Virtual Traffic Lights+ (VTL+): A Robust, Practical, and Functionally-Safe Intelligent Transportation System

ROOPAK SINHA
Department of Electrical and Computer Engineering, University of Auckland
Address: Room 301.227, 38 Princes Street, Auckland, New Zealand 1142
Phone: +64 9 373 7599 ext 84795
Fax: +64 9 373 7461
Email: r.sinha@auckland.ac.nz

PARTHA S ROOP
Department of Electrical and Computer Engineering, University of Auckland
Address: Room 303.152, 38 Princes Street, Auckland, New Zealand 1142
Phone: +64 9 373 7599 ext 88158
Fax: +64 9 373 7461
Email: p.roop@auckland.ac.nz

PRAKASH RANJITKAR
Department of Civil and Environmental Engineering, University of Auckland
Address: Room 401.1214, 20 Symonds Street, Auckland, New Zealand 1142
Phone: +64 9 373 7599 ext 83513
Fax: +64 9 373 7462
Email: p.ranjitkar@auckland.ac.nz

Submission size: 5988 words + 5 figures + 1 table

Abstract

The latest advancements in intelligent transportation systems (ITS) rely increasingly on wireless vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications to dynamically manage traffic flows on intersections. A prominent example is virtual traffic lights (*VTL*) that uses only V2V communications, and it has been shown to have potential for significantly increasing traffic flows and reducing emissions. Two key issues that can affect the adoption of desirable ITS solutions like *VTL* are *functional safety analysis*, and the management of a move from a completely non *VTL*-equipped vehicle fleet to a completely equipped vehicle fleet.

For the first issue, we propose the first model-driven engineering based modelling and verification technique for ITS, which can be used to prove functional safety with 100% coverage. Using this technique, we show that although *VTL* is safe under normal circumstances, it is very fragile when faced with unlikely, but not impossible, exceptional circumstances. For the second issue, we propose an extended algorithm called *VTL+* that uses additional V2I communication with existing infrastructure to enable effective and safe traffic flow during the *VTL* transition phase. We also find, through static analysis, that *VTL+* is more robust, and more feature-rich than *VTL*.

1. Introduction

It is critical to find social, economic or technological solutions to mitigate the negative environmental and health effects of greenhouse gas (GHG) emissions. A promising and cost-effective *left-field* solution is to radically change the way that surface traffic is controlled by using *Intelligent Transportation Systems* (ITS) that apply information and communication technologies to enhance surface transportation (1, 2, 3). Of the many enabling technologies in ITS, vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications (4) are finding increasing use not only in ITS, but also in the broader field of information and communications technology (ICT) in general (5).

Virtual traffic lights (3) (*VTL*) is a recently proposed futuristic ITS system where road-side infrastructure is replaced by in-vehicle equipment allowing cars to detect and avoid collisions at any intersection using only V2V communications. This proposal claims to achieve substantial increase in traffic flows (up to 60%) while also achieving significant emissions reduction (up to 18%). These results are demonstrated through the use of micro-simulation models of the proposed algorithm (3).

Two major issues regarding the eventual use of radical ITS proposals like *VTL* are (a) ensuring *functional safety* (6, 7), and (b) allowing *seamless adoption*. These two issues are the primary focus of the research presented in this article.

Functional safety analysis requires guaranteeing that an ITS (like *VTL*) will always allow safe (conflict-free) traffic flow. Currently, such assurances are derived from extensive simulation data obtained from a micro, meso and/or macro simulation (8). A simulation consists of providing a model of a system under test (SUT) with specific input combinations, and analyzing the outputs produced by the model. Unfortunately, even for a very small system, the total number of input combinations to be checked is prohibitively high. E.g., simulating *VTL* for every possible combination of static and dynamic parameters (vehicle speeds, positions etc.) at an intersection is not possible. In order to keep simulation-based testing to within manageable time, designers carefully choose a (very) few *representative* input combinations to simulate SUT behaviour. Since 100% coverage cannot be guaranteed, it is possible that a bug remains undetected (9), and can therefore undermine the safety of the ITS when it is deployed. It is essential to find alternative methods to ensure the functional safety of a new ITS like *VTL* before it can be adopted.

The other major issue with the use of new ITS solutions like *VTL* is their adoption. *VTL* requires every car on the road network to be equipped with V2V functionality, lane-level GPS, and on-board traffic light indicators (3). Such change cannot happen overnight on the existing fleet of cars, and until every car has been equipped, any installed *VTL* equipment will be useless. Moreover, any operational failure of on-board V2V equipment can be disastrous. This major bottleneck needs to be addressed, so that the transition from a completely non equipped to a completely *VTL*-equipped fleet is smooth, and functionally safe.

In this paper, we propose the first technique to model and analyze *VTL* for functional safety so that 100% coverage can be achieved. We use a *model-driven engineering* (MDE) process similar to those used to model and analyze safety-critical control systems such as aircraft software (10). In order to aid adoption of *VTL*, we propose *VTL+*, that can enable a partially *VTL*-enabled fleet to operate with the remaining non-equipped fleet using existing infrastructure (loops, traffic lights, pedestrian control buttons etc.). Moreover, it allows pedestrians to request and cross an intersection in a timely manner in any situation.

The main contributions of this article are summarised as follows:

1. We use MDE to graphically model the *VTL* system and generate executable code automatically. Our modelling and verification methodology is general enough to be applicable not only to *VTL* but also in the design of any other ITS system that needs to be functionally safe. To the best of our knowledge, the proposed approach lays the foundation for the first MDE-based ITS solution.
2. We use the BLAST verifier (11) to ensure the correctness of the *VTL* algorithm. We show that although under normal circumstances *VTL* is functionally safe, it is very fragile due to its over reliance on V2V communications.
3. We propose, model and verify *VTL+*, an extended *VTL* algorithm that is useful in managing traffic flow on intersections when only a part of the vehicle fleet is *VTL*-equipped. We show using verification that *VTL+* is more robust and offers more features (such as pedestrian support) than *VTL*.

The rest of this article is organized as follows. A discussion on the applicability of MDE to ITS design, and an overview of our design process appears in Sec. **Error! Reference source not found.**. Sec. **Error! Reference source not found.** provides an overview of *VTL*, and then describes how it is modelled and verified in our setting. Sec. 5 introduces *VTL+* and shows how it can also be modelled and verified in a similar fashion to *VTL*. Sec. 6 presents some interesting verification results. Finally, concluding remarks and future directions appear in Sec. 7.

2. Model-driven Engineering for ITS

An ITS like *VTL* is similar in complexity to a safety-critical distributed control system such as aircraft software. Moreover, both share identical concerns regarding functional safety. Complex control systems like aircraft software are typically designed using a precise model-driven engineering (MDE) process (10). MDE is founded on sound mathematical models based tools tools like SCADE (used to design Airbus A380) (12). These tools are ideal for modelling complex, distributed and safety-critical control systems. Moreover, they allow systems to be *statically analyzed*, where a program can be analyzed for safety properties without having to exhaustively execute or simulate it. Static analysis techniques include functional analysis (13), timing analysis (14), stack analysis etc. MDE follows a precise and automatic transformation of designs using sophisticated compilers into lower level executable code (such as C programs) which implicitly preserves the safety characteristics of the translated designs.

Mathematical models are used extensively in transportation research to primarily model the behaviour of vehicles, humans (drivers and pedestrians), and infrastructure (15), but are used in a very limited manner for traffic controller modelling. Holistic ITS modelling requires a combination of such precise plant models with precise models for control algorithms. Such holistic modelling is currently lacking and the proposed MDE approach allows for the use of such holistic modelling.

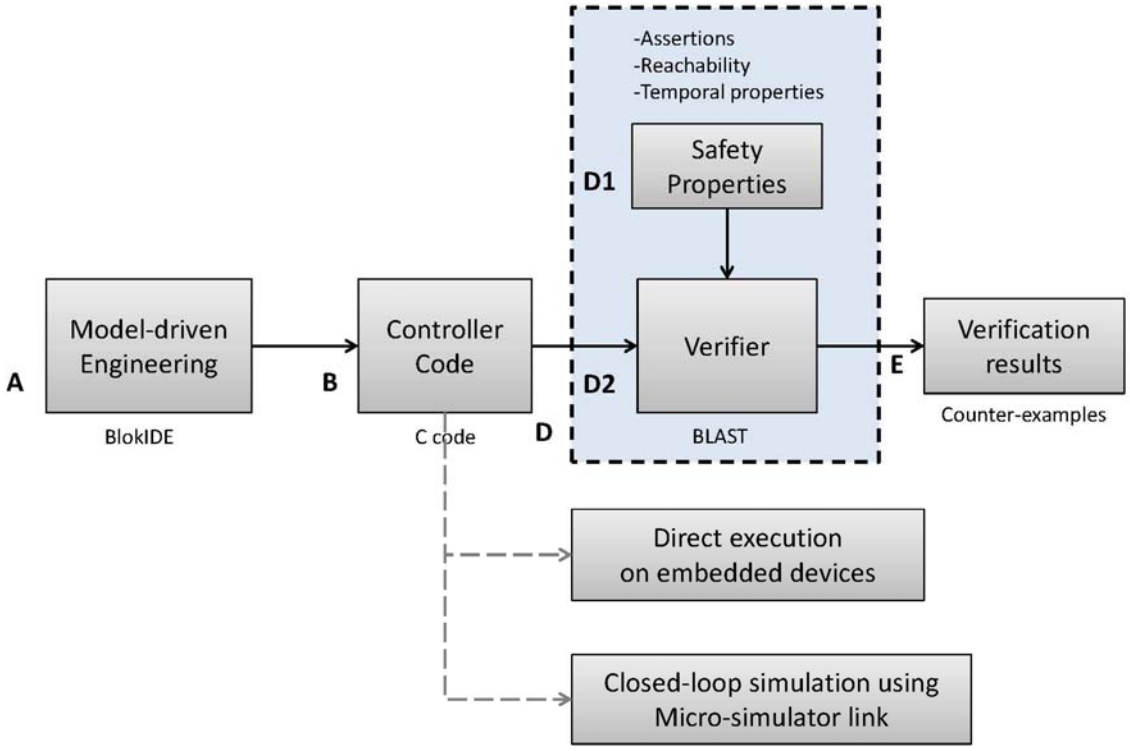


FIGURE 1: Overview of the proposed approach

An overview of our approach is provided in Fig. 1. Firstly, an MDE approach using IEC 61499 (16), the de-facto standard for designing distributed control systems, is used to develop a model of the *VTL* algorithm (Fig. 1(A)). We utilize the BlokIDE software (similar to the SCADE (12) compiler for flight control software) for the MDE of *VTL*, which uses the synchronous compiler for IEC 61499 (17). This model can be automatically translated into executable code (in the C language) that can be readily deployed or analyzed further.

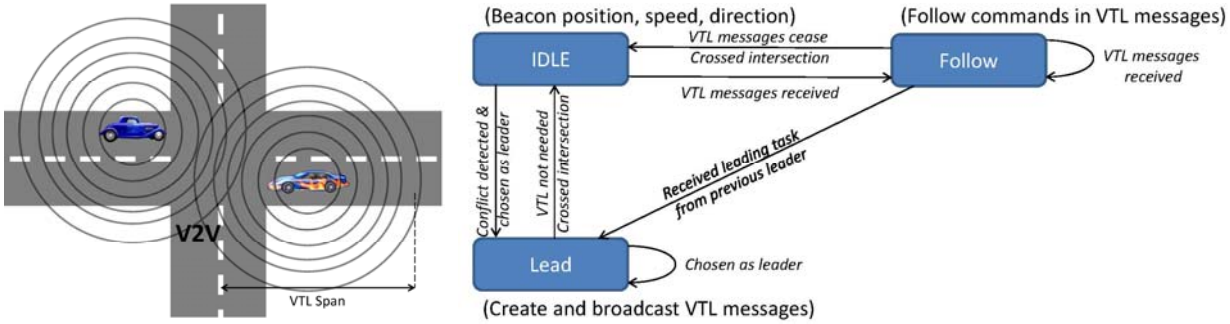
In order to check if the behaviour of the *VTL* controller is functionally safe, we use formal methods (9) (Fig. 1(D)), a collection of verification and validation (V&V) techniques that allow for the modelling of SUTs using precise mathematical models, and the manipulation of these models in a mathematical framework to provide 100% coverage of the analysis of safety-critical specifications. In our approach, we use the BLAST automatic verifier, which has been shown to verify large C programs efficiently (11). During verification, the designer provides some functional properties that the controller model must satisfy (Fig. 1(D1)). The controller code along with these properties is then provided as input to the BLAST verifier (Fig. 1(D2)), which automatically verifies if the given properties are satisfied by the model. In case a property is not satisfied, BLAST generates a *counter-example*, or an execution path in the C code that can be explored to locate the cause of the failure.

3. Modelling and Verification of Virtual Traffic Lights

3.1. Overview of Virtual Traffic Lights

Virtual Traffic Lights (*VTL*) (3, 18) use a dynamic traffic control plan (DTCP) to allow multiple vehicles to travel on a road network. Vehicle-to-vehicle (V2V) communication between individual vehicles, as shown in Fig. 2, allows the resolution of conflicts on intersections by the formation of

a negotiated travel plan. Each vehicle is required to be *VTL*-equipped, which means that it must have a GPS with lane-level accuracy to pinpoint its position, network capabilities to broadcast its parameters and communicate with other vehicles, an in-vehicle traffic-lights panel to convey the DTCP to the driver, and enough computing power to create a DTCP plan.



(a) *VTL* Layout (b) *VTL* algorithm

FIGURE 2: Virtual Traffic Lights

The main point of interest in *VTL* is a potential conflict-zone, such as an intersection, as shown in Fig. 2. For simplicity of illustration, we consider an intersection that does not allow turning. E.g., in Fig. 2, traffic can travel straight in the north-south (NS), south-north (SN), east-west (EW) or west-east (WE) directions only. The *VTL* algorithm operates as follows. Each travelling vehicle can be in one of three *VTL* modes: `IDLE` (no conflict), `Lead`, or `Follow`, as shown in Fig. **Error! Reference source not found.** which describe its behaviour when no *VTL* is present at an intersection, when it is leading the control of a *VTL*, or when it is following *VTL* commands from another leader, respectively.

In the `IDLE` mode, the vehicle periodically broadcasts its location and other parameters using appropriate beaconing methods (dictated by the network protocol used). Note that the range of the broadcast depends on the wireless network used, and can be reduced by the presence of signal barriers (e.g. buildings). In *VTL*, the *span* of an intersection is the distance (in metres) from the centre of the intersection such that all vehicles within the span of the intersection can communicate using V2V.

A conflict on an intersection is detected when two vehicles approaching an intersection may collide (if they continue on their paths). A conflict may be detected by one or more individual vehicles and results in the choosing of a *VTL*-leader. The leader is typically the first vehicle in the furthest cluster of vehicles approaching the intersection. Also, according to the *VTL*-protocol, the leader direction is shown the red light while the other conflicting direction is shown the green light. If a vehicle is chosen as the leader, it moves into its `Lead` mode, whereas all other vehicles within the span of the intersection enter their respective `Follow` modes.

In the `Lead` mode, the leader stops at the intersection and broadcasts *VTL*-messages to all other vehicles at the intersection. Once a time-out occurs, or if all conflicts have been resolved, the leader direction is shown the green light. The leader may then hand-over *VTL* leadership to another following vehicle and then crosses the intersection (and enters its `IDLE`) mode. Note that a leader cannot become a following vehicle on the same intersection - it must carry out the DTCP until it crosses the intersection.

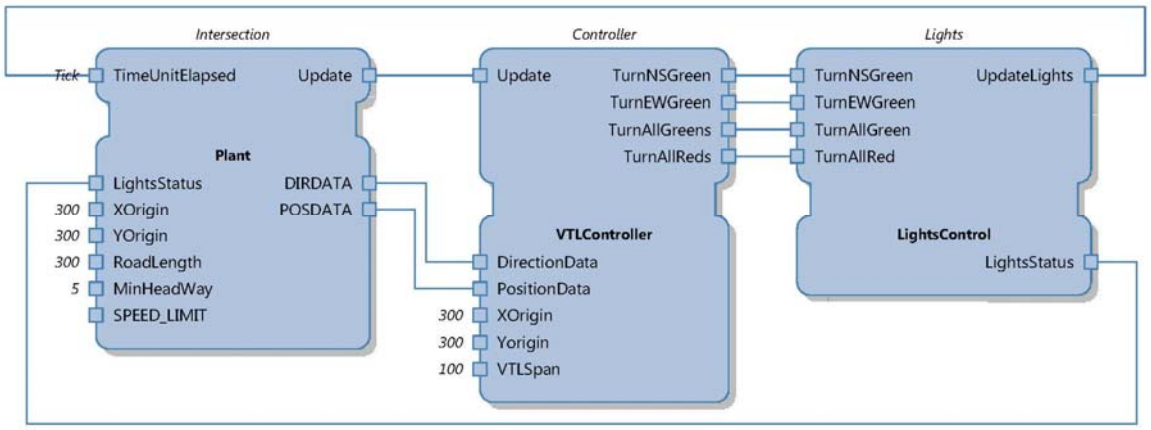
In the `Follow` mode, a vehicle receives and follows *VTL* messages received from the leader. If *VTL* messages cease (no conflicts remain) or if the vehicle crosses the intersection, it moves back into its `IDLE` mode. On the other hand, if leadership is handed over (by the previous leader), the vehicle moves into the `Lead` mode.

In order to ensure that the *VTL* algorithm works correctly, each vehicle (in any mode) must fulfil its role in the *VTL* without fail and within acceptable time limits. For simplicity, we assume that vehicles must respond within 1 second (a more accurate value can be obtained by timing analysis techniques).

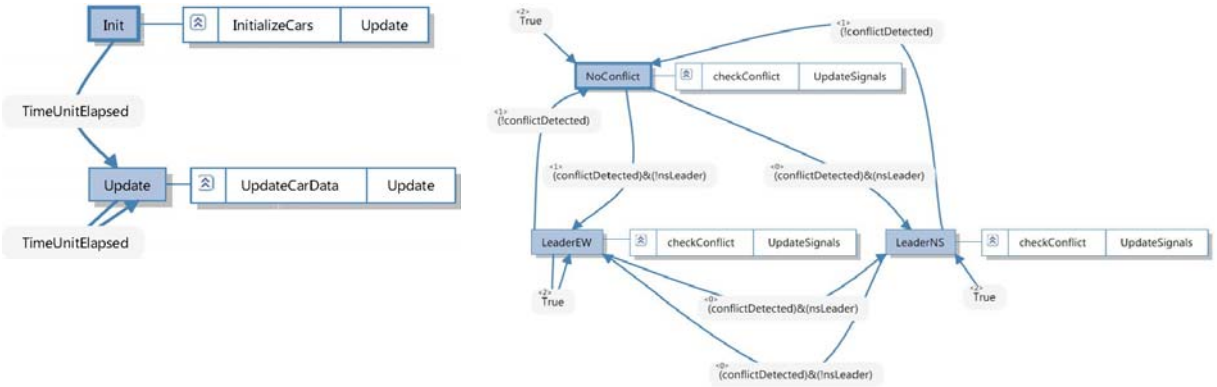
3.2. *VTL* Modelling

Intelligent transportation systems such *VTL* are akin to safety-critical distributed systems, containing multiple processing elements interacting with each other (e.g. vehicles using V2V communication). In this paper, we model *VTL* as an (16) system using the B software built around the synchronous compiler for IEC 61499 (17). We follow the model-view-controller (MVC) (19) design framework for model-drive engineering (20) where we design the road network (plant), a view (traffic lights output), and a controller (*VTL* algorithm) together as one system. MVC allows a good separation of these three orthogonal part of the design, allowing designers to refine each part (relatively) exclusively of others.

IEC 61499 has a number of benefits being based on distributed intelligence and object-oriented design concepts such as modularity, encapsulation, hierarchy as well as a precise model that can be analyzed using formal methods (21). Individual elements in an IEC 61499 system are designed visually and are called *function blocks*. A function block has a well-defined interface which describes its input events, output events, input variables, and output variables. A block can one be of two types: *composite* and *basic*. *Composite* function blocks are networks of blocks. Fig. 3 shows a simple 1-intersection *VTL* model, designed as a composite function block containing three function blocks `Plant`, `VTLController` , and `LightsControl` (model, controller and view respectively as in MVC). Blocks in a network communicate using explicit connections. For example, output event `Update` and output variable `DIRDATA` of `Plant` are connected to the input event `Update` and input variable `DirectionData` of `VTLController` respectively.



(a) Top-level network of the *VTL* model



(b) ECC for `Plant` (c) ECC for `VTLController`

FIGURE 3: *VTL* Network and execution control charts (ECCs).

Basic function blocks execute using a state machine called an execution control chart (ECC). The three blocks contained in the network shown in Fig. 3 are basic function blocks. Fig. **Error! Reference source not found.** shows the ECC of the block `Plant`. An ECC is a *state machine* containing a finite number of states. For example, the ECC for `Plant` contains 2 states *Init* and *Update* with *Init* being the initial state. Each state is associated with a sequence of actions that is executed when the state is entered. E.g,, when state `Init` shown in Fig. **Error! Reference source not found.** is entered, the algorithm `InitializeCars` is executed and the output event `Update` is emitted. A transition from the current state to a new state in an ECC triggers when an input event is present, or a boolean condition over variables evaluates to true (or both). E.g., the transition from state `Init` to state `Update` triggers when the input event `TimeUnitElapsed` is present.

3.3. Details of the *VTL* Model

As shown in Fig. 3, the *VTL* design for a single intersection contains three networked function blocks: a plant, a controller, and a traffic-lights display.

The plant block *`Plant`* relates to the model of the intersection. Each intersection has an origin - (X,Y) coordinates represented by the input variables `XOrigin` and `YOrigin`. Other inputs include the minimum headway, and the speed limit at the intersection. Intuitively, the `Plant` block operates as follows. First, a random number of vehicles (cars) are created and their positions are initialized (in state `Init`), and then regularly updated (in state *`Update`* whenever input event `TimeUnitElapsed` is present). The vehicle location data is emitted using variables `DIRDATA` and `POSDATA`. Typically, in transportation systems design, a *car-following* model is used to model the movement of vehicles (22). `Plant` includes a simplistic car-following model where cars move in single file while trying to maintain a minimum distance from the car in front. A degree of randomness embedded in the car-following model and the cars-initialization allows us to create a realistic plant model to be controlled by the *VTL* controller.

The *VTL* controller, represented by the block `VTLController` operates as per the ECC shown in Fig. **Error! Reference source not found.**. From its initial state *`NoConflict`*, whenever a conflict is detected (by executing the algorithm `checkConflict`), it moves to either state `LeaderNS` (leader in north/south approach) or `LeaderEW` (leader in east/west approach).

At each state, the appropriate *VTL* messages are broadcast as output signals `TurnAllGreens`, `TurnEWGreen` and `TurnNSGreen`. The `VTLController` assumes a fixed *VTL* span, represented by the variable `VTLSpan` (set to be 100 metres in the example shown in Fig. 3).

Finally, the `LightsControl` model simply reads *VTL* messages from `VTLController` and emits them to `Plant` (as shown by the feedback connections in Fig. 3).

The *VTL* model shown in Fig. 3 can be instantiated multiple times to create a multi-intersection system. This model is therefore *parametric* as any finite number of intersections (with individual origins, respective spans, speed limits etc.) can be created. In order to allow these individual *VTL* systems to communicate with one another, we simply allow them to share the car position and direction data (output variables `DIRDATA` and `POSDATA` of their respective plant models) amongst each other.

The BlokIDE synchronous compiler (17) generates code that executes using a *synchronous* semantics, under which system execution is divided in discrete steps, known as *ticks*. During a tick, each element in a distributed system is allowed to execute once, in a precise order, in order to guarantee determinism of execution. For example, the network shown in Fig. 3 executes the blocks in the order `Plant`, `VTLController` and `LightsControl` every tick. We determine the length of the tick to be equivalent to 1-second so that during execution, each block can communicate updated data every second (as required by *VTL*).

BlokIDE allows efficient generation of executable C code using the synchronous compiler, as well as basic (text-based) simulation facilities to test designs. Moreover, the proposed MDE approach also allows for integration with micro-simulators. For example, we are working on integrating BlokIDE with the AIMSUN micro-simulator (23) so that plant models in AIMSUN can be used in conjunction with a controller model in BlokIDE.

3.4. Verification using BLAST

The Berkeley Lazy Abstraction Software Verification Tool, or BLAST (11), is an automatic verifier of C programs. It can statically analyze C programs (without executing the code) and decides if a given safety property is satisfied or not. In case a property is not satisfied, BLAST provides a trace in the execution of the C program along which the property fails. We use BLAST to verify the *VTL* system.

The performance of most automatic verifiers suffers due to the *state explosion* problem (9). As the number of data variables within individual processing element of a system and the number of components increase, verification time increases exponentially. BLAST prevents state explosion by using *counter-example guided abstraction refinement* (CEGAR) where, beginning with a very course abstraction of the model to be verified, successively more detailed abstractions are automatically created only if needed to prove/disprove the given specification. Consequently, BLAST has been used to analyze large programs with more than 30,000 *lines of code* (loc). The C code generated from the *VTL* system (modelled in BlokIDE) is around 2200 loc, and BLAST is therefore well-suited to verify this model.

BLAST requires users to specify the safety properties to be verified. We use the following types of properties during the verification of the *VTL* system (shown in Fig. **Error! Reference source not found.**):

```
...
        if (car_dist < safe_dist)
        {
                DIST_ERR: goto DIST_ERR;
        }
...
```

(a) Reachability property (inserted in original C file)

```
...
        OutOfBounds: assert (car_pos[i]<X_MAX);
...
```

(b) Assertion property (inserted in original C file)

```
1: global int carsOnIntersection = 0;
2: event {
3:       pattern { UpdateCarData(); }
4:       action {carsOnIntersection = 0; }
5: }
6: event {
7:       pattern { car_at_origin(); }
8:       guard {carsOnIntersection == 0 }
9:       action {carsOnIntersection = 1; }
10:}
```

(c) Temporal property (exlusive of original C file)

FIGURE 4: Types of properties used to verify *VTL*.

1. *Reachability*: A reachability property requires verifying if a certain control point (a label) in a given C program can be possibly reached. Fig. 4 shows the code label `DIST_ERR` inserted in the C program for the *VTL* system, which is reached in the program if the distance between any two cars near the intersection is less than the minimum distance between them. If the label `DIST_ERR` is found to be reachable by BLAST, the corresponding safety property fails.
2. *Assertions*: An assertion property checks if a condition, evaluated based on the value of program variables, always evaluates to true at a specific control point (label). Fig. **Error! Reference source not found.** the label `OutOfBounds` (inserted in the *VTL* C program) which is used to assert the condition where a vehicle can go beyond the physical boundaries imposed by the plant (road) model. BLAST statically checks the program and verifies if this assertion if always true, i.e., cars never go out of bounds.
3. *Temporal properties*: A temporal property describes a desirable (or undesirable) property of the model being verified over time. Consider, for example, the specification shown in Fig. **Error! Reference source not found.** (written as a separate specification file). Intuitively, it checks if more than one car can be at the origin of the *VTL* intersection at any one time. On line 1, the variable `carsOnIntersection` is initialized to 0. Lines 2–5 describe an BLAST *event* where whenever the function `UpdateCarData()` is called, `carsOnIntersection` is reset to 0. Lines 6–10 describe an event where whenever the function `car_at_origin()` is called (indicating that a car is crossing), `carsOnIntersection` is set to 1 *if and only if* the value of `carsOnIntersection` was previously 0 (line 9). The above property fails when the function `car_at_origin()` is called and the value of `carsOnIntersection` is not 0. This can only happen when

during the same call to `UpdateCarData()`, more than one car is at the intersection (a collision happens).

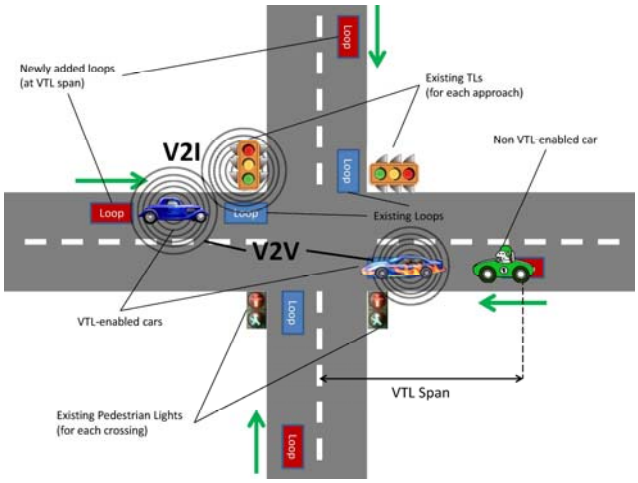
Using the above property-types, we verify a number of key safety properties of *VTL* including freedom from collisions, fairness (each car can eventually pass an intersection it intends to cross) etc. More information about verification results appears in Sec. 6.

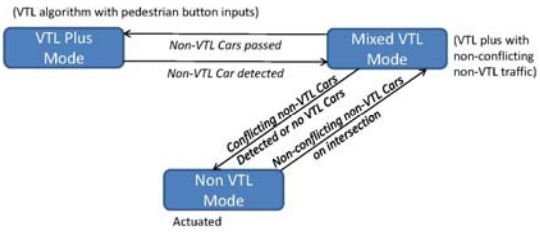
5 *VTL*$^+$

Although *VTL* has potential to reduce emissions and increase traffic flow (3), its adoption requires a drastic change in the current fleet of vehicles. *VTL*, in its proposed form, can work only when every single vehicle on a road network is *VTL*-equipped: a task that cannot be achieved overnight. In addition, the existing infrastructure present on the roads simply cannot be made obsolete overnight. Also, *VTL* in its proposed form does not take into account pedestrians. It is therefore desirable that (a) *VTL* adoption is made more gradual, (b) the existing infrastructure is used during and after the adoption of *VTL*, and (c) present pedestrian handling infrastructure is used within *VTL*.

In order to move towards a more acceptable *VTL* model, we propose *VTL+* which extends *VTL* functionality by using it with existing infrastructure. The *VTL+* layout is shown in Fig. **Error! Reference source not found.**. A *VTL+*-enabled intersection allows both *VTL*-enabled and non *VTL*-enabled cars to use the intersection. For a previously actuated intersection, *VTL+* uses existing loops located near the centre of the intersection. It also uses existing pedestrian lights and buttons to allow pedestrian traffic flow through the intersection. Existing traffic lights are retained to broadcast the intersection traffic flow plan.

A *VTL+* intersection, as shown in Fig. 5, requires some additional (but minimal) infrastructure. Firstly, the existing traffic and pedestrian lights controller should be able to communicate with *VTL*-enabled vehicles using V2I communication. In addition, we require additional loops to be placed on road segments leading to the intersection, so that each loop is exactly as far as the *VTL* span for the intersection. These additional loops allow the *VTL+* algorithm to sense the entry of a non-*VTL* car in the vicinity of the intersection. Even though this extra equipment means extra costs, the benefits of *VTL+*, i.e., a more gradual adoption of *VTL*, use of existing infrastructure, and pedestrian traffic handling outweigh them. Note that not all intersections need to be modified in this manner - any unmodified intersection will continue to be actuated, driven by give-way and/or stop signs, etc., until full *VTL* adoption is complete.





(a) *VTL+* Layout (b) *VTL+* Algorithm

FiGURE 5: *VTL+* overview

The operation of a *VTL+* intersection is described using Fig. **Error! Reference source not found.**. An intersection can be in one of the following modes at any instance of time:

1. *VTL Plus Mode*: This mode is operational when only *VTL* cars are present within the span of the intersection. In this mode, the *VTL*-algorithm shown in Fig. **Error! Reference source not found.** is executed by the cars using V2V communication. However, this algorithm is modified to take into account pedestrian button requests (through the V2I link). Also, the travel plan created by the leader is broadcast on the physical traffic lights via this V2I link.
2. *Mixed VTL Mode*: This mode is operational when both *VTL* and non-*VTL* vehicles are detected within the span of the intersection. Non *VTL* traffic is detected using the newly added loops, as shown in Fig. 5, and these vehicles must be moving in non-conflicting directions to each other. In order to avoid potential collisions, the *VTL+* algorithm marks the approaches on which non-*VTL* vehicles have been detected as active. Non *VTL*-enabled vehicles receive directions generated by the *VTL* leader from the physical traffic lights (via the V2I link). When a non-*VTL* vehicle travelling in a conflicting direction to another non-*VTL* vehicle enters the intersection span, the controller moves to the non VTL mode. Otherwise, when all non-*VTL* cars leave the intersection, the algorithm switches to VTL Plus mode. A non-*VTL* car is considered to have left a safe time (2-3 seconds) after it passes the intersection-side loop on its approach.
3. *Non VTL Mode*: This mode operates when there are no *VTL* cars within the span of the intersection or when conflicting non-*VTL* cars are detected. In this mode, the intersection works in an actuated mode (using the existing loops). The newly added loops may be additionally used to enhance the actuated intersection control by helping in predicting queue lengths on the approaches.

5.1 Modelling and Verification of *VTL+*

As with *VTL*, we use BlokIDE to create a model of the *VTL+* algorithm. A function block network similar to Fig. 3 is used. However, the block `Plant` now initializes and maintains some cars as non-*VTL* cars. The `VTLController` block is modified so that it executes the three modes presented in Fig. **Error! Reference source not found.** depending on the combination of the *VTL* and non *VTL* enabled cars read from `Plant`. Like the *VTL* model, the *VTL+* model can also be extended to multiple intersections by instantiating the *VTL+* intersection model at different origins.

The code generation and verification of *VTL+* follows a process that is identical to the one for *VTL*. First, BlokIDE is used to generate C code, and the C code is verified in BLAST using reachability, assertions, and temporal properties shown in Sec. 3.4. The verification results are discussed in the next section.

6 Results

TABLE 1: Verification results for *VTL* and *VTL+*

No.	Property	Type	*VTL*	*VTL+*
1.	Max. 1 car on intersection	Temporal	Passed	Passed
2.	Cars stay within bounds	Reachability	Passed	Passed
3.	Cars maintain minimum distance during unrestricted course of travel	Assertion	Passed	Passed
4.	Each direction shown green within 180 seconds	Temporal	Passed	Passed

5.	No collisions during network failure	Temporal	Failed	Passed
6.	No collisions if cars don't follow speed limits	Temporal	Failed	Passed
7.	Pedestrians can cross within 180 seconds of request	Temporal	–	Passed

Tab. 1 shows some interesting properties that were used to verify both the *VTL* and the *VTL+* models. The first three columns of every row present a number, description and type of property used, respectively. The fourth and fifth columns describe whether the property passed or failed on the *VTL* and *VTL+* models, respectively.

A number of observations can be made from the results shown in Tab. 1:

1. The properties (1–6) used to verify *VTL* could be reused without modification to verify *VTL+*. This was possible due to the almost identical structure of these models.
2. For some properties, only specific parts of the two models (individual functions in the C code) were needed to be verified. E.g., for properties 2 and 3 (cars remain within bounds and maintain distance), only the plant model (block `Plant` in Fig. 3) was verified.
3. In addition to safety properties (e.g. 1–3), we were able to test *fairness* of the models (e.g. properties 4 and 7). Fairness relates to ensuring that each vehicle or pedestrian requesting access to the intersection always eventually gains access.
4. Properties 5–6, which model abnormal (but not impossible) behaviour in a V2V situation, such as network failure, clearly show the fragility of *VTL*. For example, if one of the vehicles near the intersection loses network access due to a failure in its *VTL* equipment, *VTL* is unable to guarantee safe passage for all vehicles. On the other hand, whenever an exceptional behaviour is experienced, *VTL+* simply switches to Mixed or Non VTL mode, depending on its severity.
5. *VTL+* provides more functionality than *VTL*, as shown by property 7. While *VTL* has no means of tracking pedestrians, *VTL+* allows pedestrians to request and gain access under any mode.

7 Concluding Remarks

This article presents the first holistic model-driven engineering design technique to model and verify intelligent transportation systems. We choose the virtual traffic lights (*VTL*) algorithm (3) as a case study and verify its functional safety. We find that even though it is functionally safe under normal circumstances, *VTL* is fragile under non-trivial, but possible, exceptional behaviour. We also propose *VTL+*, an extension to *VTL* which can enable a smoother and realistic transition to a full *VTL* adoption by enabling the use of existing road-side infrastructure to facilitate safe traffic flow between *VTL*-equipped and non-equipped vehicles. We show that *VTL+* is more robust than *VTL* due to the availability of a degree of redundancy.

Future directions for this work include formally assessing *VTL* and *VTL+* for quantitative measures such as traffic flow improvements and emissions reductions. In addition, we also endeavor to integrate these algorithms with a micro-simulator such as (23) to further assess their effectiveness.

References

[1] Vahidi, A. and A. Eskandarian, Research advances in intelligent collision avoidance and adaptive cruise control. *Intelligent Transportation Systems, IEEE Transactions on*, Vol. 4, No. 3, 2003, pp. 143 – 153.

[2] Weiland, R. J. and L. B. Purser, Intelligent Transportation Systems. In *Transportation in the New Millennium*, 2000, p. 3.
[3] Ferreira, M. and P. M. dÓrey, On the Impact of Virtual Traffic Lights on Carbon Emissions Mitigation. *IEEE Transactions on Intelligent Transportation Systems*, Vol. PP, No. 99, 2011, pp. 1–12.
[4] Cho, W., S. I. Kim, H. kyun Choi, H. S. Oh, and D. Y. Kwak, Performance evaluation of V2V/V2I communications: The effect of midamble insertion. In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology, 2009. Wireless VITAE 2009. 1st International Conference on*, 2009, pp. 793 –797.
[5] Gifford, J. L., ICT and road transportation safety in the United States: a case of American exceptionalism. *IATSS Research*, Vol. 34, No. 1, 2010, pp. 1 – 8.
[6] Panesar-Walawege, R., M. Sabetzadeh, and L. Briand, Using Model-Driven Engineering for Managing Safety Evidence: Challenges, Vision and Experience. In *Software Certification (WoSoCER), 2011 First International Workshop on*, 2011, pp. 7 –12.
[7] Brown, S., Overview of IEC 61508. Design of electrical/electronic/programmable electronic safety-related systems. *Computing Control Engineering Journal*, Vol. 11, No. 1, 2000, pp. 6–12.
[8] Burghout, W., H. N. Koutsopoulos, and I. Andréasson, Hybrid Mesoscopic-Microscopic Traffic Simulation. *Transportation Research Record: Journal of the Transportation Research Board*, Vol. 1934, No. 1, 2005, pp. 218–255.
[9] Kern, C. and M. R. Greenstreet, Formal verification in hardware design: a survey. *ACM Trans. Des. Autom. Electron. Syst.*, Vol. 4, No. 2, 1999, pp. 123–193.
[10] Goupil, P., AIRBUS state of the art and practices on FDI and FTC in flight control system. *Control Engineering Practice*, Vol. 19, No. 6, 2011, pp. 524 – 539.
[11] Beyer, D., T. A. Henzinger, R. Jhala, and R. Majumdar, The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, Vol. 9, No. 5, 2007, pp. 505–525.
[12] *SCADE Tools*. http://www.esterel-technologies.com/, 2009, last accessed - 1.3.09.
[13] Clarke, E. M., O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2000.
[14] Roop, P. S., S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen, Tight WCRT analysis of synchronous C programs. In *CASES* (J. Henkel and S. Parameswaran, eds.), ACM, 2009, pp. 205–214.
[15] Ranjitkar, P., T. Nakatsuji, and A. Kawamura, Car-Following Models: An Experiment Based Benchmarking. *Journal of the Eastern Asia Society for Transportation Studies (EASTS 2005), Bangkok*, Vol. 6, 2005, pp. 1582–1596.
[16] International Electrotechnical Commission, *Committee Draft for Vote: IEC 61499-1 : Function Block - Part 1 Architecture*. International Electrotechnical Commission, Geneva, 2004.
[17] Yoong, L. H., P. Roop, V. Vyatkin, and Z. Salcic, A Synchronous Approach for IEC 61499 Function Block Implementation. *Computers, IEEE Transactions on*, Vol. 58, No. 12, 2009, pp. 1599 –1614.
[18] Ferreira, M. C. P., O. Tonguz, R. J. Fernandes, H. M. F. Da Conceicao, and W. Viriyasitavat, *Methods and systems for coordinating vehicular traffic using in-vehicle virtual traffic control signals enabled by vehicle-to-vehicle communications.* Patent Application, 2011.
[19] Krasner, G. E. and S. T. Pope, A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, Vol. 1, No. 3, 1988, pp. 26–49.

[20] Kent, S., Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, Springer-Verlag, London, UK, UK, 2002, IFM '02, pp. 286–298.

[21] Z. Bhatti, R. Sinha, and P. Roop, Observer based verification of IEC 61499 function blocks. In *9th IEEE International Conference on Industrial Informatics (INDIN)*, 2011, pp. 609–614.

[22] Gurusinghe, G. S., T. Nakatsuji, Y. Azuta, P. Ranjitkar, and Y. Tanaboriboon, Multiple Car-Following Data with Real-Time Kinematic Global Positioning System. *Transportation Research Record: Journal of the Transportation Research Board*, Vol. 1802, No. 1, 2002, pp. 166–180.

[23] Barceló, J. and J. Casas, Dynamic Network Simulation with AIMSUN. In *Simulation Approaches in Transportation Analysis* (R. Kitamura, M. Kuwahara, R. Sharda, and S. Voß, eds.), Springer US, Vol. 31 of *Operations Research/Computer Science Interfaces Series*, 2005, pp. 57–98.