# Modeling and simulating ERTMS/ETCS using VDM++

Christian M. Lillelund (201408354)

*School of Engineering, Aarhus University*

*Course: E18 - Modeling of Critical Systems*

## Abstract

Modeling of critical systems is an important tool in software engineering, as it allows us to design a system from specification and verify its properties in a unambiguous way early in the development process. In this report, we use the object-oriented Vienna Development Method (VDM++) to model, simulate and verify a subset of the specifications for the safety-critical European Rail Traffic Management System (ERTMS) level 2 focusing on those properties of ERTMS that concern interlocking of tracks and safety of the trains. In ERTMS, these features are controlled by the European Train Control System (ETCS) component. With VDM++, we can perform system design analysis and model verification to catch potential faults early before the system is constructed in an implementation language like Java and deployed to a real system. We start by defining requirements and invariants based on the ERTMS/ETCS specification and then use VDM++ to design a formal model followed by model-checking. The abstract implementation of ERTMS detailed in this report provide a simple but fundamental understanding of the signaling system's behavior and how to describe its constraints.

*Keywords:* ERTMS, ETCS, VDM++, Formal methods, Interlocking, Safety

## 1. Introduction

The railway domain was identified as a grand challenge of computing science in 2004 because it is understandable by the general public, provides useful features in terms of transportation and pose many concerns for design and controllability. One part of the challenge is improving the feasibility and capacity of modern railway as the world's population is increasing and rail traffic now moves cross borders.

The European Rail Traffic Management System (ERTMS) is a signaling and control system developed in the start 2000's to address the interoperability issues with cross-border rail traffic in Europe and lack of capacity with legacy systems. Currently many European countries have their own national stand-alone signaling and control system implemented by a certain set of rules that differ from each country. ERTMS is designed to replace the national systems to make rail transport more frictionless, improve rail capacity and more attractive to consumers. It consists of two primary parts, the European Train Control and Command System (ETCS) to govern the positions and safety of trains, known as the Interlocking (ITL), and the GSM-R radio communications system to send messages between trains and the Radio Block Center (RBC). There a three different levels of ERTMS, L1, L2 and L3. We consider L2 in this report. L2 introduces the RBC, a Eurobalise that register train movement and omits any track side signaling equipment. For a train to enter a new track section, it must requests a movement authority (MA) for that section from the RBC, which will be further explained in section X. ERTMS defines end of authority (EoA) as the point the train is allowed to move to. We consider this aspect as well.

Figure 1 shows the essential components of ERTMS level 2. The train control (EVC) requests movement authorities from the RBC over GSM-R. A typical national rail way will have multiple RBC's for each region. The RBC's communicate with a central interlocking service, that receives the physical location of trains using the Eurobalises, in order to either grant or deny movement authority to a train.
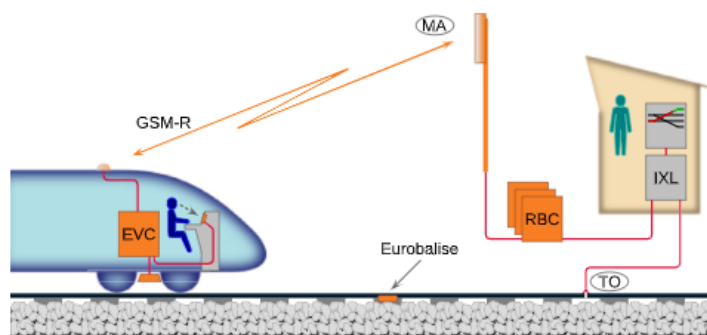


Figure 1: An overview of the ERTMS/ETCS level 2 components.

2

## 2. Requirements

The purpose of the model is to design and verify the interlocking properties and operating rules of the ETCS system, a part of ERTMS. With our model we aim to show how VDM++ can we used to create and satisfy a subset of the real requirements of ETCS interlocking, abstracting away most of the physical constraints you would have in a real system and focusing on the control mechanisms you find in the RadioBlockCenter and Interlocking components, described in the introduction. Our model should demonstrate a simulated train that interacts with the control system when traversing a set of tracks, as it would in real life, while simultaneously monitoring and adhering to the safety features of the system. We address the following requirements:

R1: A train shall not enter a track section which is occupied by another train (same track).
R2: A train shall not pass a track boundary without being given a movement authority (MA) to do so.
R3: A train shall respect the maximum permitted speed of its current track.
R4: Two trains cannot have a MA for the same track at the same time.
R5: The system shall not provide an MA for a track that is occupied by a train.
R6: The RBC shall not answer MA's for tracks it is not responsible for.
R7: The eurobalise shall report train movement to the interlocking system.

The requirements and the behavior of the ETCS control architecture, that we elaborate on in section 3, are inspired from literature X and X, that describe, model and verify many of the real functional properties of ERTMS. In this report we choose a portion of these to model and test our requirements *R1-R8*. UML will be used to show some of the logical VDM++ classes and their relationships. From the specification of ERTMS we can define a typical usage scenario for a train that wishes to enter a track. Given a train *T1*, a track TR1 from a set of tracks (i=1,..,n), an *RBC* responsible for tracks $TR_n$, a eurobalise *Eb* mounted on the track that registers physical movement and a central interlocking service *ITL*.

- *T1* wants to enter track *TR1*. *T1* requests MA for *TR1* from RBC.
- RBC contacts the ITL to verify the request.
- RBC grants MA to train *T1* for track TR1.
- *T1* enters track TR1.
- Eb registers movement and informs ITL.

This functionality will be further explored in the model architecture and design.

## 3. Model architecture

The architecture of the model represents the entire system as entities, that are later defined in VDM++ as classes, that encapsulate functionality related to the individual entity. We consider the purpose of our model, that is to express and simulate several important aspects of ERMTS/ETCS, and use the primary components of such system as our entities. These can seen in figure 2 that show a way to model the control architecture of ERTMS based in the real specification set for ERTMS level 2 detailed in X, but junction control and route guidance has been omitted.
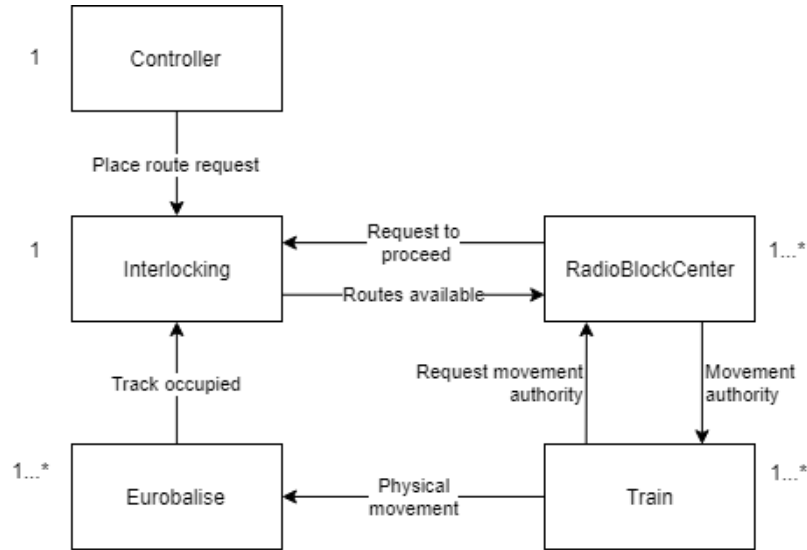


Figure 2: The ETCS control architecture partly from the real specification.

We shall describe the components in more detail before designing our model. The **controller** places route requests into the interlocking at certain time $t$ intervals. The point is to control the flow of traffic and how many routes can be available concurrently in the rail network, a way to limit congestion. A route is simply a set of tracks. **Interlocking** receives route requests and stores them in a control table containing available routes. It receives track occupation events from physical units along the track side and uses this information to update the control

table with available routes. The **RadioBlockCenter** receives MA requests from trains, checks route availability and forwards these to the interlocking. RBC's are responsible for multiple tracks, i.e. a block. The **Train** sends MA requests to the RBC before entering a new track in a route. For each track, a line segment with a geographic starting and terminal point, the train must request and obtain a MA. This is also stated in the requirements. Each track has a certain speed limit, and the train must never exceed this. The **Eurobalise** registers physical movement when trains enter and leave tracks and conveys this to the interlocking.

In a real scenario, the eurobalise sends track and speed information to passing trains and the ETCS would monitor this and eventually slow down trains, if they go too fast. We do not consider this part in our model. The ETCS architecture in figure 2, requirements and functionality will be used to create VDM++ classes, functions and invariants in the next section.

## 4. Model design

In this section we will review the model in VDM++ in a top-down approach and elaborate on interesting design choices made. The entities in our architecture are now VDM classes, types represent information state objects, and events and requests are described with operations and functions. A full class diagram can be found in Appendix A.

### 4.1. Controller

The controller controls at what rate new routes are placed into the interlocking. In a real setting, it regulates the flow of trains and adds a detail of security, as new routes that become available (free of trains) are not immediately placed into the interlocking. Listing 1 shows the how we do this in our VDM model. Operation SendRouteRequest() is executed sequentially, picks a random route from a sequence of routes and requests the route at the interlocking.

```
class Controller

instance variables
private itl : Interlocking;
private routes : seq of Interlocking'Route;

operations
public Step:() ==> ()
Step() == SendRouteRequest();
```

```
private SendRouteRequest:() ==> ()
SendRouteRequest() ==
-- send a random selected route into the Interlocking as a route request.
(dcl rn : nat := MATH`rand(len routes)+1;
  itl.RequestRoute(routes(rn));
);
...
end Controller
```

Listing 1: Definitions and SendRouteReques() operation in the Controller.

*4.2. Interlocking*

The interlocking has three primary operations with core functionality: One that accepts new route requests from the controller for routes $R_n$ and place it into its locking table (available routes) if it not already occupied. One that receive and store information from corresponding Eurobalises (track trackers) about the physical movement of trains. Lastly one that respond to the RadionBlockCenter requesting a MA (Movement Authority) on the behalf of trains. We start be defining the implemented types and instance variables (listing 2). The $trackTable$ maps tracks with a given coordinate to either a true or false value, if the track is occupied or not.

```
class Interlocking
...
types
public Order = <PROCEED_GRANTED> | <PROCEED_DENIED>;
public ProceedReply :: message : Order
routesAvaliable : set of Route;
public Track :: startX : nat
startY : nat
endX : nat
endY : nat
description : seq of char
maxSpeed : nat;
public Route = set of Track;

instance variables
private trackTable : map Track to bool;
private availableRoutes : set of Route;
inv InvNoDuplicateTrack(availableRoutes);
inv InvNoTrackAvailableAndOccupied(trackTable, availableRoutes);
```

6

```
public Interlocking: set of Route ==> Interlocking
Interlocking(rts) ==
atomic (
availableRoutes := rts;
trackTable := { tr |-> false | tr in set GetTracksInRoutes(rts)};
);
...
end Interlocking
```

Listing 2: Types and state information and constructor for the Interlocking class.

The interlocking has two invariants, $InvNoTrackAvailableAndOccupied$ that makes sure that occupied tracks in $trackTable$ and set of available routes are mutually exclusive and $InvNoDuplicateTrack$ that makes sure no duplicate track (with the same start and end coordinates) exists in the set of available routes (listing 3)

```
public InvNoTrackAvailableAndOccupied: map Track to bool
 * set of Route -> bool
InvNoTrackAvailableAndOccupied(trmap, rts) ==
forall rt in set rts & forall tr in set rt
& tr not in set dom trmap or trmap(tr) = false;

public InvNoDuplicateTrack: set of Route -> bool
InvNoDuplicateTrack(srt) ==
forall rt1 in set srt &
  forall tr1,tr2 in set rt1 & tr1 <> tr2
  => tr1.startX <> tr2.startX or tr1.endX <> tr2.endX
  or tr1.startY <> tr2.startY or tr1.endY <> tr2.endY;
```

Listing 3: Two invariants that verifies integrity of available routes and checks a route for duplicate track.

Next is the $RequestRoute()$ operation called by the controller (listing 4). If the route $rt$ is not in available routes and tracks are not occupied, we add it to $availableRoutes$ and clear its track. $GetTracksInRoutes()$ is a utility operation that returns the tracks in a route.

```
public RequestRoute: (Route) ==> ()
RequestRoute(rt) ==
if (rt not in set availableRoutes and
forall tr in set rt & tr not in set dom trackTable
```

7

```
  or trackTable(tr) = false)
  then (
  availableRoutes := availableRoutes union {rt};
  trackTable := trackTable ++ { tr |-> false |
      tr in set GetTracksInRoutes({rt})};
  ) else skip;
```

Listing 4: Definition of the SendRouteReques() operation.

The next operation $SetTrackState()$ is called by the Eurobalise when trains enter a new track $tr$ or leaves a track. This is passed as an argument by the Eurobalise. We set the track as occupied on enter and clears it on leave in the $trackTable$ (listing 5). We set a precondition that the track must be known to the interlocking.

```
public SetTrackState: Track * Eurobalise'TrainState ==> ()
SetTrackState(tr, sta) ==
-- if train enters, set specific track as occupied
-- if train leaves, clear track
if(sta = <TRAIN_ENTER>) then
trackTable := trackTable ++ {tr |-> true}
else if (sta = <TRAIN_LEAVE>) then
trackTable := trackTable ++ {tr |-> false}
pre tr in set dom trackTable;
```

Listing 5: Definition of the SendRouteReques() operation.

Finally the $RequestToProceed$ operation called by the RadionBlockCenter (listing 6). It returns a proceed granted $< PROCEED\_GRANTED$ if the route $ptr$ is in $availableRoutes$ and for all tracks in route $ptr$ neither are in the $trackTable$ as occupied. Before a proceed is granted, it removes the route $prt$ from the list of available routes as well all other routes that may contain the same track as $ptr$.

```
public RequestToProceed: Route * set of Track ==> ProceedReply
RequestToProceed(prt, respTrs) ==
-- if route is in avaliableRoutes and the route tracks are not in the dom of track
-- grant proceed by returning true, else return false
if (card availableRoutes > 0) then (
if(exists rt in set availableRoutes
& prt subset rt and forall tr in set prt
& tr in set dom trackTable) then
```

8

```
if(forall tr in set prt & trackTable(tr) = false) then (
availableRoutes := {rts | rts in set availableRoutes
  & forall ptrs in set {prt} & rts inter ptrs = {}};
return ProceedGranted(respTrs);
)
else
return ProceedDenied(respTrs)
else return ProceedDenied(respTrs)
) else (
return ProceedDenied(respTrs)
)
pre card prt > 0;
```

Listing 6: Definition of the SendRouteReques() operation.

### 4.3. RadioBlockCenter

The RadioBlockCenter receives MA requests from trains and forwards these to the interlocking, basically acting as a gateway but with its own state of available routes and responsible tracks initialized in the constructor (listing 7).

```
class RadioBlockCenter
types
public MovementAuthorityReply = <MovementAuthorityGranted> | <MovementAuthorityDen

instance variables
private respTracks : set of Interlocking`Track;
private availableRoutes : set of Interlocking`Route := {}; -- local state of avail
private itl : Interlocking;

public RadioBlockCenter : set of Interlocking`Track
* Interlocking ==> RadioBlockCenter
RadioBlockCenter(trs,pitl) ==
  respTracks := trs;
  itl := pitl;
);
...
end RadioBlockCenter
```

Listing 7: Definition of class state and constructor for RadioBlockCenter.

To receive the MA's, we define a operation $RequestMovementAuthority()$ that takes a certain route $Rt$, checks that the radio block center is responsible for

9

tracks in $Rt$, then forwards a synchronous proceed request to the interlocking and sends the reply back to the train for route $Rt$. The interlocking replies with all available routes that intersects with the responsible tracks, hence it is passed as a parameter (listing 8)

```
  RequestMovementAuthority: Interlocking`Route ==> MovementAuthorityReply
  RequestMovementAuthority(rt) ==
  (
  if(rt subset respTracks) then (
  dcl msg : Interlocking`Order;
  def mk_Interlocking`ProceedReply(message,rtr)
   = itl.RequestToProceed(rt, respTracks)
  in ( msg := message; availableRoutes := rtr; );
  if (msg = <PROCEED_GRANTED>)
  then ( return <MovementAuthorityGranted>; )
  else (
  return <MovementAuthorityDenied>;
  )
  ) else return <MovementAuthorityDenied>;
  );
```

Listing 8: Definition of the public.

### 4.4. Eurobalise

The Eurobalise is a piece of track side equipment that registers the physical movement of trains, as described previously. The class is straightforward, as it simply defines some track it is situated at and the global interlocking system (listing 9).

```
class Eurobalise
types
public TrainState = <TRAIN_ENTER> | <TRAIN_LEAVE>;

instance variables
private itl : Interlocking;
private track: Interlocking`Track;

public Eurobalise : Interlocking * Interlocking`Track
 ==> Eurobalise
Eurobalise(pitl, tr) ==
(
itl := pitl;
```

```
track := tr;
);
end Eurobalise
```

Listing 9: Types, variables, constructor for the Eurobalise.

The two operations $Enter()$ and $Leave()$ invokes the interlocking for a track $track$ when a train either enters or leaves it (listing 10.

```
public Enter: () ==> nat1
Enter() == (
  itl.SetTrackState(track, <TRAIN_ENTER>);
  return track.maxSpeed;
);

public Leave: () ==> ()
Leave() == (
  itl.SetTrackState(track, <TRAIN_LEAVE>);
);
```

Listing 10: The two operations that trains call to register their movement.

### 4.5. Train

The Train is the driver of the system and each created has a route table with one to many routes that it must traverse. When starting, it invokes the RadioBlock-Center by sending a MA request for the first route $Rt$ in its route table, and if the MA is granted, it will enter the first track in the route by calling the Eurobalise, set its current speed to match the track, move onto the end and leave it by calling the Eurobalise. The train will halt should it not obtain a MA for any $Rt$. In our simulation we follow the movement of trains by printing a log of their X and Y positional coordinates. The train has a lot of state declarations, so for clarity, we show just the sequential $Drive()$ function in listing 11. The full code for Train is found in Appendix A.

```
private Drive: () ==> ()
Drive() ==
(
if (state = <Running> or state = <WaitingForSignal>
  and len routeTable > 0) then (
    dcl currentRoute : Interlocking`Route := hd routeTable;
```

```
    if(rbc.RequestMA(currentRoute) = <MAGranted>)
    then (
      state := <Running>;
      for track in GetTracksInRoute(currentRoute) do (
        dcl currentEb : Eurobalise := transponders(track);
        currentSpeed := currentEb.Enter();
        posX := track.endX;
        posY := track.endY;
        currentEb.Leave();
      );
      routeTable := tl routeTable;
      if (routeTable = []) then (
        currentSpeed := 0;
        state := <Finished>;
      );
    ) else (
    currentSpeed := 0;
    state := <WaitingForSignal>;
    )
  ) else skip;
);
```

Listing 11: Drive() moves the train along a sequence of routes, requests MA for each route and obeys track speed limits.

## 5. Unit and combinatorial testing

A typical unit test verifies the functional properties of a class's functions and operations, often conducted in isolation with stubs for any external dependencies. A unit test invokes the unit under test (a class) with some input and evaluates whether the output satisfies what was expected. VDM++ offers unit testing options with VDMUnit, which we have used to test the four control classes in our system and shall explore in a bit. Additionally VDM++ provides a framework for combinatorial testing used to verify pre, post and invariant conditions in our classes. Both methodologies have been applied in this project and the complete set of sets can be found in the attached code, see Appendix A. We start by creating the initial environment for the tests seen in listing 12.

```
class InterlockingTest is subclass of TestCase
values
routes = {{mk_Interlocking`Track(10,5,15,5,"AB",100)},
  {mk_Interlocking`Track(15,5,20,5,"BC",110)},
```

```
   {mk_Interlocking'Track(20,5,25,5,"CD",110)},
   {mk_Interlocking'Track(25,5,30,5,"DE",110)},
   {mk_Interlocking'Track(25,5,30,5,"EF",110)}};

instance variables
private uut: Interlocking;

operations
public InterlockingTest: () ==> InterlockingTest
InterlockingTest() ==
  uut := new Interlocking(routes);
```

Listing 12: The test uses some predefined values and initiates the UUT in the constructor.

We proceed to define some functional unit tests made for the *Interlocking* in listing 13, where two examples are shown.

```
public Test_RouteInAvailableRoutes_ProceedGranted: () ==> ()
Test_RouteInAvailableRoutes_ProceedGranted() ==
(
dcl testRoute : Interlocking'Route
 := {mk_Interlocking'Track(10,5,15,5,"AB",100)};
dcl msg : Interlocking'ProceedReply
 := uut.RequestToProceed(testRoute, testRoute);
assertTrue(msg.message = <PROCEED_GRANTED>);
);

public Test_TrainEntersAndLeavesTrack_TrackChangesState: ()
 ==> ()
Test_TrainEntersAndLeavesTrack_TrackChangesState() ==
(
dcl track : Interlocking'Track
 := mk_Interlocking'Track(10,5,15,5,"AB",100);
uut.SetTrackState(track, <TRAIN_ENTER>);
assertTrue(track in set uut.GetTrackTable());
uut.SetTrackState(track, <TRAIN_LEAVE>);
assertTrue(track not in set uut.GetTrackTable());
);
```

Listing 13: Two unit tests defined for the Interlocking class.

For the combinatorial tests, they are defined in the classes and can be seen to the full extend in Appendix A. With these tests we verify preconditions, postconditions and state invariants

## 6. Simulation

- Bullet point one

- Bullet point two

1. Numbered list item one

2. Numbered list item two

### 6.1. Subsection One

Quisque elit ipsum, porttitor et imperdiet in, facilisis ac diam. Nunc facilisis interdum felis eget tincidunt. In condimentum fermentum leo, non consequat leo imperdiet pharetra. Fusce ac massa ipsum, vel convallis diam. Quisque eget turpis felis. Curabitur posuere, risus eu placerat porttitor, magna metus mollis ipsum, eu volutpat nisl erat ac justo. Nullam semper, mi at iaculis viverra, nunc velit iaculis nunc, eu tempor ligula eros in nulla. Aenean dapibus eleifend convallis. Cras ut libero tellus. Integer mollis eros eget risus malesuada fringilla mattis leo facilisis. Etiam interdum turpis eget odio ultricies sed convallis magna accumsan. Morbi in leo a mauris sollicitudin molestie at non nisl.

| Treatments | Response 1 | Response 2 |
|---|---|---|
| Treatment 1 | 0.0003262 | 0.562 |
| Treatment 2 | 0.0015681 | 0.910 |
| Treatment 3 | 0.0009271 | 0.296 |

Table 1: Table caption

### 6.2. Subsection Two

Donec eget ligula venenatis est posuere eleifend in sit amet diam. Vestibulum sollicitudin mauris ac augue blandit ultricies. Nulla facilisi. Etiam ut turpis nunc. Praesent leo orci, tincidunt vitae feugiat eu, feugiat a massa. Duis mauris ipsum, tempor vel condimentum nec, suscipit non mi. Fusce quis urna dictum felis posuere sagittis ac sit amet erat. In in ultrices lectus. Nulla vitae ipsum lectus, a gravida erat. Etiam quam nisl, blandit ut porta in, accumsan a nibh. Phasellus sodales euismod dolor sit amet elementum. Phasellus varius placerat erat, nec gravida libero pellentesque id. Fusce nisi ante, euismod nec cursus at, suscipit a enim. Nulla facilisi.

Integer risus dui, condimentum et gravida vitae, adipiscing et enim. Aliquam erat volutpat. Pellentesque diam sapien, egestas eget gravida ut, tempor eu nulla. Vestibulum mollis pretium lacus eget venenatis. Fusce gravida nisl quis est molestie eu luctus ipsum pretium. Maecenas non eros lorem, vel adipiscing odio. Etiam dolor risus, mattis in pellentesque id, pellentesque eu nibh. Mauris nec ante at orci ultricies placerat ac non massa. Aenean imperdiet, ante eu sollicitudin vestibulum, dolor felis dapibus arcu, sit amet fermentum urna nibh sit amet mauris. Suspendisse adipiscing mollis dolor quis lobortis.

$$e = mc^2 \qquad (1)$$

## 7. The Second Section

Reference to Section **??**. Etiam congue sollicitudin diam non porttitor. Etiam turpis nulla, auctor a pretium non, luctus quis ipsum. Fusce pretium gravida libero non accumsan. Donec eget augue ut nulla placerat hendrerit ac ut mi. Phasellus euismod ornare mollis. Proin tempus fringilla ultricies. Donec pretium feugiat libero quis convallis. Nam interdum ante sed magna congue eu semper tellus sagittis. Curabitur eu augue elit.

Aenean eleifend purus et massa consequat facilisis. Etiam volutpat placerat dignissim. Ut nec nibh nulla. Aliquam erat volutpat. Nam at massa velit, eu malesuada augue. Maecenas sit amet nunc mauris. Maecenas eu ligula quis turpis molestie elementum nec at est. Sed adipiscing neque ac sapien viverra sit amet vestibulum arcu rhoncus.

Vivamus pharetra nibh in orci euismod congue. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Quisque lacus diam, congue vel laoreet id, iaculis eu sapien. In id risus ac leo pellentesque pellentesque et in dui. Etiam tincidunt quam ut ante vestibulum ultricies. Nam at rutrum lectus. Aenean non justo tortor, nec mattis justo. Aliquam erat volutpat. Nullam ac viverra augue. In tempus venenatis nibh quis semper. Maecenas ac nisl eu ligula dictum lobortis. Sed lacus ante, tempor eu dictum eu, accumsan in velit. Integer accumsan convallis porttitor. Maecenas pretium tincidunt metus sit amet gravida. Maecenas pretium blandit felis, ac interdum ante semper sed.

In auctor ultrices elit, vel feugiat ligula aliquam sed. Curabitur aliquam elit sed dui rhoncus consectetur. Cras elit ipsum, lobortis a tempor at, viverra vitae mi. Cras sed urna sed eros bibendum faucibus. Morbi vel leo orci, vel faucibus orci. Vivamus urna nisl, sodales vitae posuere in, tempus vel tellus. Donec magna est, luctus non commodo sit amet, placerat et enim.