Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

TESIS DE LICENCIATURA

TSC - Traffic Simulator Compiler

Diseño e implementación de un compilador para simulación de tráfico urbano siguiendo el paradigma Cell-DEVS

Alumnos: Mariana Lo Tartaro - L.U. 14/89
César Torres - L.U. 372/89
Director: Dr. Gabriel Wainer

Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires - Argentina
http://www.dc.uba.ar

INDICE

Resumen

En el presente trabajo se diseña y desarrolla el compilador TSC (Traffic Simulator Compiler) que toma como entrada un plano de ciudad descripto en un lenguaje sencillo, y genera el modelo de simulación correspondiente para ser ejecutado mediante una herramienta de simulación basada en modelos Cell-DEVS n-dimensionales.

El lenguaje de especificación de sectores de ciudad utilizado para describir el plano se denomina ATLAS (Advanced Traffic LAnguage Specifications) y su descripción completa se encuentra en [DW99].

El simulador para el cual se generan los modelos de simulación se denomina N-CD++ y su descripción completa se encuentra en [BBW98] y en [RW99].

Lo más destacable del compilador TSC es que la estructura del código que genera es totalmente flexible, basándose en templates que se indican en el momento de invocarlo. De esta manera, variando los templates, se pueden generar diferentes modelos de simulación para el mismo sector de ciudad utilizando el mismo compilador. Esto permite que el usuario de TSC pueda analizar y sacar conclusiones sobre el comportamiento de la simulación para los diferentes modelos generados.

Otra ventaja que se obtiene del uso de templates es que cualquier cambio o extensión que se realice en futuras versiones de N-CD++ no desencadena ninguna modificación en la arquitectura del compilador TSC, sino que solamente basta con modificar los templates de generación de código.

El conjunto de templates presentados en este trabajo respeta por completo la especificación de ATLAS para la descripción de un plano de ciudad. Así mismo se presenta un conjunto de templates alternativos para la simulación de un sector de ciudad analizado y estudiado en [DVW00]. El mismo es una pequeña variación de la especificación de ATLAS.

Abstract

The present work designs and develops TSC (Traffic Simulator Compiler) compiler that, taking as input a city plan descripted in a simplified language, generates the regarding simulation model to be executed thru a simulation tool based on n-dimensional Cell-DEVS models.

The city section specification language used to describe the plan is named ATLAS (Advanced Traffic LAnguage Specifications) and its complete description can be found at [DW99].

The simulator for which the simulation models are generated is named N-CD++ and its complete description can be found at [BBW98] and [RW99].

The major point of the TSC compiler is that the code structure generated is totally flexible, based on templates indicated at invocation time. In this way, altering the templates, different simulation models can be generated for the same city section using the same compiler. This allows TSC users to be able to analize and take conclusions about the simulation behaviour for the different generated models.

Another advantage for the templates usage is that any other change or extension made in future versions of N-CD++ doesn´t unchain modifications in the TSC compiler architecture, it is only necessary to alter the templates responsible for the code generation.

The set of templates presented at this work represents completely the ATLAS specification for the city plan description. Also, an alternative set of templates is presented for a city section simulation analized and studied at [DVW00]. The last one is a little variation for the ATLAS specification.

Introducción

Para todos aquellos sistemas en los que no es posible o conveniente hallar soluciones analíticas se ha difundido el uso de metodologías y herramientas de simulación.

Las ventajas de la simulación son múltiples. Entre otras cosas puede reducirse el tiempo de desarrollo del sistema, las decisiones pueden verificarse artificialmente y un mismo modelo puede utilizarse muchas veces. La simulación es de empleo más simple que ciertas técnicas analíticas y precisa menos simplificaciones.

En la actualidad existe una gran variedad de aplicaciones complejas en las que se usan modelos y/o simulación, entre ellas el estudio del tráfico urbano. Las características comunes a estos sistemas son su complejidad y la falta de herramientas adecuadas para evaluar su desempeño.

El objetivo de este trabajo es brindar a los analizadores y estudiosos del tráfico urbano una herramienta que les permita aprovechar la ventaja del uso de simuladores, especificando los sectores de ciudad a simular en un lenguaje que respete sus propios términos (calles, cruces, semáforos, etc).

Para cumplir con este objetivo, se diseñó y desarrolló el compilador TSC (Traffic Simulator Compiler) que toma como entrada un plano de ciudad descripto en lenguaje natural, y genera los modelos de simulación correspondientes.
La estructura de estos modelos permite que los mismos puedan ser luego utilizados por la herramienta de simulación N-CD++ (basada en modelos Cell-DEVS n-dimensionales), para analizar y estudiar el comportamiento del tráfico, es decir el movimiento de los autos a lo largo de las calles del sector de ciudad especificado.

El lenguaje de especificación de sectores de ciudad utilizado para describir el plano se denomina ATLAS (Advanced Traffic LAnguage Specifications) y su descripción completa se encuentra en [DW99].
El mismo describe un sector de ciudad en términos de los elementos que lo componen. Los principales son las calles y sus cruces correspondientes, complementados con factores de control adicionales que provocan la alteración del flujo normal de los autos. Entre los mismos podemos mencionar baches, obras, lomos de burro y otros elementos que se pueden encontrar a lo largo de una calle. Además se incorporan otros controles más sofisticados como semáforos y vías de tren.

Como se mencionó anteriormente, el simulador para el cual se generan los modelos del plano, se basa en modelos Cell-DEVS n-dimensionales.
Los modelos DEVS se construyen en base a un conjunto de modelos de simulación básicos, que se combinan para formar modelos acoplados. El paradigma Cell-DEVS permite la descripción de modelos celulares a través de su definición como modelos atómicos DEVS.
La definición de modelos de tráfico utilizando el paradigma Cell-DEVS brinda múltiples posibilidades para modelar cada elemento del plano y sus acoplamientos en modelos de mayor nivel. El simulador N-CD++ permite procesar cualquiera de estos modelos. Es por esta causa que se decidió crear un compilador totalmente parametrizable para la generación de los mismos de torma flexible. Gracias a esto, se pueden generar diferentes modelos de simulación para un mismo sector de ciudad, cambiando en forma muy simple los parámetros de la herramienta TSC.

Por otra parte, tambien se incorporan todas las facilidades para la extensión y adaptación de los modelos frente a futuras versiones del simulador.

La parametrización mencionada está especificacada en un conjunto de templates de generación de código que se indica en el momento de invocar a TSC.

TSC fue construido utilizando el lenguaje de programación C++ y el compilador GCC versión 2.8.1 (http://www.sunsite.unc.edu/pub/gnu) del proyecto GNU Free Software Foundation Inc. (http://www.fsf.org).

La documentación contenida en el presente informe se organiza de la siguiente forma:

Capítulo 1	**Formalismos de Especificación**
	En este capítulo se describen y especifican los formalismos DEVS y Cell-DEVS en los cuales se basan las especificaciones de modelos de simulación generadas por el compilador TSC.
Capítulo 2	**Lenguaje ATLAS**
	En este capítulo se describe el lenguaje ATLAS de especificicación de sectores de ciudad.
Capítulo 3	**Simulador N-CD++**
	En este capítulo se describe la herramienta utilizada para ejecutar las simulaciones de los modelos generados por el compilador TSC.
Capítulo 4	**Compilador TSC**
	En este capítulo se describe la estructura y funcionamiento del compilador TSC.
Capítulo 5	**Un caso de aplicación**
	En este capítulo se describe un ejemplo real de aplicación y utilización del compilador TSC.
Capítulo 6	**Estructura detallada de TSC**
	En este capítulo se encuentra el diseño detallado de las clases que componen la estructura interna del compilador TSC.

Capítulo 1 - Formalismos de Especificación

En esta sección se incluye una definición de los paradigmas de especificación DEVS y Cell_DEVS como fueran presentados en [WG98].

1.1. DEVS

DEVS (Discrete EVents dynamic Systems) es un mecanismo de simulación jerárquica propuesto por Zeigler [Zei76]. Establece una teoría de modelado de sistemas a tiempo continuo usando modelado de eventos discretos. El paradigma provee una forma de especificar un objeto matemático llamado sistema. Éste se describe como un conjunto consistente de una base de tiempo, entradas, salidas y funciones para calcular los siguientes estados y salidas.

El formalismo define cómo generar nuevos valores para las variables y los momentos en los que estos valores deben cambiar. Puede verse como una forma de especificar sistemas cuyas entradas, estados y salidas son constantes de a trozos, y cuyas transiciones se identifican como eventos discretos.

Un modelo DEVS se construye en base a un conjunto de modelos básicos, que se combinan para formar modelos acoplados. Los modelos pueden ser de comportamiento (atómicos), o estructurales (acoplados). Un modelo acoplado especifica cómo se conectan las entradas y salidas de los componentes. Los nuevos modelos también son modelos básicos modulares, y pueden usarse para armar modelos de mayor nivel.

Un *modelo atómico* es una especificación de un modelo comportamental. Formalmente, esta especificación puede definirse como:

$$\mathbf{M = <I, \mathit{X}, S, \mathit{Y}, \delta_{int}, \delta_{ext}, \lambda, D>}$$

I = < P^X, P^Y> representa la definición de la interfaz modular del modelo. En este caso, $\forall\ j \in [1, \eta]$, $i \in \{X, Y\}$, P_j^i es una definición de un port (de entrada o salida respectivamente), donde $P_j^i = \{ (N_j^i, T_j^i)\ /\ \forall\ j \in [1, \mu], (\mu \in \mathbf{N}, \mu < \infty), N_j^i \in [i_1, i_m]$ (nombre del port), y T_j^i = Tipo del port};

X es el conjunto de eventos externos de entrada;

S es el conjunto de estados secuenciales;

Y es el conjunto de eventos externos generados para salida;

δ_{int}: S→S es la función de transición interna, que define los cambios de estado por causa de eventos internos;

δ_{ext}: QxX→S es la función de transición externa, que define los cambios de estado por causa de eventos externos. Q es el conjunto de estados totales del sistema especificado como Q = { (s,e) / s ∈ S, e ∈ [0, D(s)] }, donde **e** representa el tiempo transcurrido desde la última transición de estado con estado **s**;

λ: S → Y es la función de salida;

D: S→R^+ es la función de duración de un estado, donde D(s) es el tiempo que el modelo se queda en el estado s si no hay un evento externo.

Para especificar modelos DEVS es conveniente ver al modelo como con ports de entrada/salida que interactúan con el entorno (**I**). Cuando se reciben en los ports de entrada los eventos externos, la descripción del modelo debe determinar cómo responder.

Un modelo atómico DEVS transita entre los estados (**S**) vía sus funciones de transición. Si no hay eventos externos, los estados pueden cambiar ejecutando la función de transición interna (δ_{int}), cuya activación está determinada por la función de pasaje de tiempo (**D**) aplicada al estado actual. Antes de cada transición interna, el modelo puede generar un evento de salida, ejecutando la función de salida, (λ) que depende del estado previo a la transición. También puede haber un cambio de estado cuando hay un evento de entrada externa (δ_{ext}). La función de transición externa determina el nuevo estado basándose en el estado actual, el tiempo transcurrido en ese estado, y la entrada.

Todo modelo atómico tiene definida dos variables de estado estándar: **fase** y σ. Si no hay eventos externos, el modelo se queda en la fase durante σ, que es el tiempo que resta hasta la próxima transición interna. Si hay eventos externos, la transición externa hace que el modelo cambie de fase y σ, preparándolo para la próxima transición interna. El estado en el que se entra luego de un evento externo depende de la entrada, el estado actual, y el tiempo transcurrido en este estado, permitiendo representar el comportamiento de sistemas de tiempo continuo con eventos discretos. Por ende, la función de avance de tiempo que controla el timing de las transiciones internas simplemente devuelve el valor de σ.

Resumiendo, un estado **s** con **D**(**s**) = ∞ se dice pasivo (un estado no pasivo se dice activo). La función de transición interna δ_{int} especifica un cambio de estado en el modelo luego del tiempo dado por D. Si hay un evento externo en el instante (**t**+**e**), se ejecuta la función de transición externa δ_{ext}, y el modelo cambia inmediatamente al estado **s'**= δ_{ext}(**s**,**e**,**x**). La variable e es el tiempo transcurrido en el estado actual, y se pone en cero luego de cambio de estado.

Cada port de entrada precisa de una especificación de la transición externa, en la forma de "cuando reciba x en el port de entrada p...". La función de transición interna puede especificarse en una descripción procedural con fases y sus transiciones, con la forma "enviar y al port de salida p".

Es importante notar que no hay forma de generar una salida directamente desde un evento de entrada externa. Una salida solo puede ocurrir antes de una transición interna. Para que un evento externo provoque una salida sin demora, hay que "planificar" un estado interno con duración cero.

Los modelos básicos pueden ser acoplados en el formalismo DEVS para formar un *modelo multicomponente o acoplados*, definido por la estructura:

CM = < I, X, Y, D, {M_i}, {I_i}, {Z_{ij}}, select>

I = < P^X, P^Y> representa la definición de la interfaz modular del modelo. En este caso, $\forall$ j $\in$ [1, η], $i \in$ {X, Y}, P_j^i es una definición de un port (de entrada o salida respectivamente), donde

P_j^i = { (N_j^i, T_j^i) / $\forall$ j $\in$ [1, μ], ($\mu \in \boldsymbol{N}$, $\mu < \infty$), $N_j^i \in [i_1, i_m]$ (nombre del port), y T_j^i = Tipo del port};

X es el conjunto de eventos externos de entrada;

Y es el conjunto de eventos externos generados para salida;

D $\in \boldsymbol{N}$, D < ∞, es el conjunto de índices de modelos componentes; y $\forall$ i $\in$ D,

$\mathbf{M_i}$ es un modelo componente básico, definido como:

$M_i = <I_i, X_i, S_i, Y_i, \delta_{inti}, \delta_{exti}, \lambda_i, D_i>$

$\mathbf{I_i} \subseteq$ D es el conjunto de influenciados de i; y para cada $\forall$ j $\in$ I_i,

$\mathbf{Z_{ij}}$: Yi $\rightarrow$ Xj es la función de traducción de salida i a j.

select: D$\rightarrow$D / $\forall$ E $\neq$ {$\emptyset$}, select(E) $\in$ E es el selector ante eventos simultáneos (función de secuenciación o prioridad).

Como vemos, un modelo acoplado dice como acoplar (conectar) varios modelos componentes para formar un nuevo modelo. Este modelo puede ser empleado como componente, permitiendo construcción jerárquica. El modelo acoplado contiene, por un lado, una interfaz modular (**I**),que le permite conectarse con otros modelos básicos. Luego, se definen los conjuntos de entrada y salida del componente (**X**, **Y**), como en otros modelos básicos. Por otro lado, se define un índice (**D**) para el conjunto de componentes ($\mathbf{M_i}$) que conforman el sistema acoplado. Se determinan las influencias de cada componente ($\mathbf{I_i}$), y una especificación de acoplamiento ($\mathbf{Z_{ij}}$), en la cual la conexión de i a se especifica designando a j como el influenciado por i. Z_{ij} provee una función de traducción desde el estado de i hasta el conjunto de entradas de j. Cuando ocurre un evento interno en i, en el mismo instante envía una señal al componente j. Finalmente, la función de selección, incluye las reglas empleadas para elegir cual de los componentes inminentes (los que tienen menor tiempo al próximo evento) ejecutará su próximo evento.

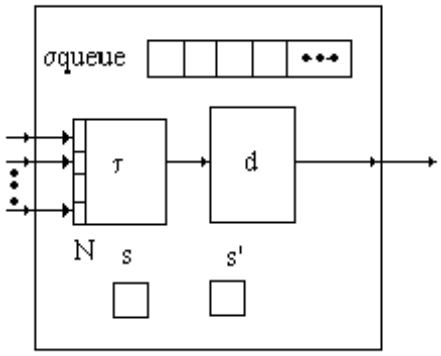
Un modelo multicomponente es equivalente a un modelo básico en el formalismo DEVS, y puede ser empleado en un modelo multicomponente mayor, ya que el formalismo es cerrado bajo acoplamiento. De esta forma, un modelo puede construirse jerárquicamente a partir de submodelos DEVS. Un modelo DEVS que no se construye usando un modelo acoplado es un modelo atómico.

1.2. Cell-DEVS

El paradigma Cell-DEVS permite la descripción de modelos celulares a través de su definición como modelos atómicos DEVS con distintos tipos de demoras. Cada celda será definida como un modelo atómico, y podrán utilizarse distintas clases de demoras para su comportamiento.

En los modelos Cell-DEVS con *demoras de transporte*, las celdas se definen como modelos atómicos, con el fin de acoplarlas con otras para formar un espacio de celdas completo. Se considera que los modelos son cerrados, ya que cada celda sólo puede influenciar o ser influenciada por un vecino.

La siguiente figura muestra informalmente los contenidos básicos de una celda atómica con demoras de transporte:



En este modelo atómico, una celda tiene η entradas en las que ocurren eventos externos. La celda memoriza los valores de todas las entradas, y cuando hay un nuevo evento externo, ejecuta la función booleana τ, que consume los valores de los eventos de entrada. Luego, el resultado del cálculo se demora durante **d** unidades de tiempo antes de ser transmitido a las celdas vecinas, planificando para ello un evento interno. Se debe usar una cola para mantener los valores de los resultados de los cálculos junto con su hora futura de planificación, ya que durante la demora pueden llegar nuevos eventos externos. Cuando llega la hora del evento interno, el valor se transmite por un port de salida.

Como los modelos influenciados sólo deben activarse cuando la celda influenciante cambia su estado, el resultado de la función de cálculo local sólo será transmitido si hay un cambio. Para permitir esta distinción, se guardan los valores presente y pasado de la celda.

Un modelo atómico como el descripto puede definirse formalmente como:

TDC = < *X*, *Y*, I, S, θ, N, d, δ_{int}, δ_{ext}, τ, λ, D >

donde para T < ∞ ∧ T ∈ {***N***, ***Z***, ***R***, ***{0,1}*** };

X ⊆ T es el conjunto de eventos externos de entrada;

Y ⊆ T es el conjunto de eventos externos de salida;

I = <η, μ, P^X, P^Y> representa la definición de la interfaz modular del modelo. En este caso,
η ∈ ***N***, η < ∞ es el tamaño de la vecindad,
μ ∈ ***N***, μ < ∞ es la cantidad de ports de entrada/salida independientes de la vecindad, y
∀ j ∈ [1, η], *i* ∈ {X, Y}, P_j^i es una definición de un port (de entrada o salida respectivamente),

P_j^i = { (N_j^i, T_j^i) / $\forall$ j $\in$ [1, $\eta+\mu$], $N_j^i \in [i_1, i_{\eta+\mu}]$ (nombre del port), y $T_j^i \in I_i$ (Tipo del port)},

I_i = { x / x $\in$ ***X*** si X } ó I_i = { x / x $\in$ ***Y*** si *i* = Y } ;

S $\subseteq$ T incluye todos los valores posibles de estados secuenciales para la celda;

θ es la definición del estado de la celda, definido como

θ = { (s, phase, σqueue, σ) /
s $\in$ S es el valor del estado para la celda,
phase $\in$ {activa, pasiva},
σqueue = { $((v_1,\sigma_1),...,(v_m,\sigma_m))$ / m $\in$ ***N*** $\wedge$ m <∞) $\wedge$ $\forall$ (i $\in$ ***N***, i $\in$ [1,m]), $v_i \in S \wedge \sigma_i \in \boldsymbol{R}_0^+ \cup \infty$}; y
σ $\in$ $\boldsymbol{R}_0^+ \cup \infty$} ;

N $\in$ S^η, es el conjunto de estados de los eventos de entrada almacenados;

d $\in$ $\boldsymbol{R}_0^+$, d < ∞ es la demora de transporte de la celda;

δ_{int}: θ $\rightarrow$ θ es la función de transición interna;

δ_{ext}: Qx*X* $\rightarrow$ θ es la función de transición externa, donde Q es el conjunto de estados definido como:

Q = { (s, e) / s $\in$ θ x N x d; e $\in$ [0, D(s)]};

τ: N $\rightarrow$ S es la función de cálculo local;

λ: S $\rightarrow$*Y* es la función de salida; y

D: θ x N x d $\rightarrow$ $\boldsymbol{R}_0^+ \cup \infty$, es la función de duración de vida del estado.

Cada celda tiene una interfaz bien definida, compuesta por un número fijo de ports numerados en orden ascendente. Por un lado, existe un conjunto de ports para establecer el acoplamiento interno del modelo de celdas, cada uno de los cuales estará conectado con un vecino. El número de estos ports de entrada y salida de cada celda será, por ende, igual al del tamaño de la vecindad. En el caso de precisarse otras entradas o salidas, se utilizarán los demás ports definidos.

Cada port en la interfaz tiene un nombre y un tipo. Los nombres están compuestos por un identificador (**X** para entradas; **Y** para salidas) y un número natural (número de port). Tanto los conjuntos de entrada y salida (***X*** e ***Y***) como el conjunto de estados secuenciales, así como los tipos de los ports, pueden representar cualquier conjunto finito de símbolos, pero se ha elegido trabajar con un conjunto de tipos básicos con sus respectivas álgebras: ***N***, ***Z***, ***R***, y ***Booleanos***. Esta elección se debe a que la mayoría de las aplicaciones usan datos en estos dominios, y que muchos conjuntos de símbolos pueden mapearse en estos.

El estado de la celda se define como un conjunto compuesto por:

El valor actual de la celda;
La fase del modelo, que puede ser activa o pasiva.
Una cola (**σqueue**) usada para mantener los tiempos de los próximos eventos y sus valores de entrada asociados; y
La variable **σ**, que representa el tiempo simulado para el siguiente evento interno planificado;

El conjunto **N** representa el conjunto de valores de entrada de la celda, que tiene la forma de una η-upla $(s_1,..., s_\eta)$ donde $s_i \in S$. Este conjunto registra los valores actuales usados para calcular el valor futuro a través de la función de cálculo local **τ**.

La función de duración de vida **D** se usa para controlar el tiempo de duración del estado de una celda. En este caso, **D**(s, phase, σqueue, σ, N, d) = t representa el tiempo durante el cual, si no hay eventos externos, el modelo atómico conservará el estado actual. Tanto la función de vida del estado como la demorade la celda tienen dominio en los números reales, ya que la base de tiempo seleccionada es continua.

Los objetivos de la ejecución de las funciones de transición (δ_{int}, δ_{ext}) y de la de salida (**λ**) son similares a los definidos para otros modelos DEVS. La función de transición interna se usa para definir cambios de estado debido a eventos internos, y la función de transición externa expresará la ocurrencia de eventos externos.

En cambio, la semántica de estas funciones será diferente a la de otros modelos DEVS. Esto se debe a que cada celda puede tener asociada una demora de transporte (**d**), que permite postergar la ejecución de la función de transición interna. La construcción de demora de transporte permite que el cambio de estado ante la ocurrencia de un evento externo sea demorado, y que el estado del sistema sólo cambie al consumirse el tiempo correspondiente a la demora. Otra diferencia es que una vez ejecutada la función de transición interna, los modelos DEVS atómicos pasan a un estado pasivo hasta recibir nuevos eventos externos. En cambio, la introducción de demoras de transporte implica que una celda debe quedar activa durante la duración de la demora, ya que en ese lapso se pueden recibir nuevos eventos externos. Esto puede provocar que haya varios eventos internos planificados a futuro.

La semántica para la función de transición externa es la siguiente: la llegada de un evento externo indica que un vecino ha cambiado. Por ende, la celda debe activarse, y calcular su función de cómputo local. La demora de transporte indica que se planificará la ejecución futura de la función de transición interna, para lo cual se almacenan los valores de la demora y de la entrada en la cola **σqueue**. Asimismo, si al hacer el cálculo el estado de la celda no cambia, sus vecinos tampoco pueden cambiar. Por ende, el estado actual no se encola para ser transmitido.

Una celda se pone en estado pasivo sólo cuando no tiene más eventos planificados, o sea, cuando la cola **σqueue** esté vacía. El valor de la variable **σ** será, por ende, igual al primer valor de esta cola. Si la celda está activa, la llegada de un nuevo evento implica que, además de encolar el elemento, los valores de σ almacenados en la cola debe ser actualizados para reflejar el tiempo transcurrido (e).

La función de transición interna es función de la demora de la celda. Esta función (y la de salida) deben activarse cuando σ=0. Como la demora de transporte ha expirado, debe transmitirse el nuevo estado de la celda. La función de salida **λ** se ejecuta antes de la función δ_{int}, como en cualquier otra especificación DEVS, la que al activarse tomará el primer elemento de la cola de espera para ser

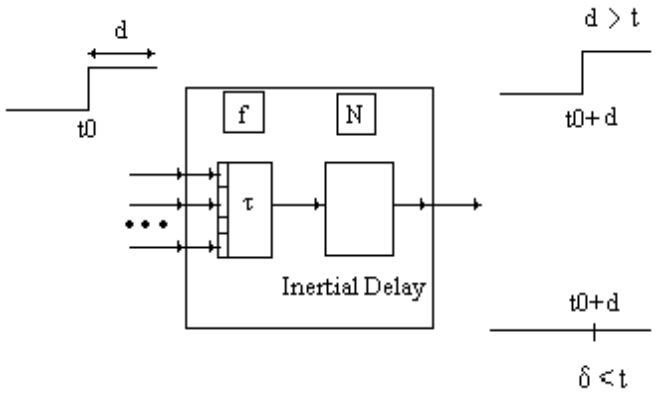
transmitido. La función de transición interna sólo elimina el primer miembro de la cola (cuyo valor ya fue transmitido), y actualiza los tiempos de la cola de espera y de la variable de estado σ.

El comportamiento detallado de estas funciones puede verse en la siguiente figura:

```
δext((s, phase, σqueue, σ, N, d), e, x) = (s, phase, σqueue, σ, N, d)
{
 /* Hay un evento de entrada: calcular τ, y si el nuevo valor es igual al anterior, no hacer nada */
    Actualizar los valores de N con el valor de x;
    s' = τ(N);              /* Calcula la función de cálculo local */

    if (s' != s) then           /* Los valores deben transmitirse sólo si cambia el estado */
        s = s';
        if (phase = pasiva) then    /* La celda está pasiva. Activarla. */
                phase = activa;
                σ = d;                      /* Demora de transporte */
        else
                  if (d < σ) σ = d endif;  /* Las demoras de una celda pueden ser variables, y podría llegar
                  un evento externo con menor demora que las ya almacenadas */
                for (ai ∈ σqueue) ai.σ = ai.σ - e;  /* La celda está activa. El tiempo transcurrido
        σ = σ - e;                          debe actualizar los valores futuros de σqueue */
        endif
        σqueue = insertar(σqueue, <s, d>); /* Inserción ordenada por σ */
    }
}

δint(s, phase, σqueue, σ) = (s, phase, σqueue, σ)
{       /* Los tiempos de σqueue deben actualizarse */
        for (ai ∈ σqueue) ai.σ = ai.σ - first(σqueue.σ);
        σqueue = tail (σqueue);

        if (empty(σqueue)) then /* No hay más eventos planificados */
                phase = pasiva;
                σ = ∞;
        else                    /* planificar el siguiente evento */
                phase = activa;
                σ = first(σqueue.σ);
        endif
}

λ(s) {
  return first(σqueue.value);
}
```

Otro tipo de demora interesante para las celdas de los modelos celulares es la *demora inercial*. Esta construcción permite representar ciertos fenómenos cuya semántica incluye comportamiento con desalojo, permitiendo descartar entradas a las celdas si su valor no se mantiene durante un período determinado. Por otro lado, si el flujo de entrada es constante durante ese tiempo (llamado demora inercial) el estado debe cambiar.

La siguiente figura ejemplifica el comportamiento de una demora inercial para una celda atómica:



Un modelo atómico de celda con demoras inerciales puede especificarse como:

IDC = < *X*, *Y*, I, S, θ, N, d, δ_{int}, δ_{ext}, τ, λ, D >

donde para T < ∞ ∧ T ∈ {***N***, ***Z***, ***R***, ***{0,1}*** };

X ⊆ T es el conjunto de eventos externos de entrada;

Y ⊆ T es el conjunto de eventos externos de salida;

I = <η, μ, P^x, P^y> representa la definición de la interfaz modular del modelo. En este caso,
η ∈ ***N***, η < ∞ es el tamaño de la vecindad,
μ ∈ ***N***, μ < ∞ es la cantidad de ports de entrada/salida independientes de la vecindad, y
∀ j ∈ [1, η], *i* ∈ {X, Y}, P_j^i es una definición de un port (de entrada o salida respectivamente),
P_j^i = { (N_j^i, T_j^i) / ∀ j ∈ [1, η+μ], $N_j^i \in [i_1, i_{\eta+\mu}]$ (nombre del port), y $T_j^i \in I_i$ (Tipo del port)},
I_i = { x / x ∈ ***X*** si *i* = X } ó I_i = { x / x ∈ ***Y*** si *i* = Y } ;

S ⊆ T incluye todos los valores posibles de estados secuenciales para la celda;

θ es la definición del estado de la celda, definido como

θ = { (s, phase, f, σ) / s ∈ S, phase ∈ {activa, pasiva}, f ∈ T, y σ ∈ $\boldsymbol{R}_0^+ \cup \infty$ } ;

N ∈ S^η, es el conjunto de estados de los eventos de entrada almacenados;

d ∈ $\boldsymbol{R}_0^+$, d < ∞ es la demora inercial de la celda;

δ_{int}: S → S es la función de transición interna;

δ_{ext}: Q*x*X → θ es la función de transición externa, donde Q es el conjunto de estados definido como:

Q = { (s, e) / s ∈ θ x N x d ∧ e ∈ [0, D(s)]};

τ: N → S es la función de cálculo local;

λ: S →*Y* es la función de salida; y

D: θ x N x d → $\boldsymbol{R}_0^+ \cup \infty$, es la función de duración de vida del estado.

Como puede verse, la mayoría de los conjuntos y funciones fueron definidos de forma similar a los presentados para modelos con demoras de transporte, pero existen algunas diferencias. Por un lado, la definición para el estado de una celda ha sido modificada. En este caso, s, phase y σ tienen el mismo significado definido anteriormente, pero f representa el valor futuro factible para la celda. Si el valor de entrada de la celda se mantiene durante la demora inercial, f se convertirá en el estado actual de la celda.

La definición de la variable **d** corresponde ahora a una demora inercial, lo que motiva un cambio en la semántica de las funciones de transición, que ahora deben modelar demoras inerciales. Para este caso, el comportamiento de las funciones de transición se ha definido como sigue:

```
δext((s, phase, f, N, d, σ), e, x) = (s, phase, f, N, d, σ)
{
        Actualizar los valores de los eventos de entrada con el valor de x;
        s' = τ(N);

        if (s != s') then
          s= s';              /* El nuevo estado es el recién calculado */

            if (phase = pasiva) then
                phase = activa;
                σ = d;    /* Demora inercial */
             else
                σ = σ - e;
                if (σ > 0 .AND. f != s)  σ = d;  /* Remoción: el estado calculado es distinto al futuro */
            endif
        endif
        f = s;      /* El estado futuro es el recién obtenido */
}

δint(s, phase, f, σ) = (s, phase, f, σ)
{
        if (σ = 0) then
                phase = pasiva;
                σ = ∞;
        endif
}

λ(s) {
        return s;
}
```

Podemos ver que el objetivo de la función de transición externa es almacenar el valor actual para la celda, y detectar si la entrada se conserva durante la demora inercial. Si no es así, la entrada anterior es removida. Al llegar la hora planificada por la demora, se transmite el valor actual de la celda.

Los modelos atómicos de celdas con distintas demoras pueden acoplarse con otros modelos para formar un *modelo Cell-DEVS multicomponente o acoplado*. Éstos son definidos como un espacio consistente de celdas atómicas conectadas por una relación de vecindad. Cada uno de los componentes es una celda como las definidas anteriormente. En esta especificación se consideran modelos cerrados (es decir, que no pueden acoplarse con otros modelos de base). Por ende, al no usarse entradas ni salidas, tampoco hay necesidad de definir una interfaz para el modelo.

Un modelo ejecutable de celdas Cell-DEVS acopladas puede definirse como:

$CC = < n, \{t_1,..., t_n\}, \eta, N, C, B, Z, select >$

donde

n es la dimensión del espacio de celdas;

$\{\mathbf{t_1,...,t_n}\}$ es la cantidad de celdas en cada una de las dimensiones;

η es el tamaño de la vecindad;

N es el conjunto de vecindad;

C es el espacio de celdas;

B es el conjunto de celdas del borde;

Z es la función de traducción; y

select es la función de selección ante eventos simultáneos.

En este caso,

$n \in N$;

$\{t_1,...,t_n\} \in N$, con $t_k < \infty \; \forall \; k \in [1,n]$;

$\eta \in N$;

$N = \{ (v_{k1},...,v_{kn}) / \forall (k \in N, k \in [1, \eta] \wedge i \in N, i \in [1, n]), v_{ki} \in Z \wedge (t_i - | v_{ki} | \geq 0) \}$;

$C = \{ C_C / c \in I \wedge C_C = < I_C, X_C, Y_C, S_C, N_C, d_C, \delta_{intC}, \delta ext_C, \tau_C, \lambda_C, D_C >$ es un componente TDC ó IDC }, siendo

$I = \{ (i_1,...,i_n) / (i_k \in N \wedge i_k \in [1, t_k]) \; \forall \; k \in [1, n] \}$ (1)

$B = \{\emptyset\}$ si el espacio de celdas es toroidal; o
$B = \{C_b / C_b \in C \text{ con } b \in I\}$, con I definido como en (1). En este caso, el conjunto B está sujeto a la restricción que $\tau_b \neq \tau_c = \tau \ \forall \ (C_c \notin B) \wedge (C_b \in B)$.

$B = \{C_b / C_b \in C \text{ con } b \in L\}$, siendo $L = \{ (i_1,...,i_n) / i_j = 0 \vee i_j = t_j \ \forall \ j \in [1, n] \}$, y sujeto a que $\tau_b \neq \tau_c = \tau \ \forall \ c \notin L$.

Definición

Si llamamos c a la posición de una celda, donde $c = (i_1,...,i_n)$; y sea $V \in \boldsymbol{Z}^n$ la n-upla definida como $V = (v_1, ..., v_n)$, con $v_i \in \boldsymbol{Z}$, $| v_i | \leq t_i$ entonces se define como $D = C +_t V$ (2) a la n-upla $D = (j_1,...,j_n)$ definida por: $j_k = (i_k + v_k) \bmod (t_k) \ \forall \ k \in [1, n]$.

Notaciones

$\forall \ k \in \boldsymbol{N}, k \in [1, \eta]$, se denotará $V_k = \{ (v_{k1},...,v_{kn}) / (v_{k1},...,v_{kn}) \in N \}$ al k-ésimo elemento de una lista de vecindad.

Se denotará P_c^{iq} al port $P_q^i \in I_c$, con $i \in \{X, Y\}$, y ($q \in \boldsymbol{N}$, $q \in [1, \eta]$), donde $I_c \subseteq C_c$, y C_c es un componente TDC ó IDC

Z: $I_C \rightarrow I_D$ es la función de traducción, definida como:

$Z(P_C{}^{Y_q}) = P_D{}^{X_q}$, $\forall$ ($q \in \boldsymbol{N}$, $q \in [1,\eta]$), donde $\forall \ V_k \in N$, $D = C +_t V_k$; y
$Z(P_C{}^{X_q}) = P_D{}^{Y_q}$, $\forall$ ($q \in \boldsymbol{N}$, $q \in [1,\eta]$), donde $\forall \ V_k \in N$, $D = C -_t V_k$.

select = $\{ (s_1, ..., s_n) / s_i \in \boldsymbol{N}, \forall \ i \in [0, n] \}$, con la restricción que select $\subseteq t_1xt_2x...xt_n \rightarrow t_1xt_2x...xt_n$.

En este caso, el espacio de celdas **C** es un modelo acoplado definido como un arreglo de modelos atómicos Cell-DEVS de tamaño fijo (t_1 x ... x t_n).

Cada celda tiene un conjunto de $\boldsymbol{\eta}$ vecinas, definidas por el conjunto de vecindad (**N**). Este conjunto está representado como una lista de tuplas de dimensión n que definen la posición relativa entre la celda origen y sus vecinos. Estos índices no pueden exceder el límite del espacio de celdas.

El conjunto **B** define el borde del espacio de celdas, y puede ser de dos tipos distintos. Si $B = \{\emptyset\}$, toda celda en el espacio tendrá el mismo comportamiento: las celdas en un borde se conectan con las que están el borde opuesto usando la relación de vecindad inversa. En cambio, si el conjunto de borde es no vacío, las celdas tendrán un comportamiento diferente a las otras del modelo (por ejemplo, pueden generar su estado, o consumir los estados de los vecinos).

La función **Z** permite definir el acoplamiento entre celdas en el modelo. Esta función traduce las salidas del q-ésimo port de salida en la celda C_c en valores para el q-ésimo port de entrada de la

celda C_D (y viceversa). Cada port de salida corresponderá a un vecino, y cada port de entrada estará asociado con una celda en la vecindad inversa [Zei76].

Los nombres de los ports se generan usando la siguiente notación: $P_C{}^{X_q}$ se refiere al q-ésimo port de entrada de la celda C_c, y $P_C{}^{Y_q}$ al q-ésimo port de salida. Estos ports corresponden con los nombres de ports denotados como X_q o Y_q para cada celda. El número de la celda con la cual debe acoplarse el modelo será generado sumando los números de la lista de vecinos al número de la celda actual. El primer port de salida de una celda estará conectado con el primer port de entrada del vecino, de acuerdo con el orden de la lista de vecinos.

La definición presentada sólo incluye permite vecindades regulares y vecinos en celdas adyacentes. Si fueran necesarios otros tipos de vecindades (incluyendo distintas vecindades para cada celda), la definición puede extenderse. Aquí,

$N = \{N_C / c \in \mathbf{I}\}$ con **I** definido como en (4)

donde

$N_C = \{ (v_{k1},...,v_{kn})_C / \forall (k \in \mathbb{N}, k \in [1, \eta_C]) \wedge (i \in \mathbb{N}, i \in [1, n]), v_{ki} \in \mathbb{Z} \wedge (ti - |v_{ki}| \geq 0) \};$

Por último se extiende la definición de los modelos Cell-DEVS acoplados cerrados a *modelos Cell-DEVS acoplados genéricos*. La idea es extender la definición de los primeros para permitir la especificación de espacios Cell-DEVS que puedan ser combinados con otros modelos básicos en una jerarquía DEVS.

Para permitir la definición múltiple de submodelos y su acoplamiento, un modelo acoplado Cell-DEVS genérico puede representarse como:

$GCC = <$ Xlist, Ylist, I, *X*, *Y*, n, $\{t_1,...,t_n\}$, η, N, C, B, Z, select $>$

En este caso,

Ylist es la lista de acoplamiento de salida;

Xlist es la lista de acoplamiento de entrada;

I representa la definición de la interfaz modular del modelo;

X es el conjunto de eventos externos de entrada;

Y es el conjunto de eventos externos de salida;

n es la dimensión del espacio de celdas;

$\{\mathbf{t_1,...,t_n}\}$ es la cantidad de celdas en cada una de las dimensiones;

$\boldsymbol{\eta}$ es el tamaño de la vecindad;

N es el conjunto de vecindad;

C es el espacio de celdas;

B es el conjunto de celdas del borde;

Z es la función de traducción; y

select es la función de selección ante eventos simultáneos.

Donde, para $T < \infty \wedge T \in \{\boldsymbol{N}, \boldsymbol{Z}, \boldsymbol{R}, \boldsymbol{\{0,1\}}\}$;

Ylist $\subseteq \{ (i_1,...,i_n) / i_k \in [1,t_k] \forall k \in [1,n], k \in \boldsymbol{N}\}$;

Xlist $\subseteq \{ (i_1,...,i_n) / i_k \in [1,t_k] \forall k \in [1,n], k \in \boldsymbol{N}\}$;

I = $< P^x, P^y >$ representa la definición de la interfaz modular del modelo. En este caso,

$\forall$ c $\in$ **I**, con **I** definido como en (4) $i \in \{X, Y\}$, P_c^i es una definición de un port,

$P_c^i = \{ (N_c^i, T_c^i) / \forall c \in i\text{list}, N_c^i \in i(c)$ (nombre del port), y $T_c^i \in T$ (Tipo del port)};

$\boldsymbol{X} \subseteq T$;

$\boldsymbol{Y} \subseteq T$;

$\mathbf{n} \in \boldsymbol{N}$;

$\{\mathbf{t_1,...,t_n}\} \in \boldsymbol{N}$, con $t_k < \infty \forall k \in [1,n]$;

$\boldsymbol{\eta} \in \boldsymbol{N}$;

N = $\{ (v_{k1},...,v_{kn}) / \forall (k \in \boldsymbol{N}, k \in [1, \eta]) \wedge (i \in \boldsymbol{N}, i \in [1, n]); v_{ki} \in \boldsymbol{Z} \wedge (ti - | v_{ki} | \geq 0) \}$;

C = { $C_c / c \in$ **I** $\wedge C_c = < I_c, X_c, Y_c, S_c, N_c, d_c, \delta_{int}c, \delta ext_c, \tau_c, \lambda_c, D_c >$ es un componente TDC ó IDC tal como los definidos anteriormente }, siendo **I** definido como en (4);

B = {$\varnothing$} si el espacio de celdas es toroidal; o
B = {$C_b / C_b \in$ C con b $\in$ **I** }, con **I** definido como en (4). En este caso, el conjunto B está sujeto a la restricción que $\tau_b \neq \tau_c = \tau \ \forall (C_c \notin B) \wedge (C_b \in B)$.

B = {$C_b / C_b \in$ C con b $\in$ L}, siendo L = $\{ (i_1,...,i_n) / i_j = 0 \vee i_j = t_j \forall j \in [1, n] \}$, y sujeto a que $\tau_b \neq \tau_c = \tau \forall c \notin L$.

Z: $I_C \rightarrow I_D$ es la función de traducción, definida como:

$Z(P_C{}^{Y_q}) = P_D{}^{X_q}$, $\forall$ ($q \in \boldsymbol{N}$, $q \in [1,\eta]$), donde $\forall\ V_k \in N$, $D = C +_t V_k$; y
$Z(P_C{}^{X_q}) = P_D{}^{Y_q}$, $\forall$ ($q \in \boldsymbol{N}$, $q \in [1,\eta]$), donde $\forall\ V_k \in N$, $D = C -_t V_k$.

select = { $(s_1, ..., s_n)$ / $s_i \in \boldsymbol{N}$, $\forall\ i \in [0, n]$ }, con la restricción que select $\subseteq t_1xt_2x...xt_n \rightarrow t_1xt_2x...xt_n$.

Se pueden encontrar algunas diferencias con la especificación de los espacios Cell-DEVS cerrados. Primero, como en todo modelo acoplado DEVS que admite entradas y salidas, han sido incluidos los conjuntos ***X*** e ***Y***. Como estos modelos pueden ser acoplados con otros modelos DEVS, también ha sido definida la interfaz **I**.

La función **Z** se sigue usando para llevar a cabo el acoplamiento interno del espacio de celdas. Finalmente, se han incluido dos nuevos conjuntos: **Xlist**, una lista de las posiciones de las celdas donde se reciben eventos externos al modelo acoplado; e **Ylist**, una lista de posiciones de celdas cuyas salidas serán recolectadas para ser enviadas a otros modelos en la jerarquía.

En modelos DEVS acoplados, la ocurrencia de eventos internos en un modelo es precedida por la ejecución de la función de salida λ. Esta función transmite el valor a otros modelos a través de ports en la interfaz. Para modelos Cell-DEVS, el comportamiento será distinto. Primero, si ocurre un evento en una celda, sus vecinos serán influenciados automáticamente a través de la ejecución de la función Z. Por otro lado, ciertas celdas en el espacio serán elegidas como celdas de entrada y salida, y serán incluidas en las listas Xlist e Ylist respectivamente. Los valores de estas celdas serán considerados como las entradas y salidas de todo el espacio de celdas.

La función select se define como una lista de posiciones en la vecindad. La lista se ordena de acuerdo con el criterio de selección a ser utilizado cuando más de una celda está activa simultáneamente.

Capítulo 2 - ATLAS

ATLAS (**A**dvanced **T**raffic **LA**nguage **S**pecifications) es el lenguaje de especificación definido en [DW99] para modelar las secciones de una ciudad utilizando los formalismos DEVS y Cell-DEVS. El objetivo del mismo es poder diseñar modelos de tráfico complejo de una forma sencilla, permitiendo además la incorporación de distintos elementos de control que alteran el flujo de los autos por las calles.

En ATLAS, una sección de ciudad se especifica como un conjunto de calles conectadas por cruces. Los autos avanzan en función de su velocidad, sobrepasando a otros autos que circulan a velocidades menores.

2.1. Calles

Una calle se especifica como una secuencia de *tramos*. Cada uno de éstos representa un sector que se extiende a lo largo de una cuadra, donde todos los carriles tienen la misma dirección de circulación y una velocidad máxima permitida. Es decir, constituyen una sección de la calle sin cruces y de mano única. Consecuentemente para construir un sector *doble mano* es necesario definir un tramo en cada dirección.

Los tramos se pueden especificar como:

$$\textbf{Tramos} = \{(p1, p2, n, a, dir, max) / p1, p2 \in \textbf{Puntos} \wedge p1 \neq p2 \wedge n, max \in \mathbb{N} \wedge a, dir \in \{0, 1\}\}$$

Por lo tanto, para cada tramo o elemento de este conjunto se deben identificar sus extremos, la cantidad de carriles, si tendrá forma recta o curva, el sentido de circulación de los vehículos y finalmente, su velocidad máxima permitida.
Entonces, un tramo t es una tupla de seis elementos:

$$\mathbf{t = (p1, p2, n, a, dir, max)}$$

donde,

p1 y p2, representan los extremos del tramo y pertenecen al conjunto Puntos que se define como:
Puntos = { (x,y) / x, y $\in \mathbb{Z}$ }
Se pide p1 ≠ p2, pues en el modelo no tiene sentido un tramo de longitud 0.

n $\in \mathbb{N}$, indica la cantidad de carriles del tramo.

dir $\in$ {0,1}, indica el sentido de circulación de los vehículos. Si dir = 1 los vehículos se desplazan hacia p2, caso contrario lo hacen hacia p1.

a $\in$ {0, 1}, indica la forma del tramo. Aquí:
a = 0 $\Rightarrow$ el tramo es un segmento recto determinado por los puntos p1 y p2.

a = 1 ⇒ el tramo queda definido por una de las semicircunferencias que se obtienen al partir la circunferencia cuyo diámetro es el segmento que une a p1 y p2. La circunferencia se parte utilizando el segmento que une a p1 y p2 como eje del corte, obteniendo 2 semicircunferencias con modelo subyacente idéntico y por ende no es necesario que la especificación diferencie ambos casos (este detalle quedará en el nivel de la interfaz con el modelador).

max ∈ ℕ, indica la velocidad máxima de circulación permitida en el tramo.

Los tramos se definen modelos Cell_DEVS de dimension igual a los carriles que contienen y con demora de transporte.

Las reglas del movimiento de los vehículos establecen que un auto siempre intenta moverse hacia adelante, si no hay suficiente espacio para esa maniobra intenta hacia adelante en diagonal izquierda y si ninguno de los 2 movimientos anteriores son posibles, intenta con la diagonal derecha. Por último, si no ha logrado avanzar por la falta de espacio entonces conservará su posición. Luego, los vehículos que avanzan derecho (sobre su propio carril) tienen prioridad de hacerlo frente a algún otro que quiera acceder a la misma posición desde otro carril. Además los automóviles que avanzan en diagonal izquierda tienen prioridad sobre los que quieren acceder a la misma posición avanzando en diagonal derecha. Por lo tanto, para que un vehículo se pueda mover derecho sólo debe pedir que haya lugar adelante. Para moverse al carril izquierdo debe verificar que tenga lugar (celda vacía) y que no haya otro auto que quiera acceder a la misma posición con movimiento recto. Por último, para un vehículo se pueda mover al carril derecho debe verificar que tenga lugar (celda vacía) y que no haya otro auto que quiera acceder a la misma posición con movimiento recto o en diagonal izquierda.

2.2. Cruces

El conjunto de los cruces se obtiene a partir de los tramos definidos como:

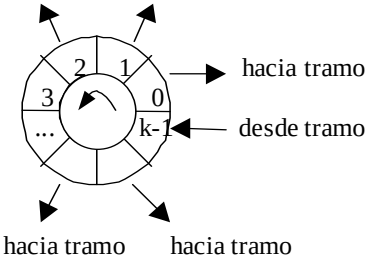
Cruces = { (c, maxc) / maxc ∈ ℕ ∧ ∃ t ,t' ∈ Tramos ∧ t = (p1, p2, n, a, dir, max) ∧ t' = (p1', p2', n', a', dir', max') ∧ t ≠ t' ∧ (p1 = c ∨ p2 = c) ∧ (p1' = c ∨ p2' = c) }

Los elementos de este conjunto se definen como puntos del espacio de dos dimensiones que representan los lugares donde se cruzan 2 ó más tramos, asociados con su velocidad máxima de circulación permitida. Pero un cruce siempre debe tener por lo menos un tramo de ingreso y uno de salida, pues si esto no sucede en realidad no se trata de una intersección de calles sino del lugar donde nacen o terminan. De esta forma los vehículos que ingresan al cruce por algún tramo siempre tendrán por lo menos una salida disponible.

Las intersecciones se representan como un anillo de celdas que se acopla con carriles de tramos. Las reglas de comportamiento de los vehículos establecen que un auto dentro de la intersección (en el anillo) tiene prioridad para acceder a una posición sobre cualquier otro vehículo que esté fuera de ella. Dentro del cruce, un vehículo gira en sentido contrario a las agujas del reloj o sale, es decir no se permite que un auto se detenga a la espera de una salida particular porque esto puede provocar que en algún momento, nadie se pueda mover por estar esperando que otro vacíe la posición (deadlock). Cada vehículo avanza hacia una celda vacía, que puede estar dentro de la intersección o

ser la primera de un carril de salida del cruce. Para modelar la elección del enlace de salida, se utiliza una función aleatoria local a la celda. Cada vez que un auto pasa por una salida, si hay lugar en el tramo, el vehículo chequea el valor que devuelve esta función para saber si debe seguir girando o salir. Caso contrario, el vehículo permanecerá en el cruce.

Cada cruce (c, maxc) ∈ Cruces, se define como un modelo Cell_DEVS de una dimensión con demora de transporte y bordes conectados, cuya estructura se presenta en la siguiente figura:



2.3. Marco Experimental

Los tramos que permiten el ingreso y salida de vehículos de la simulación se obtienen como:

TramosIngreso = { t / t = (p1, p2, n, a, dir, max) ∧ t ∈ Tramos ∧ [(dir = 0 ∧ (∄ v ∈ ℕ : (p2,v) ∈ Cruces)) ∨ (dir = 1 ∧ (∄ v ∈ ℕ : (p1,v) ∈ Cruces))] }

TramosSalida = { t / t = (p1, p2, n, a, dir, max) ∧ t ∈ Tramos ∧ [(dir = 0 ∧ (∄ v ∈ ℕ : (p1,v) ∈ Cruces)) ∨ (dir = 1 ∧ (∄ v ∈ ℕ : (p2,v) ∈ Cruces))] }

TramosIngreso y TramosSalida son subconjuntos de Tramos y se caracterizan por no tener un cruce de calles en alguno de sus extremos, el cual actuará como punto de ingreso o salida de vehículos de la simulación. Por lo tanto, estos conjuntos se derivan a partir de Tramos y Cruces.

Cada tramo de los conjuntos TramosIngreso y TramosSalida se acopla con un modelo DEVS, encargado de generar o eliminar coches de la simulación, respectivamente.

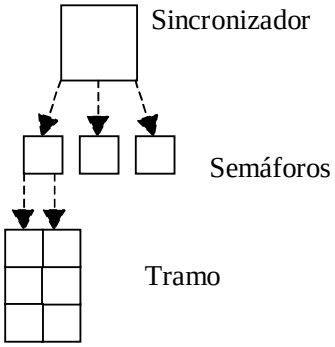
2.4. Semáforos

Los semáforos se especifican como:

CrucesSemáforos = { c / c ∈ Cruces }

Cada cruce de este conjunto representa una esquina con semáforos. Es decir, los vehículos que llegan a la intersección deben chequear el color del semáforo para determinar si pueden avanzar.

La presencia de semáforos se representa utilizando modelos adicionales al cruce y tramos afectados. Se define un modelo DEVS (Semáforo) para cada calle de la intersección, que informa del color del semáforo a las celdas del tramo. Luego, por cada cruce se construye un modelo DEVS (Sincronizador) encargado de avisar a cada semáforo cuándo le corresponde la luz verde, es decir sincroniza todos los semáforos del cruce. Estos modelos se grafican en la siguiente figura:



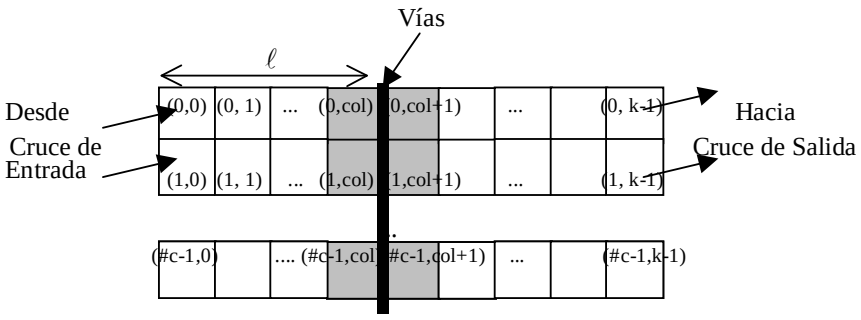
2.5. Trenes

El trazado de las vías del tren se especifica como:

RedDeVías = { Vías / Vías = { (t, ℓ, seq) / t ∈ Tramos ∧ ℓ ∈ ℕ ∧ seq ∈ ℕ } }

Cada elemento Vías ∈ RedDeVías representa el trazado de las vías de algún ramal de trenes. Para especificarlo se indican los lugares donde se ubican los pasos a nivel (intersección entre las vías y las calles donde se permite el cruce de los vehículos). Cada tupla, pn = (t, ℓ, seq), identifica la ubicación de un paso a nivel, es decir el tramo (t) y la distancia entre el comienzo del tramo y las vías (ℓ). Además indica el orden que le corresponde al paso a nivel (seq) para poder establecer la secuencia de avance del tren (en qué orden avanza sobre los pasos a nivel).

La influencia de los trenes sobre el flujo de vehículos es a través de los pasos a nivel (con y sin barreras) que impiden el avance de los autos por un determinado tiempo. Las celdas afectadas se muestran en la siguiente figura:



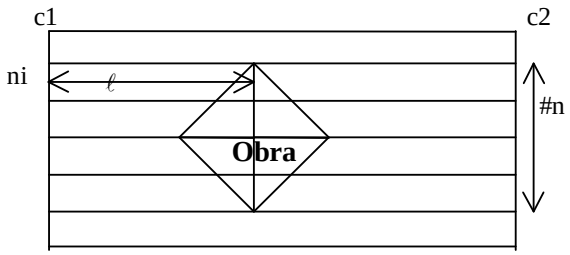
2.6. Obras

Las obras son secciones de calles deshabilitadas para la circulación de vehículos, debido a la presencia de obreros trabajando. Las obras se especifican como:

Obras = { (t, ni, ℓ, #n) / t ∈ Tramos ∧ t = (c1, c2, n, a, dir, max) ∧ ni ∈ [0, n-1] ∧ ℓ ∈ ℕ ∧ #n ∈ [1, n+1-ni] ∧ #n ≡ 1 mod 2 }

Cada tupla del conjunto, o = (t, ni, ℓ, #n), identifica el tramo (t) donde se halla la obra, el primer carril (ni) afectado por la obra, la distancia sobre el carril ni que existe entre la columna central de la obra y el comienzo del tramo, y la cantidad de carriles que ocupa la obra (#n). Cada tupla especifica un rombo sobre un tramo por donde los vehículos no pueden circular. Se pide que la cantidad de carriles sea impar para poder armar el rombo (si fuera par se obtiene un romboide y no existe una celda central) y que el carril inicial (ni) más la cantidad de carriles que ocupa (#n) no sea más grande que la cantidad de carriles totales del tramo.

La siguiente figura muestra gráficamente la especificación de una obra.



Para que los vehículos no queden atascados entre las celdas de las obras, se ha elegido la forma de rombo, pero si algún extremo de éste toca el borde superior o inferior del tramo, se deben agregar algunas celdas más. En estos casos se completa la figura del rombo con un triángulo sobre el borde correspondiente.

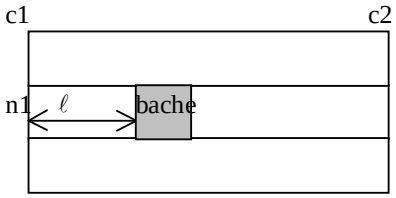
Para representar una obra o = (t, ni, ℓ, #n) en el modelo Cell-DEVS correspondiente al tramo t, se define un comportamiento diferente para las celdas dentro del rombo y las que se encuentran adelante del mismo. Las primeras que representan a la obra, tienen estado constante en 0; pues en ellas no hay vehículos. Las otras deben hacer que los autos esquiven a las celdas en obras, restringiendo sus movimientos acordemente.

2.7. Baches

Los baches se pueden especificar como:

$Baches_T$ = { (t, n1, ℓ) / t ∈ Tramos ∧ t = (c1, c2, n, a, dir, max) ∧ n1∈ [0, n-1] ∧ ℓ ∈ ℕ }

Cada tupla del conjunto, b = (t, n1, ℓ), identifica el tramo (t) y el carril (n1) donde se encuentra el bache; y el desplazamiento del bache sobre el carril, es decir la distancia sobre el carril n1 que existe entre el bache y el comienzo del tramo (representada por ℓ). La siguiente figura muestra gráficamente la especificación de un bache.



Un bache también puede ser definido sobre un cruce, especificándolo como:

$Baches_C$ = { c / c ∈ Cruces }

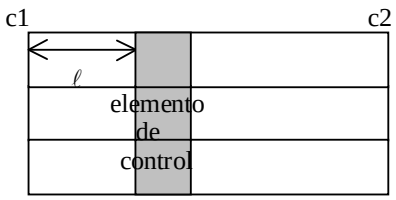
Para representar un bache en el modelo Cell-DEVS correspondiente al tramo, sólo se modifica el comportamiento de la celda que lo contiene. Para ello se utiliza una demora de transporte fija suficientemente grande que refleje que los vehículos circulan despacio debido a que la calle está rota. Un bache tendrá el tamaño de una celda y para modelar la presencia de uno más grande se deben definir varios consecutivos.

2.8. Elementos de Control

Las elevaciones transversales (lomo de burro), depresiones transversales (badén), bocacalles, irregularidades continuas (serrucho) y señales de PARE o de Escuela; se pueden especificar como:

ElementosDeControl = { (t, e, ℓ) / t ∈ Tramos ∧ ℓ ∈ ℕ ∧ e ∈ {elevación transversal, depresión transversal, bocacalles, irregularidad continua, señal de PARE, señal de Escuela} }

Cada tupla del conjunto, ec = (t, e, ℓ), identifica el tramo (t = (c1, c2, n, a, dir, max)), el tipo de elemento de control se ha especificado (e) y la distancia entre el elemento de control y el comienzo del tramo (representada por ℓ). La siguiente figura muestra gráficamente la especificación de uno de estos elementos.



Como todos estos elementos de control se especifican y mapean sobre los modelos Cell-DEVS en forma análoga, se definen en forma conjunta estableciendo una individualización sólo cuando sea necesario. Una de las características comunes es que su presencia afecta a los vehículos de todos los carriles de la columna donde se encuentra. Además se definen sobre tramos y no cruces, algunos se

ubican comúnmente en la cercanías de las esquinas (bocacalles, señal de pare) pero ninguno tiene sentido dentro la intersección.

Para reflejar la presencia de cualquiera de estos elementos de control en el modelo del tramo t, se modifica el comportamiento de las celdas de todos los carriles donde se encuentra, utilizando una demora grande y fija. Cabe destacar que el objetivo de estos elementos de control es que los vehículos disminuyan la velocidad, y esto se logra aumentando la demora de las celdas.

Capítulo 3 - Simulador N-CD++

En esta sección se describen algunos conceptos generales sobre el simulador para el cual TSC genera el modelo de simulación del plano de ciudad, así como la forma básica de invocar al simulador con los archivos de simulación y de macros que genera TSC, de forma tal de poder correr el modelo generado.
Se describe además el formato de la definición de modelos que se utiliza como entrada al simulador, especificando solamente los elementos utilizados en los templates de generación de TSC.

3.1. Sobre el simulador

N-CD++ es una herramienta para el modelado y simulación de modelos de espacios celulares utilizando el paradigma DEVS y Autómatas Celulares Asincrónicos.
Los modelos que respetan el formalismo DEVS utilizan una metodología para la construcción de sistemas complejos en función de componentes básicos. Esta construcción jerárquica facilita la comprensión del sistema y la reutilización de componentes ya verificados.
La herramienta presenta un lenguaje de especificación de modelado para la descripción de los componentes y su vinculación, detallando las características de cada uno de ellos. Dentro de esta especificación se contempla un lenguaje de expresiones de lógica trivalente para describir el comportamiento de las celdas de los modelos celulares.
Además de ser una herramienta genérica de simulación, la construcción de los modelos en forma jerárquica da la posibilidad de combinar y vincular modelos simples conformando modelos mucho más complejos, que de otra manera son más difíciles de modelar y simular. Por otra parte, además de la facilidad del modelado, la verificación de los modelos complejos se hace también más sencilla ya que la posibilidad de crearlos en combinación de otros modelos previamente verificados, asegura que si teóricamente la construcción jerárquica del modelo es correcta, el modelo en su conjunto será correcto, ya que cada uno de sus componentes también lo son.
El simulador consta de dos partes: un motor de simulación y los fuentes de los modelos atómicos que se pueden participar en alguna simulación.
El software de simulación provee el motor de simulación junto con las estructuras que permiten configurar la especificación y asociación de los modelos que componen el sistema a simular. El mismo trabaja con una interface genérica de modelos atómicos, acoplados y celulares.
El comportamiento de los modelos celulares se especifica por medio de un archivo de configuración, mientras que para los modelos atómicos debe codificarse un módulo en C++. Por esto es responsabilidad del programador generar los modelos atómicos correspondientes para llevar a cabo la simulación respetando los requerimientos del ambiente.
Al invocarlo el simulador recibe la especificación de una simulación, la ejecuta y retorna los resultados.

3.2. Forma simple de invocar al simulador

En esta sección se describe la forma más simple para invocar al simuladro con los modelos generados por TSC.
Cabe aclarar que el simulador brinda muchas opciones de invocación para generar diferentes tipos de información en cuanto a la simulación, que puede ser útil para analizar el comportamiento del plano de ciudad que se está simulando. Si el lector desea conocer en detalle la forma de trabajo del

simulador así como el total de las posibilidades que brinda en cuanto a la ejecución de simulaciones, puede referirse al Manual del Usuario y Manual del Desarrollador de N-CD++ [RW99].
Para ejecutar la simulación con los archivos generados por TSC se consideran los siguientes parámetros:

```
simu [-lmot]
        l: message log file (default: /dev/null)
        m: model file (default : model.ma)
        o: output (default: /dev/null)
        t: stop time (default: Infinity)
```

–l: Nombre de archivo en el cual se almacenarán los mensajes que reciben y emiten los modelos a lo largo de la simulación. Si se omite este parámetro el simulador no generará el log de actividad. Si se desea obtener el log por el standard output no debe especificarse ningún parámetro (–l).

–m: Nombre de archivo del cual se cargará el modelo a simular (coincide con el especificado en la opción -o de TSC al generar el modelo). Si se omite este parámetro el simulador cargará los modelos del archivo *model.ma*.

–o: nombre de archivo en el cual se grabará la salida generada por el simulador. Si se omite este parámetro el simulador no generará ninguna salida. Si se desea obtener los resultados por la salida estándar no debe especificarse ningún parámetro (–o).

–t: Hora de finalización de la simulación. Si se omite este parámetro el simulador solo se detendrá ante la falta de eventos (internos o externos). El formato de la hora debe ser HH:MM:SS:MS, donde:

HH: cantidad de horas
MM: minutos (de 0 a 59)
SS: segundos (de 0 a 59)
MS: milésimas de segundo (de 0 a 999)

3.3. Definición de Modelos

El archivo que permite definir los modelos esta compuesto por grupos de definiciones de modelos acoplados y configuración de modelos atómicos, esta última opcional. Cada definición indica el nombre del modelo (entre [ ]) y sus atributos. El grupo del modelo **[top]** es obligatorio y define el modelo acoplado de mayor nivel.
En el caso de **TSC**, **[top]** acopla todos los modelos que se corresponden con los elementos que componen el plano de ciudad.

3.3.1. Modelos Acoplados

Existen cuatro posibles propiedades a configurar: componentes (components), puertos de salida (out), puertos de entrada (in) y conexiones entre modelos (link). El nombre con el cual se define cada una de las partes del modelo es reservado y cualquier otra palabra será ignorada. La sintaxis es la siguiente:

Components: Describe los modelos que integraran el modelo acoplado. El formato es:

nombre_de_modelo@nombre_de_clase

El orden en que se especifican los modelos determina la prioridad utilizada para el envío de mensajes. Esto representa la función **select** del formalismo.

El nombre de modelo es necesario ya que es posible construir un modelo acoplado con más de una instancia del mismo modelo atómico. Por ejemplo un acoplado que posee dos colas llamadas *cola1* y *cola2*.

El nombre de clase puede hacer referencia tanto a modelos atómicos como a modelos acoplados. Estos últimos deben estar descriptos en el mismo archivo de configuración como un nuevo grupo.

En caso de no especificarse este atributo se producirá un error indicando la falta del mismo.

Out: Enumera los nombres de los puertos de salida. Este atributo es opcional ya que un modelo puede no tener puertos de este tipo.
Ejemplo: *Out* puerto1 puerto2 puerto3

In: Enumera los nombres de los puertos de entrada. Este atributo es opcional ya que un modelo puede no tener puertos de este tipo.
Ejemplo: *In* puerto1 puerto2 puerto3

Link: Describe el esquema de acoplamiento interno, externo de entrada y externo de salida. El formato esta dado por el par:

puerto_origen[@modelo] puerto_destino[@modelo]

El nombre del modelo es opcional ya que si no se indica se toma como un puerto correspondiente al acoplado en cuestión.

3.3.2. Modelos Atómicos

En esta definición se configuran los modelos atómicos participantes de la simulación. En caso de no figurar ninguna referencia se tomarán los valores por defecto que halla programado el desarrollador de dicha clase.
En el caso de TSC se definieron los modelos atómicos TSCGenerator, TSCConsumer, TSCSincroTrafficLight, TSCTrafficLight, TSCSincroRailNet, TSCRailNet.

La configuración se especifica de la siguiente forma:

```
[nombre_modelo_atomico]
nombre_variable1 : valor_var1
.
.
.
nombre_variable n : valor_var n
```

Los nombre de las variables dependen del desarrollador de la clase que se desea configurar y deben estar documentados junto con el código fuente de la misma.
Cada instancia de un tipo de modelos atómico podrá ser configurada independientemente de otras instancias del mismo tipo.

3.3.3. Modelos Celulares

Los modelos celulares son una variante de los modelos acoplados, y debido a esto se utiliza la misma especificación con el agregado de ciertos parámetros inherentes a las características de los mismos.
En el caso de TSC se definen modelos celulares para los tramos y los cruces y se utilizan los siguientes parámetros:

Dichos parámetros son:

Type : [**CELL** | **FLAT**]
Indica si el modelo celular será achatado o no. Si no se especifica se asume que es un modelo no achatado (*CELL*).

Width : entero
Permite definir la cantidad de columnas sólo para modelos celulares unidimensionales y bidimensionales.
El uso de *Width* implica necesariamente el uso de la cláusula *Height* para completar la definición de la dimensión del modelo.
Si se utiliza *Width*, la invocación de la cláusula *Dim* en la definición del mismo modelo producirá un error.

Height : entero
Permite definir la cantidad de filas sólo para un modelo celular bidimensional.
Si se desea modelar un autómata celular unidimensional, debe establecerse el valor 1 para la cláusula *Height*.
El uso de *Height* implica necesariamente el uso de la cláusula *Width* para completar la definición de la dimensión del modelo.
Si se utiliza *Height*, la invocación de la cláusula *Dim* en la definición del mismo modelo producirá un error.

In : Igual que en los modelos acoplados. Esta cláusula puede no estar definida, con lo que no existirán puertos de entrada para el modelo celular.

Out : Igual que en los modelos acoplados. Esta cláusula puede no estar definida, con lo que no existirán puertos de salida para el modelo celular.

Link : Igual que en los modelos acoplados pero para hacer referencia a una celda se debe usar el nombre del acoplado más (x_1,x_2,...,x_n) sin dejar espacios.
Ejemplos: *Link* puertoSalida puertoEntrada@nombreCelular(x_1,x_2,...,x_n)
Link puertoSalida@nombreCelular(x_1,x_2,...,x_n) puertoEntrada

Border : [**WRAPPED** | **NOWRAPPED**]
Indica si el modelo es o no toroidal. Por defecto toma el valor NOWRAPPED.
Si se utiliza un borde no toroidal, cualquier referencia a una celda fuera del espacio celular retornará el valor indefinido (**?**).

Delay : [**TRASPORT** | **INERTIAL**]
Especifica el tipo de demora usada en cada celda. Por defecto toma el valor TRANSPORT.

DefaultDelayTime : entero
Demora por defecto para los eventos externos (medida en milisegundos).

Neighbors : *nombreCelular*($x_{1,1}$, $x_{2,1}$,...,$x_{n,1}$)... *nombreCelular*($x_{1,m}$, $x_{2,m}$,...,$x_{n.m}$)
Define el vecindario para todas las celdas del modelo. Cada celda ($x_{1,i}$, $x_{2,i}$,...,$x_{n.i}$) representa un desplazamiento con respecto a la celda de origen del vecindario.
N-CD++ no impone restricciones sobre la creación del vecindario, permitiendo que las celdas que lo componen puedan no estar adyacentes a la celda de origen.
Es posible utilizar más de una sentencia **neighbors** para declarar el vecindario para un modelo celular.

Initialvalue : [*Real* | **?**]
Representa el valor inicial para todo el espacio de celdas. **?** representa al valor indefinido.

LocalTransition : *transitionFunctionName*
Indica el nombre del grupo que contiene las reglas a utilizar para la función de transición local para todas las celdas.

PortInTransition : *portName@ nombreCelular*(x_1, x_2,...,x_n) *transitionFunctionName*
Permite definir un comportamiento alternativo cuando arriba un mensaje externo por el puerto de entrada indicado de la celda (x_1, x_2,...,x_n) del modelo celular.
Si no se especifica una función asociada al puerto de la celda, al arribar un mensaje externo por el mismo, el valor de dicho mensaje será asignado a la celda usando la demora especificada por defecto en la definición del modelo.

Zone : *transitionFunctionName* { rango$_1$[..rango$_n$] }
Permite definir un comportamiento alternativo para el conjunto de celdas comprendidas dentro del rango especificado. Cada rango es algo de la forma (x_1,x_2,...,x_n) describiendo una celda, (x_1,x_2,...,x_n)..(y_1,y_2,y_n) describiendo una zona de celdas, o una lista que puede combinar una cantidad arbitraria de ambas (separando a cada elemento de la misma por un espacio en blanco).
Por ejemplo: *zone :* bache { (10,10)..(13,13) (1,3) }

En el momento de calcular el nuevo estado para una celda, si dicha celda pertenece a algún rango se usará la función definida para tal, sino se utilizará la función de transición definida en la cláusula **LocalTransition**.

3.4. Lenguaje de Especificación de Reglas

La definición de las reglas que describen un cierto comportamiento se hace en forma independiente a los modelos celulares que la utilizan. Esto permite que más de un modelo celular utilice la misma especificación como así también que varias zonas dentro de un espacio celular lo utilicen sin necesidad de redefinirla.

El lenguaje se define como un nuevo grupo dentro de la especificación, donde cada componente del grupo es una regla con la siguiente sintaxis:

rule : resultado demora { condición }

Cada regla esta compuesta por tres elementos: una *condición*, una *demora* y un *resultado*. Para calcular el nuevo estado de una celda, se toma cada una de las reglas (en el orden en que fueron definidas) y si la condición de la misma es evaluada a verdadero, entonces se evalúan su resultado y su demora, y estos valores serán los utilizados por la celda. Si la evaluación de la condición de la regla es falsa, entonces se toma la siguiente regla. Si se evalúan todas las reglas sin haber encontrado alguna válida, entonces la simulación se cancelará y se informará al usuario de tal situación. Si existe más de una regla válida se toma la primera de ellas.

Cabe considerar que si al evaluar la demora se obtiene como resultado el valor indefinido, entonces la simulación será automáticamente cancelada.

3.5. Gramática del Lenguaje

La sintaxis del lenguaje usado por *N-CD++* para la especificación del comportamiento de los modelos celulares atómicos puede definirse con la BNF mostrada en la siguiente figura, donde las palabras escritas con letras minúsculas y en negrita representan terminales, mientras que las escritas en mayúsculas representan no terminales.

```
RULELIST      = RULE
                | RULE RULELIST
RULE          = RESULT RESULT { BOOLEXP }
RESULT        = CONSTANT
                | { REALEXP }
BOOLEXP       = BOOL
                | ( BOOLEXP )
                | REALRELEXP
                | not BOOLEXP
                | BOOLEXP OP_BOOL BOOLEXP
OP_BOOL       = and | or | xor | imp | eqv
REALRELEXP    = REALEXP OP_REL REALEXP
                | COND_REAL_FUNC(REALEXP)
REALEXP       = IDREF
                | ( REALEXP )
                | REALEXP OPER REALEXP
```

```
IDREF          = CELLREF
                 |  CONSTANT
                 |  FUNCTION
                 |  portValue(PORTNAME)
                 |  send(PORTNAME, REALEXP)
                 |  cellPos(REALEXP)
CONSTANT       = INT
                 |  REAL
                 |  CONSTFUNC
                 |  ?
FUNCTION       = UNARY_FUNC(REALEXP)
                 |  WITHOUT_PARAM_FUNC
                 |  BINARY_FUNC(REALEXP, REALEXP)
                 |  if(BOOLEXP, REALEXP, REALEXP)
                 |  ifu(BOOLEXP, REALEXP, REALEXP, REALEXP)
CELLREF        = (INT, INT RESTO_TUPLA
RESTO_TUPLA    = , INT RESTO_TUPLA
                 |  )
BOOL           = t  |  f  |  ?
OP_REL         = !=  |  =  |  >  |  <  |  >=  |  <=
OPER           = +  |  -  |  *  |  /
INT            = [SIGN] DIGIT {DIGIT}
REAL           = INT  |  [SIGN] {DIGIT}.DIGIT {DIGIT}
SIGN           = +  |  -
DIGIT          = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
PORTNAME       = thisPort  |  STRING
STRING         = LETTER {LETTER}
LETTER         = a | b | c |...| z | A | B | C |...| Z
CONSTFUNC      = pi | e | inf | grav | accel | light | planck | avogadro |
                 faraday | rydberg | euler_gamma | bohr_radius | boltzmann |
                 bohr_magneton | golden | catalan | amu | electron_charge |
                 ideal_gas | stefan_boltzmann | proton_mass | electron_mass
                 neutron_mass  | pem
WITHOUT_PARAM_FUNC = truecount | falsecount | undefcount | time | random |
                     randomSign
UNARY_FUNC     = abs | acos | acosh | asin | asinh | atan | atanh | cos |
                 sec | sech | exp | cosh | fact | fractional | ln | log |
                 round | cotan | cosec | cosech | sign | sin | sinh |
                 statecount | sqrt | tan | tanh | trunc | truncUpper |
                 poisson | exponential | randInt | chi | asec | acotan |
                 asech | acosech | nextPrime | radToDeg | degToRad |
                 nth_prime | acotanh | CtoF | CtoK | KtoC | KtoF | FtoC |
                 FtoK
BINARY_FUNC    = comb | logn | max | min | power | remainder | root | beta |
                 gamma | lcm | gcd | normal | f | uniform | binomial |
                 rectToPolar_r | rectToPolar_angle | polarToRect_x | hip |
                 polarToRect_y
COND_REAL_FUNC = even | odd | isInt | isPrime | isUndefined
```

Cabe considerar que en la definición de una regla, el segundo valor, que se corresponde con la demora de la celda, puede ser un número real, ya sea en forma directa o como resultado de la evaluación de una expresión. Sin embargo, si no es un número entero, este será automáticamente truncado de tal forma que su valor sí lo sea. Por otra parte, si su valor es indefinido (**?**) entonces se informará tal situación y se producirá un error, abortando la simulación.

3.6. Preprocesador – Uso de Macros

La herramienta proporciona ciertas facilidades al lenguaje por medio de un preprocesador, que actúa sobre el archivo de definición de modelos, antes de la carga de los mismos.

La cláusula **#include** permite incluir el contenido de un archivo. Su formato es:

#include(*fileName*)

donde ***fileName*** es el nombre del archivo que contiene la definición de las macros. Dicho archivo debe encontrarse en el mismo directorio donde se encuentra el archivo de definición de modelos.

La cláusula ***#include*** debe estar contenida sólo en los archivos de definición de modelos, y puede existir más de una inclusión de distintos archivos dentro de la definición de modelos.

Las cláusulas **#BeginMacro** y **#EndMacro** permiten dar comienzo y fin a la definición de una macro.
Una definición de macro tiene la forma:

```
#BeginMacro(nombreMacro)
...
...contenido de la macro...
...
#EndMacro
```

El contenido de la macro es arbitrario, pudiendo abarcar cualquier cantidad de líneas. Las definiciones de macros no pueden estar contenidas en el mismo archivo donde son invocadas.

La cláusula **#Macro** permite el uso de una macro previamente definida, reemplazando el texto que la invoca por el contenido de dicha macro. Su formato es:

#Macro(*nombreMacro*)

El archivo de macros puede contener cualquier cantidad de macros, por más que estas nunca sean usadas en el modelo.

El texto que figura fuera de la definición de una macro es ignorado, permitiendo de esta forma incluir comentarios sobre la funcionalidad del mismo con solo escribir dicho texto antes o después de la definición.

Si una macro requerida no es encontrada en ninguno de los archivos incluidos con la cláusula ***#include***, se generará un error y la herramienta finalizará su ejecución.

Los ***#include*** pueden estar definidos en cualquier lugar del archivo, pero siempre deber estar antes de la cláusula ***#Macro*** que utilice una macro cuya descripción este contenida en el archivo referenciado por el **#include**.

Dentro de la definición de una macro no puede realizarse una invocación a otra macro.

El preprocesador permite también el uso de comentarios en cualquier parte de un archivo **.MA**. Los comentarios empiezan con el carácter '**%**', y cuando el preprocesador encuentra un comentario, ignora el string que se encuentren comprendido entre el carácter '**%**' hasta el fin de la línea.

3.7. Formato del Archivo de Log

El archivo de *log* registra el flujo de mensajes entre los modelos que participan en la simulación. Cada línea del archivo muestra el tipo de mensaje, la hora a la que se produjo, quien lo emitió y el destinatario. Esta información es común a todos los mensajes. En caso de ser un mensaje de tipo *X* ó *Y* aparecerá, además, el puerto y el valor. Para los mensajes de tipo *D* se agrega la hora del próximo evento, ó '…' en caso de que la hora sea infinito.
Los números que figuran junto al nombre del simulador asociado a cada modelo son a solo efecto de información para el desarrollador.

3.8. Incorporación de Nuevos Modelos Atómicos

Esta sección describe el mecanismo para definir e incorporar nuevos modelos atómicos a la herramienta. Sin embargo, dichos modelos no podrán ser usados para crear un modelo acoplado celular, sino para interactuar directamente con otros modelos ó para formar parte un modelo acoplado *DEVS* más general.

Para generar un nuevo modelo atómico, se debe comenzar por diseñar una nueva clase que sea derivada de la clase *Atomic* y se debe agregar al método *MainSimulator.registerNewAtomics*() el nuevo tipo de modelo atómico. Luego se deben sobrecargar obligatoriamente los siguientes métodos:

- ***initFunction***: este método es invocado por el simulador una única vez al comenzar la simulación en el tiempo cero. El objetivo es permitir la inicialización que el modelo considere necesaria. Antes de invocar al método, *sigma* vale infinito y el estado es *pasivo*.

- ***externalFunction***: este método es invocado cuando arriba un evento externo por alguno de los puertos del modelo.

- ***internalFunction***: antes de invocar a este método, *sigma* vale cero, ya que se ha cumplido el intervalo para la transición interna.

- ***outputFunction***: antes de invocar al método *sigma* vale cero, ya que se ha cumplido el intervalo para la transición interna.

- ***className***: nombre de la clase.

Estos métodos pueden invocar ciertas primitivas predefinidas, que permiten interactuar con el simulador abstracto:

- ***holdIn***(estado, tiempo): indica al simulador que el modelo debe mantenerse en el estado durante un tiempo, y que, luego de transcurrido, provocará una transición interna.

- ***passivate***(): indica al simulador que el modelo entra en modo *pasivo* y que únicamente deberá ser reactivado ante la llegada de un evento externo.

- ***sendOutput***(hora, port, valor): envía un mensaje de salida por el puerto indicado.

- ***nextChange***(): este método permite obtener el tiempo restante para su próximo cambio de estado (*sigma*).

- ***lastChange***(): este método permite obtener la hora en que se produjo el último cambio de estado.

- ***state***(): este método obtiene la fase en la que se encuentra el modelo.

- ***getParameter***(nombreModelo, nombreParámetro): este método permite acceder a los parámetros de configuración de la clase.

Capítulo 4 - Compilador TSC

4.1. Qué es TSC y cómo se utiliza

TSC es una herramienta cuya propósito es tomar como entrada una representación de un plano de ciudad realizado con el lenguaje de especificación de tráfico urbano ATLAS (Advanced Traffic Language Simulation) y dejar como salida un archivo conteniendo modelos de simulación para ese plano, con la estructura correspondiente a ser utilizada como entrada al simulador N-CD++.

TSC fue construido de forma tal que es posible configurar por completo los modelos de simulación que generará para el plano de ciudad indicado como entrada. Esto se debe a que el funcionamiento de TSC se centra en un conjunto de templates de generación que determinan la forma en que se codificarán en el archivo de entrada a N-CD++ cada uno de los elementos que componen un modelo de simulación de tráfico. El conjunto de templates a utilizar por TSC se indican en el momento de invocar al mismo.
De esta manera, si en algún momento se modifica o extiende N-CD++ cambiando la forma de especificar los modelos de simulación o agregando nueva funcionalidad a la herramienta, es suficiente con modificar o extender el archivo que contiene los templates de generación de TSC para que se genere el código adecuado.

La forma de trabajo de TSC consiste en tomar el plano de ciudad indicado al invocarlo y, basándose en los templates de generación, generar dos archivos. El primero de ellos contendrá los modelos DEVS y CellDEVS necesarios para simular el plano de ciudad correspondiente. El segundo contendrá las macros utilizadas por las reglas de cada modelo definidas en los templates de generación.
En el conjunto de templates por defecto se optó por la utilización de macros siempre que fuera posible ya que esta facilidad de N-CD++ permite evitar la repetición de reglas, haciendo código menos extenso y por lo tanto más fácil de comprender.

La forma de invocar a TSC es la siguiente:

TSC [-iomth?]

`-h:` muestra una ayuda sobre la forma de invocar a TSC.

`-?:` muestra una ayuda sobre la forma de invocar a TSC.

`-i:` nombre del archivo que contiene el plano de ciudad. Si el nombre especificado no tiene extensión, se asumirá extensión .plan. Si se omite este parámetro se tomará como entrada el archivo model.plan.

`-o:` nombre del archivo en el cual quedará el modelo a usar como entrada al simulador. Si el nombre especificado no tiene extensión, se asumirá extensión .ma. Si se omite este parámetro se asumirá el mismo nombre que el archivo de input cambiando su extensión a .ma.

-m: nombre del archivo en el cual quedarán las macros que utiliza el modelo a simular. El uso de macros es una facilidad que brinda N-CD++. Si el nombre especificado no tiene extensión, se asumirá extensión .macros. Si se omite este parámetro se asumirá el mismo nombre que el archivo de output pero cambiando su extensión a .macros.

-t: nombre del archivo que contiene los templates que usa el compilador en el momento de la generación de código. Si el nombre especificado no tiene extensión, se asumirá extensión .ini. Si se omite este parámetro se asumirá tsc.ini.

4.2. Arquitectura de TSC

TSC fue modelado utilizando el paradigma de orientación a objetos.
Está compuesto por tres clases que ejecutan las operaciones principales y por un conjunto de clases que colaboran realizando funcionalidades particulares.

Para las tres clases principales TSC instancia solamente un objeto de cada una de ellas.
La responsabilidad de estas clases es la siguiente:

- **Clase Traffic:** modela el plano de ciudad que se simulará. Contiene todos los elementos que compondrán la simulación (calles, cruces, semáforos, baches, obras, etc.).

- **Clase CodeStruct:** modela los templates de generación de código para cada uno de los elementos simulables.

- **Clase CodeGen:** genera el archivo de simulación y el de macros, basándose en el plano a simular contenido en la instancia de Traffic y en los templates de generación de código contenidos en la instancia de CodeStruct.

4.3. Formato y procesamiento del plano de ciudad

El trabajo de TSC comienza invocando a un parser (***planparse***) que debe procesar el plano de ciudad (archivo model.plan o el indicado con la opción –i) y generar la instancia de Traffic que modela este plano.
Para ello se construyó una gramática que modela el lenguaje de simulación de tráfico urbano ATLAS (Advanced Traffic Language Simulation) en el cual se basa la especificación de dicho plano. Esta gramática está totalmente descripta en la siguiente figura:

```
PROGRAM          = lambda
                   | LIST
LIST             = ELEMENT
                   | LIST ELEMENT
ELEMENT          = SEGMENTS
                   | CROSSINGS
                   | RAILNETS
```

```
                    | JOBSITES
                    | HOLES
                    | CTRELEMS
SEGMENTS          = begin segments LIST_SEGMENTS end segments
LIST_SEGMENTS     = LIST_SEGMENTS SEGMENT
                    | SEGMENT
SEGMENT           = IDENTIF = POINT , POINT , LANES , SHAPE ,
                    DIRECTION , SPEED , DELAY , PARKTYPE
CROSSINGS         = begin crossings LIST_CROSSINGS end crossings
LIST_CROSSINGS    = LIST_CROSSINGS CROSSING
                    | CROSSING
CROSSING          = IDENTIF = POINT , SPEED , TLIGHT , CROSSHOLE ,
                    DELAY , POUT
RAILNETS          = begin railnets LIST_RAILNETS end railnets
LIST_RAILNETS     = LIST_RAILNETS RAILNET
                    | RAILNET
RAILNET           = IDENTIF = LIST_RAILSEGMENTS , DELAY
LIST_RAILSEGMENTS = LIST_RAILSEGMENTS , RAILSEGMENT
                    | RAILSEGMENT
RAILSEGMENT       = ( IDENTIF , NUMBER )
JOBSITES          = begin jobsites LIST_JOBSITES end jobsites
LIST_JOBSITES     = LIST_JOBSITES JOBSITE
                    | JOBSITE
JOBSITE           = in IDENTIF : LANES , DISTANCE , LANES , DELAY
HOLES             = begin holes LIST_HOLES end holes
LIST_HOLES        = LIST_HOLES HOLE
                    | HOLE
HOLE              = in IDENTIF : LANES , DISTANCE , DELAY
CTRELEMS          = begin ctrElements LIST_CTRELEMS end ctrElements
LIST_CTRELEMS     = LIST_CTRELEMS CTRELEM
                    | CTRELEM
CTRELEM           = in IDENTIF : CTRTYPE , DISTANCE , DELAY
POINT             = ( NUMBER , NUMBER )
LANES             = NUMBER
SPEED             = NUMBER
DELAY             = NUMBER
DISTANCE          = NUMBER
POUT              = NUMBER
IDENTIF           = LETTER {LETTER|DIGIT|SYMBOL}
NUMBER            = DIGIT {DIGIT}
LETTER              a | b | c | ... | z | A | B | C | ... | Z
DIGIT             = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
SYMBOL            = - | _
DIRECTION         = go | back
SHAPE             = curve | straight
TLIGHT            = withTL | withoutTL
PARKTYPE          = parkNone | parkLeft | parkRight | parkBoth
CROSSHOLE         = withHole | withoutHole
CTRTYPE           = sawhorse|depression|intersection|saw|stop|school
```

Los diferentes elementos que componen el plano de ciudad se agrupan en secciones dentro del archivo separados por el tipo de componente. Así, tendremos una sección que define los tramos, una que define los cruces, una que define las vías, una que define las obras, una que define los baches y una que define los elementos de control.

Estas secciones no deben seguir ningún orden particular dentro de la definición del plano y puede existir más de una para el mismo tipo.
La sintaxis de cada sección se describe a continuación.

4.3.1. Sección de tramos

Esta sección es la única obligatoria dentro de la especificación del plano ya que el mismo deberá contener por lo menos un tramo para que se pueda generar un modelo de simulación correcto.
Comienza y termina con las sentencias **"begin segments"** y **"end segments"** respectivamente.
Entre estas sentencias se encontrará la definición de cada uno de los tramos con la siguiente sintaxis:

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

donde,

`id`: Alfanumérico
: Debe comenzar con una letra y puede contener letras, dígitos o los caracteres "-" y "_". Es el identificador del tramo. El compilador lo utiliza para dar nombre al modelo atómico correspondiente en el archivo de simulación.
También se utiliza para poder identificar el tramo en el resto de las secciones de este archivo si se desea asociarle obras, baches, cruces de vías o elementos de control.

`p1`: Punto
: Es un punto en el plano que identifica el comienzo del tramo.
Debe ser diferente a p2.

`p2`: Punto
: Es un punto en el plano que identifica el final del tramo.
Debe ser diferente a p1.

`lanes`: Entero
: Define la cantidad de carriles del tramo siendo el mínimo permitido de 1 carril.

`shape`: [**`curve`**|**`straight`**]
: Define el formato del tramo de acuerdo a la especificación de ATLAS.

`direction`: [**`go`**|**`back`**]
: Define el sentido de circulación del tramo. Si se define **go**, la circulación será de **p1** hacia **p2**. Si se define **back**, la circulación será de p2 a p1.

`speed`: Entero
: Define la velocidad máxima de circulación de los autos en este tramo.

`delay`: Entero
: Define la demora en la circulación de los autos en los carriles donde se permite estacionar autos. Este parámetro es ignorado en caso de que el tramo no posea zona de estacionamiento.

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

Define sobre qué lado del tramo pueden estacionarse los autos.

4.3.2. Sección de cruces

Esta sección no es obligatoria dentro de la especificación del plano.
Comienza y termina con las sentencias **"begin crossings"** y **"end crossings"** respectivamente.
Entre estas sentencias se encontrará la definición de cada uno de los cruces con la siguiente sintaxis:

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

donde,

id: Alfanumérico
Debe comenzar con una letra y puede contener letras, dígitos o los caracteres "-" y "_". Es el identificador del cruce. El compilador lo utiliza para dar nombre al modelo atómico correspondiente en el archivo de simulación.

p: Punto
Es un punto en el plano que define la ubicación del cruce.

speed: Entero
Define la velocidad máxima de circulación de los autos en este cruce.

tLight: [**withTL**|**withoutTL**]
Define si el cruce contiene o no semáforos.

crossHole: [**withHole**|**withoutHole**]
Define si el cruce contiene o no un bache.

delay: Entero
Define la demora en la circulación de los autos producida por la existencia de un bache en el cruce. Este parámetro es ignorado en caso de que el cruce no posea bache.

pout: Entero
Define la probabilidad de salir del cruce. Esta probabilidad se tomará como 1/**pout**.

4.3.3. Sección de vías

Esta sección no es obligatoria dentro de la especificación del plano.
Comienza y termina con las sentencias **"begin railnets"** y **"end railnets"** respectivamente.
Entre estas sentencias se encontrará la definición de cada una de las redes de vías con la siguiente sintaxis:

id = (t_1, d_1) {,(t_i, d_i)}, delay

donde,

id: Alfanumérico
Debe comenzar con una letra y puede contener letras, dígitos o los caracteres "-" y "_". Es el identificador de la red de vías. El compilador lo utiliza para dar nombre a los modelos atómicos correspondientes en el archivo de simulación.

t_i: Alfanumérico
Define un identificador de tramo por donde pasa la vía.

d_i: Entero
Define la distancia entre el comienzo del tramo **t_i** y la vía.

delay: Entero
Define la demora de los autos al pasar por las celdas de un tramo atravesadas por las vías de un tren.

4.3.4. Sección de obras

Esta sección no es obligatoria dentro de la especificación del plano.
Comienza y termina con las sentencias **"begin jobsites"** y **"end jobsites"** respectivamente.
Entre estas sentencias se encontrará la definición de cada una de las obras con la siguiente sintaxis:

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

donde,

t: Alfanumérico.
Es el identificador del tramo en el cual se encuentra la obra. El compilador lo utiliza para modificar la definición del modelo atómico para el tramo correspondiente.

firstlane: Entero
Define el primer carril afectado por la obra.

distance: Entero
Define la distancia sobre el primer carril que existe entre la columna central de la obra y el comienzo del tramo.

lanes: Entero impar
Define la cantidad de carriles que ocupa la obra.

delay: Entero
Este valor no es utilizado y se definió por compatibilidad con los otros elementos del plano. Se reserva para uso futuro.

4.3.5. Sección de baches

Esta sección no es obligatoria dentro de la especificación del plano.
Comienza y termina con las sentencias **"begin holes"** y **"end holes"** respectivamente.
Entre estas sentencias se encontrará la definición de cada uno de los baches con la siguiente sintaxis:

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

donde,

t: Alfanumérico
Es el identificador del tramo en el cual se encuentra el bache. El compilador lo utiliza para modificar la definición del modelo atómico para el tramo correspondiente.

lane: Entero
Define el carril donde se encuentra el bache.

distance: Entero
Define la distancia entre el comienzo del tramo y el bache.

delay: Entero
Define la demora que el bache produce en la circulación de los autos.

4.3.6. Sección de elementos de control

Esta sección no es obligatoria dentro de la especificación del plano.
Comienza y termina con las sentencias **"begin ctrElements"** y **"end ctrElements"** respectivamente.
Entre estas sentencias se encontrará la definición de cada uno de los elementos de control con la siguiente sintaxis:

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

donde,

t: Alfanumérico
Es el identificador del tramo en el cual se encuentra el elemento de control. El compilador lo utiliza para modificar la definición del modelo atómico para el tramo correspondiente.

ctrType: [**sawhorse | depression | intersection | saw | stop | school**]
Define el tipo del elemento de control (lomo de burro, badén, bocacalle, serrucho, pare, escuela)

distance: Entero
Define la distancia entre el comienzo del tramo y el elemento de control.

delay: Entero
Define la demora que el elemento de control produce en la circulación de los autos.

4.4. Validación del plano de ciudad

Durante la generación de la instancia de Traffic realizada por ***planparse***, se lleva a cabo un primer conjunto de validaciones del plano de ciudad.

Estas validaciones son:

- En el plano debe haber por lo menos un tramo.
- Un tramo no debe tener el mismo punto de comienzo que de fin.
- Dos tramos con igual inclinación superpuestos en un extremo deben tener distinta dirección.
- No deben haber dos o más cruces definidos en el mismo punto.
- Las vías, baches, obras y elementos de control deben estar definidos en un tramo existente y dentro de los límites del mismo.
- Las vías no pueden atravesar un tramo ni en la primera ni en la última celda.
- Para estacionar sobre un carril del tramo el mismo debe tener como mínimo 2 carriles y para estacionar sobre ambos carriles el mismo debe tener como mínimo 4 carriles

Una vez que fue generado el objeto de la clase Traffic que modela el plano a simular, este mismo objeto se encargará de asociar los tramos a los cruces correspondientes. Durante esta operación validará que cada cruce tenga por lo menos un tramo de entrada y por lo menos un tramo de salida.

Cabe aclarar aquí que un trabajo muy interesante para complementar a TSC es la implementación de una interface gráfica para la construcción del plano a simular. Es aquí donde lógicamente estarían incluidas todas las validaciones ya que la interface no deberá permitir que se construya un plano incoherente.

Dado que este es un trabajo posterior, optamos por validar todo lo que nos fuera posible y que no implique un análisis demasiado sofisticado que desvíe el foco de nuestro trabajo. Como ejemplo de validaciones adicionales podemos citar:

- Las obras, baches o elementos de control no deben superponerse.
- Las obras no pueden bloquear todos los carriles e impedir completamente el paso de los autos por el tramo.
- Si los tramos son de entrada/salida entonces no se deben definir elementos de control en las celdas de entrada/salida.

4.5. Estructura y procesamiento de los templates de generación

El archivo de templates de generación contiene templates por cada uno de los elementos que conforman un plano de ciudad. Por lo tanto existen templates de generación para tramos, para cruces, para baches, etc..

Antes de explicar el formato de cada uno de estos templates, es necesario hacer una aclaración sobre los modelos de simulación que generan cada uno de los elementos que componen un plano de ciudad.

Los elementos del plano que generan modelos Cell-DEVS de simulación son tramos y cruces. El resto de los elementos del plano, obras, baches, semáforos, red de vías, autos estacionados y elementos de control, alteran la estructura de dichos modelos, pero no generan nuevos modelos Cell-DEVS. Es decir, pueden por ejemplo agregar algún acoplamiento o modificar las reglas para un conjunto específico de celdas de un tramo.

Por lo tanto, los templates que se corresponden con tramos y cruces indicarán cómo se construyen los modelos correspondientes a cada tramo y a cada cruce respectivamente. Más precisamente, para generar el modelo de cada tramo de un carril se utilizará el template correspondiente a tramos de un

carril. Lo mismo para cruces, tramos de dos carriles, tramos de tres carriles, tramos de cuatro carriles y tramos de más de cuatro carriles.

El resto de los templates indican simplemente qué es lo que se agrega a estos modelos para reflejar el efecto que producen los demás elementos del plano en el comportamiento de los tramos y los cruces en los que se encuentran. Más precisamente:

- las obras, vías y elementos de control afectarán el comportamiento del tramo en el que se encuentren
- los baches afectarán el comportamiento del tramo o cruce en el que se encuentren
- los semáforos afectarán el comportamiento de los tramos de entrada al cruce en que se encuentren

La estructura del archivo que contiene los templates de generación de código está totalmente especificada por una gramática que se diseñó con tal propósito. La descripción de esta gramática se detalla a continuación:

```
CODESTRUCT         = LIST_TEMPLATES MACROS
LIST_TEMPLATES     = TEMPLATE
                     |LIST_TEMPLATES TEMPLATE
TEMPLATE           = LIST_NOTHING |-- template ID --|\n TOP_COMPONENTS
                     TOP_LINKS BEFORE_NEIGHBORS NEIGHBORS BEFORE_PORTS
                     PORTS BEFORE_LINKS LINKS BEFORE_ZONES ZONES
                     BEFORE_RULES RULES AFTER_RULES |--end template--
                     |\n
TOP_COMPONENTS     = lambda
                     | |--top components--|\n LIST_ANY_LINES
TOP_PORTS          = lambda
                     | |--top ports--|\n LIST_ANY_LINES
TOP_LINKS          = lambda
                     | |--top links--|\n LIST_ANY_LINES
BEFORE_NEIGHBORS   = lambda
                     | |--before neighbors--|\n LIST_ANY_LINES
NEIGHBORS          = lambda
                     | |--neighbors--|\n LIST_ANY_LINES
BEFORE_PORTS       = lambda
                     | |--before ports--|\n LIST_ANY_LINES
PORTS              = lambda
                     | |--ports--|\n LIST_ANY_LINES
BEFORE_LINKS       = lambda
                     | |--before links--|\n LIST_ANY_LINES
LINKS              = lambda
                     | |--links--|\n LIST_ANY_LINES
BEFORE_ZONES       = lambda
                     | |--before zones--|\n LIST_ANY_LINES
ZONES              = lambda
                     | |--zones--|\n LIST_ANY_LINES
BEFORE_RULES       = lambda
                     | |--before rules--|\n LIST_ANY_LINES
RULES              = lambda
                     | |--rules--|\n LIST_RULEBLOCKS
LIST_RULEBLOCKS    = lambda
                     |LIST_RULEBLOCKS RULEBLOCK
RULEBLOCK          = [ ID ] \n LIST_LINES
```

```
AFTER_RULES        = lambda
                     | |--after rules--|\n LIST_LINES
MACROS             = lambda
                     |LIST_NOTHING |--macros--|\n LIST_MACROS
                     LIST_NOTHING |--end macros--|\n LIST_NOTHING
LIST_MACROS        = lambda
                     |LIST_MACROS MACRO
MACRO              = LIST_NOTHING #BeginMacro( ID )\n LIST_LINES
                     #EndMacro\n
LIST_ANY_LINES     = lambda
                     |LIST_ANY_LINES LINE
                     |LIST_ANY_LINES [ ID ]\n
LIST_LINES         = LINE
                     |LIST_LINES LINE
LIST_NOTHING       = lambda
                     |LIST_NOTHING LINE
LINE               = {anyPrintableChar}\n
ID                 = {LETTER|DIGIT|SYMBOL}
SYMBOL             = - | _ | &
LETTER               a | b | c | ... | z | A | B | C | ... | Z
DIGIT              = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Cada template contiene el conjunto de líneas que se debe generar en el archivo de simulación por cada elemento de su tipo que se encuentre en el plano.

Estas líneas se encuentran agrupadas en diferentes secciones -no obligatorias- dentro del template para facilitar el armado de los modelos Cell-DEVS en el archivo de simulación.

Por ejemplo, una obra agrega un conjunto diferente de reglas en el tramo para las celdas afectadas por la obra. Esto produce una nueva instrucción **zone** en el modelo Cell-DEVS del tramo correspondiente. La línea conteniendo este **zone** debe ser intercalada en el modelo Cell-DEVS del tramo, luego de la definición de los **links** y antes del primer **zone** especificado para el tramo. La división en secciones de cada template hace más fácil esta tarea.

Las indicaciones de comienzo de cada sección, así como el nombre del template y el indicador de fin de template se encuentran encerradas entre las marcas **|--** y **--|** con el fin de diferenciarlos claramente de las líneas que deben generarse en el archivo de simulación para ese template.

Entonces, el formato de cada template será el siguiente:

|--template *identif*--|
|--top components --|
linea$_1$
linea$_2$
.
.
linea$_n$
|--top ports--|
linea$_1$
linea$_2$
.
.
linea$_n$

```
|--top links--|
linea1
linea2
  .
  .
lineaN
|--before neighbors--|
linea1
linea2
  .
  .
lineaN
|--neighbors--|
linea1
linea2
  .
  .
lineaN
|--before ports--|
linea1
linea2
  .
  .
lineaN
|--ports--|
linea1
linea2
  .
  .
lineaN
|--before links--|
linea1
linea2
  .
  .
lineaN
|--links--|
linea1
linea2
  .
  .
lineaN
|--before zones--|
linea1
linea2
  .
  .
lineaN
|--zones--|
linea1
linea2
  .
  .
lineaN
|--before rules--|
linea1
linea2
```

```
  .
  .
linea_n
|--rules--|
bloque_1
bloque_2
  .
  .
bloque_n
|--after rules--|
|--end template--|
```

donde,

|--template *identif*--|:
Define el tipo de template.
Más adelante en este documento se listan los valores posibles que puede tomar ***identif*** y qué elemento del plano representa.

|--top components --|:
Indica que el conjunto de líneas que continúan se agregan al modelo acoplado **top** para los elementos que este template genera.

|--top ports--|:
Indica que el conjunto de líneas que continúan definen los **ports** que se agregan al modelo acoplado **top** para los elementos que este template genera.

|--top links--|:
Indica que el conjunto de líneas que continúan definen los **links** que se agregan al modelo acoplado **top** para los elementos que este template genera.

|--before neighbors--|:
Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan debe ubicarse antes de definir los neighbors del modelo Cell-DEVS de este tramo o cruce.
Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección before neighbors del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--neighbors--|
Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan define los neighbors del modelo Cell-DEVS de este tramo o cruce.
Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección neighbors del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--before ports--|
Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan debe ubicarse antes de definir los ports del modelo Cell-DEVS de este tramo o cruce.
Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección before ports del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--ports--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan define los ports del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección ports del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--before links--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan debe ubicarse antes de definir los links del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección before links del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--links--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan define los links del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección links del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--before zones--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan debe ubicarse antes de definir los zones del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección before zones del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--zones--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan define los zones del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección zones del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--before rules--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan debe ubicarse antes de definir los rules del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección before rules del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--rules--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan define los rules del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección rules del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

|--after rules--|

Para templates de tramos y cruces: Indica que el conjunto de líneas que continúan debe ubicarse después de definir los rules del modelo Cell-DEVS de este tramo o cruce.

Para templates de otros elementos: Indica que el conjunto de líneas que continúan deben complementar la sección after rules del modelo Cell-DEVS del tramo o cruce afectado por este elemento.

Además de un modelo Cell-DEVS por cada tramo y cruce, el compilador genera modelos atómicos DEVS para los siguientes elementos:

- semáforos: generan semáforos y sincronizador de semáforos.
- red de vías: generan vías y un sincronizador de vías.
- tramos de entrada: generan un Generador de autos.
- tramos de salida: generan un Consumidor de autos.

El propósito de esta sección reside precisamente en la especificación de dichos modelos DEVS.

|--end template--|:

Indica que terminó la definición de este template.

$linea_i$:

Define cada una de las líneas que serán escritas en el archivo de salida. Estas líneas pueden contener macro-variables que se traducen en el momento de la generación de los modelos. El significado y utilización de las mismas se detalla más adelante en este capítulo.

$bloque_i$:

Define cada conjunto de reglas para el elemento que corresponde a este template. La sintaxis de cada bloque es:

[*identif_bloque*]
$linea_1$
$linea_2$
.
.
$linea_n$

Dónde ***identif_bloque*** puede contener macro-variables y debe corresponderse con alguno de los bloques especificados por la sentencia zone.

El compilador genera sólo una vez cada bloque de reglas en caso de no contener ninguna macro-variable entre las líneas que lo componen. Si existe alguna macro-variable, entonces se deberá generar el bloque de reglas para cada elemento ya que el valor de la macro-variable luego de ser traducida puede variar de un elemento a otro.

Al final de la especificación de los templates para cada tipo de elemento del plano encontramos un template particular que define las macros que se generarán en el archivo de macros.
Este template tiene el siguiente formato:

|--Macros--|
#BeginMacro(*identif_macro*$_1$)
$linea_1$
$linea_2$
.
.
$linea_n$
#EndMacro

```
#BeginMacro(identif_macro2)
linea1
linea2
  .
  .
lineaN
#EndMacro

  .
  .

#BeginMacro(identif_macroM)
linea1
linea2
  .
  .
lineaN
#EndMacro
|--end macros--|
```

donde,

|--Macros--|:
Indica el comienzo del template de macros.

#BeginMacro(*identif_macro*$_i$):
Indica el comienzo de la definición de una macro de nombre ***identif_macro***$_i$. El nombre puede contener macro-variables.

linea$_i$:
Se define de la misma forma que para el resto de los templates.

#EndMacro:
Indica el fin de la definición de la macro.

|--end macros--|:
Indica fin del template de macros.

Luego de procesar el archivo que contiene el plano a simular, TSC invoca a un segundo parser (***iniparse***) que procesa el archivo conteniendo los templates de generación de código (tsc.ini o el indicado con la opción –t) y genera la instancia de CodeStruct conteniendo la estructura de generación completa.
A continuación se detalla la nomenclatura utilizada para nombrar los diferentes templates de generación. Como puede observarse hay distintos templates para un mismo elemento, esto es debido a que el código a generar depende de las características y ubicación de dicho elemento. Por ejemplo, el código generado para un bache de un tramo es distinto si el bache se encuentra en la primera celda o en la última celda del tramo.

ControlSegment-***i***Lane-Cell	Elementos de control para tramos de ***i*** carriles - celdas del medio
ControlSegment-***i***Lane-Cell0	Elementos de control para tramos de ***i*** carriles - primera celda
ControlSegment-***i***Lane-Celln	Elementos de control para tramos de ***i*** carriles - última celda

Crossing	Cruces
Crossing-with-Hole	Cruces con baches
Crossing-without-Hole	Cruces sin baches
HoleSegment-***i***Lane-Cell***j***0	Baches para tramos de ***i*** carriles - carril ***j***- primera celda
HoleSegment-***i***Lane-Cell***j***n	Baches para tramos de ***i*** carriles - carril ***j***- última celda
HoleSegment-***i***Lane-Lane***j***	Baches para tramos de ***i*** carriles - carril ***j***- celdas del medio
JobsiteSegment	Tramos con obras
Segment-***i***Lane	Tramos de ***i*** carriles
Segment-***i***Lane-EndConsumer	Tramos de ***i*** carriles que terminan en un consumidor
Segment-***i***Lane-EndCrossing	Tramos de ***i*** carriles que terminan en un cruce
Segment-***i***Lane-StartCrossing	Tramos de ***i*** carriles que comienzan en un cruce
Segment-***i***Lane-StartGenerator	Tramos de ***i*** carriles que comienzan en un generador
SincroRailNet	Sincronizador de vías
TrafficLightCrossing	Cruces con semáforo
TrafficLightSegment-***i***Lane	Tramos de ***i*** carriles con semáforo
TrainSegment-***i***Lane	Tramos de ***i*** carriles con trenes
Top	Modelo acoplado TOP

Por ejemplo, si se desea generar el código para un tramo de entrada (comienza en un generador), de 1 carril y con un bache en las celdas del medio, entonces se utilizarán los siguientes templates de generación de código:

- HoleSegment-1Lane-Lane0
- Segment-1Lane-StartGenerator
- Segment-1Lane-EndCrossing
- Segment-1Lane

Como se explicó anteriormente, las líneas de los templates de generación están, a su vez, subdivididas por secciones según la posición que ocuparán dentro del archivo de salida. La generación de código se hace entonces agregando las líneas de los templates correspondientes en forma ordenada en el archivo de salida.

Entonces, para generar el código, se agregan primero las líneas de |**--top components --**| para cada uno de los cuatro modelos, luego las líneas de |**--top ports--**|, luego las líneas de |**--top links--**|, luego las de |**--before neighbors--**| y así siguiendo hasta terminar con las líneas de |**--after rules--**|.

El procesamiento de estas secciones de líneas es similar en todos los casos excepto para los bloques de reglas, en donde es necesario identificar si se están utilizando macros y generar las mismas en el archivo de macros.

4.6. Macro-variables

Los modelos de simulación de cada elemento que compone el plano de ciudad contienen información propia de ese elemento.
Algunos ejemplos de esto son:

- El tamaño del modelo celular generado para un tramo depende de la cantidad de celdas de ese tramo.
- Un bache dentro de un tramo modifica el comportamiento de la celda que contiene dicho bache.

- Las vías que cruzan un tramo alteran el comportamiento de las celdas que se encuentran antes y después de la vía en ese tramo.

Esto hace necesario que en los templates de generación de código se puedan utilizar variables que reflejen las características particulares de cada elemento del plano y que en el momento de generar el modelo para ese elemento se reemplacen por el valor correspondiente.

Por esta razón definimos un conjunto de macro-variables que pueden incluirse en las líneas de cada template. Cuando TSC encuentra una macro-variable en alguna línea, invoca un método que devuelve el valor correspondiente. Este método puede ser tan simple como devolver la cantidad de celdas de un tramo o repetir una línea tantas veces como carriles tenga un tramo.

Las macro-variables dentro de una línea son identificadas por TSC por comenzar y terminar con el símbolo **&**. Por ejemplo, **&IDENTIF&** se reemplazará por el nombre del identificador correspondiente para ese elemento.

En el caso de los cruces existen dependencias entre macro-variables, o sea que no pueden existir unas si otras no están presentes en la misma línea.

A continuación se detalla el conjunto de macro-variables existentes clasificadas según el elemento al que corresponden y se indican las dependencias entre las macro-variables de cruces. Finalmente se dan algunos ejemplos del proceso de traducción de macro-variables de TSC.

4.6.1. Macro-variables de tramos

IDENTIF	Reemplaza por el identificador
SPEED	Reemplaza por la velocidad
DELAY	Reemplaza por la demora para autos estacionados
LANE	Repite la línea una vez por cada carril y se reemplaza por el número de carril correspondiente
CELL	Repite la línea una vez por cada celda (columna) y se reemplaza por el número de celda correspondiente
FIRST_LANE	Reemplaza por el primer carril
LAST_LANE	Reemplaza por el último carril
FIRST_CELL	Reemplaza por la primera celda
LAST_CELL	Reemplaza por la última celda
LAST_CELL-1	LAST_CELL - 1
WIDTH	LAST_CELL + 1
HEIGHT	LAST_LANE + 1
ENDCROSS_IDENTIF	Identificador del cruce por donde salen los autos
STARTCROSS_IDENTIF	Identificador del cruce por el que entran los autos

4.6.2. Macro-variables de tramos con vías

RAIL_IDENTIF	Reemplaza por el identificador de vías
RAIL_DELAY	Reemplaza por la demora para las vías
RAIL_ORDER	Reemplaza por el número de orden en que se encuentra el tramo dentro de la especificación de la red de vías
BEFORE_TRAIN_CELL	Reemplaza por la celda (columna) del tramo anterior al tren

AFTER_TRAIN_CELL	Reemplaza por la celda (columna) del tramo siguiente al tren

4.6.3. Macro-variables de cruces

IDENTIF	Reemplaza por el identificador
SPEED	Reemplaza por la velocidad
DELAY	Reemplaza por la demora para los baches de los cruces
CELL	Repite la línea por cada celda del cruce. Reemplaza por el n correspondiente a la ubicación de la celda.
SEG_LANE	Dada una celda, reemplaza por el carril del tramo que se acopla a esa celda del cruce
SEG_CELL	Dada una celda, reemplaza por la celda (columna) del tramo que se acopla a esa celda del cruce
IN	Repite la línea por cada celda de IN (celdas por las que entra un tramo) del cruce. Reemplaza por el n correspondiente a la ubicación de la celda. Teniendo en cuenta las celdas de OUT estos números pueden no ser correlativos.
OUT	Repite la línea por cada celda de OUT (celdas por las que sale un tramo) del cruce. Reemplaza por el n correspondiente a la ubicación de la celda. Teniendo en cuenta las celdas de IN estos números pueden no ser correlativos.
(IN)	Reemplaza por la cantidad de celdas de IN
(OUT)	Reemplaza por la cantidad de celdas de OUT
#IN	Repite la línea (IN) veces. Reemplaza desde 0 hasta (IN) sin incluir.
#OUT	Repite la línea (OUT) veces. Reemplaza desde 0 hasta (OUT) sin incluir.
LAST_CELL	Reemplaza por la última celda
WIDTH	LAST_CELL + 1
POUT	Reemplaza por la probabilidad de salir

(IN_SEGMENTS)	Reemplaza por la cantidad de tramos que entran
#IN_SEGMENTS	Repite la línea (IN_SEGMENTS) veces. Reemplaza desde 0 hasta (IN_SEGMENTS) sin incluir.
IN_SEGMENTS	Repite la línea para cada tramo que entra al cruce. Reemplaza por el identificador de tramo
IN_SEGMENT	Dada una celda de IN, reemplaza por el identificador de tramo
SEG_LANE_IN	Dada una celda de IN, reemplaza por el carril del tramo que se acopla a esa celda del cruce
SEG_CELL_IN	Dada una celda de IN, reemplaza por la celda (columna) del tramo que se acopla a esa celda del cruce

(OUT_SEGMENTS)	Reemplaza por la cantidad de tramos que salen
#OUT_SEGMENTS	Repite la línea (OUT_SEGMENTS) veces. Reemplaza desde 0 hasta (OUT_SEGMENTS) sin incluir.
OUT_SEGMENTS	Repite la línea para cada tramo que sale del cruce. Reemplaza por el identificador de tramo
OUT_SEGMENT	Dada una celda de OUT, reemplaza por el identificador de tramo
SEG_LANE_OUT	Dada una celda de OUT, reemplaza por el carril del tramo que se acopla a esa celda del cruce
SEG_CELL_OUT	Dada una celda de OUT, reemplaza por la celda (columna) del tramo que se acopla a esa celda del cruce

Dependencias para macro-variables de cruces

Macro-variable	Depende de

SEG_LANE	CELL
SEG_CELL	CELL
IN_SEGMENT	IN, #IN o #IN_SEGMENTS
SEG_LANE_IN	IN, #IN, #IN_SEGMENTS o IN_SEGMENTS
SEG_CELL_IN	IN, #IN, #IN_SEGMENTS o IN_SEGMENTS
OUT_SEGMENT	OUT, #OUT o #OUT_SEGMENTS
SEG_LANE_OUT	OUT, #OUT, #OUT_SEGMENTS o OUT_SEGMENTS
SEG_CELL_OUT	OUT, #OUT, #OUT_SEGMENTS o OUT_SEGMENTS

4.6.4. Macro-variables de obras

JOBSITE_CELLS	Reemplaza por todas las celdas afectadas por la obra
DELAY	Reemplaza por la demora para las obras

4.6.5. Macro-variables de baches

HOLE_LANE	Reemplaza por el carril afectado por el bache
HOLE_CELL	Reemplaza por la celda (columna) afectada por el bache
DELAY	Reemplaza por la demora para los baches

4.6.6. Macro-variables de elementos de control

CONTROL_CELL	Reemplaza por la celda (columna) afectada por un elemento de control
DELAY	Reemplaza por la demora para un elemento de control

4.6.7. Macro-variables de sicronizador de trenes

IDENTIF	Reemplaza por el identificador de vías
(SEGMENT)	Reemplaza por la cantidad de tramos que atraviesa el tren
#SEGMENT	Repite la línea (SEGMENT) veces. Reemplaza desde 0 hasta (SEGMENT) sin incluir.
SEGMENT	Repite la línea para cada uno de los tramos que atraviesa el tren. Reemplaza por el identificador de tramo.

4.6.8. Ejemplo de traducciones para tramos

Dado un tramo definido como TramoA de 2 carriles y 4 celdas. Algunas traducciones sencillas son:

`[&IDENTIF&]`	[**TramoA**]
`width : &WIDTH&`	`width : 4`
in : x_c_hayauto&LANE&	in : x_c_hayauto**0** in : x_c_hayauto**1**
in : x_c_hayauto&LANE&&FIRST_CELL&	in : x_c_hayauto**00** in : x_c_hayauto**10**

El siguiente cuadro muestra la secuencia de pasos que se realizan para una traducción más compleja utilizando el mismo ejemplo:

link : y_c_hayauto@&IDENTIF&(&LANE&,&LAST_CELL&) y_c_hayauto&LANE&&LAST_CELL&
link : y_c_hayauto@**TramoA**(&LANE&,&LAST_CELL&) y_c_hayauto&LANE&&LAST_CELL&
link : y_c_hayauto@TramoA(&LANE&,**3**) y_c_hayauto&LANE&**3**
link : y_c_hayauto@TramoA(**0**,3) y_c_hayauto**0**3 link : y_c_hayauto@TramoA(**1**,3) y_c_hayauto**1**3

4.6.9. Ejemplo de traducciones para cruces

Dado un cruce uniendo en este orden:

- TramoA de 1 carril de entrada
- TramoB de 2 carriles de salida
- TramoC de 1 carril de salida
- TramoD de 3 carriles de entrada
- TramoE de 2 carriles de entrada

Algunas traducciones son:

in : x_t_hayauto&IN&	in : x_t_hayauto**0** in : x_t_hayauto**4** in : x_t_hayauto**5** in : x_t_hayauto**6** in : x_t_hayauto**7** in : x_t_hayauto**8**
out: y_t_hayauto&OUT&	out: y_t_hayauto**1** out: y_t_hayauto**2** out: y_t_hayauto**3**
&(IN)&	**6**
&(OUT)&	**3**
in : portIn&#IN&	in : portIn**0** in : portIn**1** in : portIn**2** in : portIn**3** in : portIn**4** in : portIn**5**
out : portOut&#OUT&	out : portOut**0** out : portOut**1** out : portOut**2**
&(IN_SEGMENTS)&	**3**
y_se_luz&#IN_SEGMENTS&	y_se_luz**0** y_se_luz**1** y_se_luz**2**
x_si_luz@&IN_SEGMENT&tl	x_si_luz@**TramoA**tl x_si_luz@**TramoD**tl x_si_luz@**TramoE**tl
&(OUT_SEGMENTS)&	**2**
portOut&#OUT_SEGMENTS&	portOut**0**

	portOut**1**
portOut@&OUT_SEGMENTS&	portOut@**TramoB** portOut@**TramoC**

4.7. Modelos atómicos de TSC

Los modelos atómicos DEVS utilizados por TSC que complementan a N-CD++, se corresponden con las especificaciones definidas en ATLAS.
Los mismos, como se mencionó anteriormente, son:

- **TSCGenerator**

  Es el encargado de generar los autos que ingresan a la simulación. Se acopla con un tramo de entrada, introduciendo los vehículos según una función de distribución recibida como parámetro. Los autos se generan en cada carril del tramo acoplado en forma rotativa, entrando por el siguiente carril respecto al último ingresado.

- **TSCConsumer**

  Es el encargado de eliminar y contar los autos que se retiran de la simulación. Se acopla con un tramo de salida e informa el promedio y la cantidad total de vehículos salientes por dicho tramo según una frecuencia y una unidad de tiempo recibidas como parámetro.

- **TSCSincroTrafficLight**

  Se acopla con todos los semáforos (TSCTrafficLight) de un cruce y es el encargado de la sincronización de los mismos, indicando en forma alternada a cuál le corresponde la luz verde.

- **TSCTrafficLight**

  Se acopla con un tramo de entrada a un cruce indicándole el color del semáforo a las últimas celdas del tramo y restringiendo de esta manera el avance de los autos hacia el cruce.

- **TSCSincroRailNet**

  Se acopla con todas las vías (TSCRailnet) de una red de vías y es el encargado de la sincronización del paso del tren, indicando en forma alternada la marcha del mismo a través de las vías.

- **TSCRailNet**

  Se acopla con un tramo interrumpido por una red de vías, indicándole a las celdas adyacentes a las vías la presencia o ausencia del tren, restringiendo de esta manera el avance de los autos dentro del tramo.

- **TSCCounter**

  A diferencia de los anteriores, este modelo atómico DEVS de tipo contador fue definido en [DVW00]. Se acopla con un cruce y es el encargado de contar los autos que entran y salen de

dicho cruce, informando ambos promedios según una frecuencia y una unidad de tiempo recibidas como parámetro

4.8. Detalles de la generación del modelo

El propósito de esta sección es expresar en forma más detallada algunas consideraciones tomadas al crear los templates de generación de código para el lenguaje de especifícación ATLAS y que facilitan su entendimiento. Entre las mismas se encuentran la forma de acoplar los modelos atómicos y la utilización de los modelos atómicos DEVS. En la sección siguiente se encuentra la nomenclatura utilizada para la definición de los acoplamientos.

Todos los acoplamientos necesarios entre tramos y cruces se arman durante la generación de los modelos de cruces, por lo tanto las definiciones correspondientes se encuentran solamente en los templates de los cruces.

Los semáforos están definidos en los cruces, resultando en un semáforo por cada tramo de entrada al mismo y un sincronizador que regula todos los semáforos del cruce.

La generación de semáforos se hace en dos partes:

- durante la generación de los tramos se arma el modelo del semáforo y los acoplamientos entre tramo y semáforo. El nombre del modelo del semáforo se forma con el identificador de tramo más el string **"tl"**. Para ello se utiliza el template TrafficLightSegment-***i***Lane siendo ***i*** el número de carriles del tramo.
- durante la generación de los cruces se arma el modelo del sincronizador de semáforos y los acoplamientos entre semáforo y sincronizador. El sincronizador tiene un port de salida por cada uno de los semáforos (o tramos de entrada al cruce) que sincroniza. El nombre se forma con el identificador de cruce más el string **"stl"** y el template que se utiliza es TrafficLightCrossing.

Los modelos atómicos DEVS que se utilizan son TSCSincroTrafficLight para el sincronizador de semáforos y TSCTrafficLight para cada uno de los semáforos.

La implementación de vías es muy similar a la de los semáforos, ya que hay un sincronizador que regula el paso del tren y un modelo de vía por cada tramo que se encuentra dentro de la red de vías.

La generación de vías se hace en dos partes:

- durante la generación de los tramos se arma el modelo de vía y los acoplamientos entre tramo y vía. El nombre del modelo de vía se forma con el identificador de red de vías más el número de orden en que se encuentra el tramo dentro de la red de vías. Para ello se utiliza el templates TrainSegment-***i***Lane siendo ***i*** el número de carriles del tramo.
- el sincronizador de vías se arma aparte, definiendo también los acoplamientos entre vías y sincronizador. Este sincronizador tiene un port de salida por cada una de las vías que sincroniza. El nombre se forma con el identificador de vías y el template que se utiliza es SincroRailNet.

Los modelos atómicos DEVS que se utilizan son TSCSincroRailNet para el sincronizador de vías y TSCRailNet para cada una de las vías.

La implementación de obras se reemplazó por la especificación de una zona (las celdas de la obra) en donde todas las celdas que la componen están ocupadas durante toda la simulación (mantienen el estado en 1 = celda ocupada). De esta forma los autos de las celdas anteriores al ver que las celdas de la obra están ocupadas las esquivan.

La implementación de autos estacionados se reemplazó por la especificación del carril (izquierdo, derecho o ambos) conteniendo baches con demora bien alta. De esta forma los autos permanecen durante un tiempo largo dentro de una celda, simulando que se encuentran estacionados.

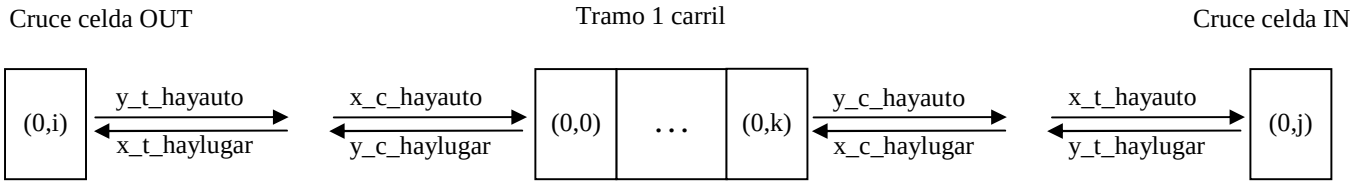
Es necesario notar que cada bache, cada elemento de control, cada cruce y cada tramo tienen su propia demora. De esta forma se pueden modelar, por ejemplo, distintos baches en donde la demora de los autos está determinada por la profundidad del pozo. Tomando este enfoque, la demora no puede estar definida en los templates de generación, sino que debe estar especificada por cada elemento de simulación y ser utilizarla a través de macro-variables. Si bien hay que repetir la demora en caso de que se quiera utilizar siempre la misma, el compilador es así más flexible, y más adelante, se podría definir una demora por defecto en una interface gráfica.

4.9. Definición de acoplamientos

En esta sección se describe gráficamente la forma en que se realizan los acoplamientos entre los diferentes modelos que componen la especificación del plano. En ATLAS estas definiciones fueron hechas de una manera general. TSC respeta estas definiciónes para sus templates por defecto pero completa los identificadores de ports con una codificación particular para no repetir nombres de ports durante la generación del código.
Para cada uno de los acoplamientos existentes se detalla en primer lugar la forma en que ATLAS define los mismos y luego la nomenclatura que se utiliza en TSC para completar esta definición.

4.9.1. Entre tramos de 1 carril y cruces

El siguiente gráfico muestra el acoplamiento entre tramos y cruces tal como se define en ATLAS:



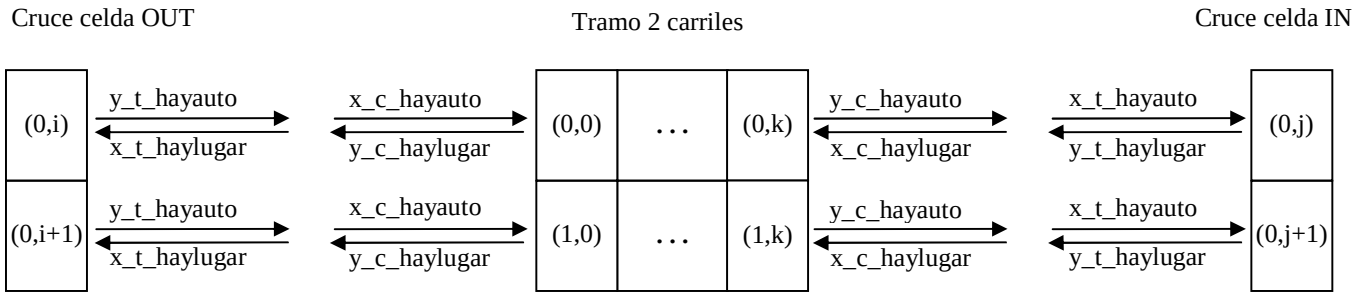
Siendo **i** la posición de la celda del cruce a la que se acopla el comienzo del tramo; y **j** la posición de la celda del cruce a la que se acopla el final del tramo.
La posición de la celda del cruce a la que debe acoplarse un tramo está determinada por la inclinación de ese tramo con respecto a ese cruce. En otras palabras, el orden de acoplamiento de los tramos que se acoplan con un mismo cruce está dado por su función de inclinación.

Para generar el código de estos acoplamientos se especifica cada puerto por separado, por lo tanto se agrega al nombre de los ports la posición de la celda en la que se produce el acoplamiento:

Tanto para los cruces como para los tramos, el nombre del modelo que se genera para especificarlos se corresponde con el identificador de los mismos.

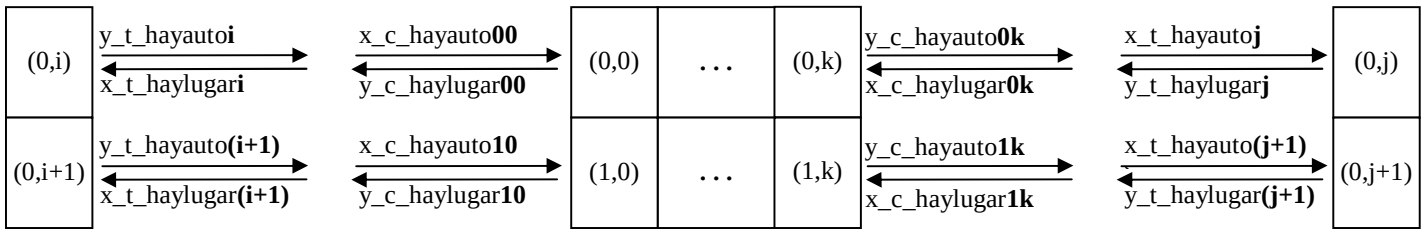
4.9.2. Entre tramos de 2 carriles y cruces

El siguiente gráfico muestra el acoplamiento entre tramos y cruces tal como se define en ATLAS.



Siendo **i** e **i+1** las posiciones de las celdas del cruce a las que se acopla el comienzo del tramo; y **j** y **j+1** las posiciones de la celdas del cruce a la que se acopla el final del tramo.
Las posiciones de las celdas del cruce a las que debe acoplarse un tramo están determinadas por la inclinación de ese tramo con respecto a ese cruce. En otras palabras, el orden de acoplamiento de los tramos que se acoplan con un mismo cruce está dado por su función de inclinación.

Para generar el código de estos acoplamientos se especifica cada puerto por separado, por lo tanto se agrega al nombre de los ports la posición de la celda en la que se produce el acoplamiento:

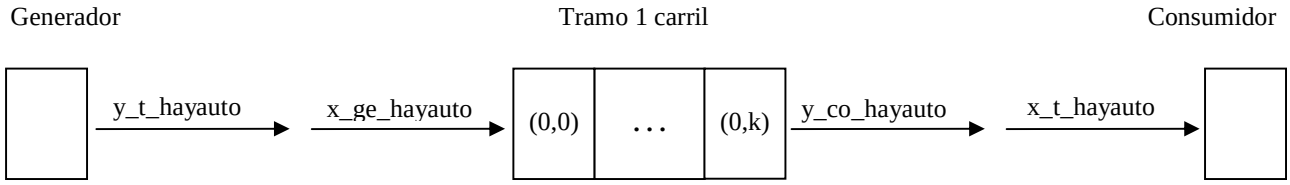


Tanto para los cruces como para los tramos, el nombre del modelo que se genera para especificarlos se corresponde con el identificador de los mismos.

Análogamente se definen los acoplamientos entre tramos de más de 2 carriles y cruces. Siempre se agrega al nombre de los ports la posición de la celda en la que se produce el acoplamiento.

4.9.3. Entre tramos de 1 carril - generador - consumidor

El siguiente gráfico muestra el acoplamiento entre tramos y cruces tal como se define en ATLAS:

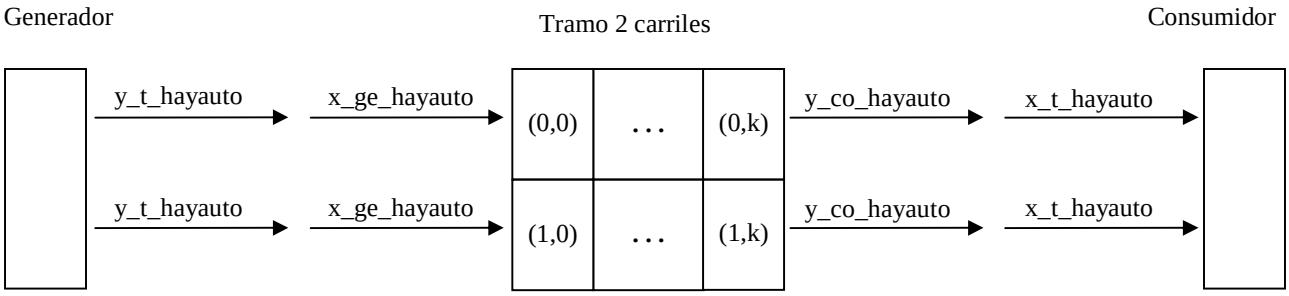


Para generar el código de estos acoplamientos se especifica cada puerto por separado, por lo tanto se agrega al nombre de los ports del tramo la posición de la celda en la que se produce el acoplamiento; y al nombre de los ports del generador y el consumidor el número del carril al que se acoplan:

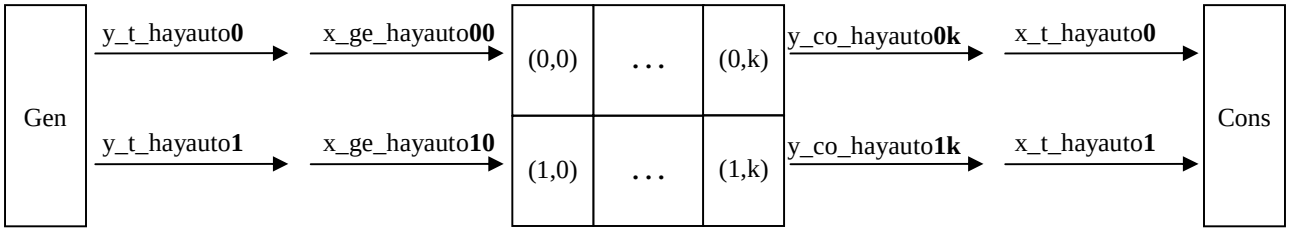
El nombre del generador se forma con el identificador de tramo más el agregado del string "Gen" y corresponde al modelo atómico DEVS TSCGenerator.
El nombre del consumidor se forma con el identificador de tramo más el agregado del string "Cons" y corresponde al modelo atómico DEVS TSCConsumer.

4.9.4. Entre tramos de 2 carriles - generador - consumidor

El siguiente gráfico muestra el acoplamiento entre tramos y cruces tal como se define en ATLAS.



Para generar el código de estos acoplamientos se especifica cada puerto por separado, por lo tanto se agrega al nombre de los ports del tramo la posición de la celda en la que se produce el acoplamiento; y al nombre de los ports del generador y el consumidor el número del carril al que se acoplan:

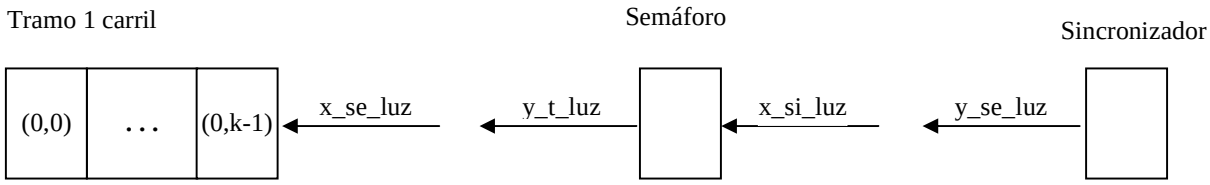


El nombre del generador se forma con el identificador de tramo más el agregado del string "Gen" y corresponde al modelo atómico DEVS TSCGenerator.
El nombre del consumidor se forma con el identificador de tramo más el agregado del string "Cons" y corresponde al modelo atómico DEVS TSCConsumer.

Análogamente se definen los acoplamientos entre tramos de más de 2 carriles y generador y consumidor. Siempre se agrega al nombre de los ports del tramo la posición de la celda en la que se produce el acoplamiento; y al nombre de los ports del generador y el consumidor el número del carril al que se acoplan.

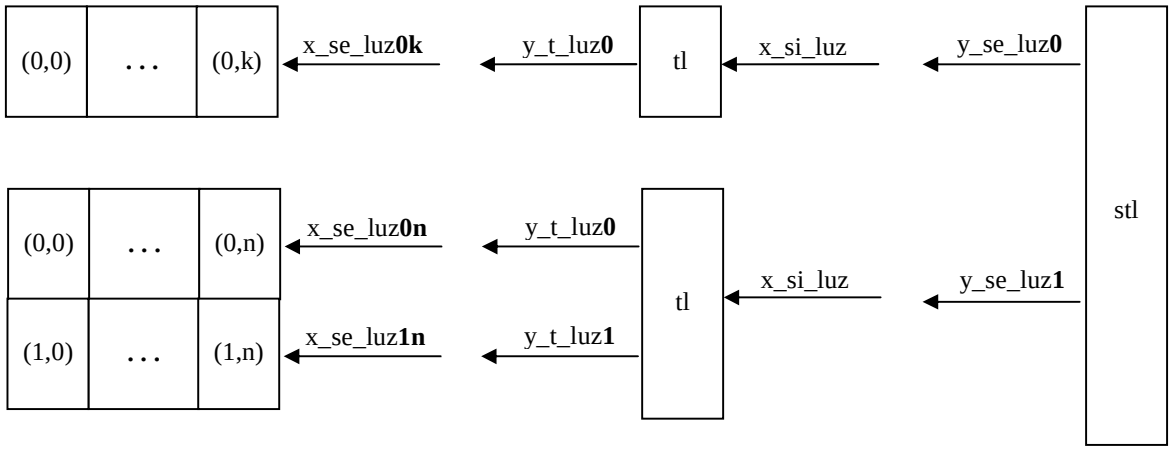
4.9.5. Entre tramos y semáforos

El siguiente gráfico muestra el acoplamiento entre tramos y semáforos tal como se define en ATLAS.



Para generar el código de estos acoplamientos se especifica cada puerto por separado. Para ello:

- se agrega al nombre de los ports de entrada a los tramos la posición de la celda de acoplamiento.
- se agrega a los ports de salida del semáforo el número del carril del tramo al que se acoplan.
- se agrega a los ports de salida del sincronizador el número del semáforo al que sincronizan. La forma de numerar los semáforos la determina el orden de inclinación de los tramos que entran al cruce donde está definido el semáforo.

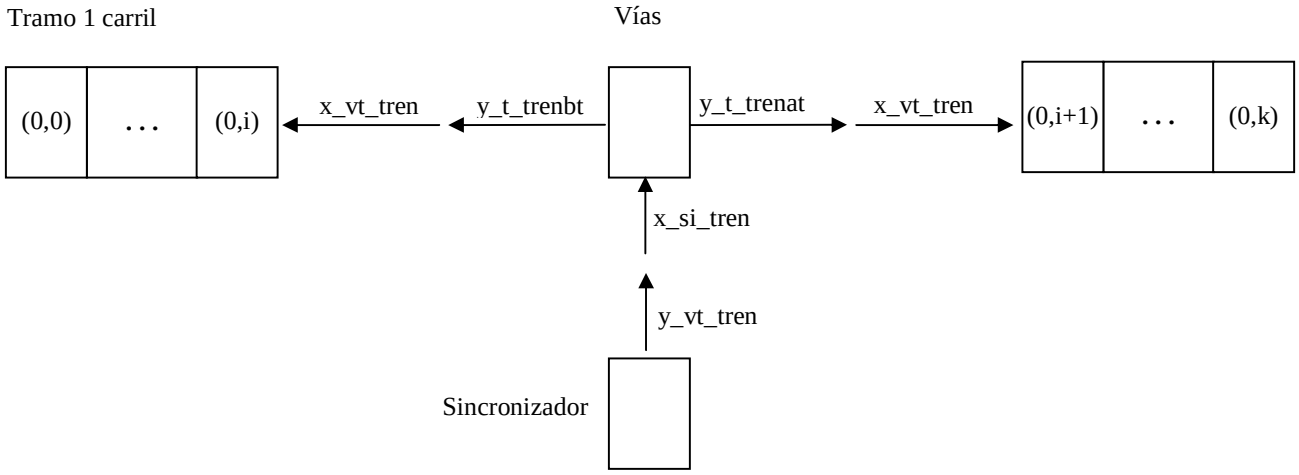


El nombre del modelo que especifica al semáforo se forma con el identificador de tramo más el agregado del string "**tl**" y corresponde al modelo atómico DEVS TSCTrafficLight.

El nombre del modelo sincronizador se forma con el identificador del cruce más el agregado del string "**stl**" y corresponde al modelo atómico DEVS TSCSincroTrafficLight.

4.9.6. Entre tramos y vías

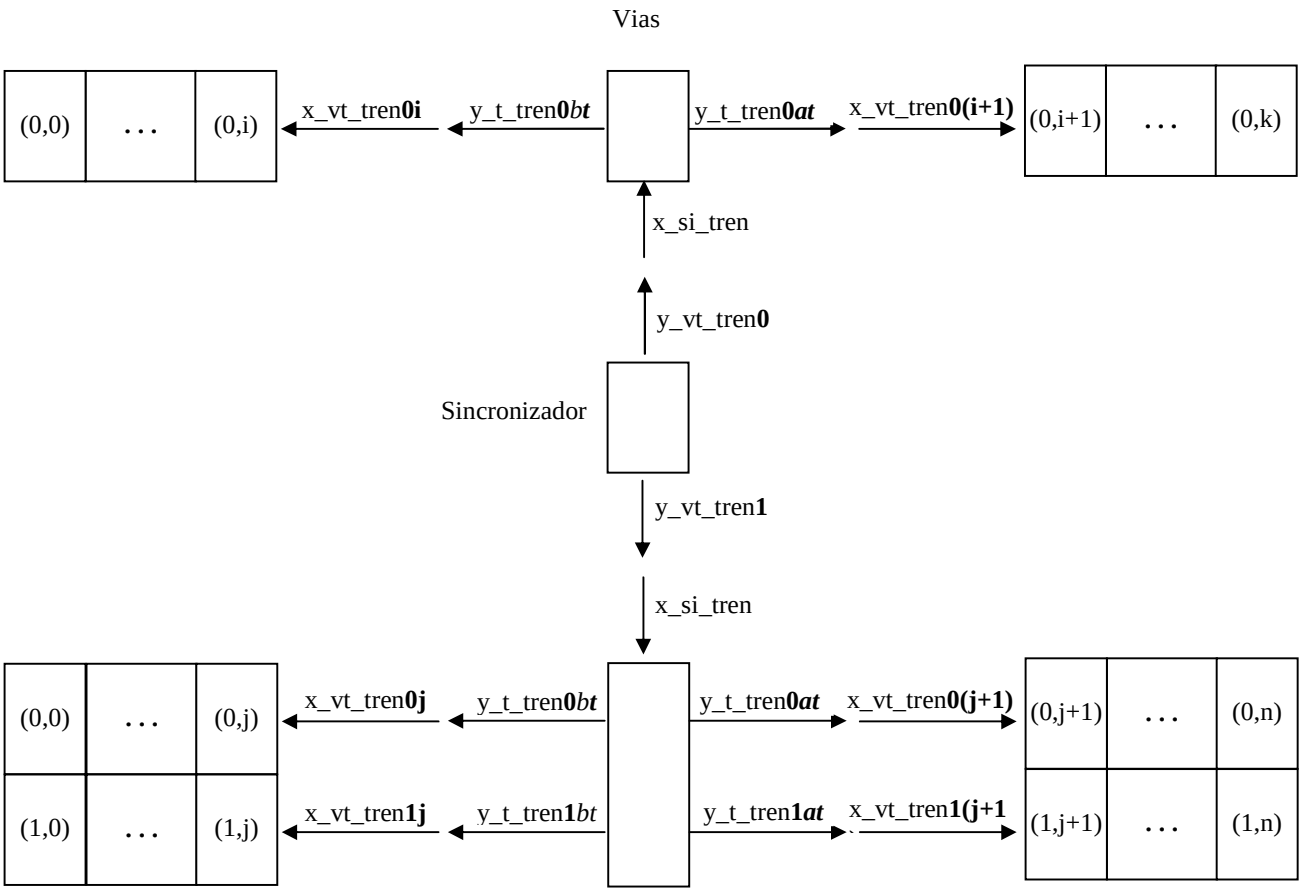
El siguiente gráfico muestra el acoplamiento entre tramos y vías tal como se define en ATLAS.



Siendo **i** la distancia de la vía al comienzo del tramo que atraviesa.

Para generar el código de estos acoplamientos se especifica cada puerto por separado. Para ello:

- se agrega al nombre de los ports de entrada a los tramos la posición de la celda a la que se acopla la vía.
- se agrega al nombre de los ports de salida de la vía el número del carril del tramo al que se acoplan.
- se agrega a los ports de salida del sincronizador el número la vía que sincroniza. La forma de numerar las vías la determina el número de orden dentro de la red de vías, del tramo al que se acoplan.



El nombre del modelo que especifica la vía se forma con el identificador de la red de vías más el número de orden correspondiente para ese tramo dentro del conjunto de vías. El modelo atómico DEVS utilizado es TSCRailnet.
El nombre del sincronizador es el identificador de la red de vías y corresponde al modelo atómico DEVS TSCSincroRailNet.

4.10. Ejemplos de especificación de un plano de ciudad

Acompañando el presente trabajo se encuentra un conjunto de ejemplos en medio magnético conteniendo una secuencia de especificaciones de planos derivadas a partir de un modelo original. La idea de presentar esta secuencia fue intentar particionar un plano para facilitar la comprensión del proceso de generación, es decir, dar un camino para llegar de las partes al todo.

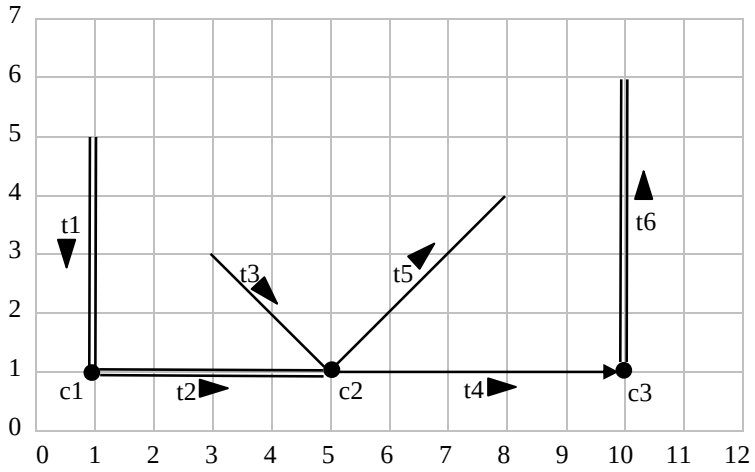
El modelo original es el archivo model.plan y el resto son subdivisiones o partes más pequeñas del mismo plano.

Sin embargo, es necesario aclarar que el código generado por el archivo model.plan no resulta de la unión total del código de los submodelos, ya que la especificación de los tramos es distinta para:

- Tramos de Entrada, comienzan en un generador y terminan en un cruce
- Tramos de Salida, comienzan en un cruce y terminan en un consumidor
- Tramos de Entrada y Salida, comienzan en un generador y terminan en un consumidor
- Tramos Intermedios, comienzan y terminan en un cruce

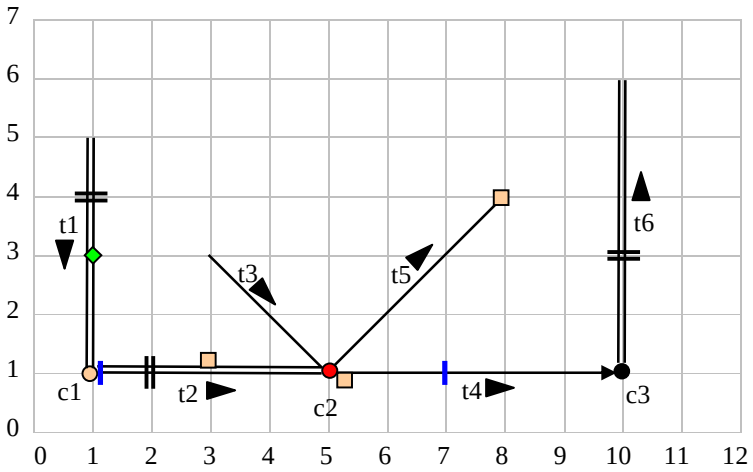
Por lo tanto el código generado para los distintos submodelos depende de cómo se haya particionado el modelo original, determinando asi la clasificacón de los tramos del submodelo.

El plano simplificado que especifíca el archivo model.plan es el siguiente:




```
Tramos
  t1 : de (1,5) a  (1,1), 2 carriles, 4 celdas
  t2 : de (1,1) a  (5,1), 2 carriles, 4 celdas
  t3 : de (3,3) a  (5,1), 1 carril,   2 celdas
  t4 : de (5,1) a (10,1), 1 carril,   5 celdas
  t5 : de (5,1) a  (8,4), 1 carril,   4 celdas
  t6 : de(10,8) a (10,1), 2 carril,   5 celdas

Cruces
  c1 : (1,1)
  c2 : (5,1)
  c3 : (10,1)
```

El plano completo que especifica el model.plan es el siguiente:



Vía

Cruce sin semáforo y sin bache

Cruce con semáforo y sin bache

Cruce sin semáforo y con bache

Bache

Obra

Elemento de control

La especificación completa del plano es la siguiente:

```
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
```

Dado el tamaño de los archivos model.ma y model.macros generados, los mismos no han sido incluídos en el presente informe. Para su visualización y la de los distintos submodelos generados recomendamos consultar el medio magnético que acompaña a este trabajo.

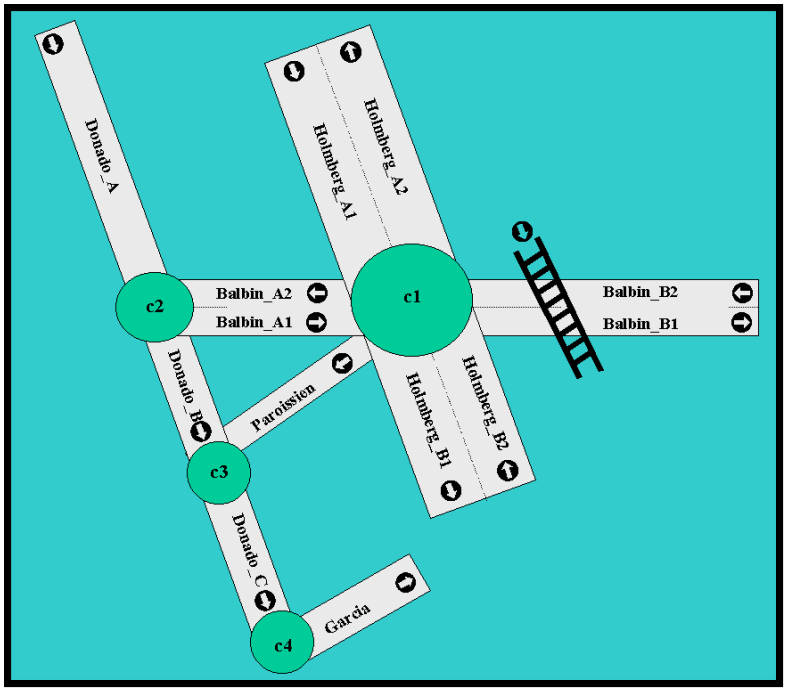
Capítulo 5 - Un caso de aplicación

Este capítulo está orientado a detallar las pruebas realizadas al finalizar la herramienta TSC para comprobar su eficiencia y flexibilidad.
Para probar TSC se tomó el sector de ciudad definido en [DVW00]. Este sector es una sección de la ciudad de Buenos Aires donde el flujo de tráfico no es significativo pero donde frecuentemente ocurren embotellamientos. El trabajo realizado en [DVW00] consistió en definir todos los modelos celulares y atómicos necesarios para poder simular el comportamiento de ese sector de ciudad utilizando N-CD++. La definición de los modelos, los ports de comunicación y las reglas de los mismos, se basó en el lenguaje de especificación de tráfico ATLAS.

La tarea que se realizó con TSC fue utilizar la herramienta para generar estos mismos modelos.
Para ello se especificó, en primer lugar, el mismo sector de ciudad en términos de tramos, cruces, semáforos, vías, etc. Por lo tanto, la especificación se hizo en los términos que utilizaría un analizador de tráfico, abstrayendse totalmente de la sintaxis necesaria para especificar cada modelo y sus reglas en el simulador.
Luego se utilizó TSC para generar todos los modelos necesarios que se utilizarían como entrada al simulador N-CD++. El resultado fue la especificación de este sector de ciudad de la forma en que se hizo en [DVW00]. El paso siguiente fue probar estos modelos utilizando el simulador y analizar el resultado obtenido. Con esto se comprobó que TSC estaba generando correctamente los modelos de simulación.
Por último se realizaron diferentes cambios en la especificación del sector de ciudad, agregando vías y semáforos. Luego de cada uno de estos cambios, se regeneraron los modelos con TSC y se realizaron las simulaciones correspondientes con el simulador analizando los resultados obtenidos. Estos resultados reflejaban el cambio realizado en forma correcta, comprobando la eficiencia de TSC en generar modelos correctos de simulación de tráfico.
En este capítulo se describe el trabajo realizado con el ejemplo de aplicación y las conclusiones a las que se arribó luego de finalizar el mismo.

5.1. El sector de ciudad

El sector de ciudad especificado en [DVW00] está compuesto por diversos elementos que incluyen calles con tramos de dos sentidos y calles con tramos de un solo sentido.
Algunos de estos tramos son de 1 carril, otros de 2 carriles y otros de 4 carriles.
Una de las calles es atravesada por una vía y, en primera instancia, ninguno de los cruces posee semáforos.
Cabe aclarar que en [DVW00] se utilizó una codificación especial para nombrar los diferentes tramos. Aquí se prefirió utilizar los verdaderos nombres de las calles, complementando los mismos con letras en el caso en que para alguna calle se especifique más de un tramo. Por ejemplo, la en el plano tenemos 3 tramos para la calle Donado. Los mismos se denominan Donado_A, Donado_B y Donado_C.
Para las calles de doble circulación, se agregó un número a los tramos correspondientes para diferenciarlos. Por ejemplo, los tramos Balbin_A1 y Balbin_A2 pertenecen a la misma cuadra de Balbín difieren en el sentido de su circulación.
A continuación se encuentra el gráfico correspondiente a este sector de ciudad:



5.2. La especificación del sector de ciudad

En primer lugar, se especificó este sector de ciudad de la forma definida en la gramática de planos de ciudad de TSC.
Se decidió en la primer especificación no incluir las vías, ya que esto permitiría luego extender los modelos agregando las mismas y comparar el código generado por TSC y los resultados de la simulación con y sin vías.
La primer especificación del sector de ciudad se detalla a continuación:

```
begin segments
  Donado_B  = (7,16),(11,25),1,straight,go,10,200,parkNone
  Donado_A  = (2,1),(7,16),1,straight,go,10,200,parkNone
  Donado_C  = (11,25),(14,34),1,straight,go,10,200,parkNone
  Balbin_A1 = (7,16),(22,16),2,straight,go,10,200,parkNone
  Balbin_A2 = (7,16),(22,16),2,straight,back,10,200,parkNone
  Paroissien  = (11,25),(22,16),1,straight,back,10,200,parkNone
  Garcia  = (14,34),(21,31),1,straight,go,10,200,parkNone
  Holmberg_A1 = (17,2),(22,16),4,straight,go,10,200,parkNone
  Holmberg_A2 = (17,2),(22,16),4,straight,back,10,200,parkNone
  Holmberg_B1 = (22,16),(24,26),2,straight,go,10,200,parkNone
  Holmberg_B2 = (22,16),(24,26),2,straight,back,10,200,parkNone
  Balbin_B1 = (22,16),(40,16),2,straight,go,10,200,parkNone
  Balbin_B2 = (22,16),(40,16),2,straight,back,10,200,parkNone
end segments

begin crossings
  c1 = (22,16),10, withoutTL, withoutHole,200, 3
  c2 = (7,16),10, withoutTL, withoutHole,200, 3
  c3 = (11,25),10, withoutTL, withoutHole,200, 3
  c4 = (14,34),10, withoutTL, withoutHole,200, 3
end crossings
```

Como puede observarse, 22 líneas fueron suficientes para especificar el plano de ciudad correspondiente a la sección de Buenos Aires a estudiar.

5.3. Generación de los modelos con TSC

Luego de la especificación de la sección de ciudad, el paso siguiente fue utilizar TSC para la generación de los modelos correspondientes que se utilizarían como entrada al simulador.
En rpimer lugar se analizó la forma en que se especificaron los modelos y las reglas en [DWV00]. Se comprobó que en este trabajo se había modificado el formato de varias de las reglas de ATLAS para adaptarlas al simulador N-CD++.
Debido a que los templates por defecto que se construyeron con TSC (y que reflejan en su totalidad ATLAS) no permitirían obtener el código generado, se decidió crear un nuevo juego de templates para poder generar los modelos de la forma en que se hizo en [DWV00].

Entre los cambios más destacables que se realizaron en los templates se encuentran:
- la unificación del conjunto de reglas para todas las celdas de un tramo del mismo carril.
- la utilización de la sentencia portIntransition y la separación del estado de los ports del bloque de reglas a ejecutar por una celda.
- el agregado de nuevos ports de comunicación entre tramos y cruces para averiguar la disponibilidad de lugar para avanzar.
- la utilización de un contador global que mide la cantidad de autos dentro de la simulación.

También se amplió el alcance del nuevo modelo extendiendo la nueva interpretación a los baches, elementos de control y semáforos. Sin embargo se respetaron algunas definiciones de ATLAS con el objetivo de generar modelos simulables más consistentes y de mejor calidad, a saber:

- Se reemplazó la utilización del modelo atómico "Generator", ubicando autos en todos los carriles al mismo tiempo, por el modelo atómico "TSCGenerator", que coloca autos por carril según una función de distribución.
- Se utilizó el modelo atómico "TSCConsumer" para retirar los autos de la simulación y contar dichos autos.
- Se incorporaron contadores a todo los cruces para medir la concentración de autos en distintos puntos del modelo.

Finalmente se utilizó TSC con este nuevo juego de templates generando en forma correcta los modelos y las conexiones entre sus ports, así como las reglas de simulación.
Esto fue una prueba muy importante para TSC ya que permitió probar la flexibilidad de la herramienta en cuanto a su adaptación al simulador subyacente. Se comprobó que sin necesidad de modificar la estructura de TSC y simplemente modificando los templates de generación se pudieron obtener los modelos deseados.

Cabe destacar que con las 22 líneas que se utilizaron para especificar el sector de ciudad y que no incluían una sola sentencia de especificación de autómatas celulares ni de modelos DEVS, se obtuvo una especificación de modelos de simulación de más de 1000 líneas (sin incluir el archivo de macros que acompaña la especificación y que también genera TSC).
El hecho de que estas 1000 líneas las tuviera que generar un analizador de tráfico (o tuviera que analizar entre ellas las sentencias a modificar en caso de querer agregar elementos a la simulación), provocaría que dicha persona tuviera que aprender la complicada sintaxis de especificación de

modelos Devs y celulares y que ocupara mucho tiempo en especificar cada modelo a simular y las relaciones entre sus ports.

Una vez generados los modelos correspondientes se corrió la simulación con N-CD++ .
Para analizar los resultados obtenidos y luego poder realizar comparaciones con el agregado de diferentes elementos a la sección de ciudad se decidió correr una simulación con un tiempo de 10 minutos, parametrizando los contadores globales y de cruces para que informaran su estado a cada minuto de la simulación.
Por otra parte se setearon los generadores de autos de la siguiente forma:
Para tramos de 1 carril: 1 auto cada 4 segundos.
Para tramos de 2 carriles: 1 auto cada 3 segundos.
Para tramos de 3 carriles: 1 auto cada 2 segundos.
Para tramos de 4 carriles: 1 auto cada 1 segundo.

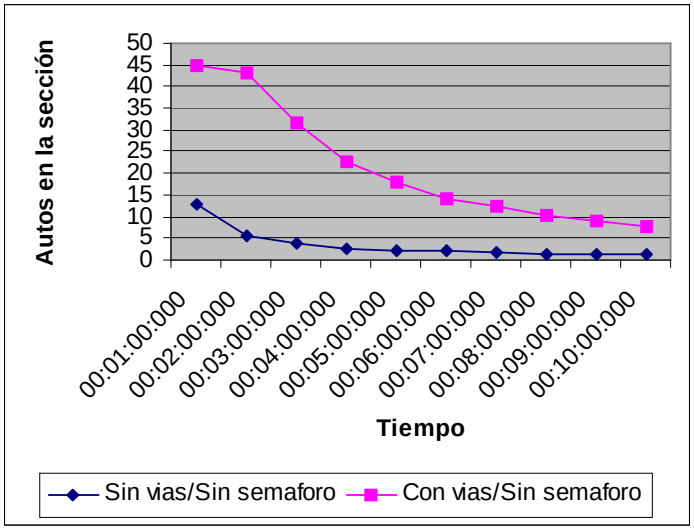
Los siguientes pasos consistieron en hacer pequeñas modificaciones en el sector de ciudad especificado para comprobar el efecto que esto produce en el comportamiento del tránsito.

5.4. Primera modificación: Agregado de vías sobre Balbín

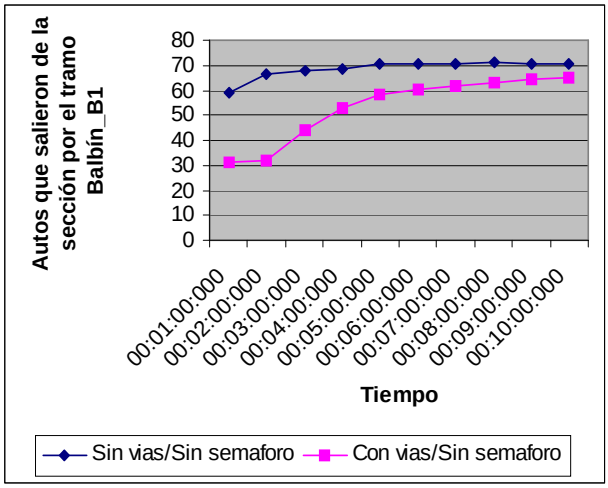
El agregado de las vías consistió en extender las especificación del sector de ciudad incorporando las siguientes líneas:

```
begin railnets
  Via_Balbin = (Balbin_B1,8),(Balbin_B2,11),200
end railnets
```

Luego de extender la especificación se utilizó TSC para regenerar los modelos de simulación y se comprobó que los mismos reflejaban correctamente el cambio.
Finalmente se utilizó N-CD++ para simular el comportamiento del tráfico con esta modificación.
Al término de la simulación se analizó el efecto del agregado de las vías que atraviesan la calle Balbin.
Para realizar este análisis fue construida una serie de gráficos basados en los datos emitidos por la simulación:



En este primer gráfico podemos observar el efecto que produjo el agregado de vías en la simulación con respecto al promedio de autos en el secto de ciudad en cada minuto.
En los primeros minutos se observa un gran embotellamiento de autos en la sección, debido seguramente al paso de los trenes por la vías agregadas.
Para confirmar esto, construimos un segundo gráfico:



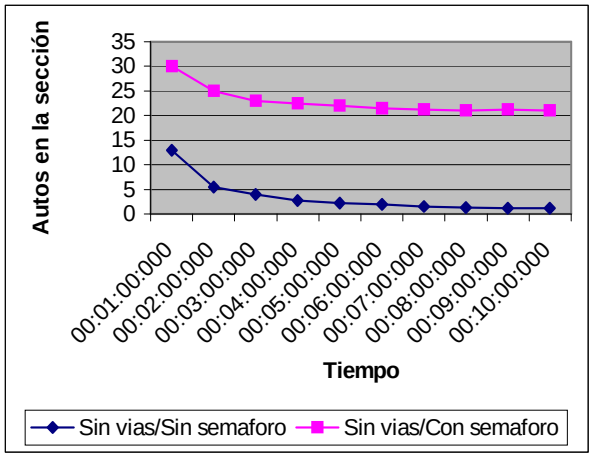
Aquí podemos observar que efectivamente la circulación de autos por este tramo se vió seriamente afectada por la presencia de las vías sobre el mismo. En los primeros minutos las vías provocaron que la mitad de los autos no salieran por el tramo de la forma en que lo hacían cuando el mismo no tenía vías. Esta situación comenzó a normalizarse hacia el final de la simulación.

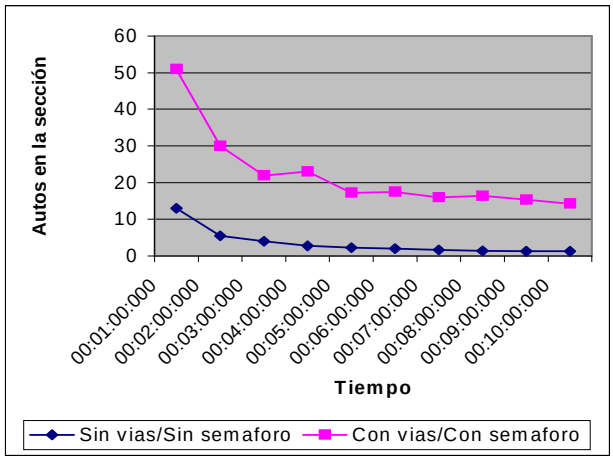
5.5. Segunda modificación: Agregado de semáforos a los cruces c1 y c2

El agregado de semáforos consistió en un cambio en la especifiación de de los cruces c1 y c2.
Este cambio está indicado en **negrita** en el siguiente cuadro:

```
  c1 = (22,16),10, withTL, withoutHole,200, 3
  c2 = (7,16),10, withTL, withoutHole,200, 3
```

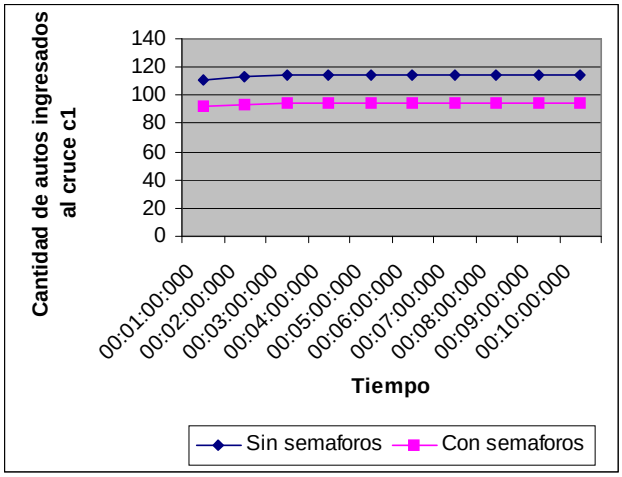
Este agregado fue hecho tanto en la especificación original como en la especificación que contenía las vías sobre la calle Balbín.
Luego del cambio se regeneraron los modelos con TSC de la misma forma en que lo habíamos hecho con el agregado de las vías.
Finalmente se corrieron las simulaciones, tanto para los modelos sin vías como para los modelos con vías, pero siempre con semáforos sobre c1 y c2.
Al finalizar las simulaciones se analizó el comportamiento del tráfico construyendo nuevos gráficos:





En estos gráficos se observa que la cantidad de autos que aún se mantenían dentro de la sección de ciudad en cada minuto se incrementó al colocar los semáforos en los cruces c1 y c2.
Esto puede interpretarse como que los semáforos, si bien tienen una gran utilidad en cuanto al ordenamiento del tráfico, hacen que el flujo de autos en la sección sea más lento y por lo tanto pueden producir más situaciones de demora.
Inclusive en el segundo de estos gráficos se ve que la combinación de vías sobre Balbín con los semáforos (uno de ellos en c1, cruce por el cual pasa la calle Balbín) produce una situación de demora aún peor, provocando menor salida de autos de la simulación.

El siguiente gráfico sobre el promedio de los autos ingresados al cruce c1 en cada minuto, también muestra resultados significativos:



Aquí se observa que la cantidad de autos ingresados al cruce disminuye al implementar semáforos en el mismo.

5.6. Conclusiones sobre el caso de aplicación

Este ejemplo de aplicación permitió hacer un testeo profundo de la herramienta TSC y su capacidad para generar modelos de simulación de tráfico en base a la especificación de un sector de ciudad.
Las diferentes pruebas de generación realizadas y su posterior testeo en el simulador permitió evaluar eficientemente los siguientes puntos de TSC:

5.6.1. Eficiencia

Se comprobó que TSC generó correctamente los modelos atómicos y celulares necesarios para cada uno de los elementos del sector de ciudad.
Lo mismo sucedió con todos los ports de comunicación entre los diferentes modelos y las reglas de comportamiento de cada modelo celular.
Los cambios realizados en la especificación del sector de ciudad se reflejaron correctamente en los nuevos modelos generados, sin necesidad de hacer cambios posteriores en los archivos de entrada al simulador.

5.6.2. Adaptabilidad

Si bien los templates originales de TSC no reflejaban en su totalidad los modelos y las reglas utilizados en [DWV00], un nuevo juego de templates con los cambios necesarios permitió que TSC generase el código deseado.
Este fue uno de los principales objetivos al realizar TSC: que se puedan generar diferentes modelos para los mismos elementos de simulación con solo uilizar otro juego de templates de generación y sin necesidad de cambiar la herramienta.

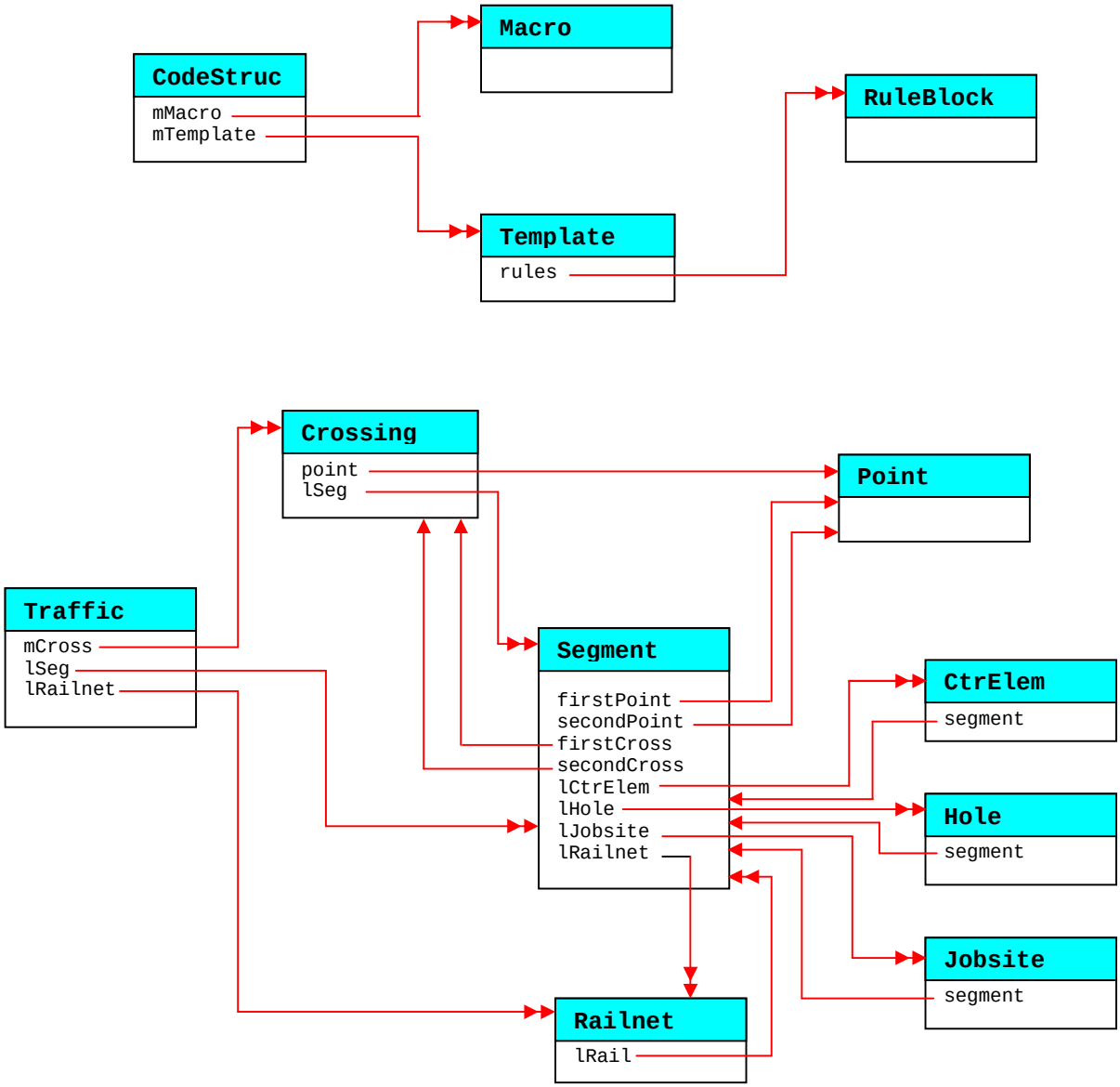
5.6.3. Abstracción

Se logró que una especificación sencilla de un sector de ciudad (basándose en el lenguaje ATLAS) generara un conjunto correcto de modelos sin necesidad de escribir en ningún momento ninguna sentencia de simulación que incluyera términos tales como cells, ports, rules, transitions, etc.
Esto comprueba el buen nivel de abstracción que pueden obtener los analizadores de tráfico al poder utilizar sus propios términos (calles, vías, semáforos, etc) cuando desean utilizar el simulador para realizar sus análisis.

Capítulo 6 - Arquitectura detallada de TSC

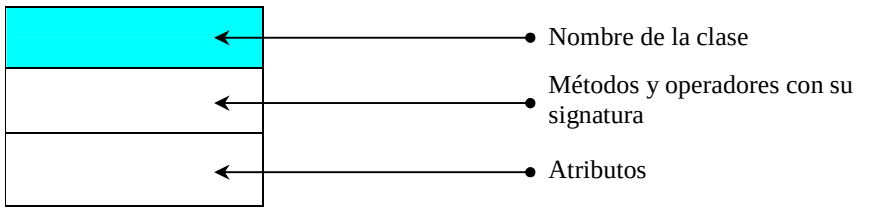
En este capítulo se describe en forma detallada cada una de las clases que componen la estructura interna de TSC, describiendo sus responsabilidades, atributos y métodos además de la jerarquía de agregación entre las mismas.

6.1. Jerarquía de Agregación entre Clases



6.2. Detalle de las clases que componen TSC

En esta sección se documentan, por orden alfabético, cada una de las clases que componen TSC.
Esta documentación se compone de un cuadro con información de la clase, las responsabilidades de la misma y un detalle de cada método, operador y atributo.
El cuadro con información de la clase tiene la siguiente estructura:



Los operadores se distinguen de los métodos por estar escritos en letra *itálica.*
Tanto los métodos como los atributos escritos en **negrita** son públicos. El resto son privados o protegidos.

6.2.1. CodeGen

```
void          generate()
void          generateCrossing(Crossing*)
void          generateLine(list<string>*,string)
void          generateLineCrossing(Crossing*,list<string>*,string)
void          generateLineCrossingCall(Crossing*,string)
void          generateLineCtrElem(CtrElem*,list<string>*,string)
void          generateLineHole(Hole*,list<string>*,string)
void          generateLineJobsite(Jobsite*,list<string>*,string)
void
generateLineRail(Segment*,Railnet*,list<string>*,string)
void          generateLineRailnet(Railnet*,list<string>*,string)
void          generateLineSegment(Segment*,list<string>*,string)
void          generateLineSegmentCall(Segment*,string)
void          generateRailnet(Railnet*)
void          generateRuleCrossingCall(Crossing*,string)
void          generateRuleSegmentCall(Segment*,string)
void          generateSegment(Segment*)
void          generateTop ()
list<string> *(Template::*call)()
void          clearDepWasUsed()
```

Responsabilidad de CodeGen

Genera el archivo de salida del compilador que contiene el modelo de simulación del plano de ciudad.
Es instanciada por el módulo principal de TSC (***tsc.cc***) y utilizada una vez que se parsearon tanto el archivo de templates (mediante el módulo ***iniparse***) como el archivo del plano de ciudad (mediante el módulo ***planparse***).

Estos parsers dejan instanciadas a ***CodeStruct*** y ***Traffic*** respectivamente. Las instancias de estas dos clases son las que utiliza la instancia de ***CodeGen*** para generar el código contenido en el archivo del modelo de simulación.

Métodos de CodeGen

`generate`	Genera el archivo de salida. Para ello envía mensajes privados para la generación de cada tipo diferente de elemento de simulación.
`generateCrossing`	Es invocado por el método **generate()** para comenzar la generación de código de los cruces. Se encarga de separar cada sección de los templates e invocar al método **generateLineCrossingCall()** por cada una de ellas. Solamente para la generación de las reglas invoca al método **generateRuleCrossingCall()**.
`generateLine`	Método final para la generación de código del modelo ***top*** en el archivo de salida.
`generateLineCrossing`	Método final para la generación de código de los cruces en el archivo de salida.
`generateLineCrossingCall`	Es invocado por el método **generateCrossing()** para continuar con la generación de código de los cruces. Se encarga de recuperar el conjunto de líneas del cruce y de los elementos adicionales para dicho cruce para una sección determinada del template. Luego invoca a los métodos finales de generación de código por elemento.
`generateLineCtrElem`	Método final para la generación de código de los elementos de control en el archivo de salida.
`generateLineHole`	Método final para la generación de código de los baches en el archivo de salida.
`generateLineJobsite`	Método final para la generación de código de las obras en el archivo de salida.
`generateLineRail`	Método final para la generación de código de las vías de un tramo en el archivo de salida.
`generateLineRailnet`	Método final para la generación de código de las redes de vías en el archivo de salida.
`generateLineSegment`	Método final para la generación de código de los tramos en el archivo de salida.
`generateLineSegmentCall`	Es invocado por el método **generateSegment()** para continuar con la generación de código de los tramos. Se encarga de recuperar el conjunto de líneas del tramo y de los elementos adicionales para dicho tramo para una sección determinada del template. Luego invoca a los métodos finales de generación de código por elemento.
`generateRailnet`	Es invocado por el método **generate()** para comenzar la generación de código de las vías. Se encarga de separar cada sección de los templates e invocar al método **generateLineRailnet()** por cada una de ellas
`generateRuleCrossingCall`	Es invocado por el método **generateCrossing()** para la generación de las reglas del cruce y las reglas de los elementos adicionales para dicho cruce.
`generateRuleSegmentCall`	Es invocado por el método **generateSegment()** para la

	generación de las reglas del tramo y las reglas de los elementos adicionales para dicho tramo.
generateSegment	Es invocado por el método **generate()** para comenzar la generación de código de los tramos. Se encarga de separar cada sección de los templates e invocar al método **generateLineSegmentCall()** por cada una de ellas. Solamente para la generación de las reglas invoca al método **generateRuleSegmentCall()**.
generateTop	Es invocado por el método **generate()** para comenzar la generación del modelo acoplado ***top***. Se encarga de separar cada sección de los templates e invocar al método **generateLine()** por cada una de ellas
(Template::*call)	Se utiliza para llamar en forma indirecta a los distintos métodos **get** de la clase Template.
clearDepWasUsed	Marca como no usadas todas las macros y todos los bloques dependientes (que contienen macrovariables).

6.2.2. CodeStruct

Macro*	**getMacro(string)**
Template*	**getTemplate(string)**
map<string,Macro>	***getMMacro()**
map<string,Template>	***getMTemplate()**
void	**addMacro(Macro*)**
void	**addTemplate(Template*)**
void	**print()**
map<string,Macro>	mMacro
map<string,Template>	mTemplate

Responsabilidad de CodeStruct

Almacena los templates de generación de código para cada uno de los elementos simulables y para cada una de las macros que se definen en el plano de ciudad utilizado como entrada a TSC.
Es instanciada por el parser que procesa el plano de ciudad (***planparse***) y colabora con la clase ***CodeGen*** para generar el archivo de entrada al simulador.

Atributos de CodeStruct

mMacro	Mapa de las Macros definidas en el plano de ciudad. Como clave del mapa se está utilizando el string que define el nombre de la macro.
mTemplate	Mapa de Templates definidos en el plano de ciudad. Como clave del mapa se utiliza el string que define el nombre del template.

Métodos de CodeStruct

getMacro	Retorna la Macro cuyo nombre se especifica como parámetro.
getTemplate	Retorna el Template cuyo nombre se especifica como parámetro.
getMMacro	Retorna el mapa de Macros.
getMTemplate	Retorna el mapa de Templates.
addMacro	Agrega la Macro que recibe como parámetro al mapa de Macros.

addTemplate	Agrega el Template que recibe como parámetro al mapa de Templates.
print	Imprime por standard output información sobre la instancia.

6.2.3. Crossing

int	**getDelay()**
string	**getHole()**
string	**getIdent()**
Point	**getPoint()**
int	**getPout()**
int	**getQcell()**
int	**getSpeed()**
string	**getTLight()**
list<Segment*>	***getLSeg()**
int	**getQcellIn()**
int	**getQcellOut()**
int	**getQSegIn()**
int	**getQSegOut()**
Segment	***getSegment(int)**
int	**getSegmentLane(int)**
void	**setDelay(int)**
void	**setHole(string)**
void	**setIdent(string)**
void	**setPoint(Point)**
void	**setPout(int)**
void	**setQcell()**
void	**setSpeed(int)**
void	**setTLight(string)**
void	**addSeg(Segment *)**
bool	**isInSeg(Segment*)**
bool	**isOutSeg(Segment*)**
void	**print()**
bool operator	***== (const Crossing &) const***
bool operator	***!= (const Crossing &) const***
bool operator	***< (const Crossing &) const***
bool operator	***<= (const Crossing &) const***
bool operator	***> (const Crossing &) const***
bool operator	***>= (const Crossing &) const***
int	delay
string	hole
string	ident
Point	point
int	pout
int	qcell
int	speed
string	tLight
list<Segment*>	lSeg

Responsabilidad de Crossing

Modela un cruce.

El parser del plano de ciudad crea una instancia de esta clase por cada cruce que se encuentra en el plano.

Atributos de Crossing

`delay`	Demora de circulación en el cruce en caso de existir baches.
`hole`	Indica si el cruce tiene o no baches. Su valor es "withHole" cuando tiene baches y "withoutHole" en caso contrario.
`ident`	Identificador del cruce.
`point`	Punto en el que se encuentra el cruce.
`pout`	Probabilidad de salir del cruce (1/***pout***).
`qcell`	Cantidad de celdas del cruce.
`speed`	Velocidad máxima de circulación en el cruce.
`tLight`	Indica si el cruce tiene o no semáforo. Su valor es "withTL" cuando tiene semáforo y "withoutTL" en caso contrario.
`lSeg`	Lista de los tramos que llegan o salen del cruce ordenados por inclinación.

Métodos de Crossing

`getDelay`	Retorna el atributo ***delay***.
`getHole`	Retorna el atributo ***hole***.
`getIdent`	Retorna el atributo ***ident***.
`getPoint`	Retorna el atributo ***point***.
`getPout`	Retorna el atributo ***pout***.
`getQcell`	Retorna el atributo ***qcell***.
`getSpeed`	Retorna el atributo ***speed***.
`getTLight`	Retorna el atributo ***tLight***.
`getLSeg`	Retorna el atributo ***lSeg***.
`getQcellIn`	Retorna la cantidad de celdas asociadas a tramos de entrada al cruce.
`getQcellOut`	Retorna la cantidad de celdas asociadas a tramos de salida del cruce.
`getQSegIn`	Retorna la cantidad de tramos de entrada al cruce.
`getQSegOut`	Retorna la cantidad de tramos de salida del cruce.
`getSegment`	Retorna el tramo al que pertenece el carril que se asocia con la celda ***n*** del cruce, donde ***n*** es el entero especificado como parámetro.
`getSegmentLane`	Retorna el número de carril dentro del tramo correspondiente que se asocia con la celda ***n*** del cruce, donde ***n*** es el entero especificado como parámetro.
`setDelay`	Instancia el atributo ***delay*** con el parámetro especificado.
`setHole`	Instancia el atributo ***hole*** con el parámetro especificado.
`setIdent`	Instancia el atributo ***ident*** con el parámetro especificado.
`setPoint`	Instancia el atributo ***point*** con el parámetro especificado.
`setPout`	Instancia el atributo ***pout*** con el parámetro especificado.
`setQcell`	Instancia el atributo ***qcell*** sumando la cantidad de carriles que tiene cada uno de los tramos que entran o salen del cruce.
`setSpeed`	Instancia el atributo ***speed*** con el parámetro especificado.
`setTLight`	Instancia el atributo ***tlight*** con el parámetro especificado.
`addSeg`	Agrega el tramo especificado como parámetro a la lista de tramos del cruce respetando el orden de inclinación.
`isInSeg`	Retorna verdadero si el tramo especificado como parámetro es de entrada al cruce.
`isOutSeg`	Retorna verdadero si el tramo especificado como parámetro es de salida del cruce.

`print` Imprime por standard output información sobre la instancia.

Operadores de Crossing

`==`	Un cruce ***a*** es igual a un cruce ***b***, si el punto en el que se encuentra ***a*** es igual al punto en el que se encuentra ***b***.
`!=`	Un cruce ***a*** es distinto de un cruce ***b***, si el punto en el que se encuentra ***a*** es distinto al punto en el que se encuentra ***b***.
`<`	Un cruce ***a*** es menor que un cruce ***b***, si el punto en el que se encuentra ***a*** es menor al punto en el que se encuentra ***b***.
`<=`	Un cruce ***a*** es menor o igual que un cruce ***b***, si el punto en el que se encuentra ***a*** es menor o igual al punto en el que se encuentra ***b***.
`>`	Un cruce ***a*** es mayor que un cruce ***b***, si el punto en el que se encuentra ***a*** es mayor al punto en el que se encuentra ***b***.
`>=`	Un cruce ***a*** es mayor o igual que un cruce ***b***, si el punto en el que se encuentra ***a*** es mayor o igual al punto en el que se encuentra ***b***.

6.2.4. CtrElem

```
int       getDelay()
int       getDistance()
Segment  *getSegment()
string    getType()
void      setDelay(int)
void      setDistance(int)
void      setSegment(Segment*)
void      setType(string)
void      print()
```

```
int       delay
int       distance
Segment  *segment
string    type
```

Responsabilidad de CtrElem

Modela un elemento de control.
El parser del plano de ciudad crea una instancia de esta clase por cada elemento de control que se encuentra en el plano.

Atributos de CtrElem

`delay`	Demora que produce el elemento de control en la circulación de los autos.
`distance`	Distancia del elemento de control al comienzo del tramo.
`segment`	Tramo en el que se encuentra el elemento de control
`type`	Tipo de elemento de control. Los valores posibles son "sawhorse", "depression", "intersection", "saw", "stop", "school".

Métodos de CtrElem

`getDelay` Retorna el atributo ***delay***.

getDistance	Retorna el atributo ***distance***.
getSegment	Retorna el atributo ***segment***.
getType	Retorna el atributo ***type***.
setDelay	Instancia el atributo ***delay*** con el parámetro especificado.
setDistance	Instancia el atributo ***distance*** con el parámetro especificado.
setSegment	Instancia el atributo ***segment*** con el parámetro especificado.
setType	Instancia el atributo ***type*** con el parámetro especificado.
print	Imprime por standard output información sobre la instancia.

6.2.5. FileAdmin

void	**open()**
void	**close()**
void	**setFileName(string)**
void	**setMacroFileName(string)**
void	**write(string,string)**
fstream	*file
string	fileName
string	lines
string	linesTopC
string	linesTopL
string	linesTopP
fstream	*macroFile
string	macroFileName
string	rules
string	macros
streampos	top

Responsabilidad de FileAdmin

Es la única clase que trata con los archivos generados como salida, tanto el de simulación como el de macros.
Almacena en sus atributos todo el código que se escribirá en estos archivos y realiza el trabajo de apertura, escritura y cierre de los archivos físicos.
Es usada cuando la instancia de ***CodeGen*** recorre la instancia de ***Traffic***, para almacenar el código generado para cada uno de los elementos simulables.

Atributos de FileAdmin

file	Archivo de simulación.
fileName	Nombre del archivo de simulación.
lines	Código correspondiente a todos los modelos.
linesTopC	Código correspondiente a la parte "components" del modelo ***top***.
linesTopL	Código correspondiente a la parte "links" del modelo ***top***.
linesTopP	Código correspondiente a la parte "ports" del modelo ***top***.
macroFile	Archivo de macros.
macroFileName	Nombre del archivo de macros.
rules	Código correspondiente a las reglas.
macros	Código correspondiente a las macros.

Métodos de FileAdmin

open	Abre los archivos de simulación y de macros para su generación.
close	Escribe en los archivos de simulación y de macros los atributos que contienen el código que se generó. Luego cierra ambos archivos.
setFileName	Instancia el atributo ***fileName*** con el parámetro especificado.
setMacroFileName	Instancia el atributo ***macroFileName*** con el parámetro especificado.
write	Concatena el string recibido en el primer parámetro con uno de los atributos de la clase. Este atributo depende de lo que contenga el string especificado como segundo parámetro de la siguiente forma: Si el segundo parámetro contiene "body", el primer parámetro se concatena con el atributo ***lines***. Si el segundo parámetro contiene "rule", el primer parámetro se concatena con el atributo ***rules***. Si el segundo parámetro contiene "macro", el primer parámetro se concatena con el atributo ***macros***. Si el segundo parámetro contiene "topC", el primer parámetro se concatena con el atributo ***linesTopC***. Si el segundo parámetro contiene "topL", el primer parámetro se concatena con el atributo ***linesTopL***.

6.2.6. Hole

int	**getDelay()**
int	**getDistance()**
int	**getLane()**
Segment	***getSegment()**
string	**getCell()**
void	**setDelay(int)**
void	**setDistance(int)**
void	**setLane(int)**
void	**setSegment(Segment*)**
void	**print()**
int	delay
int	distance
int	lane
Segment	*segment

Responsabilidad de Hole

Modela un bache.
El parser del plano de ciudad crea una instancia de esta clase por cada bache que se encuentra en el plano.

Atributos de Hole

delay	Demora que el bache produce en la circulación de los autos.
distance	Distancia del bache al comienzo del tramo.
lane	Carril donde se encuentra el bache.
segment	Tramo en el que se encuentra el bache

Métodos de Hole

getDelay	Retorna el atributo ***delay***.
getDistance	Retorna el atributo ***distance***.
getLane	Retorna el atributo ***lane***.
getSegment	Retorna el atributo ***segment***.
getCell	Retorna un string conteniendo la celda en la que se encuentra el bache con el formato "(***l***,***d***)", donde ***l*** es el carril y ***d*** la distancia al comienzo del tramo.
setDelay	Instancia el atributo ***delay*** con el parámetro especificado.
setDistance	Instancia el atributo ***distance*** con el parámetro especificado.
setLane	Instancia el atributo ***lane*** con el parámetro especificado.
setSegment	Instancia el atributo ***segment*** con el parámetro especificado.
print	Imprime por standard output información sobre la instancia.

6.2.7. Jobsite

int	**getDelay()**
int	**getDistance()**
int	**getLane()**
int	**getQlane()**
Segment	***getSegment()**
string	**getCells()**
void	**setDelay(int)**
void	**setDistance(int)**
void	**setLane(int)**
void	**setQlane(int)**
void	**setSegment(Segment*)**
void	**print()**
int	delay
int	distance
int	lane
int	qlane
Segment	*segment

Responsabilidad de Jobsite

Modela una obra.
El parser del plano de ciudad crea una instancia de esta clase por cada obra que se encuentra en el plano.

Atributos de Jobsite

delay	No utilizado. Reservado para uso futuro.
distance	Distancia de la obra al comienzo del tramo.
lane	Primer carril afectado por la obra.
qlane	Cantidad de carriles que ocupa la obra.
segment	Tramo en el que se encuentra la obra.

Métodos de Jobsite

getDelay	Retorna el atributo ***delay***.
getDistance	Retorna el atributo ***distance***.

getLane	Retorna el atributo ***lane***.
getQlane	Retorna el atributo ***qLane***.
getSegment	Retorna el atributo ***segment***.
getCells	Retorna un string conteniendo todas las celdas afectadas por la obra. Las celdas están separadas por comas y cada una tiene el formato "(***l***,***d***)", donde ***l*** es el carril y ***d*** la distancia al comienzo del tramo.
setDelay	Instancia el atributo ***delay*** con el parámetro especificado.
setDistance	Instancia el atributo ***distance*** con el parámetro especificado.
setLane	Instancia el atributo ***lane*** con el parámetro especificado.
setQlane	Instancia el atributo ***qLane*** con el parámetro especificado.
setSegment	Instancia el atributo ***segment*** con el parámetro especificado.
print	Imprime por standard output información sobre la instancia.

6.2.8. LineConvertor

```
string    convertCrossing(string,Crossing*)
string    convertCtrElem(string,CtrElem*)
string    convertForLine(string,string,int)
string    convertHole(string,Hole*)
string    convertJobsite(string,Jobsite*)
string    convertLine(string,string,string)
string    convertRail(string,Segment*,Railnet*)
string    convertRailnet(string,Railnet*)
string    convertSegment(string,Segment*)
string    getVariable(string)
```

Responsabilidad de LineConvertor

Se encarga de convertir las líneas de los templates de generación, reemplazando las macrovariables que contienen las mismas con los valores correspondientes a cada elemento de simulación.
Ante cada mensaje que se le envía se encarga de convertir el string que recibe como primer parámetro resolviendo las macrovariables en base a los elementos de simulación recibidos en el resto de los parámetros.

Métodos de LineConvertor

convertCrossing	Resuelve las macrovariables de la linea especificada en el primer parámetro en base al cruce recibido como segundo parámetro.
convertCtrElem	Resuelve las macrovariables de la linea especificada en el primer parámetro en base al elemento de control recibido como segundo parámetro.
convertForLine	Concatena ***i*** repeticiones del string especificado como primer parámetro, reemplazando el string especificado como segundo parámetro por el número de iteración correspondiente, considerando la primera iteración como la número 0 y la última como la ***i***-1; donde ***i*** es el entero especificado como tercer parámetro.
convertHole	Resuelve las macrovariables de la linea especificada en el primer parámetro en base al bache recibido como segundo parámetro.
convertJobsite	Resuelve las macrovariables de la linea especificada en el primer parámetro en base a la obra recibida como segundo parámetro.
convertLine	Reemplaza todas las ocurrencias del string especificado como segundo

	parámetro por el string especificado como tercer parámetro, en el string especificado como primer parámetro.
convertRail	Resuelve las macrovariables de la linea especificada en el primer parámetro en base al tramo recibido como segundo parámetro y a la red de vías recibida como tercer parámetro.
convertRailnet	Resuelve las macrovariables de la linea especificada en el primer parámetro en base a la red de vías recibida como segundo parámetro.
convertSegment	Resuelve las macrovariables de la linea especificada en el primer parámetro en base al tramo recibido como segundo parámetro.
getVariable	Retorna la primer macrovariable encontrada en el string especificado como parámetro. Si no hay macrovariables en el string retorna la cadena vacía.

6.2.9. Macro

string	**getName()**
bool	**getWasUsed()**
list<string>	***getLines()**
bool	**getIsDependent()**
void	**setName(string)**
void	**setWasUsed(bool)**
void	**addLine(string c)**
void	**print()**
string	name
bool	wasUsed
list<string>	lines

Responsabilidad de Macro

Modela las macros contenidas en los archivos de template.

Atributos de Macro

name	Nombre de la macro.
wasUsed	Verdadero si la macro ya fue utilizada durante la generación de código, falso en caso contrario.
lines	Líneas que componen la macro.

Métodos de Macro

getName	Retorna el atributo ***name***.
getWasUsed	Retorna el atributo ***wasUsed***.
getLines	Retorna el atributo ***lines***.
getIsDependent	Retorna verdadero si la macro contiene macrovariables, falso en caso contrario.
setName	Instancia el atributo ***name*** con el parámetro especificado.
setWasUsed	Instancia el atributo ***wasUsed*** con el parámetro especificado.
addLine	Agrega una linea en el atributo ***lines***.
print	Imprime por standard output información sobre la instancia.

6.2.10. Point

int	**getx()**
int	**gety()**
void	**setx(int)**
void	**sety(int)**
void	**print()**
bool operator	***== (const Point &) const***
bool operator	***!= (const Point &) const***
bool operator	***< (const Point &) const***
bool operator	***<= (const Point &) const***
bool operator	***> (const Point &) const***
bool operator	***>= (const Point &) const***
int	x
int	y

Responsabilidad de Point

Modela un punto en el plano.

Atributos de Point

x Coordenada x del punto.
y Coordenada y del punto.

Métodos de Point

getx Retorna el atributo ***x***.
gety Retorna el atributo ***y***.
setx Instancia el atributo ***x*** con el parámetro especificado.
sety Instancia el atributo ***y*** con el parámetro especificado.
print Imprime por standard output información sobre la instancia.

Operadores de Point

== ***a*** = ***b*** si $a_x = b_x$ y $a_y = b_y$, donde a_x y a_y son las coordenadas ***x*** e ***y*** del punto ***a*** respectivamente; y b_x y b_y son las coordenadas ***x*** e ***y*** del punto ***b*** respectivamente.
!= ***a*** != ***b*** si a_x != b_x o a_y != b_y, donde a_x y a_y son las coordenadas ***x*** e ***y*** del punto ***a*** respectivamente; y b_x y b_y son las coordenadas ***x*** e ***y*** del punto ***b*** respectivamente.
< ***a*** < ***b*** si $a_y < b_y$ o $a_y = b_y$ y $a_x < b_x$,donde a_x y a_y son las coordenadas ***x*** e ***y*** del punto ***a*** respectivamente; y b_x y b_y son las coordenadas ***x*** e ***y*** del punto ***b*** respectivamente.
<= ***a*** <= ***b*** si ***a*** < ***b*** o ***a*** = ***b***.
> ***a*** > ***b*** si $a_y > b_y$ o $a_y = b_y$ y $a_x > b_x$,donde a_x y a_y son las coordenadas ***x*** e ***y*** del punto ***a*** respectivamente; y b_x y b_y son las coordenadas ***x*** e ***y*** del punto ***b*** respectivamente.
>= ***a*** >= ***b*** si ***a*** > ***b*** o ***a*** = ***b***.

6.2.11. Railnet

int	**getDelay()**
string	**getIdent()**
list<pair<Segment*, int>>	***getLRail()**

int	**getQsegs()**
void	**setDelay(int)**
void	**setIdent(string)**
void	**addRail(Segment*,int)**
int	**findDistance(string)**
int	**findRailOrder(string)**
void	**print()**
int	delay
string	ident
list<pair<Segment*, int>>	lRail

Responsabilidad de Railnet

Modela una red de vías.

Atributos de Railnet

delay Demora que la vía produce en la circulación de los autos.
ident Identificador de la red de vías
lRail Lista de pares ordenados modelando los tramos por los que pasa la red de vías. El primer elemento del par es el identificador del tramo y el segundo la distancia de la vía al comienzo del tramo.

Métodos de Railnet

getDelay	Retorna el atributo ***delay***.
getIdent	Retorna el atributo ***ident***.
getLRail	Retorna el atributo ***lRail***.
getQsegs	Retorna la cantidad de tramos por los que pasa la red de vías.
setDelay	Instancia el atributo ***delay*** con el parámetro especificado.
setIdent	Instancia el atributo ***ident*** con el parámetro especificado.
addRail	Agrega un par tramo-distancia al atributo ***lRail***. El tramo es especificado como primer parámetro y la distancia como segundo parámetro.
findDistance	Retorna la distancia de la vía al comienzo del tramo cuyo identificador se recibe como parámetro. Si la vía no pasa por ese tramo retorna 0.
findRailOrder	Retorna el número de orden en la red de vías del tramo cuyo identificador se recibe como parámetro. Si la vía no pasa por ese tramo retorna 0.
print	Imprime por standard output información sobre la instancia.

6.2.12. RuleBlock

list<string> *	**getRules()**
bool	**getWasUsed()**
bool	**getIsDependent()**
void	**setWasUsed(bool)**
void	**addRule(string c)**
void	**print()**
list<string>	rules
bool	wasUsed

Responsabilidad de RuleBlock

Modela los bloques de reglas de los modelos de simulación.

Atributos de RuleBlock

`rules`	Lista de reglas que componen este bloque.
`wasUsed`	Verdadero si el bloque de reglas ya fue utilizado durante la generación de código, falso en caso contrario.

Métodos de RuleBlock

`getRules`	Retorna el atributo ***rules***.
`getWasUsed`	Retorna el atributo ***wasUsed***.
`getIsDependent`	Retorna verdadero si el bloque de reglas contiene macrovariables, falso en caso contrario.
`setWasUsed`	Instancia el atributo ***wasUsed*** con el parámetro especificado.
`addRule`	Agrega una regla al atributo ***rules***.
`print`	Imprime por standard output información sobre la instancia.

6.2.13. Segment

```
int              getDelay()
string           getDirec()
Crossing        *getFirstCross()
double           getFirstIncl()
Point            getFirstPoint()
string           getForm()
string           getIdent()
string           getParkType()
int              getQcell()
int              getQlane()
Crossing        *getSecondCross()
double           getSecondIncl()
Point            getSecondPoint()
int              getSpeed()
list<CtrElem>   *getLCtrElem()
list<Hole>      *getLHole()
list<Jobsite>   *getLJobsite()
list<Railnet*>  *getLRailnet()
Crossing        *getEndCross()
double           getEndIncl()
Point            getEndPoint()
Point            getMajorPoint()
double           getMinorIncl()
Point            getMinorPoint()
Crossing        *getStartCross()
double           getStartIncl()
Point            getStartPoint()
string           getTLight()
void             setDelay(int)
```

void	**setDirec(string)**
void	**setFirstCross(Crossing*)**
void	**setFirstPoint(Point)**
void	**setForm(string)**
void	**setIdent(string)**
void	**setParkType(string)**
void	**setQcell()**
void	**setQlane(int)**
void	**setSecondCross(Crossing*)**
void	**setSecondPoint(Point)**
void	**setSpeed(int)**
void	**addCtrElem(CtrElem*)**
void	**addHole(Hole*)**
void	**addJobsite(Jobsite*)**
void	**addRailnet(Railnet*)**
void	**calcIncls()**
Railnet	***findRailnet(string)**
void	**print()**
int	delay
string	direc
Crossing	*firstCross
double	firstIncl
Point	firstPoint
string	form
string	ident
string	parkType
int	qcell
int	qlane
Crossing	*secondCross
double	secondIncl
Point	secondPoint
int	speed
list<CtrElem>	lCtrElem
list<Hole>	lHole
list<Jobsite>	lJobsite
list<Railnet*>	lRailnet

Responsabilidad de Segment

Modela los tramos.
El parser del plano de ciudad crea una instancia de esta clase por cada tramo que se encuentra en el plano.

Atributos de Segment

delay	Demora de la circulación de autos por el tramo.
direc	Dirección del tramo. Los valores posibles son "go" y "back".
firstCross	Cruce donde comienza el tramo.
firstIncl	Inclinación del tramo en el punto donde comienza el mismo.
firstPoint	Punto del plano donde comienza el tramo.
form	Formato del tramo. Los valores posibles son "straight" y "curve".

ident	Identificador del tramo.
parkType	Define sobre qué lado del tramo pueden estacionarse los autos. Los valores posibles son "parkLeft", "parkRight" y "parkBoth".
qcell	Cantidad de celdas del tramo.
qlane	Cantidad de carriles del tramo.
secondCross	Cruce donde termina el tramo.
secondIncl	Inclinación del tramo en el punto donde termina el mismo.
secondPoint	Punto del plano donde termina el tramo.
speed	Velocidad máxima de circulación de autos en este tramo.
lCtrElem	Lista de elementos de control que posee el tramo.
lHole	Lista de baches que posee el tramo.
lJobsite	Lista de obras que posee el tramo.
lRailnet	Lista de vías que pasan por el tramo.

Métodos de Segment

getDelay	Retorna el atributo ***delay***.
getDirec	Retorna el atributo ***direc***.
getFirstCross	Retorna el atributo ***firstCross***.
getFirstIncl	Retorna el atributo ***firstIncl***.
getFirstPoint	Retorna el atributo ***firstPoint***.
getForm	Retorna el atributo ***form***.
getIdent	Retorna el atributo ***ident***.
getParkType	Retorna el atributo ***parkType***.
getQcell	Retorna el atributo ***qCell***.
getQlane	Retorna el atributo ***qLane***.
getSecondCross	Retorna el atributo ***secondCross***.
getSecondIncl	Retorna el atributo ***secondIncl***.
getSecondPoint	Retorna el atributo ***secondPoint***.
getSpeed	Retorna el atributo ***speed***.
getLCtrElem	Retorna el atributo ***lCtrElem***.
getLHole	Retorna el atributo ***lHole***.
getLJobsite	Retorna el atributo ***lJobsite***.
getLRailnet	Retorna el atributo ***lRailnet***.
getEndCross	Retorna el cruce por el que salen los autos del tramo.
getEndIncl	Retorna la inclinación del tramo en el punto por donde salen los autos.
getEndPoint	Retorna el punto del tramo por donde salen los autos.
getMajorPoint	Retorna el mayor entre el punto de comienzo y de final del tramo.
getMinorIncl	Retorna la menor entre la inclinación del punto de comienzo y la inclinación del punto de final del tramo.
getMinorPoint	Retorna el menor entre el punto de comienzo y de final del tramo.
getStartCross	Retorna el cruce por el que ingresan los autos al tramo.
getStartIncl	Retorna la inclinación del tramo en el punto por donde ingresan los autos.
getStartPoint	Retorna el punto del tramo por donde ingresan los autos.
getTLight	Retorna "withTL" si el tramo no es de salida y el cruce al que llega posee semáforo. Retorna "withoutTL" en caso contrario.
setDelay	Instancia el atributo ***delay*** con el parámetro especificado.
setDirec	Instancia el atributo ***direc*** con el parámetro especificado.
setFirstCross	Instancia el atributo ***firstCross*** con el parámetro especificado.
setFirstPoint	Instancia el atributo ***firstPoint*** con el parámetro especificado.
setForm	Instancia el atributo ***form*** con el parámetro especificado.

setIdent	Instancia el atributo ***ident*** con el parámetro especificado.
setParkType	Instancia el atributo ***parkType*** con el parámetro especificado. Si se especifica "parkBoth" en tramos de menos de cuatro carriles o cualquier tipo de estacionamiento en tramos de menos de dos carriles, retorna error.
setQcell	Instancia el atributo ***qCell*** calculando la cantidad de celdas de acuerdo a las fórmulas especificadas en ATLAS.
setQlane	Instancia el atributo ***qLane*** con el parámetro especificado. Si se especificó 0, retorna error.
setSecondCross	Instancia el atributo ***secondCross*** con el parámetro especificado.
setSecondPoint	Instancia el atributo ***secondPoint*** con el parámetro especificado.
setSpeed	Instancia el atributo ***speed*** con el parámetro especificado.
addCtrElem	Agrega un elemento de control al atributo ***lCtrElem***.
addHole	Agrega un bache al atributo ***lHole***.
addJobsite	Agrega una obra al atributo ***lJobsite***.
addRailnet	Agrega una vía al atributo ***lRailnet***.
calcIncls	Instancia los atributos ***firstIncl*** y ***secondIncl*** calculando las inclinaciones de acuerdo a las fórmulas especificadas en ATLAS.
findRailnet	Retorna la red de vías cuyo identificador se recibe como parámetro. Si esa red de vías no pasa por el tramo, retorna NULL.
print	Imprime por standard output información sobre la instancia.

6.2.14. Template

```
string            getName()
bool              getWasUsed()
list<string>     *getAfterRules()
list<string>     *getBeforeLinks()
list<string>     *getBeforeNeighbors()
list<string>     *getBeforePorts()
list<string>     *getBeforeRules()
list<string>     *getBeforeZones()
list<string>     *getLinks()
list<string>     *getNeighbors()
list<string>     *getPorts()
list<RuleBlock> *getRules()
list<string>     *getTopComponents()
list<string>     *getTopLinks()
list<string>     *getTopPorts()
list<string>     *getZones()
void              setName(string)
void              setWasUsed(bool)
void              addAfterRules(string)
void              addBeforeLinks(string c)
void              addBeforeNeighbors(string c)
void              addBeforePorts(string c)
void              addBeforeRules(string)
void              addBeforeZones(string)
void              addLinks(string)
void              addNeighbors(string c)
void              addPorts(string c)
void              addRules(RuleBlock*)
```

void	**addTopComponents(string c)**
void	**addTopLinks(string c)**
void	**addTopPorts(string c)**
void	**addZones(string)**
void	**print()**
string	name
bool	wasUsed
list<string>	afterRules
list<string>	beforeLinks
list<string>	beforeNeighbors
list<string>	beforePorts
list<string>	beforeRules
list<string>	beforeZones
list<string>	links
list<string>	neighbors
list<string>	ports
list<RuleBlock>	rules
list<string>	topComponents
list<string>	topLinks
list<string>	topPorts
list<string>	zones

Responsabilidad de Template

Modela los templates de generación que contienen el código que se generará para cada uno de los elementos de simulación.

Atributos de Template

name	Nombre del template. Identifica para qué elemento de simulación es el código que genera el mismo.
wasUsed	Indica si el template ya fue utilizado durante la generación de código.
afterRules	Código a incluir después de la definición de reglas.
beforeLinks	Código a incluir antes de la definición de links.
beforeNeighbors	Código a incluir antes de la definición de neighbors.
beforePorts	Código a incluir antes de la definición de ports.
beforeRules	Código a incluir antes de la definición de reglas.
beforeZones	Código a incluir antes de la definición de zones.
links	Código para la definición de links.
neighbors	Código para la definición de neighbors.
ports	Código para la definición de ports.
rules	Código para la definición de rules.
topComponents	Código a incluir en la definición de componentes del modelo ***top***.
topLinks	Código a incluir en la definición de links del modelo ***top***.
topPorts	Código a incluir en la definición de ports del modelo ***top***.
zones	Código para la definición de zones.

Métodos de Template

getName	Retorna el atributo ***name***.
getWasUsed	Retorna el atributo ***wasUsed***.

getAfterRules	Retorna el atributo ***afterRules***.
getBeforeLinks	Retorna el atributo ***beforeLinks***.
getBeforeNeighbors	Retorna el atributo ***beforeNeighbors***.
getBeforePorts	Retorna el atributo ***beforePorts***.
getBeforeRules	Retorna el atributo ***beforeRules***.
getBeforeZones	Retorna el atributo ***beforeZones***.
getLinks	Retorna el atributo ***links***.
getNeighbors	Retorna el atributo ***neighbors***.
getPorts	Retorna el atributo ***ports***.
getRules	Retorna el atributo ***rules***.
getTopComponents	Retorna el atributo ***topComponents***.
getTopLinks	Retorna el atributo ***topLinks***.
getTopPorts	Retorna el atributo ***topPorts***.
getZones	Retorna el atributo ***zones***.
setName	Instancia el atributo ***Name*** con el parámetro especificado.
setWasUsed	Instancia el atributo ***wasUsed*** con el parámetro especificado.
addAfterRules	Agrega una línea en el atributo ***afterRules***.
addBeforeLinks	Agrega una línea en el atributo ***beforeLinks***.
addBeforeNeighbors	Agrega una línea en el atributo ***beforeNeighbors***.
addBeforePorts	Agrega una línea en el atributo ***beforePorts***.
addBeforeRules	Agrega una línea en el atributo ***beforeRules***.
addBeforeZones	Agrega una línea en el atributo ***beforeZones***.
addLinks	Agrega una línea en el atributo ***links***.
addNeighbors	Agrega una línea en el atributo ***neighbors***.
addPorts	Agrega una línea en el atributo ***ports***.
addRules	Agrega una línea en el atributo ***rules***.
addTopComponents	Agrega una línea en el atributo ***topComponents***.
addTopLinks	Agrega una línea en el atributo ***topLinks***.
addTopPorts	Agrega una línea en el atributo ***topPorts***.
addZones	Agrega una línea en el atributo ***zones***.
print	Imprime por standard output información sobre la instancia.

6.2.15. TemplateName

string	**getTNCrossing()**
string	**getTNCrossingHole(Crossing*)**
string	**getTNCrossingTL()**
string	**getTNCtrElem(CtrElem*)**
string	**getTNHole(Hole*)**
string	**getTNJobsite(Jobsite*)**
string	**getTNRail(Segment*)**
string	**getTNSegment(Segment*)**
string	**getTNSegmentEnd(Segment*)**
string	**getTNSegmentStart(Segment*)**
string	**getTNSegmentTL(Segment*)**
string	**getTNSincroRailnet()**
string	**getTNTop()**

Responsabilidad de TemplateName

Arma el nombre de template que se corresponde con cada uno de los elementos de simulación que componen el plano de ciudad.
Es utilizada en el momento en que se está generando el código de cada elemento de simulación para determinar el template a aplicar para la generación de ese elemento.

Métodos de TemplateName

`getTNCrossing`	Retorna el nombre del template para un cruce.
`getTNCrossingHole`	Retorna el nombre del template para un cruce con o sin baches.
`getTNCrossingTL`	Retorna el nombre del template para un cruce con semáforo
`getTNCtrElem`	Retorna el nombre del template para un elemento de control.
`getTNHole`	Retorna el nombre del template para un bache.
`getTNJobsite`	Retorna el nombre del template para una obra.
`getTNRail`	Retorna el nombre del template para una red de vías.
`getTNSegment`	Retorna el nombre del template para un tramo.
`getTNSegmentEnd`	Retorna el nombre del template para un tramo de entrada.
`getTNSegmentStart`	Retorna el nombre del template para un tramo de salida.
`getTNSegmentTL`	Retorna el nombre del template para un tramo con semáforo.
`getTNSincroRailnet`	Retorna el nombre del template para un sincronizador de vías.
`getTNTop`	Retorna el nombre del template para ***top***.

6.2.16. TException

`string`	**`getMsg()`**
`string`	**`getType()`**
`void`	**`setMsg(string)`**
`void`	**`setType(string)`**
`void`	**`print()`**
`TException(string,string)`	
`string`	`message`
`string`	`type`

Responsabilidad de TException

Maneja las excepciones producidas por el compilador.

Atributos de TException

`message`	Mensaje de excepción.
`type`	Tipo de excepción.

Métodos de TException

`getMsg`	Retorna el atributo ***message***.
`getType`	Retorna el atributo ***type***.
`setMsg`	Instancia el atributo ***message*** con el parámetro especificado.
`setType`	Instancia el atributo ***type*** con el parámetro especificado.
`print`	Imprime por standard output el tipo y el mensaje de error.
`TException`	Contructor.

6.2.17. Traffic

list<Railnet*>	***getLRailnet()**
list<Segment*>	***getLSeg()**
map<Point, Crossing*>	***getMCross()**
void	**addRailnet(Railnet*)**
void	**addSeg(Segment*)**
void	**addCross(Crossing*)**
Segment	***findSeg(string)**
void	**validate()**
void	**print()**
void	printCrossings()
void	printRailnet()
void	printSegments()
list<Railnet*>	lRailnet
list<Segment*>	lSeg
map<Point,Crossing*>	mCross

Responsabilidad de Traffic

Modela el plano de ciudad que se simulará. Contiene todos los elementos que componen la simulación (calles, cruces, semáforos, baches, obras, etc.).

Atributos de Traffic

lRailnet	Lista de redes de vías del plano.
lSeg	Lista de tramos del plano.
mCross	Lista de cruces del plano.

Métodos de Traffic

getLRailnet	Retorna el atributo ***railnet***.
getLSeg	Retorna el atributo ***lSeg***.
getMCross	Retorna el atributo ***mCross***.
addRailnet	Agrega una red de vías al atributo ***lRailnet***.
addSeg	Agrega un tramo al atributo ***lSeg***.
addCross	Agrega un cruce al atributo ***lCross***.
findSeg	Retorna el tramo cuyo identificador se especifica como parámetro.
validate	Valida el plano.
print	Imprime por standard output información sobre el plano.
printCrossings	Imprime por standard output información sobre los cruces del plano.
printRailnet	Imprime por standard output información sobre las redes de vías del plano.
printSegments	Imprime por standard output información sobre los tramos del plano..

Problemas y limitaciones detectados en N-CD++

A continuación se detallan las falencias identificadas en el funcionamiento de N-CD++ que producen limitaciones en la escritura y ejecución de los modelos de simulación correspondientes a ATLAS.

- se detectaron problemas utilizando demoras random en función de la velocidad para las celdas, lo que ocasionaba que los autos avanzaran "pisándose" entre sí. Esto se solucionó utilizando velocidades fijas para modelar el avance de los autos.
- la demora de transporte/inercial se define a nivel de modelo atómico y no a nivel de reglas.
- la función **portvalue** solamente puede usarse con un **portIntransition**, no permitiendo su utilización dentro de un **localtransition** o un **zone**.
- luego de procesar un la función portvalue el valor recibido por el puerto de entrada se "consume", es decir hay una única oportunidad para preguntar por su valor y tomar una determinada acción. A partir de ese momento el valor no se encuentra nunca más disponible para la simulación.
- se encontraron casos de conflicto en las sentencias **localtransition**, **portIntransition** y **zone** para especificar modelos simulables. El simulador da distintos errores dependiendo del orden en que se escriben estas sentecias.
- utilizando la sentencia **send** dentro del bloque de reglas determinadas por una sentencia **zone** la simulación aparentemente termina bien, pues no da ningún mensaje de error. Sin embargo, al analizar el log de salida, se observa que el tiempo de duración indicado no finalizó y que la simulación se corta abruptamente.
- la sentencia **portvalue**(x-c-hayauto) produce error, en cambio la sentencia **portvalue**(x_c_hayauto) funciona correctamente.
- la sentencia **zone** {(0,0)} produce error, en cambio la sentencia **zone** { (0,0) } funciona correctamente.
- la sentencia **out**:nombrePuerto produce error, en cambio la sentencia **out** : nombrePuerto funciona correctamente.
- la sentencia **localtransition** debe estar siempre presente, a pesar de existir sentencias **zone** definidas para todas las celdas.

Futuras extensiones a TSC

Si bien TSC permite especificar un plano de ciudad de una forma sencilla, sería interesante el diseño e implementación de una interfaz gráfica para modelar este plano.
Mediante esta interfaz un analizador de tráfico podría construir los tramos, cruces y demás elementos visualizando el sector de ciudad a medida que lo construye, sin necesidad de indicar a mano y en formato de texto cada uno de los parámetros que se utilizan en la especificación de cada elemento del plano (celdas de comienzo y final de cada tramo, celdas dentro de un tramo por la que pasa una vía, etc).
Por otra parte esta interfaz gráfica podría ocuparse de validar la estructura del plano en el momento mismo de su construcción, sin necesidad de que estas validaciones las realice TSC una vez que todo el plano fue especificado. Además, la visualización del plano que se está construyendo evitaría que se cometan errores de construcción que es factible que se comentan cuando uno especifica el mismo en formato de texto.

Otra extensión interesante para TSC consiste en la incorporación de una herramienta para la construcción de los templates de generación de código. La misma podría validar qur no falten templates para ningún elemento del plano y que la sintaxis de los mismos sea correcta. Si bien TSC incluye un parser para la gramática de especificación de templates, esta herramienta evitaría que los errores de construcción de los mismos se detecten recién en el momento de utilizarlos durante la generación de algún modelo de simulación. Por otra parte no siempre una simulación incluye todos los elementos de especificación de sectores de ciudad disponibles (no todas las simulaciones incluyen todos los elementos de control, semáforos y vías) y quizás se detecte un error en la estructura de los templates mucho tiempo después de haberlos construido, por ejemplo en el momento de utilizar un elemento de control que no se había incluido hasta ese momento en ninguna simulación.

Conclusiones

El control de tráfico es un tema ampliamente investigado por las ciencias de la computación en el área de simulación. El análisis y estudio de los modelos generados pueden usarse directamente para mejorar la calidad de vida a través de un ordenamiento más adecuado del tránsito.

El análisis de estos modelos puede ayudar entre otras cosas a medir el flujo y concentración de los autos en áreas críticas de la ciudad, las consecuencias derivadas de los choques, los desvíos en la circulación por la presencia de hombres trabajando en obras viales y los efectos y alcances de los elementos de regulación de tráfico existentes

El estudio de todos estos factores tiene aplicaciones reales inmediatas que pueden derivar en la reducción de embotellamientos, en la detección de zonas no propicias para estacionar, en alteraciones en la dirección del movimiento de los autos y en construcción de puentes o distribuidores de tránsito.

TSC en combinación con N-CD++, permite la creación de modelos de tránsito de una manera sencilla, permitiendo abstraer sectores de ciudad que presenten problemas de circulación de autos en modelos sencillos y computables para su análisis y estudio.

A través del ejemplo de aplicación mencionado en el capítulo 5 se ha comprobado el cumplimento de todos los objetivos planteados durante la etapa de diseño del compilador TSC, entre ellos,

- Eficiencia y correctitud en la generación de modelos atómicos y modelos celulares necesarios para la simulación de un sector de ciudad.
- Adaptabilidad ante cualquier cambio deseado en la especificación de los modelos celulares debido a la implementación de templates de generación de código.
- Abstracción de la complejidad de la especificación de modelos en favor de un lenguaje sencillo y natural.

Todos estos puntos dan una idea de la gran potencia y robustez del compilador TSC para la generación de modelos de simulación de tráfico.

Un trabajo muy interesante para complementar a TSC es la implementación de una interface gráfica para la construcción del plano a simular, facilitando aún más la tarea de definir sectores de ciudad para su posterior simulación. Con esta interface, con el compilador TSC y con el simulador N-CD++ se le puede proveer a los analizadores y estudioso del tráfico urbano, de un valioso conjunto de herramientas capaces de facilitar la investigación y solución de todos los inconvenientes derivados de la circulación de autos en cualquier sector de ciudad.

Bibliografía

[DW99] A.Davidson, G. Wainer
"Definición de un lenguaje de especificación para simulación de tráfico urbano siguiendo el paradigma Cell-DEVS".
Tesis de Licenciatura, Departamento de Computación, FCEN/UBA, 1999

[BBW98] A.Barylko, G. Beyoglonian, G. Wainer
"CD++, una herramienta de implementación de modelos Cell-DEVS binarios"
Tesis de Licenciatura, Departamento de Computación, FCEN/UBA,1998

[RW99] D. Rodriguez, G. Wainer
"N-CD++, implementación de modelos Cell-DEVS n-dimensionales"
Tesis de Licenciatura, Departamento de Computación, FCEN/UBA,1999

[ASU90] A. Aho, R. Sethi,J Ullman
"Compiladores, principios técnincas y herramientas"
Addison Wesley, 1990

[SB98] B. Stroustrup
"El lenguaje de programación C++"
Addison Wesley, 1998

[O97] Steve Oualline
"Practical C++ Programming"
O´REILLY, 1995. Correcciones posteriores en 1997

[W96] G. Wainer
"Introducción a la Simulación de Eventos Discretos"
Technical Report 96-005, Departamento de Computación, FCEN/UBA

[WD00] G. Wainer, A.Davidson
"Traffic control specifications using discrete events cellular models"
Departamento de Computación, FCEN/UBA, 2000

[WD00] G. Wainer, A.Davidson
"Specifying truck movement in traffic models using Cell-DEVS"
Departamento de Computación, FCEN/UBA, 2000

[DVW00] A. Diaz, V.Vázquez, G. Wainer
"Application of ATLAS language in models of urban traffic"
Departamento de Computación, FCEN/UBA, 2000

[WG98] G. Wainer, N. Giambiasi
"Specification, modelling and simulation of timed Cell-DEVS models"
Technical Report 97-007, Departamento de Computación, FCEN/UBA, 1998

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
"Design Patterns"
Addison Wesley, 1996

[Zei76] B. Zeigler
"Theory of modeling and simulation"
Wiley, 1976