of the 1st Open International Conference on Modeling and Simulation*, Clermont-Ferrand, France, 2005.
27. Wainer G, Morihama I., Passuello V. Automatic verification of DEVS models. *Proceedings of the SISO Spring Interoperability Workshop*, 2002.
28. Nutaro J, Hammonds P. Combining the model/view/control design pattern with the DEVS formalism to achieve rigor and reusability in distributed simulation. *Journal of Defense Modeling and Simulation*.

DOI: 10.1002/spe

29. Kim TG, Cho SM, Lee WB. DEVS framework for systems development. *Discrete Event Modeling and Simulation: Enabling Future Technologies*. Springer: Berlin, 2001.
30. Zeigler BP, Fulton D, Nutaro J, Hammonds P. M&S enabled testing of distributed systems: Beyond interoperability to combat effectiveness assessment. *Proceedings of the 9th Annual M&S Workshop*. ITEA: White Sands, NM, 2003.
31. Lo Tartaro M, Torres C, Wainer G. Defining models of urban traffic using the TSC tool. *Proceedings of the 2001 Winter Simulation Conference*. IEEE Press: Washington, DC, 2001.
32. Wainer G. CD++: A toolkit to define discrete-event models. *Software—Practice and Experience* 2002; **32**(3):1261–1306.
33. Díaz A, Vázquez V, Wainer G. Application of the ATLAS language in models of urban traffic. *Proceedings of the 34rd Annual Simulation Symposium*, Seattle, WA, 2001.
34. Wainer G, Borho S, Pittner J. Defining and visualizing models of urban traffic. *Proceedings of the SCS 1st Mediterranean Multiconference on Modeling and Simulation*, Genoa, Italy, 2004.
35. Chopard B, Droz M. *Cellular Automata Modeling of Physical Systems*. Cambridge University Press: Cambridge, 1998.
36. Wolfram S. *A New Kind of Science*. Wolfram Media, 2002.
37. Simon P, Gutowitz H. A cellular automaton model of bi-directional traffic. *Physical Review E* 1998; **57**(2):2441–2444.
38. Dupuis A, Chopard B. Parallel simulation of traffic in Geneva using cellular automata. *Virtual Shared Memory for Distributed Architectures*, Kühn E (ed.). Nova Science: Commack, NY, 2001; 89–107.
39. Campari E, Levi G, Maniezzo V. Cellular automata and roundabout traffic simulation. *Proceedings of the ACRI'2004*, Amsterdam, Netherlands, 2004 (*Lecture Notes in Computer Science*, vol. 3305). Springer: Berlin, 2004.
40. Li X, Jia B, Gao Z, Jiang R. A realistic two-lane cellular automata traffic model considering aggressive lane-changing behavior of fast vehicle. *Physica A* 2006; **367**:479–486.
41. Zeigler B, Moon Y, Kim D, Ball G. The DEVS environment for high-performance modeling and simulation. *IEEE Computational Science and Engineering* 1997; **4**(3).
42. Wagner P, Nagel K, Wolf P. Realistic Multi-line traffic rules for cellular automaton. *Physica A* 1997; **234**:687.
43. Chopard B, Queloz PA, Luthi P. Cellular automata model of car traffic in two-dimensional street networks. *Journal of Physics A: Mathematical and General* 1996; **29**:2325–2336.
44. Ames A, Nadeau D, Moreland J. *VRML 2.0 Sourcebook*. Wiley: New York, 1996.

DOI: 10.1002/spe