# Vehicle Platooning Simulations with Functional Reactive Programming

Bernd Finkbeiner
Saarland University
Germany

Felix Klein
Saarland University
Germany

Ruzica Piskac
Yale University
CT, USA

Mark Santolucito
Yale University
CT, USA

## ABSTRACT

Functional languages have provided major benefits to the verification community. Although features such as purity, a strong type system, and computational abstractions can help guide programmers away from costly errors, these can present challenges when used in a reactive system. Functional Reactive Programming is a paradigm that allows users the benefits of functional languages and an easy interface to a reactive environment. We present a tool for building autonomous vehicle controllers in FRP using Haskell.

## CCS CONCEPTS

•**Computer systems organization** →**Embedded and cyber-physical systems;** •**Software and its engineering** →*Embedded software; Real-time systems software;*

## KEYWORDS

FRP, Autonomous Vehicles

**ACM Reference format:**
Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2016. Vehicle Platooning Simulations with Functional Reactive Programming. In *Proceedings of 2017 1st International Workshop on Safe Control of Connected and Autonomous Vehicles (SCAV 2017), Pittsburgh, PA USA, April 2017 (SCAV 2017),* 5 pages.
DOI: http://dx.doi.org/10.1145/3055378.3055385

## 1 INTRODUCTION

Autonomous vehicles are considered to be one of the most challenging types of reactive systems currently under development [1, 27, 31]. They need to interact reliably with a highly reactive environment and crashes cannot be tolerated. Life critical decisions have to be made instantaneously and need to be executed at the right point in time.

The development of autonomous vehicles and other cyberphysical systems is supported by a wide spectrum of programming and modeling methodologies, including synchronous programming languages like Lustre [9] and Esterelle [3], hardware-oriented versions of imperative programming languages like SystemC [18], and visual languages like MSCs and Stateflow-charts [13, 14]. The question of which programming paradigm is best-suited to write easy-to-understand, bug-free code is still largely unresolved.

In the development of other forms of critical software, outside the embedded domain, developers increasingly turn to functional programming (cf. [12]). The strong type system in functional languages largely eliminates runtime errors [8]. Higher-order functions like `map` often eliminate the need for explicit index counters, and, hence, the risk of "index out of bounds" errors. Functional purity reduces the possibility of malformed state that can cause unexpected behavior.

While mathematical models of embedded and cyberphysical systems often rely on functional notions such as stream-processing functions [5, 6], the application of functional programming in the practical development of such systems has, so far, been limited. One of the most advanced programming language in this direction is Ivory, which was used in the development of autonomous vehicles [26]. Ivory is a restricted version of the C programming language, embedded in Haskell. It provides access to the low level operations necessary for embedded system programming, but still enforces good programming practice, such as disallowing pointer arithmetic, with a rich type system.

Ivory does not, however, have an explicit notion of time. It cannot deal directly with the integration of continuous and discrete time, which is fundamental for the development of a cyberphysical system. For example, in a car, continuous signals, such as the velocity or acceleration, mix with the discrete steps of the digital controller.

In this paper, we investigate the use of functional programming in a domain where the interaction between continuous and discrete signals is of fundamental importance. We build a vehicle controller capable of both autonomous vehicle control and multi-vehicle communication, such as the coordination in platooning situations.

Our approach is based on Functional Reactive Programming (FRP) [16, 17]. The fundamental idea of FRP is to extend the classic building blocks of functional programming (*e.g.* monads, arrows, applicatives) with the abstraction of a *signal* to describe time-varying values. FRP programs can be exceptionally efficient. For example, a network controller recently implemented as an FRP program on a multicore processor outperforms any other such controller existing today [29].


DOI: http://dx.doi.org/10.1145/3055378.3055385

**Figure 1: A screenshot of Haskell controlling the autonomous vehicle in the TORCS simulator.**

We have built a library, Haskell-TORCS, to use FRP to control a vehicle inside a simulation. The library interfaces Haskell FRP programs to TORCS, The Open Racing Car Simulator, an open-source vehicle simulator [32]. TORCS has been used in the Simulated Car Racing Championship competition [4], as well as other autonomous vehicle research projects [7, 23, 24, 33]. Through Haskell-TORCS, the Haskell program has access to the sensors and actuators of the car, as well as communication channels between different vehicles. Such a simulator is a critical component of modern autonomous vehicle research, especially towards the goal of safe platooning algorithms [19].

We report on our experience with two case studies, one in which we implement a controller for a solo car, and another for multi-vehicle platooning using a communication channel between the cars. Our controller successfully navigates the TORCS preloaded tracks with reasonable speed and finesse while avoiding collisions (see Fig. 1). Furthermore, the case study illustrates that the functional approach indeed leads to elegant, easily understandable, and safe code. The ability to run full simulations for solo and platooning vehicles is a critical piece to advancing the state of the art in using FRP for autonomous vehicle control.

## 2 FRP

The most common solution for the construction of reactive systems in an imperative setting are call-back frameworks, embedded into a loop. The call-backs are either used to query the state of variables, or to change them. This imperative approach is well suited for rapid prototyping of small systems. However, tracing behaviors over time quickly becomes unmanageably complex for larger systems.

Functional Reactive Programming instead introduces a concrete abstraction of time that allows the programmer to safely manipulate time-varying values. The key abstraction is given by a *signal*, providing the programmer with a simple type interface:

```
type Signal a = Time → a
```

For example, the type `Signal Image` represents a video, while `Signal Steer` captures a steering wheel operated over time. To better understand how our library works, we now introduce the basic concepts and terminology from FRP.

**Listing 1: Basic Arrowized FRP syntax**

```
myDriver :: SF Image Steer
myDriver = proc image → do
  basicSteer    <-     turn  -< image
  adjustedSteer <- arr avoid -< (image, basicSteer)
  returnA -< adjustedSteer
```

### 2.1 Arrowized FRP

There are many types of FRP based on different abstractions from type theory. Expressive abstractions, such as monads, allow for complex manipulation of signal flows [28]. However, for most applications they are far too expressive. We instead focus on an FRP library, Yampa, which uses the arrow abstraction, or so called Arrowized FRP [17]. Arrows generally run faster and with little need for manual optimization [34], but are fundamentally less expressive than a monadic FRP [21]. This more restrictive language is in fact a benefit, as it makes it harder for the programmer to introduce errors. As we will see in the sequel, Yampa is still powerful enough to write complex controllers to drive an autonomous vehicle, or even to communicate with other vehicles. At the same time, the syntax is clear and accessible enough to make for an easy introduction to the FRP paradigm.

Along with signals, Yampa also introduces the abstraction of a *signal function (SF)*. This is a transformer from one signal to another.

```
type SF a b = Signal a → Signal b
```

Using the previous signals, imagine a type for a steering function, which operates based on a video stream, such as

```
turn :: SF Image Steer
```

This function processes video and uses it to decide how to steer. We omit an implementation, as the details of the data transformation are not relevant to the structure of the FRP code.

Haskell provides special syntax for Arrowized FRP, which mimics the structure of control flow charts. The syntax provides a composition environment, in which the programmer just manages the composition of arrow functions. Inputs are read in from the right hand side, and piped to the left hand side (`output <- function -< input`). A demonstration is given in Listing 1.

The example introduces

```
avoid :: (Image, Steer) → Steer
```

a pure function that adjusts the basic steering plan based on the image to avoid any obstacles. In Listing 1, this `avoid` function is lifted to the signal level using:

```
arr :: (a → b) → SF a b
```

The function `turn` is already on the signal level (has an SF type). Hence, we do not need to lift it.

### 2.2 Stateful FRP

To avoid obstacles on the road, we might write an `avoid2` function as shown in Listing 2, which requires two images to calculate the adjusted steering command. For this, we need a mechanism to maintain state between each processing step. Two images would be necessary to filter noise in the image, or calculates the velocity of an approaching obstacle. To implement it, we use an abstraction called `ArrowLoop` to save the previous state of the image for the next

**Listing 2: Using ArrowLoop to send feedback**

```
myDriver :: SF Image Steer
myDriver = proc image → do
  rec
    oldI          <- iPre null  -< image
    basicSteer    <-      turn  -< image
    adjustedSteer <- arr avoid2 -< (image, oldI,
         basicSteer)
  returnA -< adjustedSteer
```

processing step. The syntax is presented in Listing 2. Intuitively, `ArrowLoop` gives us a recursive computation, as also indicated by the `rec` keyword[1].

The predefined function `iPre` takes an initial state, in our case an empty image, and saves images for one time step, each time it is processed. This way, we create a feedback loop that is then used in the updated `avoid` function. At the same time, the `rec` keyword is used to denote a section of arrow code with mutual dependencies[2].

## 3 HASKELL-TORCS

TORCS, The Open Racing Car Simulator, is an existing open source vehicle simulator [32] that has bindings for various languages [4]. We provide the first bindings for Haskell, and further extend this into a full library for multi-vehicle simulations. The library is an open source library, called Haskell-TORCS, and publicly available at https://github.com/santolucito/Haskell-TORCS. We now explain the functionality provided by our library, and highlight the ability of FRP to create modular and flexible controllers with clean code for autonomous vehicles.

### 3.1 Basics

To interface with Haskell-TORCS, a user must implement a controller that will process the `CarState`, which contains all the data available from the sensors. The controller should then output a `DriveState`, which contains all the data for controlling the vehicle. This transformation is succinctly described as the now familiar *signal function*. The core functionality of Haskell-TORCS is captured in the function `startDriver`, which launches a controller in the simulator. This function automatically connects a `Driver` to TORCS, which results in continuous `IO()` actions, the output type of this function.

```
type Driver = SF CarState DriveState
startDriver :: Driver → IO ()
```

The sensor and output data structures contain all the typical data available in an autonomously controlled vehicle. `CarState` includes fields like `rpm` to monitor the engine, or `track` to simulate an array of LiDAR sensors oriented to the front of the vehicle. `DriveState` includes fields like `accel` to control the gas pedal, or `steering` to control the angle of the steering wheel. A full description of the interface is available in the Simulated Car Racing Competition Manual [22].

[1]We elide the technical details for the purposes of this presentation and refer the interested reader to [25].

[2]Without the keyword, there is an unresolvable dependency loop.

**Listing 3: A complete basic controller in Yampa**

```
{-# LANGUAGE Arrows, MultiWayIf, RecordWildCards #-}
module TORCS.Example where
import TORCS.Connect
import TORCS.Types

main = startDriver myDriver

myDriver :: Driver
myDriver = proc CarState{..}  → do
  rec
    oldG <- iPre 0 -< g
    g <- arr shifting -< (rpm, oldG)
    s <- arr steering -< (angle, trackPos)
    a <- arr gas -< (speedX, s)
  returnA -< defaultDriveState
    { accel = a, gear = g, steer = s }

shifting :: (Double, Int) → Int
shifting (rpm, g) = if
  | rpm > 6000 → min 6 (g + 1)
  | rpm < 3000 → max 1 (g - 1)
  | otherwise  → g

steering :: (Double, Double) → Double
steering (spd, trackPos) = let
  turns = spd * 14 / pi
  centering = turns - (trackPos * 0.1)
  clip x = max (-1) (min x 1)
 in
  clip centering

gas :: (Double, Double) → Double
gas (speed, steer) =
  if speed < (100 - (steer * 50)) then 1 else 0
```

### 3.2 Case Study : Driving

As a demonstration of the Haskell-TORCS library in use, we implemented a simple controller, shown in Listing 3. The code is complete and immediately executable as-is together with an installation of TORCS. Our controller successfully navigates, with some speed and finesse, a vehicle on track, as shown in Fig. 1 along with a video demonstration[3]. The controller uses `ArrowLoop` to keep track of the current gear of the car. Although the gear is available as sensor data, it is illustrative to keep track locally of this state. In general, the `ArrowLoop` can be used to maintain any state that may be of interest in a future processing step. Additionally, notice all of the data manipulation functions are pure, and lifted via the predefined function `arr`.

One major advantage of FRP is this separation of dependency flow and data level manipulation. This abstraction makes it possible to easily reason about each of the components without worrying about confounding factors from the other. For example, if a programmer wants to verify that the steering control is correct, it is semantically guaranteed that the only function that must be checked is `steering`. Because of Haskell's purity, this is the only place where the steering value is changed. This significantly reduces the complexity of verification or bug tracking in case of an error.

[3]http://www.marksantolucito.com/torcsdemo

**Listing 4: Communicating between controllers**

```
request :: Double → Message
request dist =
  if dist < 3 then "faster" else ""

adjustSpeed :: (Communications, Double) → Double
adjustSpeed (comms, oldSpeed) =
  if any (map (== "faster") comms) then s + 10 else s
```

### 3.3 Case Study : Communication for Platoons

Thanks to functional languages' exceptional support for parallelism, controlling multiple vehicles in a multi-threaded environment is exceedingly simple. In our library API, the user simply uses `startDrivers` rather than `startDriver`, and passes a list of `Driver` signal functions "driving" together. In this way, we easily let various implementations race against each other, or build a vehicle platooning controller. In the latter, the user can even extend the implementation to simulate communication between the vehicles.

Our library already provides a simple interface for simulating communication between vehicles. In order to broadcast a message to the other vehicles in the simulation, the controller simply writes a message to the `broadcast` field of `DriveState`. That message is then sent to all other vehicles as soon as possible, and received in the `communication` field of the input `CarState`.

A fragment of communication code is given in Listing 4, to pass messages between vehicles. In this fragment, a vehicle checks if a collision is imminent, and can request for the other cars in the platoon to go faster and move out of the way. Every vehicle also checks if any other car has requested for the platoon to speed up, and will adjust its own speed accordingly. These functions can be added to a controller, like the one in Listing 3, with little effort.

We allow all vehicles in the simulation to communicate irrespective of distance and with zero packet loss. However, users are free to implement and simulate unreliable communications, or distance constraints.

### 3.4 Implementation

Haskell-TORCS uses Yampa [10] as the core FRP library, though its structure can easily be adapted to any other Haskell FRP library.

TORCS uses a specialized physics engine for vehicle simulations, that includes levels of detail as fine grained as tire temperatures effect on traction. When TORCS is used in the Simulated Car Racing Championship competition [4], each car is controlled via a socket that sends the sensor data from the vehicle and receives and processes the driving commands. So too, Haskell-TORCS communicates over these sockets to control vehicles inside the TORCS simulations.

In addition to the core controller functionality, we have also augmented Haskell-TORCS with the ability to test vehicle platooning algorithms that utilize cross-vehicle communication. The communication channels are realized via a hash map, using the `Data.Hashmap` interface, from vehicle identifiers to messages. Each vehicle is given write permissions to their unique channel, where all other vehicles have read-only permissions. The access is mutually exclusive, which is ensured by Haskell's `MVar` implementation, a threadsafe shared memory library. This ensures that there will never be packet loss in the communication.

## 4 RELATED WORK

TORCS has been proven to provide an expressive framework for the research community [7, 23, 24]. Notably, it has even been used for formal verification of platoons [19, 33]. None of these works have used FRP as the language for the controller. With the assistance of FRP, we build vehicle controllers in a principled way that allows users to manipulate sensor data in a transparent and well structured environment.

To the best of our knowledge this is the first FRP-based vehicle simulator. Although there are many bindings to various vehicle simulators, these tend to use imperative languages. For instance, TORCS allows users to directly edit the source code and add a new car in C++. There are also TORCS bindings for Python, Java, and Matlab, which have been used in the SCRC competition [4].

FRP specifically has been proposed as a tool for vehicle control [20, 35], where FRP was extended to prioritize functions for timing constraints. However, due to the lack of a compatible simulator, the vehicle simulation never was implemented. FRP has also been used for embedded systems [15] and networking [30]. The FRP networking library took advantage of Haskell's multicore support and significantly outperformed competing tools written in C++ and Java.

The videogame Grand Theft Auto (GTA) [2] has also been used to train image recognition software for autonomous vehicles [11]. While GTA is a professionally produced game with more attractive graphics, it is proprietary software not designed for autonomous vehicle research. The only available sensor data are gameplay images, which are a limited model for autonomous vehicles. Using GTA as a meaningful control simulator would still be a valuable tool, but we leave this to future work.

## 5 CONCLUSIONS

We have presented a library to write autonomous vehicle controllers in FRP that supports cross-vehicle communication. This work opens the door for further research in using the powerful FRP paradigm for building safer, more reliable controllers for one of the most critical applications in reactive systems.

## Acknowledgments

Supported by the European Research Council (ERC) Grant OSARES (No. 683300) and by the National Science Foundation (NSF) Grant CCF-1302327.

## REFERENCES

[1] Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2016. Compositional Synthesis with Parametric Reactive Controllers. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016.* 215–224.

[2] Leslie Benzies. 2013. Grand Theft Auto V. www.rockstargames.com/V/. (2013).

[3] Gérard Berry and Laurent Cosserat. 1984. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984 (Lecture Notes in Computer Science)*, Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel (Eds.), Vol. 197. Springer, 389–448.

[4] Mohommad Reza Bonyadi, Samadhi Nallaperuma, Daniele Loiacono, and Frank Neumann. 2015. Simulated Car Racing Championship. http://cs.adelaide.edu.au/~optlog/SCR2015/index.html. (2015).

[5] Manfred Broy. 2012. Engineering Cyber-Physical Systems: Challenges and Foundations. In *Complex Systems Design & Management, Proceedings of the Third International Conference on Complex Systems Design & Management CSD&M 2012,*

*Paris, France, December 12-14, 2012*, Marc Aiguier, Yves Caseau, Daniel Krob, and Antoine Rauzy (Eds.). Springer, 1–13.

[6] Manfred Broy and Ketil Stølen. 2001. *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement*. Springer.

[7] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. 2009. Learning drivers for TORCS through imitation using supervised methods. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Milano, Italy, 7-10 September, 2009*. 148–155.

[8] Luca Cardelli. 1996. Type Systems. *ACM Comput. Surv.* 28, 1 (March 1996), 263–264.

[9] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 178–188. http://dl.acm.org/citation.cfm?id=41625

[10] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. ACM, 7–18.

[11] Artur Filipowicz, Jeremiah Liu, and Alain Kornhauser. 2017. Learning to Recognize Distance to Stop Signs Using the Virtual World of Grand Theft Auto 5. *Transportation Research Board, 96th Annual Meeting* (2017).

[12] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. 2009. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming* 19, 01 (2009), 27–45.

[13] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274.

[14] David Harel and PS Thiagarajan. 2003. Message sequence charts. In *UML for Real*. Springer, 77–105.

[15] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, 8–16.

[16] Paul Hudak. 2000. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press.

[17] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*. Springer, 159–187.

[18] Open SystemC Initiative and others. 2006. IEEE standard SystemC language reference manual. *IEEE Computer Society* (2006), 1666–2005.

[19] Maryam Kamali, Louise A Dennis, Owen McAree, Michael Fisher, and Sandor M Veres. 2016. Formal verification of autonomous vehicle platooning. *arXiv preprint arXiv:1602.01718* (2016).

[20] Zeinab Kazemi and Albert MK Cheng. 2016. A Scratchpad Memory-Based Execution Platform for Functional Reactive Systems and Its Static Timing Analysis. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*. IEEE, 176–181.

[21] Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science* 229, 5 (2011), 97–117.

[22] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. 2013. Simulated Car Racing Championship: Competition Software Manual. *CoRR* abs/1304.1672 (2013). http://arxiv.org/abs/1304.1672

[23] Jorge Muñoz, Germán Gutiérrez, and Araceli Sanchis. 2010. A human-like TORCS controller for the Simulated Car Racing Championship. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG 2010, Copenhagen, Denmark, 18-21 August, 2010*. 473–480.

[24] Enrique Onieva, David A. Pelta, Javier Alonso, Vicente Milanés, and Joshué Pérez. 2009. A modular parametric architecture for the TORCS racing engine. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Milano, Italy, 7-10 September, 2009*. 256–262.

[25] Ross Paterson. 2001. A new notation for arrows. *ACM SIGPLAN Notices* 36, 10 (2001), 229–240.

[26] Lee Pike, Patrick Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. 2014. Programming languages for high-assurance autonomous vehicles. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages meets Program Verification*. ACM, 1–2.

[27] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M. Murray, and Sanjit A. Seshia. 2015. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*. 239–248.

[28] Atze van der Ploeg. 2014. Monadic functional reactive programming. *ACM SIGPLAN Notices* 48, 12 (2014), 117–128.

[29] Andreas Voellmy and Junchang Wang. 2012. Scalable software defined network controllers. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Aug. 2012), 289–290.

[30] Andreas Voellmy and Junchang Wang. 2012. Scalable software defined network controllers. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 289–290.

[31] Tichakorn Wongpiromsarn, Sertac Karaman, and Emilio Frazzoli. 2011. Synthesis of provably correct controllers for autonomous vehicles in urban environments. In *14th International IEEE Conference on Intelligent Transportation Systems, ITSC 2011, Washington, DC, USA, October 5-7, 2011*. 1168–1173.

[32] Bernhard Wymann, Eric Espie, and Christophe Guionneau. 2017. Torcs: The open racing car simulator, v1.3.4. http://torcs.sourceforge.net/index.php. (2017).

[33] Zhixiang Xu, Jiemei Jiang, and Yonggui Liu. 2016. Experimental research of vehicle-platoon coordination control based on TORCS platform. In *Control Conference (CCC), 2016 35th Chinese*. IEEE, 7404–7409.

[34] Jeremy Yallop and Hai Liu. 2016. Causal commutative arrows revisited. In *Proceedings of the 9th International Symposium on Haskell*. ACM, 21–32.

[35] Xingliang Zou, Albert MK Cheng, and Yu Jiang. 2016. P-FRP task scheduling: A survey. In *Declarative Cyber-Physical Systems (DCPS), CPSWeek Workshop on*. IEEE, 1–8.