Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

"Verificación Formal del Diseño de Tráfico Vehicular"

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

HELAINE FLORES CERVANTES

ASESOR:

M.C. Liliana Badillo Sánchez

CO-ASESOR:

M.C. José Andrés Vázquez Flores

PUEBLA, PUEBLA 2010

Agradecimientos

Gracias a Dios

Por darme la oportunidad de terminar con esta etapa de mi vida

Gracias a mis padres y hermanos

Por todo su apoyo que me brindaron en todo momento.

Gracias a mi asesor

Por sus consejos y paciencia, sin su ayuda y conocimientos no estaría en donde me encuentro ahora.

Gracias a mi co-asesor

Por su apoyarme en este proyecto, muchas gracias.

Índice

Introducción

Los sistemas de hardware y software son cada vez más grandes y complejos, por ello es cada vez más fácil que se produzcan fallas en las aplicaciones, y estas se vuelvan cada vez más difíciles de detectar.

Es necesario realizar una comprobación del sistema para asegurarnos que un sistema computacional haga lo que tenga de hacer. El área de Verificación Formal estudia los fundamentos teóricos y la implementación de técnicas de verificación de sistemas computacionales utilizando técnicas matemáticas para asegurar la validez de un sistema.

Para realizar la verificación formal se construye un modelo matemático de los posibles comportamientos del sistema, después se especifica en un lenguaje formal los comportamientos deseables del sistema y por ultimo se verifica si el modelo satisface las especificaciones.

Una especificación formal sirve para descubrir inconsistencias como requerimientos contradictorios u omisiones en la funcionalidad del sistema a construir y analizar propiedades del sistema antes de su construcción

En este documento se muestra la importancia del uso de un método formal conocido como model-checking o verificador de modelos, para comprobar el buen funcionamiento del control del tráfico vehicular. El verificador de modelos a utilizar en este trabajo es SPIN, el cual es una herramienta que utiliza el lenguaje de programación PROMELA para especificar el algoritmo de control de tráfico vehicular.

Capítulo1 Marco Teórico

En este capítulo se describen los términos: métodos formales y la verificación de modelos. Estos elementos se encuentran fuertemente ligados con el desarrollo de este trabajo y en conjunto forman el marco teórico que respalda esta tesina.

1.1Métodos Formales

En Ciencias de la Computación los métodos formales son técnicas matemáticas usadas para la especificación, desarrollo y verificación de software (y hardware). Su uso en el diseño puede contribuir a la robustez[1] y precisión de programas. Dado su alto costo se utilizan únicamente en el desarrollo de sistemas de alta integridad donde la seguridad es de vital importancia [1].

Los métodos formales se utilizan en diferentes niveles:

Nivel 0: Especificación formal. Es una descripción matemática de software o hardware, que describe qué debe hacer y el sistema, y no el cómo lo debe de hacer.
Nivel 1: Desarrollo formal y verificación formal. Para producir un programa de manera formal. Ejemplo: una prueba de propiedades de refinamiento de la especificación al programa. Es el más apropiado en sistemas de alta integridad.
Nivel 2: Demostradores de teoremas. Para llevar a cabo pruebas formales realizadas automáticamente. Es demasiado caro y práctico solamente si el costo de errores es extremadamente alto (i.e. partes críticas en el diseño de microprocesadores).

Los métodos formales se clasifican de la siguiente manera:

- Semántica denotacional, el significado de un sistema se expresa con la teoría matemática de dominios (estudia casos especiales de conjuntos parcialmente ordenados).

[1] Por **robustez** de un programa se entiende la ausencia de fallos en su ejecución con diferentes datos de entrada durante intervalos de tiempo predeterminados.

- Semántica operacional, el significado de un sistema se expresa como una secuencia de acciones de un modelo computacional más simple.
- Semántica axiomática, el significado del sistema es expresado en términos de precondiciones y pos condiciones e invariantes las cuales son verdaderas antes que el sistema realice una tarea. Los críticos dicen que esta semántica nunca describe lo que un sistema hace, solamente lo que es verdadero antes y después de su ejecución.

Los métodos formales se aplican en varios puntos durante el proceso de desarrollo:

- Especificación. Para dar una descripción formal del sistema a ser desarrollado, a cualquier nivel de detalle deseado. Es usada para guiar futuras actividades de desarrollo (siguientes puntos). Adicionalmente, se puede usar para verificar que los requerimientos del sistema que se está desarrollado han sido completa y exactamente especificados.
- Desarrollo. Una vez que la especificación formal ha sido llevada a cabo, se usa como guía mientras el sistema está siendo desarrollado. Ejemplos:
    - Si la especificación formal es en semántica operacional, el comportamiento observado puede ser comparado con el de la especificación.
    - Si la especificación formal es en semántica axiomática, las precondiciones y pos condiciones se pueden convertir en aserciones en el código ejecutable.
- Verificación. Una vez que se ha llevado a cabo el desarrollo guiado por la especificación, puede ser usada para probar propiedades sobre el mismo (es deseable hacerlo mediante la inferencia del sistema desarrollado).
- Demostración realizada por esfuerzo humano. En algunas ocasiones, la motivación para demostrar que un sistema es correcto no es la obvia necesidad de reasegurarse de su funcionalidad, sino el deseo de entenderlo mejor. Algunas demostraciones son producidas en el estilo de demostración matemática: escritas a mano usando lenguaje natural y usando un nivel de informalidad común a dichas demostraciones. Una buena demostración es la que puede ser leída y entendida por cualquier lector con conocimientos del área.
- Demostraciones automáticas. Se dividen en las siguientes categorías:

- Demostración automática de teoremas. Un sistema produce una prueba formal dada la descripción del sistema, un conjunto de axiomas lógicos y un conjunto de reglas de inferencia.
- Análisis Estático (Static Analysis). Es realizado sin ejecutar el programa. En la mayoría de los casos es sobre una versión del código fuente y en otros en el código objeto, también se considera la documentación.

  El término es usualmente aplicado al uso de una herramienta automática con análisis humano llamado comprensión del programa. Algunas personas consideran a las métricas de software y la ingeniería inversa formas de análisis estático. Ejemplos de implementaciones son: *verificación de modelos*, que considera sistemas que tienen estados finitos o pueden ser reducidos a estados finitos por abstracción; *interpretación abstracta*, modela el efecto que cada sentencia tiene en un estado en una máquina abstracta; uso de aserciones en código como fue sugerido por Hoare.

- Análisis dinámico (Dynamic analysis). Se realiza ejecutando programas en un procesador virtual o real.

1.2 Verificación de modelos

Otro enfoque de razonamiento formal de programas es la verificación de modelos (model checking)[1][2]. Esta técnica se utiliza para demostrar que se mantengan algunas propiedades a través del tiempo de un modelo formalmente especificado. Algunos ejemplos de utilización de model checking se encuentran en [3][4][5].

El modelo puede ser alguno de los siguientes:

- Del programa (cada comando es un estado).
- Una abstracción del programa.
- Un modelo de la especificación.
- Un modelo del dominio.

Verificación de modelos está basada en la lógica temporal lineal (LTL). Busca una solución entre todos los posibles estados del modelo dado. Da respuesta a algunas preguntas acerca de propiedades temporales de un programa.

Se construye el modelo A de un problema o sistema. L(A) denota todos los posibles comportamientos y L(P) el conjunto de comportamientos válidos (i.e. aquellos donde la propiedad P es satisfactible).Para demostrar que el modelo A siempre satisface P, es suficiente que $L(A) \subseteq L(P)$ o equivalentemente $T(A) \cap L(P) = \varnothing$. Si la intersección anterior es no vacía, se busca un contraejemplo válido.

1.3 Verificación de modelos temporal

En la mayoría de los sistemas, la verdad de algunas fórmulas es dinámica, es decir, cambia con el tiempo.

Existen dos tipos de lógica temporal:

- Árbol lógico de computación (CTL, Computation Tree Logic). El tiempo se representa como un árbol que tiene raíz en el momento actual y se extiende hacia el futuro.
- Lógica temporal lineal. El tiempo es un conjunto de trayectorias. Una trayectoria es una secuencia de instantes en el tiempo. La lógica temporal lineal se relaciona con la teoría de autómatas finitos, la cual se utiliza para modelar sistemas.

1.4 Lógica temporal lineal (LTL)

La lógica temporal (LT) es una extensión de la lógica clásica y se utiliza para describir un sistema de reglas y simbolismo, para la representación y razonamiento de proposiciones cuantificadas en términos de tiempo. La lógica temporal lineal (LTL) modela el tiempo como una secuencia de estados, que se extienden infinitamente. Esta secuencia de estados se conoce como camino. En general el futuro no está determinado, por lo que se consideran diferentes caminos que representan posibles futuros. LTL contiene a los operadores mostrados en la Tabla 1.

Siguiente	$\bigcirc$
Eventualmente	$\Diamond$
Siempre	$\Box$
Fuerte hasta	$\mathbf{U}$
Débil hasta	$\mathbf{W}$

Tabla 1. Operadores de la Lógica Temporal Lineal (LTL).

1.5 Sintaxis de LTL

Una fórmula bien formada (wff) en LTL se define recursivamente como:

- $\top, \bot$ son fórmulas bien formadas.
- Si **p** es un símbolo proposicional que representa una propiedad falsa o verdadera en cualquier estado del modelo **A**, p es una fórmula bien formada.

- Si **p** y **q** son fórmulas bien formadas entonces también:

$$\neg \mathbf{p},\ \mathbf{p} \wedge \mathbf{q},\ \mathbf{p} \vee \mathbf{q},\ \mathbf{p} \rightarrow \mathbf{q},$$
$$\bigcirc \mathbf{p},\ \Diamond \mathbf{p},\ \Box \mathbf{p},\ \mathbf{p}\ \mathbf{U}\ \mathbf{q},\ \mathbf{p}\ \mathbf{W}\ \mathbf{q}$$

- Son todas.

1.6 Semántica de LTL

El símbolo $\vDash$ expresa que una secuencia de estados satisface una fórmula bien formada en LTL. Lo anterior significa que:

- $\sigma[i] \vDash p \Leftrightarrow$ la propiedad *p* permanece en la secuencia de estados $\{ \sigma_i, \sigma_{i+1}, \ldots \}$.
- $\sigma \vDash f \Leftrightarrow$ la fórmula temporal f permanece en la secuencia de estados σ.

Con esta notación se definen formalmente los operadores temporales de *LTL* a continuación:

Siempre: $\sigma \vDash \Box p \Leftrightarrow \forall i \geq 0.\ (\sigma[i] \vDash p)$*Eventualmente*: $\sigma[i] \vDash \Diamond p \Leftrightarrow \exists i \geq 0.\ (\sigma[i] \vDash p)$

◊*p* establece la garantía de que la propiedad *p* eventualmente se convertirá verdadera al menos una vez.

Siguiente: $\sigma[i] \vDash \mathbf{O}p \Leftrightarrow \sigma[i+1] \vDash p$

O*p* establece que la propiedad *p* es verdadera en el estado siguiente.

Capítulo 2 Herramienta SPIN

SPIN [1] es una herramienta que asiste en la verificación formal. Fue desarrollado por el grupo de métodos formales y verificación de los Laboratorios Bell, dando comienzo su desarrollo en 1980. Para llevar a cabo el proceso de verificación, Spin utiliza un lenguaje propio para especificar abstracciones de sistemas. Este lenguaje se denomina PROMELA (PROcess MEta LANguage)[6].

Spin acepta especificaciones de diseño por medio del lenguaje PROMELA (similar en sintaxis al lenguaje C), que, a su vez, acepta especificaciones de corrección en la sintaxis del estándar LTL (Lógica Temporal Lineal).Los pasos para verificar un programa son los siguientes:

- Se construye el programa usando PROMELA.
- El programa es traducido por SPIN a un autómata finito
- Expresamos la propiedad a verificar usando lógica temporal
- La propiedad es traducida por SPIN a un nuevo autómata
- SPIN verificará que se cumpla la propiedad, en caso contrario, SPIN mostrará a la secuencia de líneas de código que nos llevan al error [6].

2.1 Instalación de SPIN

La instalación de SPIN en Linux se realiza tecleando las siguientes líneas de comandos:

cd/home/usuario

gunzip spin514.tar.gz

tar-xf spin514.tar

isntalar librerias buil-essential

cd spin5.1.4

sudo apt-get install byacc

make

sudo cp-l spin /usr/local/bin

2.3. Verificación de modelos en SPIN

Se utilizó el método formal verificación de modelos para verificar que el algoritmo propuesto es correcto (*i.e.* termina). El verificador de modelos que se utilizó es la herramienta SPIN [1]. Ésta utiliza el lenguaje de programación PROMELA [7] para especificar algoritmos. Los pasos para verificar un algoritmo en *Spin* son los siguientes:

1. Se codifica el algoritmo en el lenguaje PROMELA (similar al lenguaje *C*).
2. *Spin* mapea automáticamente el algoritmo a un autómata finito.
3. El estado del algoritmo no deseado (*never claim*) se expresa en lógica temporal (*LTL*).
4. *Spin* traduce a un autómata la fórmula en *LTL* para el algoritmo y se verifica el algoritmo.

SPIN verifica que el lenguaje del autómata del algoritmo está contenido en el lenguaje del autómata de la propiedad a verificar. Si se cumple lo anterior el algoritmo es correcto. En caso contrario, SPIN muestra la secuencia de líneas de código que contiene el error.

De acuerdo con el proceso anterior, primero se codifica el algoritmo propuesto en el lenguaje PROMELA. SPIN genera internamente el autómata finito a partir del código en PROMELA.

Después se especificó la propiedad a verificar del algoritmo. El comportamiento deseado se expresa en LTL con una fórmula.

Spin efectúa automáticamente el último paso para la verificación del algoritmo. Se demuestra que no se alcanza el estado no deseado en cualquier ejecución del algoritmo, por lo tanto el algoritmo termina (*i.e.* es correcto).

Capítulo 3 Modelo de Control de Tráfico Vehicular

En éste capítulo se describe el contexto y la formulación del problema que se trata en este trabajo de investigación, así como la solución propuesta y su formalización en el verificador de modelos automático SPIN.

3.1 Descripción del problema

En la intersección West End que se encuentra en Edimburgo [8] convergen las siguientes calles (ver figura 1):

1. Princes Street.

2. Lothian Road.

3. Rutland Street.

4. Shandwick Place.

5. Queensferry Street.

6. Hope Street.

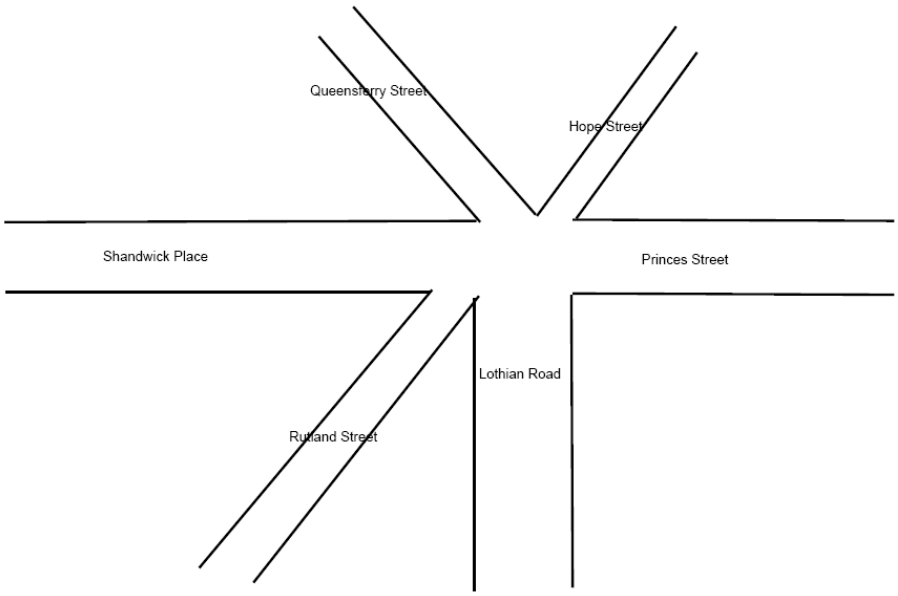

Figura 1. Intersección West End

Se utilizan luces de control de tráfico para controlar el flujo de tráfico en esta intersección. Es importante que las señales de las luces de tráfico no permitan un comportamiento no deseado. Para asegurar que las luces de tráfico se comporten de manera correcta, es necesario construir y verificar un modelo del flujo de tráfico.

Existen seis etapas por cada ciclo de las señales de tráfico. Para el modelo se consideran las siguientes premisas:

- No existen peatones cruzando las calles.
- Todas las luces se ponen en rojo antes de pasar a un nuevo estado en el ciclo de las luces de tráfico.
- El área donde se interceptan las calles es lo suficientemente grande como para permitir "vueltas amplias". Un ejemplo de ello es si el tráfico de Shandwich Place fluye hacia Hope Street, es posible que el tráfico de Princes Street de vuelta hacia Rutland Street.

Además la intersección West End tiene las siguientes restricciones:

1. No se permiten las "vueltas en u".
2. La calle Rutland Street es de un solo sentido, por lo que el tráfico solo puede circular en sentido contrario de la intersección.
3. El tráfico de Queensferry Street no puede voltear hacia:
    a. Princes Street
    b. Shandwich Place
    c. Lothian Road
    d. Rutland Street
4. El tráfico de Hope Street no puede voltear hacia:
    a. Princes Street
    b. Queensferry Street

c. Lothian Road

d. Rutland Street

5. El tráfico de Shandwick Place no puede voltear hacia Rutland Street.

3.2 Solución propuesta

En la materia de Automated Reasoning de School of Informatics of the University of Edinburgh [8] se propuso el problema de corregir un modelo pr opuesto de la intersección West End.

Las tareas que se realizaron para corregir el modelo propuesto para que no permitiera algún comportamiento no deseado son las siguientes:

Tarea 1

En el archivo NeverUTurns se especifica un never claim (ver figura 2) para verificar que el modelo WestEnd no permite vueltas en U. Para realizar esta tarea se corrieron los siguientes comandos:

spin –a – N NeverUTurns WestEnd

cc –o pan pan.c

./ pan –e

./pan –r1

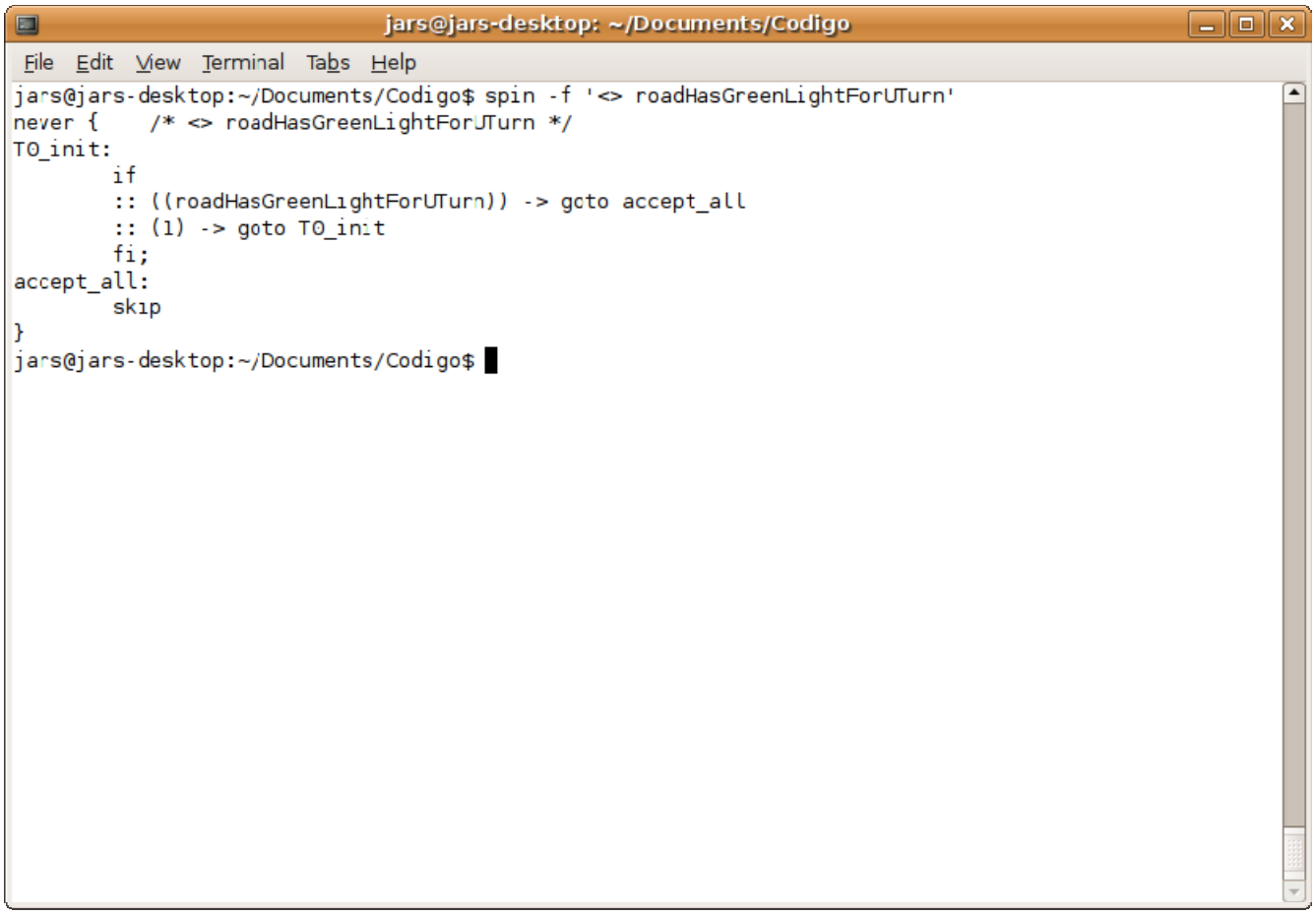

Figura 2. Never claim generado por Spin para verificar la restricción 1.

La formula en LTL para verificar que no se permitan este tipo de vueltas es la siguiente:

```
#define greenForUTurns
(travelling_from[PRINCES_STREET].going_to[PRINCES_STREET]==GREEN ||
travelling_from[LOTHIAN_ROAD].going_to[LOTHIAN_ROAD]==GREEN ||
travelling_from[RUTLAND_STREET].going_to[RUTLAND_STREET]==GREEN ||
travelling_from[SHANDWICK_PLACE].going_to[SHANDWICK_PLACE]==GREEN ||
travelling_from[QUEENSFERRY_STREET].going_to[QUEENSFERRY_STREET]==GREEN ||
travelling_from[HOPE_STREET].going_to[HOPE_STREET]==GREEN )
never { /* <>greenForUTurns */
TO_init:
if
:: greenForUTurns -> goto accept_all
:: (1) -> goto TO_init
fi;
accept_all:
skip}
```

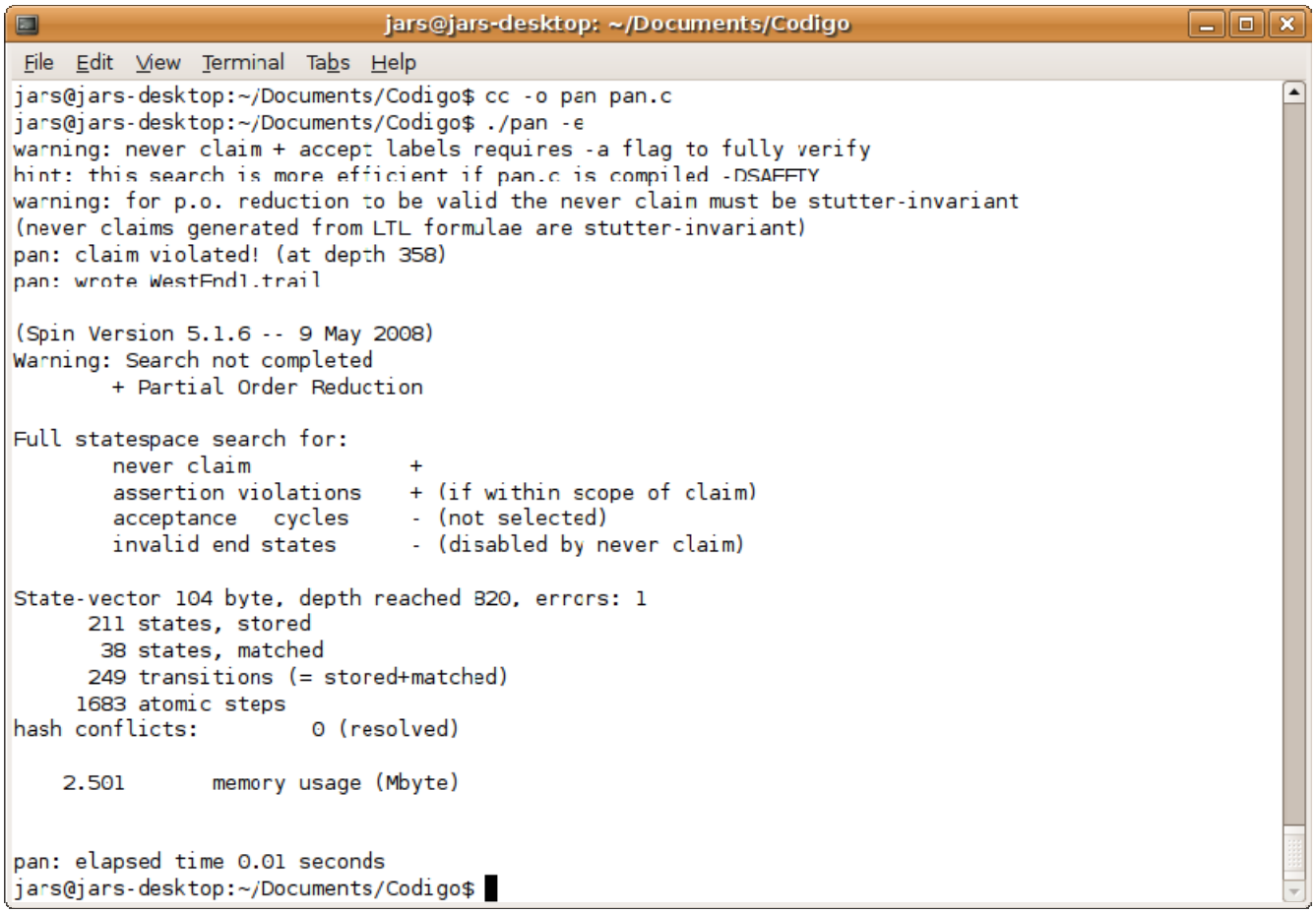

Figura 3. Verificación del modelo por Spin para vueltas en U.

En el modelo WestEnd se detecto una vuelta en U (ver figura 3), en la calle SHANDWICK_PLACE. La solución fue cambiar la bandera a falso.

traveling_from[SHANDWICK_PLACE].going_to[SHANDWICK_PLACE] = false;

Al cambiar la línea anterior se obtuvo un modelo correcto (ver figura 4).

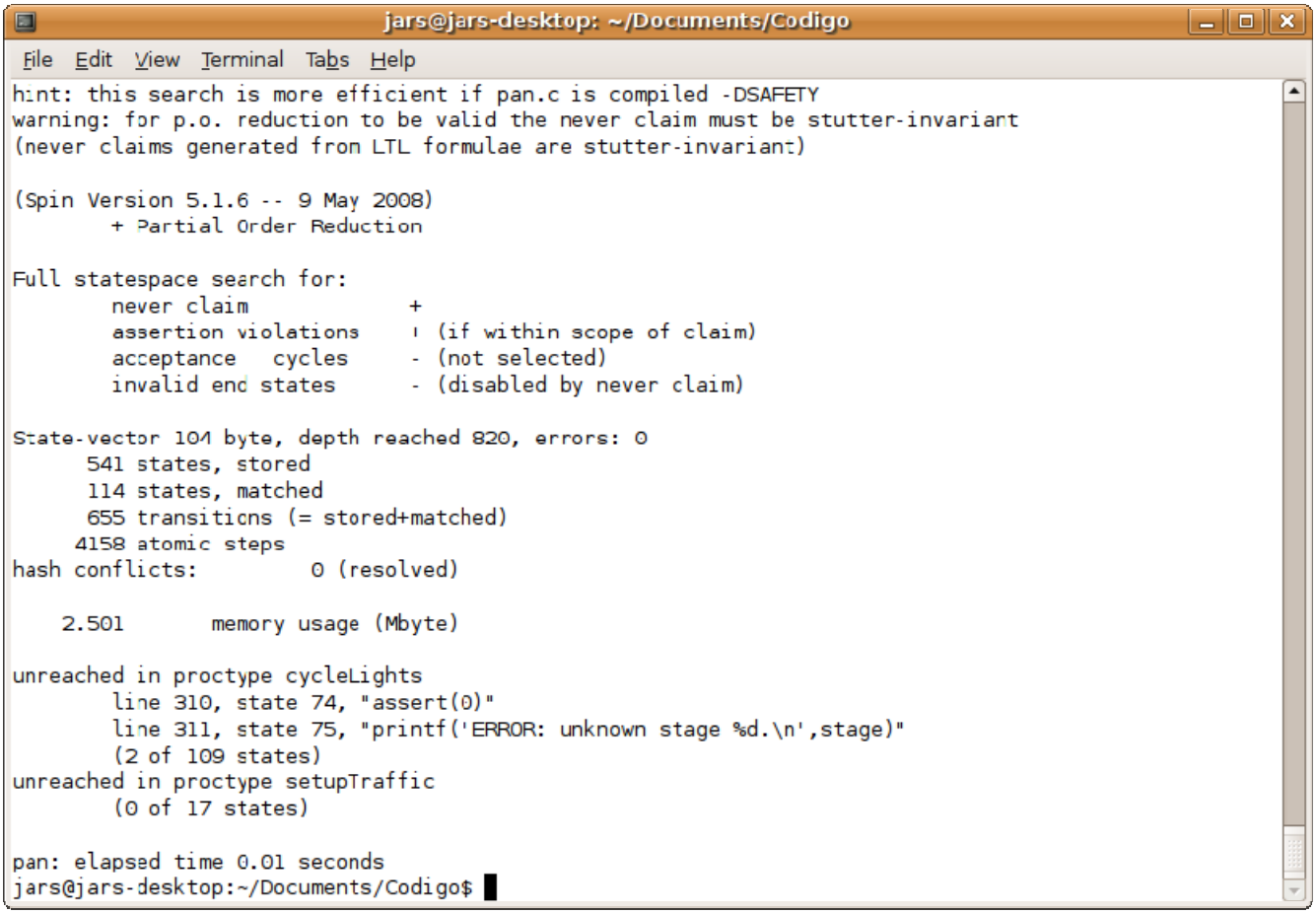

Figura 4. Verificación del modelo corregido en Spin.

Tarea 2

Se especificó un never claim (ver figura 5) para verificar el modelo para satisfacer la restricción 2.

spin –a – N RutlandOneWay WestEnd

cc –o pan pan.c

./ pan –e

./pan –r1

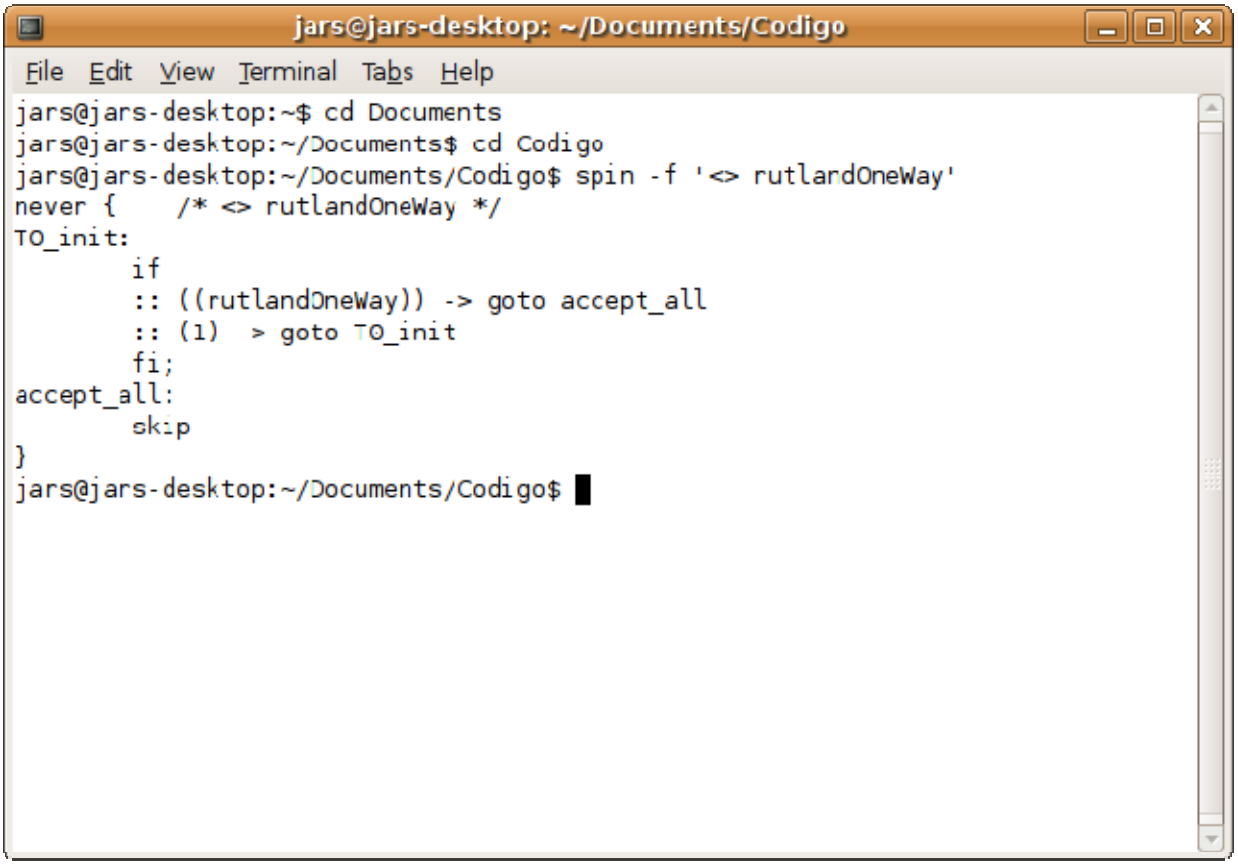

Figura 5. Never claim generado por Spin para verificar la restricción 2.

La formula LTL a verificar en el archivo RutlandOneWay es la siguiente:

```
int randomRoad;
active proctype setupTraffic() {
  setRoadRandomly(randomRoad);
  setupTrafficDone = true;
}

#define rutlandOneWay \
   (travelling_from[RUTLAND_STREET].going_to[randomRoad] == GREEN)

never {    /* <> rutlandOneWay */
T0_init:
      if
      :: ((rutlandOneWay)) -> goto accept_all
      :: (1) -> goto T0_init
      fi;
accept_all:
      skip
}
```

En el modelo WestEnd se detecto una vuelta equivocada en Rutland.

traveling_from[RUTLAND_STREET].going_to[LOTHIAN_ROAD] = false;

Tarea 3

Se generó un never claim (ver figura 6) y una formula LTL en el archivo NeverDisallowed para verificar que el modelo satisface las restricciones 3, 4 y 5.

spin –a – N NeverDisallowed WestEnd

cc –o pan pan.c

./ pan –e

./pan –r1

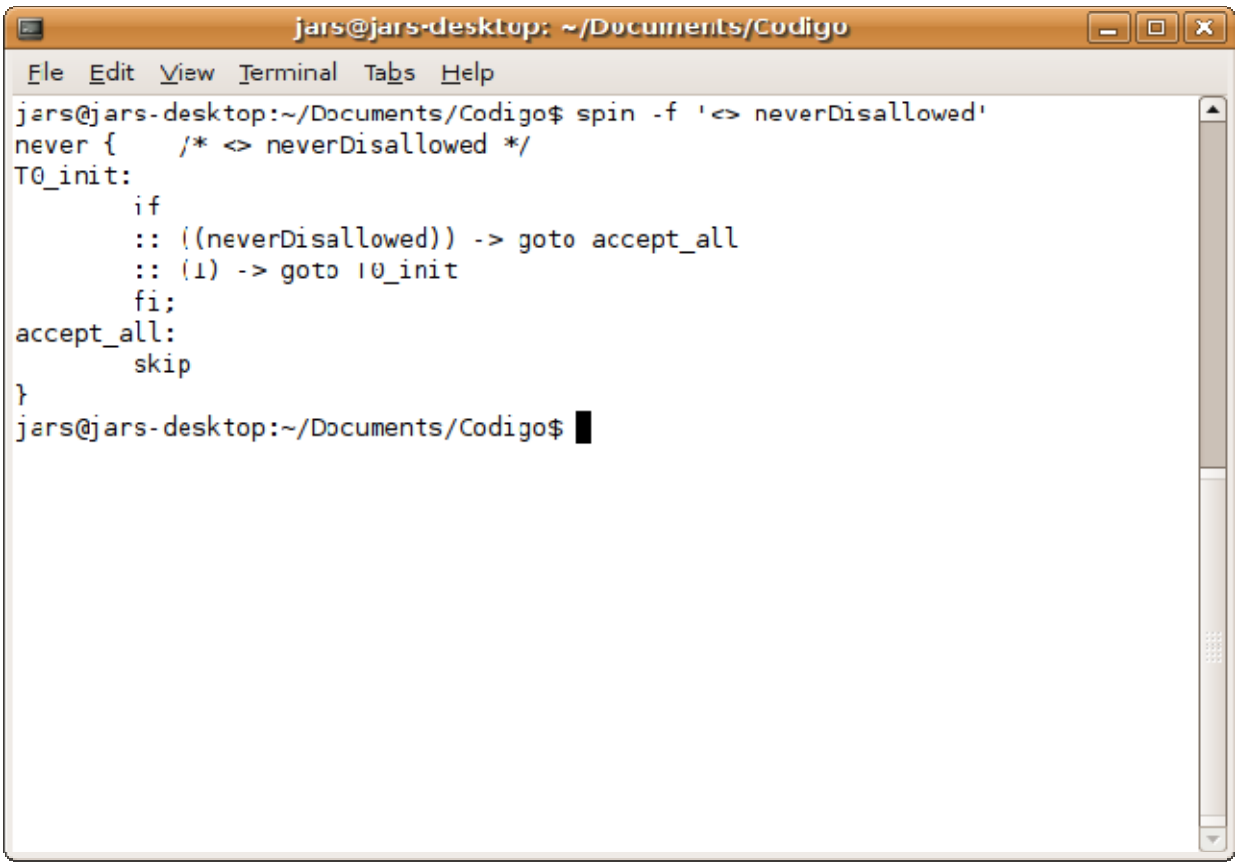

Figura 6. Never claim generado por Spin para verificar las restricciones 3, 4 y 5.

La formula LTL para comprobar que no se violen las restricciones 3,4 y 5 es la siguiente:

```
int randomRoad;
active proctype setupTraffic() {
  setRoadRandomly(randomRoad);
  setupTrafficDone = true;
}

#define neverDisallowed \
   (travelling_from[randomRoad].going_to[randomRoad]                  ==                GREEN                ||
travelling_from[QUEENSFERRY_STREET].going_to[PRINCES_STREET]          ==          GREEN          ||
travelling_from[QUEENSFERRY_STREET].going_to[SHANDWICK_PLACE]         ==          GREEN          ||
travelling_from[QUEENSFERRY_STREET].going_to[LOTHIAN_ROAD]            ==          GREEN          ||
travelling_from[QUEENSFERRY_STREET].going_to[RUTLAND_STREET]          ==          GREEN          ||
travelling_from[HOPE_STREET].going_to[PRINCES_STREET]                 ==                GREEN                ||
travelling_from[HOPE_STREET].going_to[QUEENSFERRY_STREET]             ==          GREEN          ||
travelling_from[HOPE_STREET].going_to[RUTLAND_STREET]                 ==                GREEN
||travelling_from[SHANDWICK_PLACE].going_to[RUTLAND_STREET] == GREEN )


never {
T0_init:
      if
      :: ((neverDisallowed)) -> goto accept_all
      :: (1) -> goto T0_init
      fi;
accept_all:
      skip
}
```

En el modelo WestEnd se detecto que se violaron las restricciones en:

traveling_from[QUEENSFERRY_STREET].going_to[LOTHIAN_ROAD] = false;

traveling_from[HOPE_STREET].going_to[PRINCES_STREET] = false;

traveling_from[SHANDWICK_PLACE].going_to[RUTLAND_STREET] = false;

Conclusiones

Técnicas como la simulación[2] y las pruebas de software; no son suficientemente robustas para asegurar el buen funcionamiento de un sistema bajo cualquier circunstancia. En la historia existen varios ejemplos de fallas en sistemas computacionales, algunas con pérdidas humanas, otras con pérdidas materiales.

La técnica de verificación de modelos, involucra varios temas de la ciencia de la computación y matemáticas, como son: lógica temporal, autómatas, algoritmos y estructuras de datos, teoría de puntos fijos, representación simbólica, entre otros.

El verificador de modelos SPIN permite realizar la verificación de sistemas desde hace casi tres décadas con resultados positivos. Este software ganó el *System Software Award* (Premio de Sistemas de Software) de la *ACM* (Association for Computing Machinery) en el 2001[9].

La técnica de verificación de modelos no es un conjunto de técnicas teóricas sin trascendencia en la práctica, al contrario, es una de las tantas aplicaciones de la teoría computacional. Una de sus ventajas principales, es que no es necesario diseñar una extensa etapa de pruebas al sistema construido. Éste trabajo es un ejemplo de cómo dicha técnica puede ser transferida hacia la industria con impacto tecnológico.

Fue necesario realizar un análisis del modelo propuesto, para encontrar solución al problema planteado en el objetivo. Además de proponer formulas en LTL para modelar el comportamiento no deseado. A partir de este análisis, se propusieron las mejoras correspondientes para que el modelo sea correcto.

[2] "La simulación es el diseñar y desarrollar un modelo computarizado de un sistema o proceso y conducir experimentalmente con este modelo con el propósito de entender el comportamiento del sistema del mundo real o evaluar varias estrategias con los cuales puedan operar el sistema".

La aportación del presente trabajo es la corrección del modelo presentado como propuesta para el control de tráfico de la intersección West End, además de la ejemplificación de cómo diseñar y verificar un modelo de algún sistema que tiene estados y que se comporta dinámicamente.

Los paradigmas de la computación están en constante desarrollo para adecuarse a las necesidades de la industria y nuevas tecnologías. Esto ha hecho que las técnicas formales de verificación y diseño hayan retomado cierto auge, ya que se ha comprobado que tienen una gran utilidad en la reducción de costos y esfuerzos en el desarrollo de sistemas.

Referencias

[1] Hossam A. Gabbar. "*Modern Formal Methods and Aplications*". Springer. 2006.

[2] Michael Huth, Mark Ryan."*Logic in Computer Science Modelling and Reasoning abuout Systems*". Cambridge University Press. 2004.

[3] M. Aguilar-Cornejo, J.L. Quiroz-Fabian, G. Román-Alonso."*Verificación formal de un algoritmo de procesamiento paralelo de imágenes médicas*". Área de Computación y Sistema. División de CBI,UAM-I. http://www.izt.uam.mx/contactos/n64ne/imag-med.pdf . Consultada en Agosto del 2009.

[4] P. Ammann, P.E. Black y W. Majurski, "*Using Model Checking to Generate Tests from Specifications"*. En *Proceedings of the Second IEEE International Conference on Formal Engineering Methods*. Brisbane - Australia, (1998).

[5] José Garcia-Fanjul, Javier Tuya, Claudia de la Riva."*Generación de casos de prueba para composiciones de servicios web especificadas en BPEL*". Departamento de Informática, Universidad de Oviedo. Consultada en septiembre 2009 http://in2test.lsi.uniovi.es/pris2006/PRIS2006-FanjulTuyaDelaRiva.pdf

[6] Gerard J. Holzmann. "The *SPIN Model Checker:Primer and Reference Manual*". Addison-Wesley. 2004

[7] Gerard J. Holzmann, G.J. "*Design and Validation of Computer Protocols*". Englewood Cliffs, N.J.: Prentice Hall, 1991.

[8]Automated Reasoning de School of Informatics of the University of Edinburgh. http://www.inf.ed.ac.uk/teaching/courses/ar/coursework/. Consultada el 7 de Mayo de 2009.

[9] Association for Computing Machinery. http://awards.acm.org/software_system/ . Consultada en Octubre 2009.

Anexo A. Código Fuente

West End

```
typedef PermittedDestinations {
  bool going_to[6]
};
PermittedDestinations travelling_from[6];
int PRINCES_STREET = 0,
    LOTHIAN_ROAD = 1,
    RUTLAND_STREET = 2,
    SHANDWICK_PLACE = 3,
    QUEENSFERRY_STREET = 4,
    HOPE_STREET = 5;
bool GREEN = true,
     RED = false;
mtype = { Princes_Street, Shandwick_Place, Lothian_Road, Queensferry_Street, Hope_Street, Rutland_Street };
mtype NAME_OF_STREET[6];
inline setupStreetNames() {
  NAME_OF_STREET[0] = Princes_Street;
  NAME_OF_STREET[1] = Lothian_Road;
  NAME_OF_STREET[2] = Rutland_Street;
  NAME_OF_STREET[3] = Shandwick_Place;
  NAME_OF_STREET[4] = Queensferry_Street;
  NAME_OF_STREET[5] = Hope_Street;
}
inline stopTraffic( ) {
  atomic{
    int i, j;
    i = 0;
    do
```

```
    :: i<6 -> {
      j = 0;
      do
        :: j<6 -> {
          travelling_from[i].going_to[j] = RED;
          j++;
        }
        :: else -> break
      od;
      i++;
    }
    :: else -> break;
  od;
 }
}
bool setupLightsDone = false;
bool setupTrafficDone = false;
inline waitForSetupDone() {
 do
 :: (setupLightsDone && setupTrafficDone) -> break;
 :: else -> skip;
 od;
}
typedef Route {
 int from, to;
}
inline setRoadRandomly(x) {
 if
   :: true -> x=PRINCES_STREET;
   :: true -> x=SHANDWICK_PLACE;
   :: true -> x=LOTHIAN_ROAD;
```

```
    :: true -> x=QUEENSFERRY_STREET;
    :: true -> x=HOPE_STREET;
    :: true -> x=RUTLAND_STREET
  fi;
}
inline setRouteRandomly(x) {
  atomic {
    setRoadRandomly(x.from);
    setRoadRandomly(x.to);
  }
}
mtype = { TRAFFIC_INFO, TRAFFIC_INFO_CHECKING, TRAFFIC_ERROR }
inline printRoute(m, x) {
  atomic {
        skip;
    printm(m);
    printf(" from ");
    printm(NAME_OF_STREET[x.from]);
    printf(" to ");
    printm(NAME_OF_STREET[x.to]);
    printf("\n");
  }
}
#define LIGHT_FOR_ROUTE(route) \
   (travelling_from[route.from].going_to[route.to])
int stage = 0;
int NUM_STAGES_IN_CYCLE = 6;
active proctype cycleLights() {
  setupStreetNames();
  setupLightsDone = true;
  waitForSetupDone();
```

```
restart_cycle:
  atomic { skip; /* just for pretty-printing */
    printf("TRAFFIC_INFO setting lights for stage %d\n", stage);
  }
  if
  :: (stage%NUM_STAGES_IN_CYCLE == 0) ->
    atomic {

      travelling_from[PRINCES_STREET].going_to[SHANDWICK_PLACE] = true;
      travelling_from[PRINCES_STREET].going_to[RUTLAND_STREET] = true;
      travelling_from[PRINCES_STREET].going_to[LOTHIAN_ROAD] = true;
      travelling_from[SHANDWICK_PLACE].going_to[PRINCES_STREET] = true;
      travelling_from[SHANDWICK_PLACE].going_to[QUEENSFERRY_STREET] = true;
      travelling_from[SHANDWICK_PLACE].going_to[HOPE_STREET] = true;
    }
  :: (stage%NUM_STAGES_IN_CYCLE == 1) ->
    atomic {
      travelling_from[PRINCES_STREET].going_to[QUEENSFERRY_STREET] = true;
      travelling_from[PRINCES_STREET].going_to[HOPE_STREET] = true;
      travelling_from[LOTHIAN_ROAD].going_to[RUTLAND_STREET] = true;
      travelling_from[LOTHIAN_ROAD].going_to[SHANDWICK_PLACE] = true;
    }
  :: (stage%NUM_STAGES_IN_CYCLE == 2) ->
    atomic {
      travelling_from[SHANDWICK_PLACE].going_to[QUEENSFERRY_STREET] = true;
      travelling_from[SHANDWICK_PLACE].going_to[SHANDWICK_PLACE] = false;
      travelling_from[SHANDWICK_PLACE].going_to[HOPE_STREET] = true;
      travelling_from[SHANDWICK_PLACE].going_to[PRINCES_STREET] = true;
      travelling_from[SHANDWICK_PLACE].going_to[LOTHIAN_ROAD] = true;
      travelling_from[RUTLAND_STREET].going_to[LOTHIAN_ROAD] = false;
```

```
    travelling_from[LOTHIAN_ROAD].going_to[RUTLAND_STREET] = true;
    travelling_from[LOTHIAN_ROAD].going_to[SHANDWICK_PLACE] = true;
    travelling_from[LOTHIAN_ROAD].going_to[QUEENSFERRY_STREET] = true;
   }
  :: (stage%NUM_STAGES_IN_CYCLE == 3) ->
   atomic {
    travelling_from[PRINCES_STREET].going_to[LOTHIAN_ROAD] = true;
    travelling_from[SHANDWICK_PLACE].going_to[QUEENSFERRY_STREET] = true;
    travelling_from[SHANDWICK_PLACE].going_to[HOPE_STREET] = true;
    travelling_from[SHANDWICK_PLACE].going_to[RUTLAND_STREET] = false;
    travelling_from[LOTHIAN_ROAD].going_to[PRINCES_STREET] = true;
    travelling_from[LOTHIAN_ROAD].going_to[RUTLAND_STREET] = true;
    travelling_from[LOTHIAN_ROAD].going_to[SHANDWICK_PLACE] = true;
   }
  :: (stage%NUM_STAGES_IN_CYCLE == 4) ->
   atomic {
    travelling_from[PRINCES_STREET].going_to[LOTHIAN_ROAD] = true;
    travelling_from[LOTHIAN_ROAD].going_to[PRINCES_STREET] = true;
    travelling_from[LOTHIAN_ROAD].going_to[RUTLAND_STREET] = true;
    travelling_from[SHANDWICK_PLACE].going_to[PRINCES_STREET] = true;
    travelling_from[SHANDWICK_PLACE].going_to[QUEENSFERRY_STREET] = true;
    travelling_from[QUEENSFERRY_STREET].going_to[HOPE_STREET] = true;
    travelling_from[QUEENSFERRY_STREET].going_to[LOTHIAN_ROAD] = false;
   }
  :: (stage%NUM_STAGES_IN_CYCLE == 5) ->
   atomic  {
    travelling_from[PRINCES_STREET].going_to[LOTHIAN_ROAD] = true;
    travelling_from[LOTHIAN_ROAD].going_to[PRINCES_STREET] = true;
    travelling_from[LOTHIAN_ROAD].going_to[HOPE_STREET] = true;
    travelling_from[LOTHIAN_ROAD].going_to[QUEENSFERRY_STREET] = true;
    travelling_from[LOTHIAN_ROAD].going_to[SHANDWICK_PLACE] = true;
```

```
    travelling_from[LOTHIAN_ROAD].going_to[RUTLAND_STREET] = true;
    travelling_from[HOPE_STREET].going_to[PRINCES_STREET] = false;
   }
  :: else -> {
    printf("ERROR: unknown stage %d.\n", stage);
    assert(false);
   }
  fi;
  stopTraffic();
  atomic { skip;
   printf("TRAFFIC_INFO lights stopped\n");
  }
  stage++;
  if
  ::stage<=6 -> goto restart_cycle;
  ::else skip;
  fi;
}
```

NeverUTurnBasic

```
Route someRoute;

active proctype setupTraffic() {


  setRouteRandomly(someRoute);

  setupTrafficDone = true;

  waitForSetupDone();

  printRoute(TRAFFIC_INFO_CHECKING, someRoute);

}

type = { TRAFFIC_ERROR__U_Turn_Allowed };

#define uTurnHasGreenLight

  ((someRoute.from == someRoute.to) && (LIGHT_FOR_ROUTE(someRoute)==GREEN))

never {

  waitForSetupDone();

  do

  :: uTurnHasGreenLight

        -> break;

  :: else -> skip;

  od;

  printRoute(TRAFFIC_ERROR__U_Turn_Allowed, someRoute);


}
```

rutlandOneway

```
int randomRoad;

active proctype setupTraffic() {

 setRoadRandomly(randomRoad);

 setupTrafficDone = true;

}

#define rutlandOneWay \

   (travelling_from[RUTLAND_STREET].going_to[randomRoad] == GREEN)

never {

T0_init:

      if

      :: ((rutlandOneWay)) -> goto accept_all

      :: (1) -> goto T0_init

      fi;

accept_all:

      skip

}
```

NeverDisallowed

```
int randomRoad;

active proctype setupTraffic() {

 setRoadRandomly(randomRoad);

 setupTrafficDone = true;

}

#define neverDisallowed \

   (travelling_from[randomRoad].going_to[randomRoad] == GREEN ||
travelling_from[QUEENSFERRY_STREET].going_to[PRINCES_STREET] == GREEN ||
travelling_from[QUEENSFERRY_STREET].going_to[SHANDWICK_PLACE] == GREEN ||
travelling_from[QUEENSFERRY_STREET].going_to[LOTHIAN_ROAD] == GREEN ||
travelling_from[QUEENSFERRY_STREET].going_to[RUTLAND_STREET] == GREEN ||
travelling_from[HOPE_STREET].going_to[PRINCES_STREET] == GREEN ||
travelling_from[HOPE_STREET].going_to[QUEENSFERRY_STREET] == GREEN ||
travelling_from[HOPE_STREET].going_to[RUTLAND_STREET] == GREEN
||travelling_from[SHANDWICK_PLACE].going_to[RUTLAND_STREET] == GREEN )
```

```
never {
T0_init:
	if
	:: ((neverDisallowed)) -> goto accept_all
	:: (1) -> goto T0_init
	fi;
accept_all:
	skip
}
```