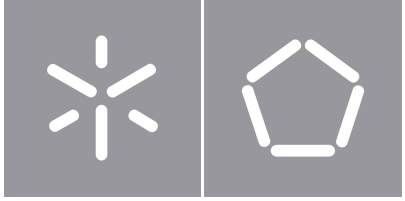
University of Minho
School of Engineering

Rui Carlos Azevedo Carvalho

Adding Uncertainty to Real-Time Programming

august 2023



University of Minho
School of Engineering

Rui Carlos Azevedo Carvalho

Adding Uncertainty to Real-Time Programming

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
Renato Neves

august 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

I would like to start by expressing my gratitude towards Professor Renato Neves for the all the guidance throughout this project and for the incredible help in every step of this dissertation. His vast expertise was invaluable not only for completing this dissertation but also to greatly improve my knowledge in this research area.

I would also like to thank Juliana Souza, who acted like a co-supervisor, not only for the scientific contribution which served as the foundation for this dissertation, but also for all the knowledge, related to the more theoretical parts of programming languages and mathematics, and support during this whole year.

Finally, I would like to thank my parents, Carlos and Arminda, as well as my brother Luís, for the incredible support and encouragement they always gave me during every part of my life which was fundamental to keep me focused on my education which helped me a lot to now be able to complete this dissertation.

This work was financed by National Funds through FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference PTDC/CCI-COM/4280/2021. The author was also financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, august 2023

Rui Carlos Azevedo Carvalho

Abstract

In this dissertation we cover the implementation in Haskell of an interpreter for a while-language capable of handling both hybrid and probabilistic effects. The interpreter is supported by both operational and denotational semantics which were devised in this dissertation as well.

We started by studying a pre-existing syntax and operational semantics of a programming language capable of performing wait calls and probabilistic choices through a random-number-generator. We then redefined this semantics to another one that is more suitable for statistical analysis in programming. Next we performed another iteration over these two semantics, more specifically we extended them to support full hybrid behaviour, traditionally used to encode interactions between digital devices and physical processes such as movement and time.

We also devised two denotational semantics corresponding to the operational semantics mentioned before, as a way of providing a mathematical abstraction, through the use of monads, to the programs of our language.

Not only this, we also implemented a domain specific language embedded into Haskell, which thus provides to the hybrid programmer all the expressive power that Haskell offers in addition to a palette of combinators designed specifically for the hybrid domain. Such gives rise to an expressivity power much greater than what the aforementioned while-language can provide.

Lastly, we presented and analysed several deterministic hybrid programs, such as cruise controllers, and added subtle probabilistic elements to them that reflect certain real-word scenarios. Such an addition lead from one possible execution to several possible executions; and most notably some of the latter revealed safety issues introduced by the probabilistic elements.

All in all this dissertation has both theoretical and practical contributions that form a stepping stone towards a rigorous engineering discipline of probabilistic hybrid systems.

Keywords Formal Methods, Hybrid Systems, Cyber-Physical Systems, Theory of Programming Languages, Functional Programming

Resumo

Através desta dissertação descrevemos uma implementação em Haskell de um interpretador para uma linguage *while* capaz de lidar tanto com o efeito híbrido como com o efeito probabilístico. O interpretador é suportado por semânticas operacionais e denotacionais que foram definidas também nesta dissertação.

Primeiramente, abordamos uma sintaxe e semântica operacional pré-existentes de uma linguagem capaz de realizar chamadas de espera e escolhas probabilísticas, através de um gerador de números aleatórios. De seguida, redefinimos esta semântica numa outra que é mais apropriada para a análise estatística dos programas. Consequentemente, realizamos uma outra iteração sobre estas semânticas sendo que as estendemos para suportarem totalmente o comportamento híbrido, tradicionalmente utilizado para representar interações entre dispositivos digitais e físicos, tais como movimento e tempo.

Criámos também duas semânticas denotacionais, correspondentes às semânticas anteriores, de modo a fornecer uma abstração matemática, através da utilização de mónadas, para os possíveis programas da nossa linguagem.

Para além disto, foi implementada uma *domain specific language* incorporada em Haskell, o que providencia ao programador híbrido todo o poder que o Haskell oferece conjugado com combinadores definidos especialmente para o domínio híbrido. Tal dá origem a um poder expressivo muito maior do que aquele que a linguagem *while* supramencionada pode providenciar.

Por fim, abordaram-se vários exemplos de programas híbridos determinísticos, tais como sistemas de *cruise control*, aos quais foram adicionados elementos probabilísticos subtis que refletem cenários do mundo real. Esta adição fez com que fossemos de uma execução para uma variedade de execuções possíveis; e mais notávelmente alguns dos sistemas revelaram problemas relacionados com a segurança quando os elementos probabílisticos foram introduzidos.

Em suma, esta dissertação apresenta contribuições teóricas e práticas que se traduzem num passo em direção a uma dísciplina de engenharia rigorosa sobre sístemas híbridos probabilísticos.

Palavras-chave Métodos Formais, Sistemas Híbridos, Sistemas Ciber-Físicos, Teoria das Linguagens de Programação, Programação Funcional

Contents

List of Figures

Acronyms

DSL Domain specific language.

ODE Ordinary differential equations.

PRE Probabilistic execution.

RNG Random number generator.

RTE real-time execution.

Chapter 1

Introduction

1.1 Context and motivation

Hybrid systems, the *sine qua non* of this dissertation, are characterised by the following unique feature: they harbour behaviour with both discrete and continuous components, typically observed in the interaction between classical programs and processes of a physical nature, such as time, velocity, and temperature. Such systems appear in a vast spectrum of fields, ranging all the way from the medical area, with insulin pumps and pacemakers, to district-wide infrastructures like water and electric grids. Other cases be found for example in autonomous driving, which gets more and more attention everyday [38, 47].

As hybrid systems become more prevalent, the associated programming paradigm, known as *hybrid programming*, also starts to become more relevant. This is fuelling the development of new formalisms, techniques and tools for the design and analysis of such systems [25, 38, 44]. There are however several challenges associated with this paradigm. A core one is on how to create a suitable abstraction layer that encodes the interaction between a physical process and a digital program. Another challenge is on how to guarantee the correctness of hybrid systems. Since hybrid systems involve both discrete and continuous behaviour, these challenges are highly difficult to overcome because they generally require techniques from computer science combined with techniques from real-analysis [38, 47, 25].

1.2 Contributions

The problem that motivates this dissertation revolves around the relevance of hybrid systems (specially in today's world) and the fact that the majority of tools able to analyse such systems cannot handle the inherent uncertainty they possess – note that even an almost trivial instruction such as *wait call* of one second does not guarantee that the computer will halt for *exactly* one second.

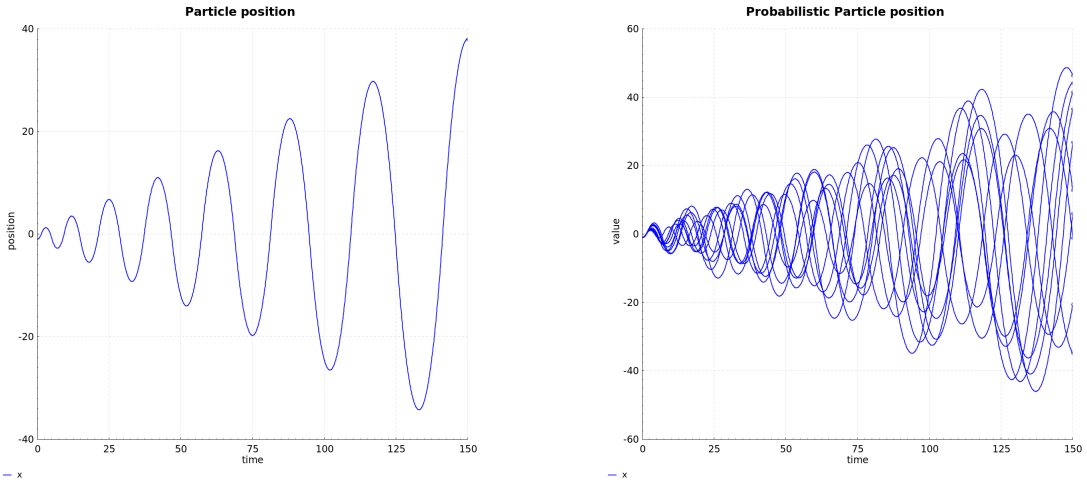
This project thus consists in the implementation, in Haskell, of a hybrid programming language that

harbours *probabilistic* operations. To be more specific, the main contributions of the present dissertation are the following:

- The implementation of parser and an interpreter for a while-language capable of handling both hybrid and probabilistic behaviour.
- The implementation of a flexible visualisation module for a proper analysis of the behaviour of probabilistic hybrid systems.
- A thorough study of possible operational semantics for the while-language above. This resulted in the development of two complementary semantics for the language, both integrated in the interpreter. One semantics is based on random-number-generators and performs real-time simulations and the other works explicitly with distributions and does not perform real-time wait calls. The former runs slower (when timed events are present) but might be useful in cases where we wish to include the errors associated with wait calls to the operating system, while the latter is faster but only takes into consideration the theoretical wait times.
- We gave first steps towards denotational counterparts to the two previously mentioned operational semantics. This helps to root our interpreter in strong mathematical foundations, from which we can capitalise to reason about probabilistic hybrid programming. For example, reasoning about program equivalence often recurs to denotational semantics [25].
- Then in order to boost the flexibility and expressivity of the aforementioned while-language, we implemented a domain specific language (DSL) embedded in Haskell, which allows the use of elements present in the while-language in conjunction with functions from Haskell, such as folds or maps, as well as combinators created specifically for the DSL.
- Finally we developed several hybrid programs combined with probabilistic elements, using both the while-language and the DSL. We analysed and reported how probabilistic behaviour affects the respective results. Notably, some programs presented no problems when using the deterministic approach, but as soon as small levels of uncertainty were introduced, we obtained right away trajectories that indicated safety related problems.

This tool expands the modelling capabilities of Lince [5] allowing us to see different trajectories, by adding a few elements of uncertainty. Figure 1 shows this in action, as in the left side we have a deterministic trajectory, where we try to maintain a particle at a fixed distance, but due to error accumulation when

accelerating or decelerating it, the particle tends to get further and further away [5]. As we mentioned before we cannot expect an action to be made in an exact amount of time, there is always some kind of uncertainty in these moments. The tool we built, which we named ProbLince, allows us to specify this uncertainty in the model, leading to a multitude of trajectories. In some of these the particle moves away faster than others, which leads us to observe the strength of uncertainty and what kind of problems we can detect by having it ingrained in our models.



(a) Naive particle position

(b) Probabilistic naive particle position

Figure 1: Simulation of a particle position with ProbLince

1.3 Roadmap

This dissertation is divided into seven chapters with the following content:

- Chapter 1: goes over the context, motivation, main contributions and structure of this dissertation.
- Chapter 2: covers the state of the art of the techniques used to implement our parser and interpreter. It also gives an overview of probabilistic and hybrid programming, and establishes the mathematical background, namely elements of category theory, such as Kleisli categories and monads.
- Chapter 3: describes the syntax and operational semantics of our while-language, which mixes probabilistic elements with hybrid behaviour.
- Chapter 4: goes over the implementation details such as the architecture of the developed tool, and how some key parts of both the parser and interpreter were implemented.

- Chapter 5: discusses the implementation details of the DSL, describes its combinators and shows how the DSL it differs from the previous while-language.
- Chapter 6: presents some case studies, how they were implemented in our tool, and the results obtained.
- Chapter 7: concludes the dissertation and discusses possible lines of future work.

Finally there are two appendices. The first contains the grammar used by the parser, while the second has the installation commands and user guide for the implemented tools.

Chapter 2

State of the Art

In this chapter we reveal some of the key concepts in the literature, which support this dissertation and the tool we implemented.

We start by going over the concept of a monad and how it can be used to capture computational effects, as well as the techniques to combine them into a single monad capable of handling all the different effects at once.

Afterwards, we cover the definition of probabilistic programming and its characteristics. We also uncover some well known languages for implementing probabilistic programs, but more importantly some implementations which follow a monadic approach.

In the final two sections we dive into the concept of hybrid systems, by revealing what they are, their importance and the challenges associated with them. We will also go over the notion of hybrid programming and how we can use monads to capture the hybrid effect and why this approach is an advantage when compared to other ways of modelling hybrid systems.

2.1 Core concepts of monad-based programming

2.1.1 The notion of a monad

Before going over the actual definition of a monad, we start by looking at a fruitful connection between monads and programming. In [36, 37] E. Moggi established an important connection between functions and complex programs, by pointing out that the latter could be seen as functions (or more generally, morphisms) $A \to T\ B$, which map values of some generic type A to *computations* $T\ B$ over a generic type B. This provides an uniform and flexible view of various side effects. One of the most common ones are exceptions, as they appear in many software products. If we think for example about the $head$ function, which outputs the first element of a given list, we quickly notice the existence of the empty list

case, where an exception should be raised together with an appropriate message. So the final computation over a list corresponds to two possibilities, either we output the first element of the list or an exception.

Even though there are many side effects (which means there are several maps like the one mentioned above) these present intrinsic common characteristics: indeed in every instance we can consider how computations should be handled along sequential composition, and how an input value should be embedded into a computation [37]. This yields naturally the following definition.

Definition 1 (Kleisli triple)**.** A Kleisli triple over a category C is comprised of the following elements [36, 37]:

- A mapping on objects $Obj(\mathsf{C}) \to Obj(\mathsf{C})$;
- For every C-object X a morphism $\eta_X : X \to T\ X$, which allows the embedding of a value into a computation;
- A function $(-)^*$ which lifts a morphism $A \to T\ B$ to a morphism $T\ A \to T\ B$, corresponding to the way computations are handled along sequential composition.

Moreover $(-)^*$ needs to respect a certain set of coherence laws detailed in [36, 37].

It is at this point that the notion of a monad naturally appears in the context of programming: Kleisli triples and monads are in bijective correspondence to each other.

Definition 2 (Monad)**.** A monad over a category C is a triple comprised of the following elements [37]:

- A functor $T : \mathsf{C} \to \mathsf{C}$;
- A natural transformation $\eta : \mathsf{Id} \to T$;
- A natural transformation $\mu : TT \to T$.

Moreover certain coherence laws about the interaction between η and μ need to be respected. These are detailed for example in [37].

Very briefly, from a Kleisli triple we obtain the endofunctor T of the equivalent monad by defining $Tf = (\eta \cdot f)^*$, and for every C-object A the morphism $\mu_A : TT\ A \to T\ A$ is defined by $(id_{TA})^*$. Conversely if we take a monad and want to obtain the corresponding Kleisli triple we have to apply a constraint to the endofunctor T so it only deals with objects of the category and the Kleisli lifting $(-)^*$ is defined by $f^* = \mu \cdot Tf$. See more details in [37]. This bijective correspondence between Kleisli

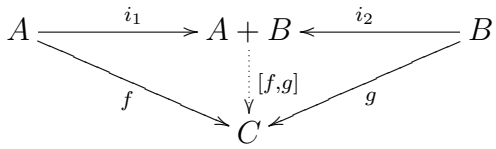
triples and monads is useful, because in certain aspects it is easier to work with one view instead of the other [37].

Below we have the definitions of some well-known monads, which are crucially used in various software products – we will work in the category of sets and functions, but many of these examples can be actually generalised to categories satisfying certain conditions.

Definition 3 (Exception monad)**.** Exceptions can be viewed as the possibility of obtaining two kinds of result: the concrete value, which corresponds to the result of the computation, or a message, which is an error or exception $e \in \mathbb{E}$ that was thrown during the execution of the program. In [37] we have a Kleisli triple which defines this type of computation, but we can consider the following monadic definition of the exception monad ($\mathcal{E}[\mathbb{E}]$, in which $\mathbb{E}$ is the set of exceptions):

- $E[\mathbb{E}]\ X = X + \mathbb{E}$
- $\eta_X\ =\ i_1$
- $\mu_X = [id, i_2]$

Note in the definition above the use of the injection functions i_1 and i_2, as well as the function $[f, g]$. These are detailed in [39, 42] and correspond to a coproduct diagram:



Definition 4 (State monad, $\mathcal{S}$)**.** The state monad provides mutable states $s \in \mathbb{S}$ to the programmer. One use case of this monad was presented in [52] where P. Wadler uses it to count the number of reduction steps a computation takes to terminate. Its definition is given again in [37], but by adapting it into the monadic view we obtain:

- $S\ X\ =\ (X \times \mathbb{S})^{\mathbb{S}}$
- $\eta_X(x)\ =\ \lambda s.(x, s)$
- $\mu_X(g)\ =\ \lambda s.f(s')$ with $g(s) = (f, s')$

Definition 5 (Writer monad, $\mathcal{W}$)**.** The writer monad is used for aggregating data along computations. This is useful for example to debug programs or to keep track of which resources were used. Technically the monad is parametrised by a monoid $(\mathbb{M},\ \cdot\ , i)$ and the Kleisli triple definition is presented in [37]; by adapting the latter to the monadic perspective we obtain:

- $W\ X\ =\ X \times \mathbb{M}$
- $\eta_X(x)\ =\ (x, i)$
- $\mu_X((x, m'), m)\ =\ (x, m\ \cdot\ m')$

Definition 6 (Generalised writer monad)**.** Consider a monoid $(\mathbb{M},\ \cdot\ , i)$. A monoid module is a set $\mathbb{E}$ equipped with a map $\triangleright\ :\ \mathbb{M} \times \mathbb{E}\ \rightarrow\ \mathbb{E}$ (monoid action) that satisfies the laws $i \triangleright e\ =\ e$ and $(m \cdot n) \triangleright e = m \triangleright (n \triangleright e)$. Every monoid module yiels a Kleisli triple defined by [25, 23],

- $W[\mathbb{M}]\ X = X \times \mathbb{M} + \mathbb{E}$
- $\eta_X(x) = i_1\ (x, i)$
-
  - $f^*(i_1\ (x, m)) = i_1\ (y, m \cdot n)$ if $f(x) = i_1\ (y, n)$
  - $f^*(i_1\ (x, m)) = i_2\ m \triangleright e$ if $f(x) = i_2\ e$
  - $f^*(i_2\ e) = i_2\ e$

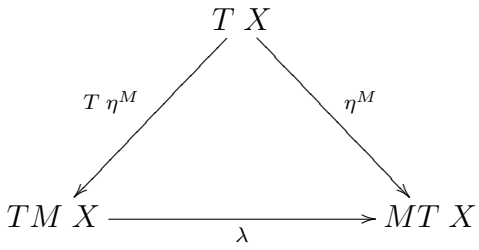
Where $W[\mathbb{M}]$ is the functor of the generalised writer monad ($\mathcal{W}[\mathbb{M}]$).

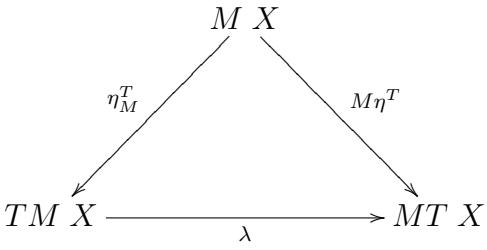
2.1.2 Monad composition

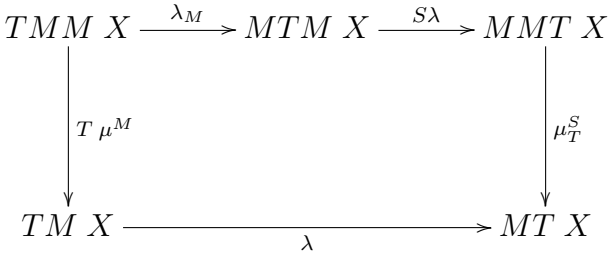
As already mentioned, the implementation of interpreters is a frequent case [52, 32] in which different monads are need to capture different functionalities of the underlying language. For example, we may have a language that is able to not only throw exceptions but also to read internal stores. The first functionality uses the previously defined exception monad whilst the second involves the also previously defined state monad [52]. Indeed this problem of considering different functionalities in a single interpreter was already highlighted in different contexts [52, 49], in all cases *modularity* being a desirable tenet. In other words, the problem calls for techniques to combine the functionalities (*i.e.* the corresponding monads) in a principled way, a quest that has been taken using different approaches in the past years. We briefly overview some of them next.

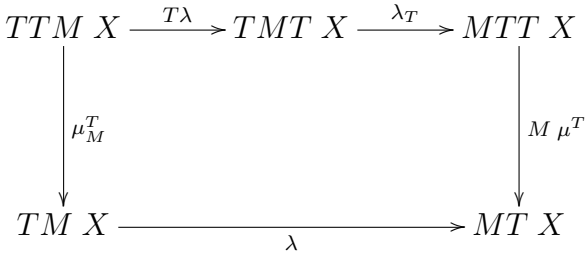
A standard way of combining monads is to see them as equational theories (whenever possible) and to take advantage of existing results from categorical algebra to combine them. This allows us to consider for example coproducts and tensors of monads (in algebraic form) [35]. One disadvantage of this approach is that in some cases it is not applicable [17]. Two other standard ways of combining monads is via the notion of a distributive law [17, 12, 16, 43] and monad transformers [34, 32]. We will focus on the last two, because we take advantage of them in our work.

Definition 7 (Distributive law)**.** The definition of the distributive law is given in [17]. If we have two generic monad instances, $\mathcal{M}$ and $\mathcal{T}$, and we wish to combine them into the monad $\mathcal{MT}$, then we would need to first define a function which allows the following transformation: $TM\ X \xrightarrow{\lambda} MTX$. In order to prove that this function is in fact a distributive we need to prove that the diagrams below commute:









The distributive law yields a new monad $\mathcal{MT}$ defined by the following triple [17]: $(MT, \eta_T^M \cdot \eta^T, M\ \mu^T \cdot \mu_{TT}^M \cdot M\lambda_T)$. Unfortunately such laws present limitations, specifically there might not always exist a distributive law between two monads, a prime example being the powerset and distribution monads [55]. Some alternative theories were thus created to cover some of these limitations, such as weak distributive laws [35, 41, 17]. Additionally, in [55] Zwart and Mardsen reveal a set of theorems which demonstrate the situations where it is impossible to define a distributive law, even covering some concrete monadic combinations where the specification of a distributive law fails.

Although not always rooted in mathematical foundations, monad transformers are used extensively in Haskell programming, specially in the cases concerning interpreters (mentioned above) [52]. These differ from distributive laws, as the idea is to create a transformer associated with different monads, such as the state monad transformer or the exception monad transformer [35, 34]. The following example gives a code fragment of the state monad transformer which receives a monad $\mathcal{T}$ and returns a new monad able to manipulate internal states:

Example 1 (State monad transformer)**.** The type below is the result of instantiating the state monad transform with a generic monad `t` (`StateT t a`), using Haskell's notation, (see more details in [34, page 338] and [35, page 136]):

```
State -> t (a, State)
```

The combined monad supports both the state monad operations as well as operations associated with the monad `t`. An interesting aspect to notice is that this transformer does correspond to any distributive law (see more in [35]), for the reason that the monad `t` is sandwiched between $(a, State)$ and $(-)^{State}$.

A downside of monad transformers is that we need to define the transformer for each monad we want to use, which in some cases its quite difficult to know how to do or even impossible (as it happens with the `IO` monad) [35]. Also a thing to be careful about monad transformers is that the order in which they are used matters [35]. In other words, if we have the stack of transformers `StateT State (ExceptT String Identity) a` we will obtain a value of the following type:

```
State -> Identity (Either String (a, State))
```

and now we switch the order of both transformers we to `ExceptT String (StateT State Identity ) a` we get,

```
State -> (Identity (Either String a, State))
```

Both two types are significantly different, and thus the order in which monad transformers are applied should be properly planned.

As mentioned before, in order to use monad transformers one needs to define them for the monads we want to use. In particular their definition requires us to define the lifting operation. Generically, this transformation follows the type `lift :: t a -> mt (t a)` which is described in more detail in [32]. In this case `t` is a generic monad and `mt` is the monad transformer. Let us illustrate this with an example.

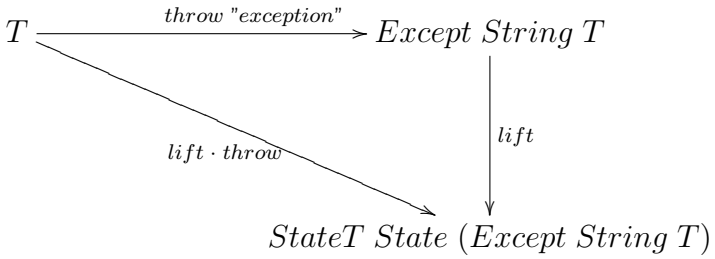
Example 2 (Application of the lift function)**.** As we mentioned before, every monad transformer has an associated lift function. In the case of the state monad transformer (`StateT`) that function has the following signature (with `t` being a generic monad):

```
lift :: t a -> StateT State t a
```

If we admit the exception monad transformer has a function named `throw`, which receives a message (String) and throws it as an exception, we would obtain a computation of type `Except String a`. If we wish combine this effect with the effect of the state monad, we could use the monad transformer mentioned above, resulting in the following signature:

```
StateT String (Except String) a
```

Now if we wanted to use the functions associated with the exception monad, such as the `throw` function, we have to use the `lift` function mentioned before, as we need to include an exception monadic computation into the state transformer:



In this case, T is the functor of a generic monad.

2.1.3 The power of monads in programming

Monad-based programming presents a large number of advantages. An important one is that functional languages tend to be pure, which means they do not allow side effects, by default, on their functions. As a result, it becomes problematic to implement certain features or programs that are usually easy to implement in other programming languages. Monads allow us to overcome this as they allow the introduction of side effects in a modular way as they handle all the complexity of performing impure operations [51, 52].

There are a lot of use cases for monad which illustrate their usefulness, but in order to be exhaustive, we will dive in into two which are relevant for the implementation of ProbLince. The first one, are the previously mentioned monadic interpreters and the second is monadic parsing.

Modular interpreters

The construction of an interpreter boils down to implementing a function that maps generic terms (of the underlying programming language) into computations. Let us illustrate this via a small example.

Example 3 (Simple monadic interpreter)**.** The following block of code presents the signature of a simple interpreter such as the one in [34], using Haskell's syntax. Monad $\mathcal{T}$ incorporates the effects needed to perform the operations of the language (we only consider the trivial effect for now). The interpretation function receives an arithmetic term (*i.e.* a generic term of the underlying 'programming language') and outputs a real number (`Double`) wrapped in a monadic computation.

```
type T = Identity
data AritTerm = Value Double
              | Sum AritTerm AritTerm
              | Sub AritTerm AritTerm

interpret :: AritTerm -> T Double
```

Listing 2.1: Simple monadic interpreter

In this case `AritTerm` designates arithmetic terms, specifically real numbers and operators for addition and subtraction. According to the specification above, the only thing the interpreter can do is to perform the calculation of arithmetic expressions and return their values. However, there are some expressions that should return an exception, such as division by 0. By updating the monad above to include the exception monad transformer, we extend the interpreter's capability to perform exception handling:

```
type T = ExceptT String Identity

data AritTerm = Value Double
              | Sum AritTerm AritTerm
              | Sub AritTerm AritTerm
              | Div AritTerm AritTerm

interpret :: AritTerm -> T Double
interpret (Value v) = return v
interpret (Sum t1 t2) = (+) <$> interpret t1 <*> interpret t2
interpret (Sub t1 t2) = (-) <$> interpret t1 <*> interpret t2
interpret (Div term1 term2) = do
    valueT1 <- interpret term1
```

```
        valueT2 <- interpret term2
        if valueT2 == 0
        then throwE " Division by 0 exception "
        else return $ valueT1 / valueT2
```

Listing 2.2: Adding error handling to the base monadic interpreter

As mentioned before there many *monad transformers* and not just `ExceptT`. The process of adding new computational effects however is always analogous: the monad gets updated to include a new transformer which adds new functionalities. This allows to build the interpreter modularly, with the obvious precaution on the order in which monad transformers are applied.

Monad-based Parsing

Another good example that shows the potential of monads is *monadic parsing* – this example is specially relevant for our interpreter as we also need to build a parser capable of handling the syntax of our while-language. In a nutshell, monadic parsing revolves around the idea of building complex parsers from simple ones via the use of combinators. D. Leijen and E. Meijer present in [33] a *Haskell* library, called Parsec [9], that allows the implementation of such parsers and furthermore provides mechanisms for producing detailed error messages. The whole library revolves around a monad built for this purpose (the `Parser` monad). There is a multitude of combinators that are capable of parsing characters, strings, digits, among others. However the most interesting feature is how they can be combined to build more complex parsers, a functionality facilitated by the underlying monad.

The choice combinator `(<|>)` is one of the main combinators of this library. The combinator receives two parsers and starts by applying the first. If it fails to parse the input string right in the beginning, without parsing a single character, then it applies the second one. However, if the first parser can make a partial parse of the input, then the returned value is just what the first parser was able to parse from the input [33]. To illustrate this, if there is a parser `x` that can parse the string "abc" and a parser `y` which parses the string "d", then applying the combinator `x <|> y` on the input string "abcd" would result in "abc".

Another interesting aspect to analyse is how this library is capable of handling left recursive grammars, as it is normal for them to be written that way [33]. A good example of a left recursive grammar is the one present in in [33, page 4] which implements simple arithmetic expressions:

```
    expr := expr "+" factor
    factor := "(" expr ")" | number
```

Listing 2.3: Left-recursive grammar for arithmetic expressions

The grammar above allows the parsing of inputs such as "1 + 2" or "1 + (2 + 3)". This parser is left-recursive, because the first step of the `expr` parser is to call itself again. The implementation of a parser for this language resorting to Parsec is not as simple as it seems. If we try to make a parser following the specification above we would end up with the following result:

```
exprParser :: Parsec String st Int
exprParser = (string "\n" *> return 0) <|>
    do
        v1 <- exprParser
        string "+"
        v2 <- factorParser
        return (v1 + v2)

factorParser :: Parsec String st Int
factorParser = between (string "(") (string ")") exprParser <|> ((read) <$>
    many digit)
```

Listing 2.4: parser with infinite recursion

The parser would check if its the end of the line, returning zero in that case, or it would apply a parser which would only be calling himself over and over again. To avoid this situation and still maintain the design of the left-recursive grammar, Parsec exports the `chainl1` combinator.

```
exprParser :: Parsec String st Int
exprParser = chainl1 factorParser (string "+" *> return (+))

factorParser :: Parsec String st Int
factorParser = between (string "(") (string ")") exprParser <|> ((read) <$>
    many digit)
```

Listing 2.5: Usage of the chainl function

This combinator receives two parsers, one for the `factor` and another for the "+" operator. In the cases the latter does not occur, then it returns the value of the `factor` parser. On the other hand, when the "+" is present (*e.g.* the operation "2 + 3") then the parser for the "+" operator returns a left associative function (in this case the (+) function) which is applied to the the value of the `factor` parser. Moreover, this combinator keeps applying these parsers until it reaches a point where does nothing else to parse, thus solving the aforementioned recursive problem.

To illustrate this, think of the operation "2 + 3". The result of applying the `chainl1` combinator with the parsers above, results in the value "(+) 2 3" embedded in a `Parsec` monad. Alternatively, if we had the

operation "2 + 3 + 4", the end result would be "(+) ((+) 2 3) 4" (again embedded in the monad `Parsec`).

Finally there is another use case that is very frequent in parsing systems, namely the *lookahead* operation. Parsers for LL grammars, which are parsed from left to right and need some sort of lookahead value, which looks a few tokens in advance, so they can find the right parsing rule to apply [33]. This is due to the fact that, sometimes, there are two rules which can be applied when parsing a certain string or file [33]. As an example, if we have a parser `x` which parses numbers and another parser `y` which parses strings that start by a number followed by letters (*e.g.* "2abc") and we decide to use the choice operator mentioned previously (`x <|> y`), the result would be the number "2" from the string and the rest would not be parsed. The result we wished for, however, is the application of the second parser. In contrast, if we switched the order of both combinators then if the input string was just a number, the combinator would consume the first number and then return an error as the second combinator could not be used. Parsec solves this issue by implementing the `try` operator: when used with a parser `x` it makes it so that when `x` fails, even when it consumes any type of input, everything is backtracked to the point where it was applied [33]. This would solve the problem in the previous example by applying the `try` operator to the first element of the `(<|>)` combinator: `try x <|> y`.

2.2 Probabilistic programming

2.2.1 An overview

Probabilistic programming is a paradigm that allows the usage of statistical concepts, such as distributions, and elements of programming, such as conditionals and while-loops, to build complex statistical models. This provides modularity and allows to capitalise on previous results on programming theory. For example, this paradigm allows certain fields, such as machine learning, to be able to compute and tweak statistical models referring to convoluted problems via certain programming techniques [40, 56, 26, 50].

Probabilistic programming languages usually include two important functions: one used to sample from a distribution (often called a *sampler*), and the other (called *conditioning*) to integrate posterior observations inside some condition or after some while loop, altering the probabilities of obtaining certain samples, as certain events become more likely to happen [40, 56]. These languages also implement some statistical primitives, like the *Bernoulli* or *Normal* distributions, and normally the problem is how to sample from these distributions. Some tools or libraries only deal with discrete probabilities, such as the one detailed in [20]. This is a monad-based library for probabilistic programming based on the idea of mapping events to their respective probabilities. In this case, getting a sample is relatively easy but in other cases

where continuous distributions are being used we cannot have the same type of implementation. This is due to not being always feasible to associate probabilities to every event. In this case these programming languages provide inference algorithms which are capable of retrieving samples from those distributions [56].

As examples of fully-fledged probabilistic programming languages we have Anglican [50, 54] and Church [26]. We also mention LazyPPL [40], Monad Bayes [56] and [20] as monad-based approaches which are Haskell libraries. While the latter only allows the use of discrete distributions the first two are able to handle both discrete and continuous ones. Moreover, Monad Bayes also lets us use more inference algorithms when compared to LazyPPL which only uses a version of the *Metropolis-Hastings* algorithm [40, 56].

2.2.2 Monads for probabilistic programming

We list now some monads used in probabilistic programming. The first one is the core element of the aforementioned library [20]. It revolves around the concept of associating to a given event the likelihood of it happening, creating the distribution monad ($\mathcal{D}$). This monad is defined (in Kleisli triple from) as follows: first given a function $\mu : A \to [0, 1]$ we define $\text{supp}(\mu) = \{a \in A \mid \mu(a) > 0\}$. Then we define the components of the Kleisli triple,

- $D\ A\ =\ \{\mu :\ A \to [0, 1] \mid \sum_{a\ \in\ \text{supp}(\mu)}\ \mu(a) = 1$ and $\text{supp}(\mu)$ is finite$\}$
- $\eta_X(a)(x) = \begin{cases} 1 & \text{if } x = a \\ 0 & \text{otherwise} \end{cases}$
- $f^*\ (\mu)(x) = \sum_{a \in \text{supp}(\mu)} \mu(a)\ \cdot\ f(a)(x)$

This monad is particularly important for our work, because we will use it in the operational semantics of our language, while the other libraries will be used in the the implementation part of our interpreter (the presentations of the following monads for probabilistic programming will thus be less formal and briefer).

Another approach is the one present in [40], where the core monad is called `Prob`. Contrary to the previous case, this library is capable of handling both discrete and continuous distributions. It also exports the functions sample and score with the behaviour described before, and the *Metropolis-Hastings* algorithm as a function which allows to draw samples from the distributions. One interesting aspect of this implementation is that it is lazy, so these samples are only going to get drawn when they are needed (when doing plots, for example) [40].

This lazy sampling technique was ported into the Monad Bayes library [7], through the `Sampler` monad, and is in general a more complete library. It revolves around the use of monadic type classes, extending the standard Haskell monad type class, which enforce the implementation of different probabilistic operations in instances of these classes [56].

Some of this classes allow us to sample or condition (in this cased it is called score) certain distributions, such as the `MonadDistribution` and `MonadFactor`, respectively. In both cases they are referred to as inference representations, and when both are combined we obtain the `MonadMeasure` class. Furthermore, we also have inference transformers, which add more functionalities to the representations, similarly to way monad transformers allow the combination of two monads [56, 7].

The library already instantiates some monad transformers under the classes mentioned above, such as the case for the writer monad transformer. Note, in the case of these instances they only happen when the transformers are instantiated by monads which are also instances of the `MonadDistribution` class. The `Enumerator` inference transformer (which is also a monad) presents an interesting functionality, as it register all the possible executions when sampling from a certain distribution. Moreover, it also allows the association of these executions to their respective probabilities of occurring [7].

In the block of code below we have a function which samples a value from a discrete binary distribution, including it in a writer computation. Note the lack of use of the lifting functions mentioned before, as the instantiation of the writer monad transformer with the inference representation implements it by default:

```
drawUniform :: WriterT String Enumerator Double
drawUniform = uniformD [0,1]
```

Listing 2.6: Drawing a writer computation from a discrete binary distribution

If we apply the function `explicit . runWriterT $ drawUniform` we obtain the following list: `[((0.0,"")` `,0.5),((1.0,""),0.5)]`. This indicates a distribution of two possible results, each of them with 50% of probability of occurring.

The `Sampler` monad mentioned before can be used similarly to the `Enumerator` monad, and will create a lazy sampler of these computations from the discrete distribution. By updating the example above we obtain the following:

```
drawUniform :: WriterT String Sampler Double
drawUniform = uniformD [0,1]
```

Listing 2.7: Implementing a lazy sampler of writer computations

Now we can use the functions associated with the `Sampler` monad such as the independent function,

which returns an infinite list of indenpendent samples from a distribution, and the sampler function, which transforms a `Sampler` monadic computation into an `IO` computation:

```
ghci> take 10 <$> (sampler . independent . runWriterT $ drawUniform)
[(0.0,""),(1.0,""),(0.0,""),(1.0,""),(1.0,""),(0.0,""),(0.0,""),(0.0,"")
    ,(0.0,""),(0.0,"")]
```

Listing 2.8: Obtaining 10 samples of writer computations from a binary distribution

In the example above we have taken the 10 first results of a list of infinite independent samples. Now, in this case we do not have the guarantee that our samples are biased or not. Because of this, we can use the inference algorithms of the library, such as the *Metropolis-Hastings* algorithm, to obtain a more adequate collection of samples [56].

2.3 Hybrid systems

2.3.1 The concept and main challenges

Hybrid systems are characterised by producing behaviour with both discrete and continuous components. From an engineering perspective, one can regard them as systems comprised of two parts: a physical one, which can be the motion of a vehicle or the temperature of a heater, and a digital part which controls (and is influenced by) the first [45, 44]. Specifically, the latter takes in the information of the cyber-physical interaction and uses it to make decisions that affect the behaviour of the physical part of the system. Acceleration and deceleration or stopping a cooling process are prime examples of this, among many others. A recurring concrete example of a hybrid system is that of a vehicle whose velocity is being monitored and controlled by a computer. In this case, the velocity corresponds to the physical effect that is changing at every time instant (continuous behaviour) and when it reaches a certain value the software triggers the brakes which slows down the vehicle (discrete behaviour) [46, 44]. Such a behavioural pattern is common in automotive systems.

There are many challenges associated with the design and analysis of such systems. These mostly arise from the complexity that gets introduced by combining two kinds of behaviour (*i.e.* continuous and discrete) which interact with each other in intricate and often surprising ways. Consequentially, this leads to the number one difficulty which is to find good formalisms to model correctly hybrid behaviour. Of course these formalisms are also required to facilitate the verification of the correctness of the model in a rigorous way. It is important to perform this verification for both types of behaviour simultaneously: doing

it so separately may result in incorrect conclusions [38, 46]. On one hand, just looking at the continuous behaviour results in imprecisions as the discrete behaviour affects, even if just from time to time, the continuous behaviour. On the other hand, by discretising things we loose information about the system that is crucial for the verification process [46].

2.3.2 Their role in modern technology

The relevancy of hybrid systems is largely associated with the vast range of fields they exist in and the number of examples we can find today, which will only continue to grow in the future. These systems can be seen in the medical field, with devices ranging all the way from pacemakers to robots capable of assisting surgeries in operating rooms. Additionally, self-driven vehicles (including cars, planes, and satellites) are also hybrid systems, as they have software on board that is responsible for changing the speed and movement of the vehicle in order to adapt to the circumstances of the surrounding environment [38, 47]. Yet another example is that of smart power grids and how they are used to control the flow of electricity to the places where it is most needed. In order to do this a proper management system and devices are installed for the purpose of monitoring the flow. This application could become very useful as the number of people that produce their own power, through solar panels for example, is increasing. These panels produce energy that could be used by the smart grid to perform a more efficient administration of the electricity flow [30]. Other examples can be checked in [31].

Notably in the power grid example malfunctions can lead to power outages, with direct impact not only on the general population but also on factories and other types of services [30, 47]. Actually all the examples above are critical systems, in the sense that a malfunction can result in catastrophic events. Safety concerns thus become a central aspect in the design of these systems [30]. Simple tests to ensure this safety are not enough as they might not cover all the different scenarios which these systems will face and so, some bugs might still be present in the final implementation [45]. In the words of E. Dijkstra "Program testing can be used to show the presence of bugs, but never to show their absence!" [19, 1970, page 7]. Due to this fact there is a pressing need to find more suitable ways of designing and verifying hybrid systems [45].

2.3.3 Modelling and analysis

As previously mentioned, there is a pressing need to find good formalisms for the design and analysis of hybrid systems. Extensive research has been done in this direction in the past few years. We give a brief overview next.

Hybrid automata and Kleene algebra

One of the main topics of this research revolves around the idea of modelling hybrid systems as automata with hybrid characteristics [25]. Specifically this theory is described in [27] where T. Henzinger presents the idea of modelling a hybrid system as a graph in which the nodes represent locations and the edges represent possible transitions (discrete behaviour) between locations. Continuous behaviour is specified via differential equations inside locations.

Example 4 (Hybrid automaton that models a car braking system)**.** The following hybrid automaton is inspired in the thermostat example present in [27] and in the train control example in [46] and models a car braking system:

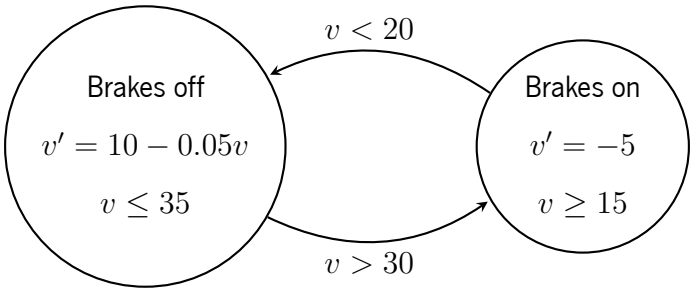


Figure 2: Car braking system modelled by an hybrid automaton

The automaton presents two valid locations in which the system can be in. In the first one the brakes are off and the car is gaining speed according to the respective differential equation. As for the second one, the braking system is triggered and thus speed decreases. There is one transition each way dictating when the braking system can be (de)activated. Finally, there is information in the locations namely conditions (often called invariants) which have to be true when the system is in that location. Note that when the system is in a location and the respective invariant becomes false the system is forced to leave that location. This example shows perfectly how the formalism of hybrid automata tries to incorporate the prominent aspect of hybrid systems: continuous behaviour (physical process) is specified by the differential equations in the locations and the discrete behaviour (digital process) affects it by switching the location and thus the 'active' differential equation.

The authors in [23, 25] refer to another relevant topic in the quest for hybrid systems formalisms: the idea of creating algebras to model hybrid systems, such as the ones present in [29] and [46], based on Kleene algebra theory.

Non-determinism vs. a modular perspective of hybrid behaviour

The aforementioned formalisms have an important drawback: non-determinism is present by *default* in the corresponding models. This makes it difficult to study *pure* hybrid computation which is always mingled with non-determinism [23]. An alternative approach, advocated for example in [23, 25], is to study first pure hybrid computation and subsequently its combination with other effects (including non-determinism). It is actually possible to draw a parallel between this idea of stacking effects and a central topic of monad-based programming, namely that of monad composition. In this case, we could have for example a monad for hybrid behaviour and another for non-determinism or probabilistic behaviour and consider a proper combination [23]. This more modular approach towards hybrid systems has the benefit of separating challenges conceptually and at different levels. This has implications for example in the development of semantics for hybrid systems [23, 25]. Also for our case it has strong implications in what concerns the addition of probabilistic effects to hybrid systems: it is certainly easier to combine probabilities with pure hybrid behaviour than to combine probabilities with hybrid behaviour mingled with non-determinism. Indeed the task of combining probabilities with non-determinism alone has already made extensive amounts of ink flow [16].

2.4 Hybrid programming

Now that we have established what a hybrid system is and what monad-based programming consists of, we analyse the connection between these two fields. More concretely in this section we recall the concept of hybrid programming and how it can be frammed in a monadic setting.

2.4.1 An overview

Hybrid programming is based on the idea of extending programming languages with differential equations. Similarly to probabilistic programming (recall our previous description), this allows to take advantage of concepts and techniques from programming theory to model and analyse hybrid systems [25].

Example 5 (Hybrid Program)**.** An example of a hybrid program is given in [25, page 1 (Online version)]:

```
while true do {
    if v <= 10
    then (v' = 1 for 1)
    else (v' = -1 for 1)
}
```

Listing 2.9: Cruise controller

It implements a cruise control system in which the velocity of the vehicle (variable v) is controlled by accelerating the vehicle for one time unit if the observed value is less or equal than 10, or decelerating for the same amount of time if the velocity rises above 10. The program shows exactly the key points of hybrid programming in the sense that it allows the use of classical computing elements, in this case while-loop and conditionals, with other elements that are connected to a physical process, *i.e.* the continuous change in velocity through the use of differential equations.

As an example of hybrid languages we have HybCore, a deterministic language that follows a monadic approach, specifically it involves a monad that frames hybrid behaviour as a computational effect. This language is particularly useful as it integrates the tool Lince, which allows to analyse hybrid programs via simulations of their execution [25]. There are also hybrid programming languages that follow a non-monadic approach and are also non-deterministic, such as the ones described in [46, 15]. Additionally there are DSLs embedded into Haskell, such as Yampa [14, 28], which uses combinators to compose specific forms of hybrid behaviour. Yampa shows an interesting approach to how a DSL could be implemented; in fact it inspired the development of our own DSL (detailed in Chapter 5).

In order to verify a hybrid program it is important that the underlying programming language has a well-established semantics. The latter can be divided into different categories but here we focus on two: operational semantics and denotational semantics [25, 53].

Operational semantics reveals the reduction steps that occur in a program's execution and how they affect the internal state of the computer on which the program is running. This is useful to develop interpreters for the language as it shows exactly how a program should behave over time [37, 25]. As for the denotational semantics, it consists on providing mathematical foundation to the language. This particularly useful, as it allows us to reason about the programs written in this language abstractly, without being encumbered by syntactic particularities. This is useful to not only better understand the language but also to reason about the correctness of programs and the equivalency between two programs. In the case of the HybCore, the denotational semantics revolves around a specific monad which will be detailed next [25, 23, 37].

2.4.2 Monads for hybrid programming

Now that both types of semantics were briefly introduced, we will focus more in the denotational part as it is where we can observe how a specific monad can be used to capture hybrid effects. Interestingly, this monad is a particular instance of the generalised writer monad (which described in the definition 6), and involves a monoid-module of trajectories [23, 25] which we detail next. The basic idea of this monoid-module is that hybrid behaviour can be encoded in trajectories $[0, r) \to \mathbb{R}^n$ where $r \in \mathbb{R}_{\geq 0}$ and n is a natural number. The value r can be seen as the duration of the hybrid effect whilst n corresponds to the number of variables being changed over time. It is possible that the hybrid effect runs *ad infinitum* (think for example of while-loops) so we also wish to consider trajectories of the type $[0, \infty) \to \mathbb{R}^n$. Now, there is clearly a trivial trajectory $[0, 0) \to \mathbb{R}^n$ which corresponds to an instantaneous hybrid effect. And we can concatenate trajectories which corresponds in some sense to 'extending over time' an hybrid effect with another. Let us give a formal definition of this description

Definition 8 (Monoid module of trajectories)**.** Fix a natural number n. The monoid component is the triple $(\coprod_{r \in \mathbb{R}_{\geq 0}} [0, r) \Rightarrow \mathbb{R}^n, +\!\!+, ! : [0, 0) \to \mathbb{R}^n)$ where $+\!\!+$ is defined by,

$$(r_1, e_1) +\!\!+ (r_2, e_2) = (r_1 + r_2, t \mapsto \text{ if } t < r_1 \text{ then } e_1\ (t) \text{ else } e_2\ (t - r_1))$$

The module component is given by the set $\coprod_{r \in [0,\infty]} [0, r) \Rightarrow \mathbb{R}^n$ together $+\!\!+$ extended in the obvious way as the module operation.

In the next chapter we will give concrete details on how this monad can be used to give semantics to a hybrid programming language. For now, it is relevant to mention that although this monad is suitable to handle hybrid effects, it raises problems in its Haskell implementation. More concretely, after joining two trajectories with the multiplication function, one cannot discern exactly the instant where the first trajectory ends and the second one starts. This leads to problems related to the visualisation of trajectories. We thus changed the definition of the previous monoid-module to another which is capable of handling hybrid effects and does not carry the described problem. We will give more details about this problem and the altered monoid-module in the following chapter.

Chapter 3

A probabilistic while language with hybrid behaviour

As mentioned in Introduction, our main goal is to implement a language that harbours both probabilistic and hybrid effects. In order to accomplish this we will first look at a pre-existent language that supports probabilistic choice and a very specific hybrid effect: that of wait calls which halt a machine for a specific amount of seconds [48]. We will first detail the syntax of this language and then its semantics.

Subsequently, we will go over the necessary changes to the syntax and semantics of the language which will allows us to fully incorporate both the hybrid and probabilistic effects.

Finally, the last section of this chapter covers the denotational semantics which serves as a mathematical foundation for this language, using a monadic approach such as the one present in [25].

3.1 Combining probabilistic behaviour with program duration

3.1.1 Syntax

We first focus on the syntactic structure [48] of what we call deterministic terms. These can be either real numbers ($\mathbb{R}$), variables (of a predetermined countable set *Var*) or applications of arithmetic operations. More formally deterministic terms are described by the grammar:

```
d ::= a          a ∈ ℝ
    | x          x ∈ Var
    | d op d     op ∈ {+,-,×,÷}
```

Listing 3.1: Deterministic terms syntax

The programming language supports more general terms which can be either deterministic terms or probabilistic terms; the latter case amounts to a `coin()` operator which implements an unbiased coin toss by returning a random binary value. These more general terms also allow the usage of basic arithmetic operations:

```
t ::= d                deterministic term
    | coin()           sample from {0,1}
    | d op d           op ∈ {+,-,×,÷}
```

Listing 3.2: Terms syntax

Booleans conditions are also present in the language. We can observe in the grammar below that one is able to perform comparisons between deterministic terms or use standard logical operators like the 'or operator (||)' between tests:

```
b ::= true
    | false
    | d == d | d < d | d > d      comparison of deterministic terms
    | b && b | b || b |!b         Boolean combinations of tests
```

Listing 3.3: Tests syntax

Finally we present the grammar associated to programs, which reveals not only standard program constructs but wait calls as well:

```
e ::= skip
    | x := t                  assignment
    | e ; e                   sequential composition
    | if b then e else e      conditional
    | while b do e            while loop
    | wait t do e             wait calls
```

Listing 3.4: Programs syntax

3.1.2 Operational semantics

In this subsection we present operational semantics for the previous language in two different styles: the first uses a random number generator (RNG) m to calculate the output of a program that starts its execution in some initial state s. Technically, the RNG consists of an infinite list of random binary values which will be used to interpret the coin toss operation. States on the other hand consist of maps between variables and real numbers, and can be seen as the computer's memory. The second style of semantics discards the RNG and returns not a single output but a distribution of outputs. Note that, the first operational semantics is part of a pre-existent work [48], as we mentioned before.

Notably, the language presented above reveals some cases where errors might occur (*e.g* division by 0). In this case, we designed both the operational and denotational semantics as if they do not happen, because we are only trying to capture the hybrid and probabilistic behaviours. With respect to the implementation, we implemented the capability of handling errors to produce a more complete tool (catching the division by 0 cases), even though these semantics do not specify them.

Both styles of semantics have their merits. The first is closer to how probabilistic languages are actually implemented and has a good performance w.r.t. execution. The second has a worse performance but is closer to a monadic style of programming which allows us to appeal to several results about monads (for example monad composition). In both cases the semantics boil down to a set of deductive rules describing the outputs of a program under specific assumptions – the computation of the output a program will thus consist in a proof tree of these deductive rules.

Let us make the previous explanations about the two semantics more concrete by presenting them in full detail next. We start with the semantics that uses a RNG. The expression $(e, s, m) \Downarrow (s', m', d)$ denotes that program e in initial state s and with generator m returns the new state s' in d seconds; its execution consumes part of the generator which results in m'. Furthermore, the operation $[\![t]\!]\ (s, m)$ obtains the value of the term t when given a state s and the probabilistic resource m. The same thing applies to the test terms, where $[\![b]\!]\ (s, m)$ obtains the boolean value of the term b. Let us now present the deductive rules for this language [48].

The first rule refers to the assignment of values to variables. In this case, we assign the value t to the variable x_i. Firstly, this program needs to obtain the value of the term t using the initial resources (s, m), which gives a value a and the possibly altered value of the probabilistic resource m'. The output of this program will then change the state of the variables by assigning the value a to the variable x_i. The program is instantaneous which results in it being associated with a duration of zero seconds:

$$\text{(asg)}\ \frac{[\![t]\!](s, m) = (a, m')}{(x_i := t, s, m) \Downarrow (s[i \to a], m', 0)}$$

The following rule refers to the situation where we are sequencing the program $skip$ with any other. Note that $skip$ is a trivial command that does not take any time to execute and does not alter any of the resources. In contrast, the generic program e can be a wait call or a variable assignment or even a program which does a coin toss, so we can have different types of outcome. The outcome of the sequence between of $skip$ and e will be the outcome of the program e:

$$\text{(skip)}\ \frac{}{(skip, s, m) \Downarrow (s, m, 0)}$$

As for the sequencing between two generic programs e_1 and e_2 we need to run the first program and then feed its outcome to the second one. In the end, the duration of this sequence will be the sum of the duration of both programs:

$$\text{(seq)}\ \frac{(e_1, s, m) \Downarrow (s', m', d') \qquad (e_2, s', m') \Downarrow (s'', m'', d'')}{(e_1; e_2, s, m) \Downarrow (s'', m'', d' + d'')}$$

The following rule covers the wait call mechanism. It requires first to obtain the value of the term t, which can change the probabilistic resource m as explained before. Then we need to execute the program e and obtain its result as this will be used in the conclusion of the rule. Finally, the final duration will be equal to the sum between the time it took to run the program e and the value of the term t:

$$\text{(wait)}\ \frac{[\![t]\!](s, m) = (a, m') \qquad (e, s, m') \Downarrow (s', m'', d')}{(\text{wait } t \text{ do } e, s, m) \Downarrow (s', m'', a + d')}$$

As for the if statement rules, they are divided into two, the case where the associated Boolean condition is true and the one where it is false. If the first case occurs the program which will be executed will be e_1 and so the output of the conditional statement will be the output of this program. A similar case happens when the condition is false, but in this case it is e_2 that executes:

$$\text{(if-true)}\ \frac{[\![b]\!](s, m) = true \qquad (e_1, s, m) \Downarrow (s', m', d')}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m) \Downarrow (s', m', d')}$$

$$\text{(if-false)}\ \frac{[\![b]\!](s, m) = false \qquad (e_2, s, m) \Downarrow (s', m', d')}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m) \Downarrow (s', m', d')}$$

As for the while loop rules, we can also verify the existence of two separate rules, one where the condition of the loop is true and so we need to perform one iteration of the program inside its body, and another where the condition is false and so the loop ends. In the first case the output of the program is the output of the program e, whilst in the latter the program does nothing to both the memory and the probabilistic resource and has a duration of zero:

$$\text{(wh-true)}\ \frac{[\![b]\!](s, m) = true \qquad (e; \text{while } b \text{ do } e, s, m) \Downarrow (s', m', d')}{(\text{while } b \text{ do } e, s, m) \Downarrow (s', m', d')}$$

$$\text{(wh-false)}\ \frac{[\![b]\!](s, m) = false}{(\text{ while } b \text{ do } e, s, m) \Downarrow (s, m, 0)}$$

We will now present the second style of semantics, which is our redefinition of the semantics above, by discarding the RNG and returning not a single output but a distribution of outputs. In this setting it is often convenient to represent a distribution $\mu : A \to [0, 1]$ in algebraic form,

$$\mu(a_1) \cdot a_1 + \cdots + \mu(a_n) \cdot a_n$$

where $\{a_1, \ldots, a_n\} = \text{supp}(\mu)$. Then in this semantics the expression $(e, s) \Downarrow \sum_i^n p_i \cdot (s_i, d_i)$ denotes that program e in initial state s outputs a distribution over states and durations. Like with the first semantics we will now present the corresponding deductive rules and their rationale. So starting with the assignment rule, the difference we observe is that the term t will now produce a distribution over the real numbers. The assignment will produce a distribution of outputs, and obviously, in all of the possible outcomes of this distribution the time it takes to run is always zero as the assignment is an instantaneous operation.

$$\text{(asg)}\ \frac{[\![t]\!](s) = \sum_i^n p_i \cdot a_i}{(x_k := t, s) \Downarrow \sum_i^n p_i \cdot (s[k \to a_i], 0)}$$

As previous explained, the output of the program corresponding to the sequence of the program $skip$ with a generic program e will always result in the output of the program e. So, the only thing we need to update is that now the program e will produce a distribution of outcomes.

$$\text{(skip)}\ \frac{}{(skip, s) \Downarrow 1 \cdot (s, 0)}$$

As for the sequence of two generic programs, e_1 and e_2, we will feed the outcome of the first to the latter as before. However, the first program will now produce a distribution and so we need to apply the program e_2 to each one of these possible outcomes which will result in a new distribution. So, in the end we will have a distribution of distributions. To "flatten" this result we can multiply the probability of each event present in the distribution produced by e_2 with the probability of that distribution happening (given by e_1):

$$\text{(seq)}\ \frac{(e_1, s) \Downarrow \sum_i^n p_i \cdot (s_i, d_i) \qquad \forall_{i \leq n} (e_2, s_i) \Downarrow \sum_j^m q_j \cdot (s_j, d_j)}{(e_1; e_2, s) \Downarrow \sum_i^n \sum_j^m p_i \cdot q_j \cdot (s_j, d_i + d_j)}$$

The wait call rule follows the same logic as before, in the sense that the interpretation of the term results in a distribution of possible values. Therefore, we will apply the program e to each one the possible values of the distribution and it will result again in a distribution of distributions which we need to "flatten" like in the previous rule:

$$\text{(wait)}\ \frac{[\![t]\!](s) = \sum_i^n p_i \cdot a_i \qquad (e, s) \Downarrow \sum_j^m q_j \cdot (s_j, d_j)}{(\text{wait } t \text{ do } e, s) \Downarrow \sum_i^n \sum_j^m p_i \cdot q_j \cdot (s_j, a_i + d_j)}$$

The semantic rules for if statements and while loops are analogous to the previous operational semantic rules except that now they will produce distributions of outcomes instead of a deterministic ones:

$$\text{(if-true)}\ \frac{[\![b]\!](s) = true \qquad (e_1, s) \Downarrow \sum_i^n p_i \cdot (s_i, d_i)}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m) \Downarrow \sum_i^n p_i \cdot (s_i, d_i)}$$

$$\text{(if-false)}\ \frac{[\![b]\!](s) = false \qquad (e_2, s) \Downarrow \sum_i^n p_i \cdot (s_i, d_i)}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m) \Downarrow \sum_i^n p_i \cdot (s_i, d_i)}$$

$$\text{(wh-true)}\ \frac{[\![b]\!](s) = true \qquad (e;\ \text{while } b \text{ do } e, s) \Downarrow \sum_i^n p_i \cdot (s_i, d_i)}{(\text{while } b \text{ do } e, s) \Downarrow \sum_i^n p_i \cdot (s_i, d_i)}$$

$$\text{(wh-false)}\ \frac{[\![b]\!](s) = false}{(\ \text{while } b \text{ do } e, s) \Downarrow 1 \cdot (s, 0)}$$

3.2 Incorporating full hybrid behaviour

Now that we developed operational semantics for a language capable of handling both coin tosses and wait calls, we will focus on adding a new layer of complexity which is the incorporation of trajectories (resulting in full hybrid behaviour mixed with coin tosses). This will allow us to model more interesting hybrid systems such as cruise controllers.

3.2.1 Changes to the syntax

Before moving on to the new operational semantics we need to alter the syntax of the previous language to accommodate differential statements. We will use the latter to encode the continuous dynamics of physical processes such as time, movement, and temperature. To this effect, we will adopt the idea present in the HybCore language [23, 25], specifically the use of systems of differential equations that are 'active' for predetermined amount of time given by a real number. Actually we go further than HybCore by allowing this predetermined value to be encoded not just as a real number but as a probabilistic term.

```
e ::= skip
    | x := t                                  assignment
    | e ; e                                   sequential composition
    | if b then e else e                      conditional
    | while b do e                            while loop
    | wait t do e                             wait calls
    | x0' = dt0,..., x'n = dtn for t system of differential equations
```

Listing 3.5: New programs syntax

In the previous grammar, each system of differential equation is comprised of variable derivatives on the left-hand side and specific terms on the right-hand side that will be described next via the following grammar.

```
dt ::= v                    real number
     | x                    x ∈ Var
     | _time_               Time keyword
     | dt op dt             op ∈ {+,-,×,÷}
```

Listing 3.6: Syntax of differential terms

With this grammar the language supports differential equations such as $x' = 2 \cdot v$ and $y'(t) = 3 \cdot t$. To specify for example the continuous dynamics of the movement of a vehicle we can write down the system $p' = v, v' = a$ where p is the vehicle's position, v the velocity, and a the acceleration. If we then wish to consider this continuous dynamics for precisely one second we simply write down as a program `p' = v, v' = a for 1`.

3.2.2 Operational semantics

After updating the syntax we need to revisit the operational semantics in order to accommodate differential statements. An interesting characteristic present in HybCore [25] is that it allows the specification of infinite while loops (*i.e.* the corresponding Boolean condition is always true), without resulting in what is called unproductive divergence (*i.e.* the program runs *ad infinitum* without yielding any information). To incorporate this characteristic, we adopt a similar semantics to the one presented in [25, Figure 2, page 7 (Online version)] which has the additional benefit of providing a clear set of rules for implementing the interpreter.

Unlike the previous semantics for wait calls (which informs about the duration of a program) the new semantics needs to accommodate information about differential statements. As we will see this requires that the new semantics receives as parameter the time instant t at which the program at hand needs to be evaluated, or in other words the output that the program provides at time instant t. Then by a successive application of the semantic rules this time instant will be decremented (according to the durations of program instructions) until reaching 0 and at this stage we return the current output. Let us analyse this procedure via an example, by specifying a program which performs a wait call of two seconds:

```
x := 1;
wait 2 do {
    x := coin()
}
```

Listing 3.7: wait call example

It is clear that for any time instant $t < 2$ the semantics should return 1 for variable x and for $t = 2$ return either 0 or 1 depending on the RNG. Now, if we alter this program and include it in an infinite while loop we obtain the following:

```
x := 1;
while true do {
    wait 2 do {
        x := coin()
    }
}
```

Listing 3.8: Infite loop of wait calls

If we wish to evaluate at $t = 4$ we need to unfold the loop twice, each iteration taking precisely two seconds. The semantic rules handle this loop unfolding, in particular they calculate how many times the loop needs to be unfolded by subtracting from the point of evaluation the duration of each iteration (of the loop) until reaching 0. In general the semantic rules will also need to determine when the desired time instant of evaluation has been surpassed. This can happen for example in the previous case if $t = 3$ since each loop iteration has the duration of two seconds. To signal this occurrence in program evaluation the semantics rules will use a $stop$ flag. The $skip$ flag will be used to signal that the point of evaluation has not been surpassed. Such a strategy was also adopted in [25].

Taking into account the previous description, in the new semantics the expression $(e, s, m, t) \Downarrow (r, s', m', 0)$ $(r \in \{skip, stop\})$ indicates that program e with initial state s and RNG m outputs s' at time instant t. The value m' is the generator that results from the application of m to the coin tosses that arise from evaluating e. Then the expression $(e, s, m, t) \Downarrow (r, s', m', t')$ with $t' \neq 0$ denotes a *partial evaluation* of the program: *i.e.* the evaluation is unfinished and s' is not necessarily the output at time instant t but a previous time instant namely $t - t'$.

We now analyse each semantic rule in detail and compare it with the original duration-based semantics. We start by looking at the assignment rule, which will now use the aforementioned time instant of evaluation d in conjunction with the other elements of the original semantics. Assignments are instantaneous and thus there is no time progression. This is reflected in the corresponding semantic rule which is presented next.

$$\text{(asg)}\ \frac{[\![t]\!](s, m) = (a, m')}{(x_i := t, s, m, d) \Downarrow (skip, s[i \to a], m', d)}$$

Like before, the semantic rule for the program $skip$ does not make any changes to the input, furthermore it is instantaneous. This corresponds to the following rule.

$$\text{(skip)}\ \frac{}{(skip, s, m, d) \Downarrow (skip, s, m, d)}$$

In the case of the sequence operator, we now divide the rule into two. One where the first program of the sequence originates an output with the continuation flag set to $skip$ and another with the flag set to $stop$. In the first situation the corresponding output is given to the second program e_2 and the final output is that of e_2. If the flag is $stop$ then the evaluation should stop and thus the output of e_1 is returned.

$$\text{(seq-skip)}\ \frac{(e_1, s, m, d) \Downarrow (skip, s', m', d') \qquad (e_2, s', m', d') \Downarrow (skip, s'', m'', d'')}{(e_1; e_2, s, m, d) \Downarrow (skip, s'', m'', d'')}$$

$$\text{(seq-stop)}\ \frac{(e_1, s, m, d) \Downarrow (stop, s', m', d')}{(e_1; e_2, s, m, t) \Downarrow (stop, s', m', d')}$$

Next, to not overburden notation we abbreviate the differential statement $x_1' = t_1, \ldots, x_n' = t_n$ for t simply into $\overline{x}' = \overline{u}$ for t. Recall as well that given a system of differential equations $x_1' = t_1, \ldots, x_n' = t_n$ for t we assume the existence of a unique solution $\phi : \mathbb{R}^n \times [0, \infty) \to \mathbb{R}^n$. Then to interpret this differential statement, we first interpret the term t and compare the resulting value with the point of evaluation. If the value of the term is equal or smaller, then the computation proceeds with a $skip$ flag, and the variables are altered according to the solution ϕ. We also subtract the value of the term from the point of evaluation, in other words we get closer to the point of evaluation. If on the other hand the value of the term is strictly greater than the point of evaluation, the computation stops with the $stop$ flag and the variables are updated accordingly.

$$\text{(diff-skip)}\ \frac{[\![t]\!](s, m) \Downarrow (a, m') \qquad a \leq d}{(\overline{x}' = \overline{u} \text{ for } t, s, m, d) \Downarrow (skip, s[\overline{x} \to \phi(s, a)], m'', d - a)}$$

$$\text{(diff-stop)}\ \frac{[\![t]\!](s, m) \Downarrow (a, m') \qquad a > d}{(\overline{x}' = \overline{u} \text{ for } t, s, m, d) \Downarrow (stop, s[\overline{x} \to \phi(s, d)], m'', 0)}$$

Regarding conditionals and while-loops, the rules are completely analogous to those of the original semantics. They are presented next.

$$\text{(if-true)}\ \frac{[\![b]\!](s, m) = true \qquad (e_1, s, m, t) \Downarrow (r, s', m', t')}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m, t) \Downarrow (r, s', m', t')}\ (r \in \{skip, stop\})$$

$$\text{(if-false)}\ \frac{[\![b]\!](s, m) = false \qquad (e_2, s, m, t) \Downarrow (r, s', m', t')}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m, t) \Downarrow (r, s', m', t')}\ (r \in \{skip, stop\})$$

$$\text{(wh-true)}\ \frac{[\![b]\!](s, m) = true \qquad (e;\ \text{while } b \text{ do } e, s, m, t) \Downarrow (r, s', m', t')}{(\text{while } b \text{ do } e, s, m, t) \Downarrow (r, s', m', t')}\ (r \in \{skip, stop\})$$

$$\text{(wh-false)}\ \frac{[\![b]\!](s, m) = false}{(\ \text{while } b \text{ do } e, s, m, t) \Downarrow (skip, s, m, t)}$$

We have just provided an operational semantics for probabilistic hybrid programs based on a RNG. Analogously to the previous section, we will now pursue an alternative view in which the generator is discarded and the outputs become probabilistic. We analyse each semantic rule in detail next.

Regarding the assignment rule, the outcome consists in a distribution of tuples where the continuation flag is always $skip$ and time never progresses:

$$\text{(asg)}\ \frac{[\![t]\!](s) = \sum_i^n p_i \cdot a_i}{(x_j := t, s, d) \Downarrow \sum_i^n p_i \cdot (skip, s_i[j \to a_i], d)}$$

As for the rule of program $skip$, we removed the probabilistic resource m and changed the output to become a Dirac distribution with the same value as the input.

$$\text{(skip)}\ \frac{}{(skip, s, d) \Downarrow 1 \cdot (skip, s, d)}$$

The rule for the sequential composition of two programs suffered major changes because we can not separate it into two rules as before. This is because we now have to take into account that some of the events in the distribution produced by the evaluation of program e_1 can have $stop$ as the continuation flag. So, we need to analyse first which events present the $skip$ flag and only in those cases evaluate program e_2. To tackle this assume that $(e_1, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i)$ and let $k \subseteq n$ be the subindex of the events in the distribution that possess the $skip$ flag. Then the semantic rule is defined by,

$$\text{(seq)}\ \frac{(e_1, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i) \qquad \forall_{i \in k}\ (e_2, s_i, d_i) \Downarrow \mu_i}{(e_1; e_2, s, d) \Downarrow \sum_{i \in k} p_i \cdot \mu_i + \sum_{i \in n \setminus k} p_i \cdot (stop, s_i, d_i)}$$

Let us now focus on the rule concerning differential statements. Analogously to previous rule we need to join both the diff-skip and diff-stop rules, present in the previous semantics. This happens because in a differential statement $\overline{x}' = \overline{u}$ for t the term t originates a distribution of values and in some cases those values surpass the point of evaluation (given by the element d). Thus let $[\![t]\!](s) = \sum_i^n p_i \cdot a_i$ and let $k \subseteq n$ be the subindex of the values a_i that not surpass d. We then define:

$$\text{(diff)}\ \frac{[\![t]\!](s) \Downarrow \sum_i^n p_i \cdot a_i}{(\overline{x}' = \overline{u} \text{ for } t, s, d) \Downarrow \sum_{i \in n} \begin{cases} p_i \cdot (skip, s_i[\overline{x} \to \phi(s_i, a_i)], d - a_i) & \text{if } i \in k \\ p_i \cdot (stop, s_i[\overline{x} \to \phi(s_i, d)], 0) & \text{otherwise} \end{cases}}$$

Finally the rules for conditionals and while loops are very simple to update as shown next.

$$\text{(if-true)}\ \frac{[\![b]\!](s) = true \qquad (e_1, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i)}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i)}$$

$$\text{(if-false)}\ \frac{[\![b]\!](s) = false \qquad (e_2, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i)}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i)}$$

$$\text{(wh-true)}\ \frac{[\![b]\!](s) = true \qquad (e;\ \text{while } b \text{ do } e, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i)}{(\text{while } b \text{ do } e, s, d) \Downarrow \sum_i^n p_i \cdot (r_i, s_i, d_i)}$$

$$\text{(wh-false)}\ \frac{[\![b]\!](s) = false}{(\text{while } b \text{ do } e, s, d) \Downarrow 1 \cdot (skip, s, d)}$$

3.3 Denotational semantics

After developing a suitable operational semantics for a language capable of handling both hybrid and probabilistic behaviour, we also developed a denotational semantics for the language. Although the specification of a denotational semantics was not necessary to implement this language (the operational semantics are enough to do it), we decided to go ahead and do it in order to create a mathematical foundation for the language.

3.3.1 Denotational semantics based on a random number generator

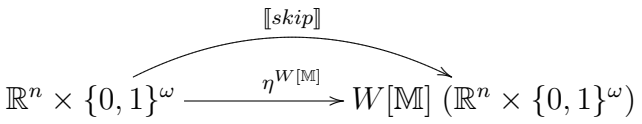
Similarly to the operational semantics we also divided the denotational semantics in two cases: one in which we use a RNG and the other in which the outputs of a program are distributions and not singular values. Analogously to [25], we will use a function $p \mapsto [\![p]\!]$ which maps a program p to its interpretation $[\![p]\!]$ (also known as denotation). This function will be used to interpret every program construct, such as conditionals and while-loops, and the set of all resulting interpretations is the denotational semantics of our while-language. We start by describing the denotational semantics based on the RNG.

In a nutshell, the semantics based on a RNG regards program denotations $[\![p]\!]$ as functions of the type $\mathbb{R}^n \times \{0,1\}^\omega \to W[\mathbb{M}](\mathbb{R}^n \times \{0,1\}^\omega)$. In more detail, we use the generalised writer monad instantiated with the monoid-module of trajectories to produce and concatenate the trajectories that programs output. Then the inputs for these programs are pairs comprised of the values of all variables in the computation (*i.e.* the internal memory) $\mathbb{R}^n$ and an infinite list of binary values $\{0,1\}^\omega$, which will be used to make random decisions (*i.e.* it corresponds to the random number generator which will be used in coin tosses). Finally the programs output the same type of pair, corresponding to the new internal memory of the machine and the RNG.

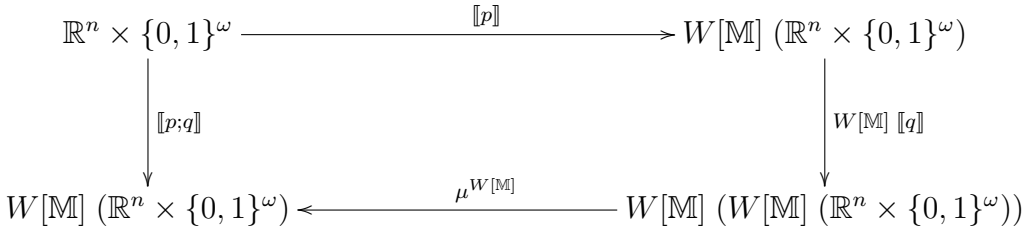
It is also important to mention that both denotational semantics assume that systems of differential equations in the language always have a unique solution $\phi : \mathbb{R}^n \times [0, \infty) \to \mathbb{R}^n$.

Let us now analyse the semantics of each program construct. Regarding the program "skip", it

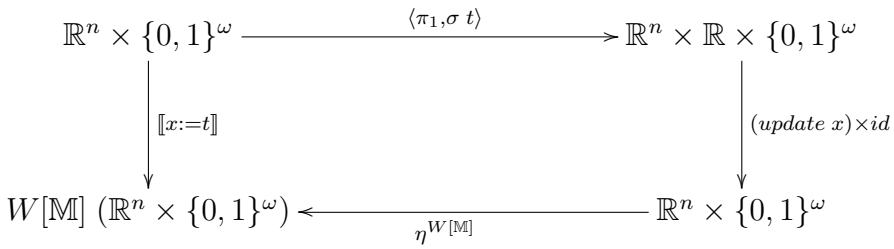
corresponds to just applying the unit function of the monad, because this program intuitively does nothing:



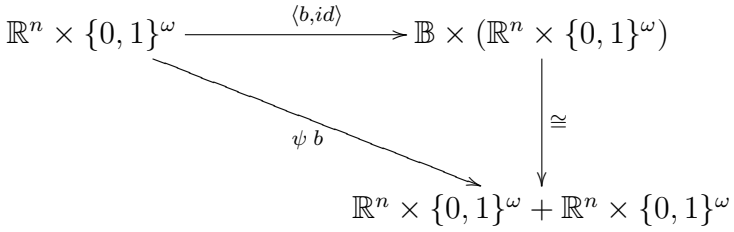
As for the sequence of two programs p and q we start by first interpreting the first program and then interpreting the second one. Note that we need to use the multiplication function of the adopted monad to merge these two computations into one:



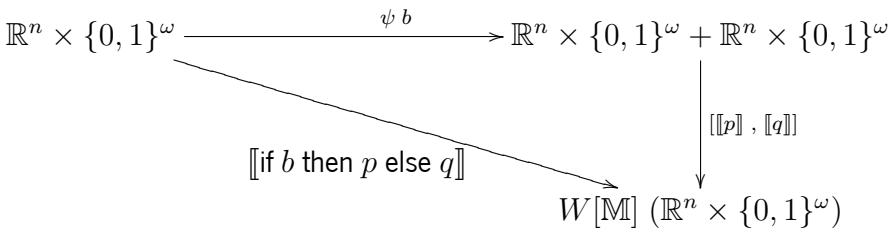
In what regards assigments $x := t$, we first need to interpret the term t. We do this via a function σ of the type $\mathbb{R}^n \times \{0,1\}^\omega \to \mathbb{R} \times \{0,1\}^\omega$ which was already used in the previous operational semantics. Afterwards, a function $update$ is used to change the value of the variable x according to the value provided by σ. To sum up, we have:



Next, in order to interpret conditionals we will need a function ψ to interpret the Boolean terms associated to conditionals. Such a function is described in the following diagram:



Then for the interpretation function of conditionals we have the diagram:



Finally we show to interpret differential statements $x'_1 = t_1, \ldots, x'_n = t_n$ for t. As previously mentioned we can assume the existence of a unique solution $\phi : \mathbb{R}^n \times [0, \infty) \to \mathbb{R}^n$, and we also have the function $\sigma : \mathbb{R}^n \times \{0,1\}^\omega \to \mathbb{R} \times \{0,1\}^\omega$. With these data is straightforward to build a function of the type $\mathbb{R}^n \times \{0,1\}^\omega \to \coprod_{r \in [0,\infty)} (\mathbb{R}^n)^{[0,r]} \times \{0,1\}^\omega$ which given an initial state s and probabilistic resource m returns a trajectory corresponding to the solution of the system of differential equations with initial condition s and restricted to the duration specified by t (the probabilistic resource on the output corresponds to the consumption of m caused by interpreting t). Let us denote this function as f. Now the denotation of the program $x'_1 = t_1, \ldots, x'_n = t_n$ for t is simply the morphism composition,

$$\begin{array}{ccc}
\mathbb{R}^n \times \{0,1\}^\omega & \xrightarrow{\quad f \quad} & \coprod_{r \in [0,\infty)} (\mathbb{R}^n)^{[0,r]} \times \{0,1\}^\omega \\
\Big\downarrow {\scriptstyle [\![x'_1 = t_1, \ldots, x'_n = t_n \text{ for } t]\!]} & & \Big\downarrow {\scriptstyle \cong} \\
W[\mathbb{M}](\mathbb{R}^n \times \{0,1\}^\omega) & \xleftarrow{\quad i_1 \quad} & \coprod_{r \in [0,\infty)} (\mathbb{R}^n)^{[0,r)} \times (\mathbb{R}^n \times \{0,1\}^\omega)
\end{array}$$

After defining the diagrams above we can summarise the denotational semantic rules in the following list of rules:

$$\begin{aligned}
&[\![skip]\!] = \eta^{W[\mathbb{M}]} \\
&[\![x := t]\!] = \eta^{W[\mathbb{M}]} \cdot ((update\ x) \times id) \cdot \langle \pi_1, \sigma\ t \rangle \\
&[\![p; q]\!] = \mu^{W[\mathbb{M}]} \cdot W[\mathbb{M}]\ [\![q]\!] \cdot [\![p]\!] \\
&[\![\texttt{if } b \texttt{ then } p \texttt{ else } q]\!] = [[\![p]\!], [\![q]\!]] \cdot \cong \cdot \langle b, id \rangle \\
&[\![x'_1 = t_1, \ldots, x'_n = t_n \texttt{ for } t]\!] = i_1 \cdot \cong \cdot f
\end{aligned}$$

Listing 3.9: Summary of the random number generator denotational semantics

Note that there is no interpretation for wait calls as they are a specific case of a system of differential equation where the derivative of every variable is equal to zero: $x'_1 = 1, \ldots, x'_n = 0$ for t.

Additionally there is also no interpretation of while loops. This is due to the complexity associated to their interpretation, which usually involves fixpoint constructs and is out of the scope of this dissertation. Keeping it brief, in this case we would need to prove for example that the monad used as the basis of this interpretation is an Elgot monad [22] as it was done in [23, 25, 24]. Because the specification of the denotational semantics is already an extra component beyond this dissertation, we leave the completion of the denotational semantics with the interpretation of while loops as a future research line.

3.3.2 Denotational semantics based on distributions

Before moving on to the definition of the denotational semantics, we needed to define a new monad capable of handling distributions. This is done by combining the generalised writer monad with the distribution monad. In order to achieve this combination, we need to reference the concept of strong monads. This is because both monads in the case fit into this category. Strong monads differ from regular monads as they possess a new natural transformation, τ, called strength. This natural transformation allows the mapping of type $A \times TB$ to values of type $T\ (A \times B)$ [37, 51].

Theorem 1. The following diagram shows a possible distributive law between a generic monad $\mathcal{T}$ and the generalised writer monad $\mathcal{W}[\mathbb{M}]$:

$$\begin{array}{ccc} T\ X \times \mathbb{M}\ +\ \mathbb{E} & \xrightarrow{\tau + id} & T(X \times \mathbb{M}) + \mathbb{E} \\ \downarrow{\scriptstyle \lambda} & & \downarrow{\scriptstyle id + i_2} \\ T\ (X \times \mathbb{M}\ +\ \mathbb{E}) & \xleftarrow{[T\ i_1, \eta^T]} & T\ (X \times \mathbb{M}) + (X \times \mathbb{M} + \mathbb{E}) \end{array}$$

By observing the diagram above it is possible to observe that the following function is a distributive law between both monads,

$$[T\ i_1, \eta^T] \cdot (id\ +\ i_2) \cdot (\tau\ +\ id)\ =\ [T\ i_1 \cdot \tau, \eta^T \cdot i_2]$$

Proof. The proof of this theorem can be observed in the appendix C.1.

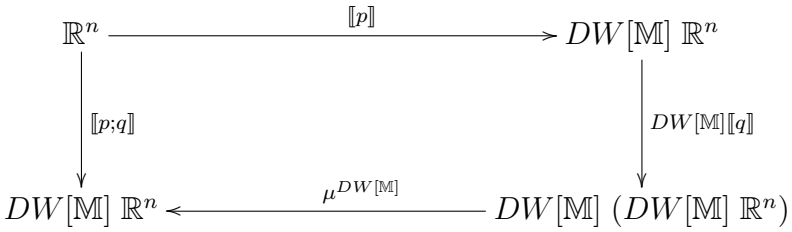
Now that we have a distributive law between these two monads, we can now replace the generic monad $\mathcal{T}$ with the distribution monad ($\mathcal{D}$) and create what we call the probabilistic generalised writer monad ($\mathcal{DW}[\mathbb{M}]$, in which $\mathbb{M}$ is a monoid-module).

Analogous to the previous denotational semantics, we will use the interpretation function $[\![p]\!]$ but this time we will drop the randomly generated infinite list and use the probabilistic version of the generalised writer monad mentioned above: $\mathbb{R}^n \to DW[\mathbb{M}](\mathbb{R}^n)$.

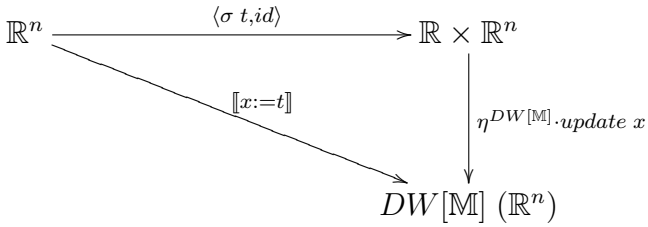
Like previously, the $skip$ function does not do anything, so we will just use the unit of the monad $DW[\mathbb{M}]$.

$$\begin{array}{ccc} & \overset{[\![skip]\!]}{\frown} & \\ \mathbb{R}^n & \xrightarrow{\eta^{DW[\mathbb{M}]}} & DW[\mathbb{M}]\ \mathbb{R}^n \end{array}$$

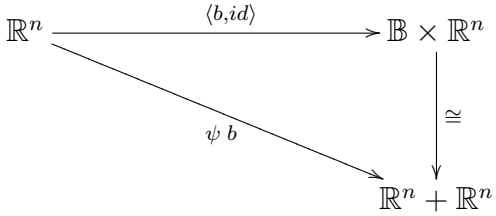
The sequence between two programs, p and q, is interpreted as a function which starts by interpreting the program p, followed by the program p applied to the resulting memory of variables. Finally, the μ function of the monad is used to merge the two computations.



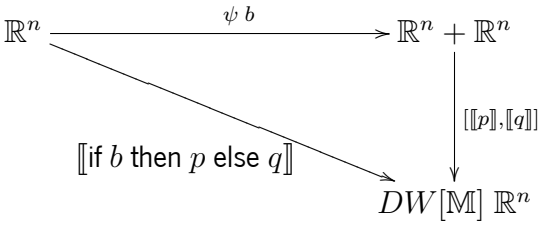
The assignment interpretation starts by interpreting the term t by using a similar function to the one present in the denotational semantics based on a RNG: σ of type $\mathbb{R}^n \to \mathbb{R}$. Afterwards we also use the function $update$ to alter the value of the variable x to be equal to the calculated value of term t.



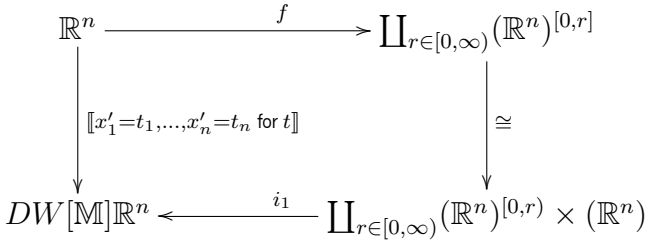
Again the interpretation of if statements uses the function ψ to obtain a coproduct between memories of variables, corresponding to the cases where the boolean value of the test b is either true or false:



By using this function we can apply the interpretation of either the program p or q to the coproduct which will result in a single monadic computation:



With respect to the interpretation of systems of differential equations, we again have the functions f and ϕ (both defined in subsection 3.3.1), as well as the function σ (redefined above), which we can use to interpret the differential statements:



The following set of rules reveal the denotational semantics for simulating hybrid programs:

$$
\begin{aligned}
&[\![skip]\!] \;=\; \eta^{DW[\mathbb{M}]} \\
&[\![x \;:=\; t]\!] \;=\; \eta^{DW[\mathbb{M}]} \;\cdot update\; x \cdot \langle \sigma\; t,\; id \rangle \\
&[\![p; q]\!] \;=\; \mu^{DW[\mathbb{M}]} \cdot DW[\mathbb{M}]\; [\![q]\!] \cdot [\![p]\!] \\
&[\![\texttt{if}\; b\; \texttt{then}\; p\; \texttt{else}\; q]\!] \;=\; [[\![p]\!]\;,\; [\![q]\!]] \cdot \cong \cdot \langle b, id \rangle \\
&[\![\overline{x} := \overline{u}\; \texttt{for}\; t]\!] \;=\; i_1 \cdot \cong \cdot f
\end{aligned}
$$

Listing 3.10: Summary of the denotational semantics based on distributions

As it happened in the previous semantics, the rules for the wait calls and while loops are also absent, for the exact same reasons.

Chapter 4

An interpreter for a probabilistic hybrid language

In this chapter we present the main details of our interpreter for a probabilistic hybrid while-language. We start with a broad overview of the system's architecture. We then discuss how the operational semantic rules – the core engine of the interpreter – were implemented. We give special focus to the most prominent rules namely sequential composition and those concerning differential statements.

4.1 Implementation architecture

The implementation of our interpreter can be divided into three main components. The first one is the parser of the language: it takes a probabilistic hybrid program, written in the syntax previously described, and parses it into a Haskell structure. The interpreter then takes this structure and uses the operational semantics to evaluate the program. Finally the visualisation component draws the histograms and/or plots extracted from the data obtained from the evaluation. On the whole, this provides a flexible and automated ways of analysing the program at hand, with the obvious advantage of finding potential problems early in the game. Let us analyse these three components in more detail.

The parser was implemented via the aforementioned parsec library [9]. It is at this stage that we find possible syntactic errors in the program and report them back to the user if they exist.

The second component has the task of evaluating the parsed program. Recall that we defined two corresponding operational semantics in the previous chapter, one that uses a RNG and the other that returns probabilistic outputs. Both styles of semantics are implemented in our tool. However, when implementing them we decided to add another characteristic to the first semantics, which consists in performing real-time wait calls to the operating system when calculating systems of differential equations or wait commands. This allows the programmer to take into consideration the delays in the operating system if he wishes to do so. From now on we will name this form of evaluation as real-time execution (RTE). As for the second semantics, we implemented it as it is, so it does not perform real-time wait calls.

This second form of evaluation was named probabilistic execution (PRE).

Thus the evaluator takes the parsed program and depending on the user choice, it can either evaluate with one semantics or the other. Because we are dealing with systems of ordinary differential equations (ODE) we used an Haskell library to solve them. This ODE solver [4] uses internally the GNU scientific library [3].

Finally the visualisation component receives the relevant data from the evaluator and creates either histograms or plots, depending on the user input, via the charts library [2].

Figure 3 depicts these components and how they interact, as well as the inputs and outputs of each one. We have decided to name our interprer ProbLince as an homage to the tool Lince which also interprets hybrid programs but does not support probabilistic behaviour [25].

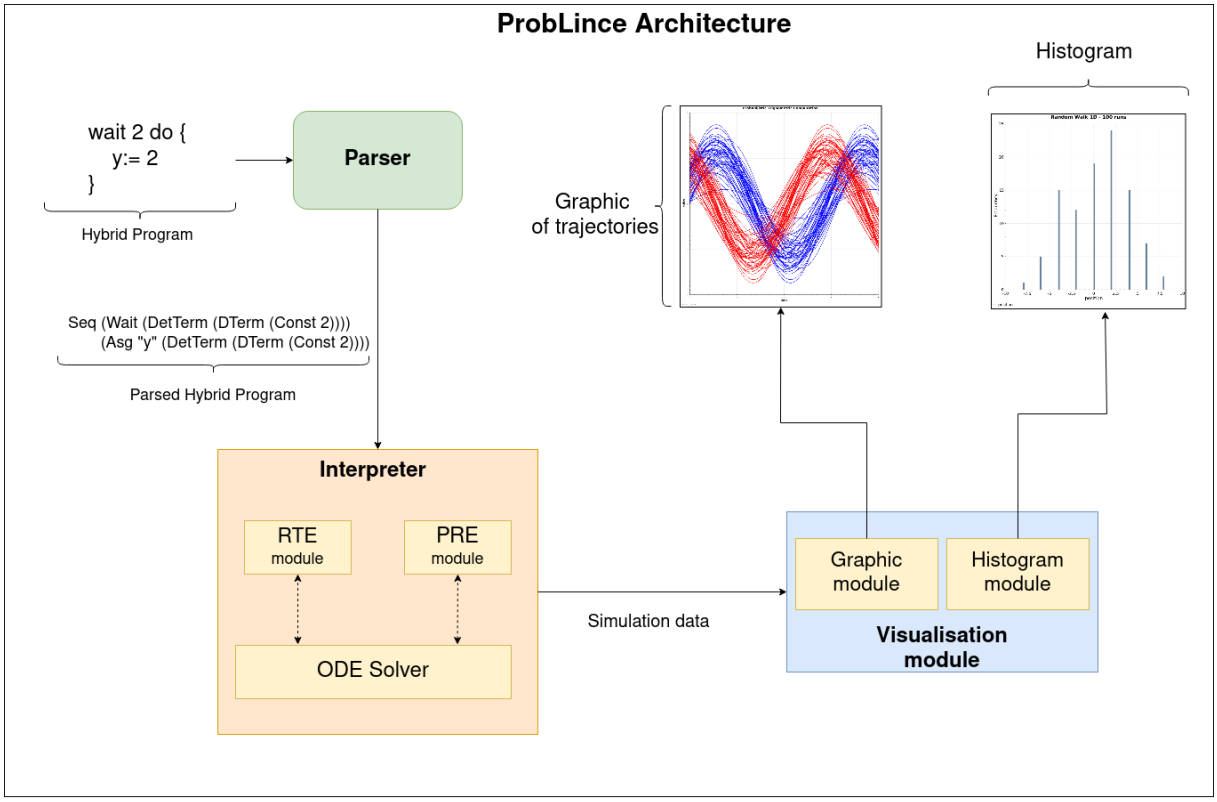


Figure 3: Implementation architecture

4.2 Parser

As mentioned before we are using parsec to implement the parser for the language. Recall that this library allows to implement simple parsers, for example for deterministic or Boolean terms, and then combine them through a variety of combinators. These parsers will parse the input text and structure the data according to a pre-determined Haskell datatype. Let us take the parser of terms as an example.

First we defined an Haskell data type called "Term" which incorporates all the possible values we can face when parsing a term, such as the coin operator and the deterministic term. In relation to the syntax showed in the previous chapter, we took the liberty of adding new, convenient elements, like the normal and rand (see [18]) operators which correspond to normal and uniform distributions (note that both cases are continuous distributions), respectively. We also have arithmetic operations such as multiplication and subtraction. All possibilities of what a term can be are detailed in the following code fragment.

```
data Term = DetTerm DTerm
          | COIN
          | RAND
          | NORMAL Term Term
          | NegateT Term
          | AddT Term Term
          | SubT Term Term
          | MultT Term Term
          | DivT Term Term deriving Show
```

Listing 4.1: Term data type

After defining this structure we implemented the respective parser allowing us to transform the input text into a value of this data type. The following code fragment shows the main function for parsing a term and how we dealt with the precedences of multiplication and division in relation to addition and subtraction. In order to do this we used the "chainl1" combinator, for which we already explained via an example, how it solves the problem of precedence.

```
parseTerm :: Parsec String st Term
parseTerm = chainl1 parseAritTerm op
    where
        op = try (spaces >> string "+" >> spaces >> return AddT)
          <|> try (spaces >> string "-" >> spaces >> return SubT)

parseAritTerm :: Parsec String st Term
parseAritTerm = chainl1 parseAritFactor op
    where
        op = try (spaces >> string "*" >> spaces >> return MultT)
          <|> try (spaces >> string "/" >> spaces >> return DivT)
```

Listing 4.2: Term parser

Let us now take a look at another part of our parsing system, namely the parser of programs. Similarly

to before, we defined a data type called “Instruction” which corresponds to the possible programs of our language, such as assignments and differential statements. The only change we did in relation to the syntax of the language was creating a declaration instruction, which is very similar to the assignment case. We did this in order to force the declaration of a variable before using it.

```
data Instruction = SKIP
                 | Decl String Term
                 | Asg String Term
                 | Seq Instruction Instruction
                 | Cond Test (Instruction,Instruction)
                 | Wh Test Instruction
                 | Wait Term Instruction
                 | Diff DiffSystem Term deriving Show
```

Listing 4.3: Data type correspondent to the possible programs of our language

As for the parser itself we again had to handle precedence, this time connected to the sequence operator: the idea is to parse the program “p;q” into the construct “`Seq P Q`”, in which “P” and “Q” are the parsed versions of “p” and “q”, respectively. If we do not apply any type of precedence when parsing the sequence operator, then the programs “p” and “q” would be parsed independently and we could not apply the left associative `Seq` constructor to their results. Because of this we separated both the parser for the sequence operator, through the function “`parseSeq`”, and the other programs through the function “`parseInstruction`”. Notably, the latter uses the `<|>` combinator to define a choice between the different parsers for the different possible types of program; also `try` operator as a lookahead is needed because different programs may start with the same prefix.

```
parseSeq :: Parsec String st Instruction
parseSeq = (chainl1 (L.whiteSpace hybridLangLexer *> parseInstruction) op)
    where
        op = char ';' *> return Seq

parseInstruction :: Parsec String st Instruction
parseInstruction = try parseSkip
                <|> try parseDecl
                <|> try parseAsg
                <|> try parseCond
                <|> try parseWh
                <|> try parseWait
                <|> parseDiff
```

Listing 4.4: Parsers for the sequential operator and other programs

Let us analyse this operation via an example, if we have a program which consists of a sequence between a declaration and an attribution such as "`var x := 0; y := 0`", then by applying the sequence parser the `chainl1` combinator would start by applying the parser for instructions (in this case the declaration parser), to the first program, resulting in "`Decl (DetTerm (Var "x")) (DetTerm (Const 0.0) )`". Subsequently, the sequence parser would parse the ";" operator, followed by the application of the instruction parser to the second program. The latter returns the structure "`Asg (DetTerm (Var "x")) (DetTerm (Const 0.0))`". Finally the combinator applies the left associative "`Seq`" constructor to these results obtaining,

```
Seq (Decl (DetTerm (Var "x")) (DetTerm (Const 0.0))) (Asg (DetTerm (Var
"x")) (DetTerm (Const 0.0)))
```

The interpreter will then know that this result is the sequence between the original two programs, and will feed the outcome of the first to the second when evaluating them.

4.3 Evaluation

In this section we will go over some of the evaluator's details, such as which monads were used and how the rules of the operational semantics were implemented. As mentioned before, this evaluator uses either the operational semantics based on a RNG or the semantics that returns distributions of outputs.

4.3.1 Evaluation based on a random number generator

As mentioned previously, we followed a monadic approach when building the interpreter. Recall also that the monoid of trajectories (definition 8) has limitations when using it in our implementation, with implications to the visualisation module. Due to this fact, we made slight changes to the monoid of trajectories so that instead of joining two trajectories in a single one, we keep a list of all trajectories. Formally we have,

Definition 9 (Alternative monoid of trajectories)**.** Fix a natural number n. The monoid component is the triple $((\coprod_{r\in\mathbb{R}_{\geq 0}}(\mathbb{R}^n)^{[0,r)})^*, +\!\!+, [])$ where $+\!\!+$ is list concatenation.

The implementation of this monoid is given in the following code fragment. In it we define a new data type which consists of a list of pairs, where the first element corresponds to the total duration of the

trajectory and the second to a function which maps time instants to memories. In this case `Map String Double` can also be seen as `[(String,Double)]`, mapping each variable (the `String` corresponds to its name) to the respective value. We chose to use this structure instead of the standard Haskell list, as it presents some optimizations specially when it comes to the append operation [6].

```
data Trj = Trj [(Int, Int -> Map String Double)]

instance Semigroup Trj where
    Trj t1 <> Trj t2 = Trj (t1 <> t2)

instance Monoid Trj where
    mempty = Trj []
```

Listing 4.5: Trajectory monoid implementation

In our implementation we had to combine different kinds of monads to obtain the desired results in a modular way. Specifically, we used the `IO` monad to interact with the operating system and to perform real-time wait calls, among other things. We used a state monad to handle the remaining time in the evaluation and continuation flags. We also used the previous writer monad of trajectories to track all the occurring trajectories. We used monad transformers to join all of these individual monads, a popular procedure in the development of interpreters (as mentioned before). We detail this next.

As there is no `IO` monad transformer, the `IO` monad has to be in the bottom of the monad stack. We then use the exception monad transformer instantiated with the `IO` monad, which gives rise to computations of type `IO (Either Exception a)`. This was done to produce a more complete tool, and to signal basic errors such as division by 0, even though it is not formally specified in the semantics. We then apply the writer monad transformer to the previous monad which results in computations of the type,

```
IO (Either Exception (a,Trj)).
```

Finally we apply the state monad transformer to this last monad to obtain computations of type,

```
S -> IO (Either Exception ((a,S), Trj))
```

The resulting monad is denoted by `RteM` and is presented in the following code block. The same block also presents the `Resources` type corresponding to the three probabilistic resources, given as infinite lists of probabilistic values, as well as the memory of variables encoded by a `Map` structure.

```
data ProbElems = ProbElems {
                        coin :: [Double],
```

```
                    rand :: [Double],
                    normal :: [Double]
                }

type Resources = (ProbElems, Map String Double)

type RteM = StateT (Int, Bool) (WriterT Trj (ExceptT RunException IO))
```

Listing 4.6: RTE monad definition

Both the coin and rand lists were generated through the use of the Monad Random library [8]. This library exports the function `getRandomRs` which generates an infinite list of random values between a range inside a monadic computation. The use of `evalRandIO` (also from the monad random library [8]) transforms the monadic computation into a `IO` one. Lastly, the normally distributed infinite list was created via the Monad Bayes library [7], where we created a list filled with independent samples of normally distributed values, with a mean of 0 and a standard deviation of 1. Although these lists are infinite, the lazy characteristic of Haskell only generates the values when they are going to be used which allows us to keep drawing the head values from them as it is specified in the operational semantics.

```
infCoinList <- evalRandIO $ getRandomRs (0,1)
infRandList <- evalRandIO $ getRandomRs (0.0,1.0)
infNormalList <- sampler . independent $ B.normal 0.0 1.0
```

Listing 4.7: Infinite probabilistic resources implementation

Next, one of the key aspects of this operational semantics is how it performs real-time wait calls. Let us look precisely at the function `waitCallRTE` which is invoked when we have an hybrid program that performs a wait operation such as "wait 2.0 do (..)". If we observe the operational semantics we notice that we have to compare the duration of the wait command with the time left in the execution. Depending on the result of this condition we have to perform a real-time wait call which is done by the `threadDelay` function. We measure the time this command takes to finish and create a trajectory with this duration. Specifically, the corresponding duration gets paired with a constant function which returns the same memory of variables, as the wait call does no alterations to it. This is is what following code block does.

```
waitCallRTE :: Int -> Resources -> RteM Resources
waitCallRTE dur (r,m) = do
    (time, st) <- get
    a <- lift . lift . lift $ getCurrentTime
```

```
    if time <= dur then do
        lift . lift . lift . threadDelay $ time
        put (0, False)

    else do
        lift . lift . lift . threadDelay $ dur
        put (time - dur, True)

    b <- lift . lift . lift $ getCurrentTime
    lift . tell $ toTrj ((floor $ diffUTCTime b a * 1000000.0)) (const m)
    return (r,m)
```

Listing 4.8: Evaluation of wait calls

Another good example of how the semantic rules were implemented is the sequence operation. The code below shows that we first obtain the result of the evaluation of the first program with a probabilistic resource and a memory state given as input. Subsequently, we check the continuation flag (encoded as as a Boolean value stored in the mutable state of the monadic computation). If the flag is true, it means we obtained the value $skip$ in the operational rules, and so we can perform the execution of the second program of the sequence operation. On the other hand if the condition is false, it means we obtained a $stop$ flag and so the evaluation stops immediately.

```
seqRTE :: Instruction -> Instruction -> Resources -> RteM Resources
seqRTE inst1 inst2 r = do
                r' <- programRTE inst1 r
                s <- get
                case p2 $ s of
                    True -> programRTE inst2 r'
                    False -> return r'
```

Listing 4.9: Implementation of the rule for sequential composition

4.3.2 Evaluation based on distributions over outputs

We now focus on the evaluation of programs based on the second style of operational semantics, *i.e.* the one that does not provide a single output but a distribution of outputs. Like in the previous case, we use a modular, monadic approach to implement the evaluator – this results in a monad that is very similar to the one used in the previous case (`RteM`), but does not incorporate the `IO` monad because there are

no real-time wait calls now. Not only this, it incorporates a new monad capable of handling distributions, present both in the operational and denotational semantics based on distributions over outputs. Indeed if we look at this denotational semantics we can verify the use of the 'distribution monad' $\mathcal{D}$ which encodes distributions as a map between events and the probability of them happening. This is fine if we are dealing only with discrete distributions. However since we also wish to use the `rand` and `normal` operators, we have to use a monad which handles these types of distributions as well. Another problem with the distribution monad is its performance when a large number of events are present. The Monad Bayes library allows us to build a monad that is capable of fixing both these problems. Next, we describe in more detail the combined monad that was used for the evaluation.

In the code block below we see the monad `Prob` which is defined as a `Weighted Sampler`. As mentioned before, `Sampler` is a monad which allows us to use do lazy sampling. As we are using the Metropolis-Hastings algorithm to obtain samples, we need to use the `Weighted` inference representation, which allows us to obtain unbiased samples. Since we use this new monad, we do not need to have a `probElems` instance (like in the previous implementation of an evaluator), as the infinite probabilistic resources are not used any more.

```
type Prob = Weighted Sampler

type Memory = Map String Double

type PreM = StateT (Int, Bool) (WriterT Trj (ExceptT RunException Prob))
```

Listing 4.10: Monad stack for the PRE monad

The following code fragment shows the implementation of the wait call function, which is similar to the corresponding one for the previous evaluator: the main difference is that now we do not perform real-time wait calls to the operating system. Note for example that the if statement below, which tries to determine whether to produce a "skip" or ""stop", flag is analogous to the one used in the previous evaluator.

```
waitCallPRE :: Memory -> Int -> PreM Memory
waitCallPRE m dur = do
    (time, st) <- get
    if time <= dur then do
        put (0, False)
        lift . tell $ toTrj time (const m)
    else do
        put (time - dur, True);
        lift . tell $ toTrj dur (const m)
```

```
    return m
```

Listing 4.11: Evaluation of wait calls using the PRE execution mode

As a concluding note, the code block below shows the implementation of the while-loop rule, where we check the corresponding Boolean condition using the `bind` and `cond` operations. If the condition is true we perform a sequence operation between the program inside the loop's body and the loop itself. Otherwise we simply return the memory of variables, thus stopping the evaluation of the while-loop.

```
whPRE :: Test -> Instruction -> Memory -> PreM Memory
whPRE test inst m = testPRE test m
                  >>= (\b -> cond (const b) (programPRE (Seq inst (Wh test
   inst))) return m)
```

Listing 4.12: Evaluation of while-loops

Chapter 5

Designing an embedded domain specific language

In this chapter we discuss the implementation details of another tool we developed, which has the benefit of providing more expressive power to the programmer in comparison to the previous language. Specifically it corresponds to a DSL embedded in Haskell which allows the user to take advantage of all features of Haskell to write probabilistic hybrid programs – note that previously we were restricted to a while-language. This allows us to use many features present in Haskell (*e.g.* folds or maps), which means we can write down hybrid programs in an entirely different manner.

We will go over some of the new data types that were created specifically for this tool, as well as the available methods of program evaluation, and a set of combinators for combining different probabilistic hybrid computations.

5.1 Overview

There are many DSLs available, such as [13, 11], or even the previously mentioned Yampa language [28]. These can be divided into two types: they can be either shallowly or deeply embedded into another language. In the first type we have a translation of the program into an intermediate structure, which is then evaluated. The second type does not use this structure, which means the terms are directly evaluated by the language [21]. For our case we adopt the second type by deeply embedding our language into Haskell.

The monads developed before are still a central part of our DSL, but with slight adjustments which we briefly explain next. We changed the mutable state of both monads to accommodate either probabilistic resources (in the case of evaluations based on random number generators) or the memory (in the case of evaluations based on probabilistic outputs). Obviously we had to adapt the functions exported in the monad modules to now use the resources or memory inside the mutable state. This is relevant for example when adding a new variable to the program. The resulting monad stacks are described in the following code block.

```
type RteM = StateT ((Int, Bool), Resources) (WriterT Trj (ExceptT
    RunException IO))

type PreM = StateT ((Int, Bool), Memory) (WriterT Trj (ExceptT RunException
     Prob))
```

Listing 5.1: The new monad stacks

5.2 Representing discrete events and main combinators

A new data type we created for this DSL was the `DEvent`, which corresponds to possible discrete events. This data type has two constructors one referring to the wait call and another to a system of differential equations.

```
data DEvent = Wait | DiffSystem [String] (Double -> [Double] -> [Double])
```

Listing 5.2: Discrete events data type

There also function such as "`preDEvent`" and "`rteDEvent`" which receive one of these discrete events as input as as a certain duration, and produce a RTE or PRE monadic computation, essentially building the trajectory of the variables.

These trajectories can either correspond to evaluations based on a random number generator (in the RTE case) or evaluations based on distributions (in the PRE case) over outputs. Next, now that we have these individual evaluations, we implemented a specific set of combinators to combine them. As an example we have the `sequential` and `infinite` combinators. The first is the counterpart of the sequence operator in the while-language: it receives two evaluations and composes them based on the continuation flag of the first evaluation. The second combinator uses the sequential combinator and a recursive call to itself to create an infinite sequence of evaluations. Obviously, the execution will stop once we reach the time instant we wish to evaluate (the sequential combinator performs this check).

```
sequential :: RteM () -> RteM () -> RteM ()
sequential e1 e2 = do
    e1
    continue <- stop
    if continue then
        e2
    else
        return ()
```

Listing 5.3: Sequential combinator

In the same way we have combinators for evaluations, we also have combinators for discrete events. One of the more interesting ones is the interleaving combinator, which runs two events for small periods of time and checks a condition between each run. Once the condition is true an evaluation is triggered, as presented next in code.

```
interleave :: DEvent -> DEvent -> Double -> RteM Bool -> RteM () -> RteM ()
interleave e1 e2 dur c e3 = do
                    b <- c
                    if b then
                        e3
                    else
                        sequential (sequential (rteDEvent e2 dur) (
   rteDEvent e1 dur)) (interleave e1 e2 dur c e3)
```

Listing 5.4: Interleave combinator

This combinator is particularly interesting to model the collision of particles or balls. For example in Figure 4 it is possible to observe two trajectories, corresponding to two balls. The blue line corresponds to a ball which is dropped from a height of 800 m and the red line to another which is thrown from a height of 1000 m. By using the interleaving combinator their trajectories will be simulated for a few seconds and after that a check will be done in order to know if those two collide. If it happens then the collision effect will take place and if it does not then the simulation will keep on going. If we wanted we could combine this interleaving operation with the `infinite` combinator to continue with the simulation until hitting a certain duration or the ground.

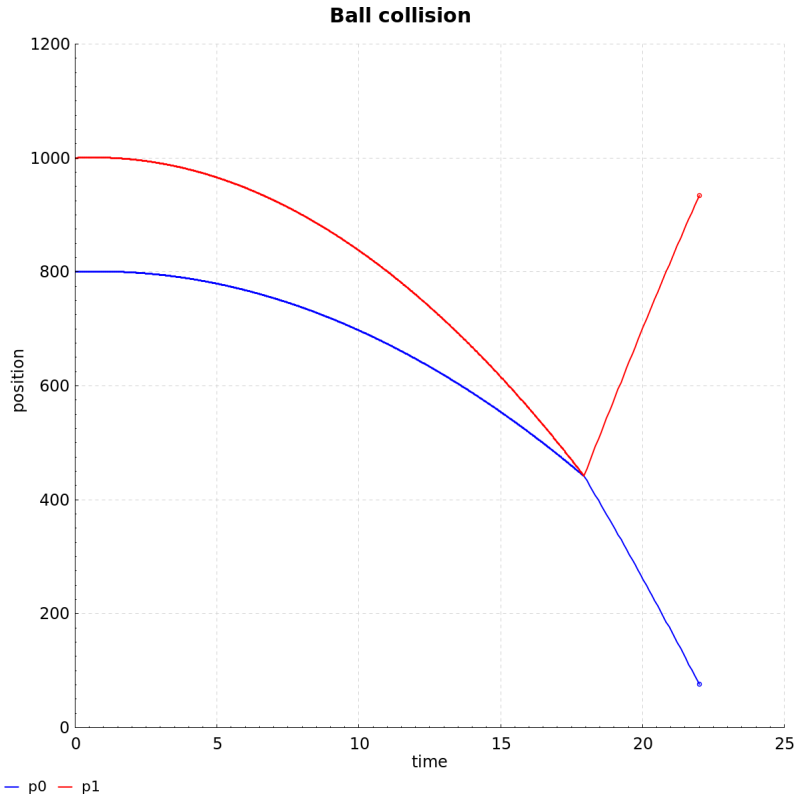


Figure 4: Interleaving combinator test - ball collision

There are some other combinators implemented, like the `after` combinator which performs a wait call for a certain duration passed as input followed by an evaluation, or the `untilC` combinator which performs an evaluation until a prescribed condition is met.

Because our DSL is embedded into Haskell we can further extend the list of combinators by Haskell's capabilities. As an example, we could create a combinator which receives a list of evaluations, and joins them all into one monadic computation by using Haskell's `foldl` function and the `sequential` combinator. We do not pursue this further in the dissertation, because the implementation of the DSL itself is already out of its scope.

Chapter 6

Case studies

In this chapter we explore our interpreter ProbLince via several case-studies. We start by presenting programs that model different distributions and compare them against the expected results. Afterwards, we implement more complex hybrid programs, starting the analysis of deterministic approaches and then adding a layer of uncertainty. We also analyse the role that uncertainty plays in these programs and how it may break some safety aspects of those systems.

6.1 Modelling some known distributions

In this section we present the implementation of different probabilistic programs in our language and verify whether the output we observe is the one expected.

6.1.1 Uniform distribution

The following program draws from a Bernoulli trial to determine how much time it should wait before updating variable "x":

```
var x := 0;

wait coin() do {
    x := 2
}
```

Listing 6.1: Simple coin toss program with wait call

Since the coin operator value is obtained either by drawing a value from a randomly generated binary list or by sampling a value from a binary distribution, we expect to observe a uniform distribution when analysing how much time the program took to halt. More specifically, we expected to observe that in half of the cases the program lasts 0 seconds and 1 second in the other half. The following histogram provided

by ProbLince is the result of evaluating the program and performing 100 samples out of the resulting distribution:

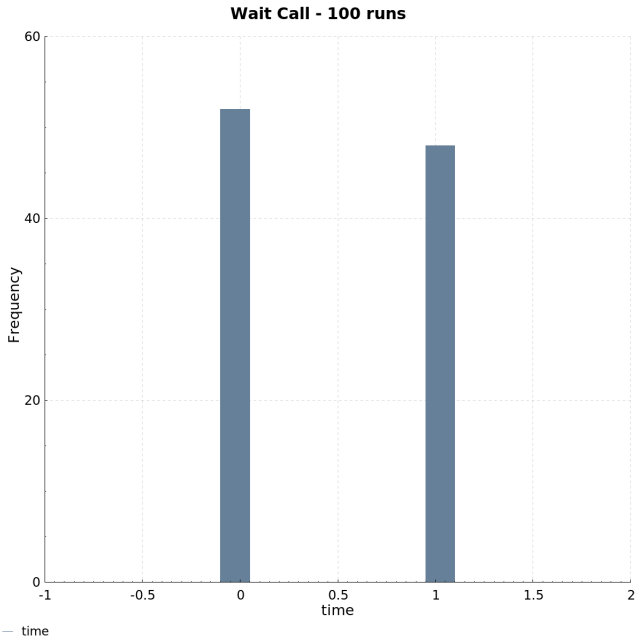


Figure 5: wait call for 100 runs

By observing these results we see that the results are not completely uniform, which seems to go against what we intended. This is because we simulated the program for a low number of trials. The following histograms show two more evaluation of this program where we used 1000 and 10000 trials. Through the results below we can verify that the distribution gets closer and closer to the uniform distribution as we increase the amount of samples:

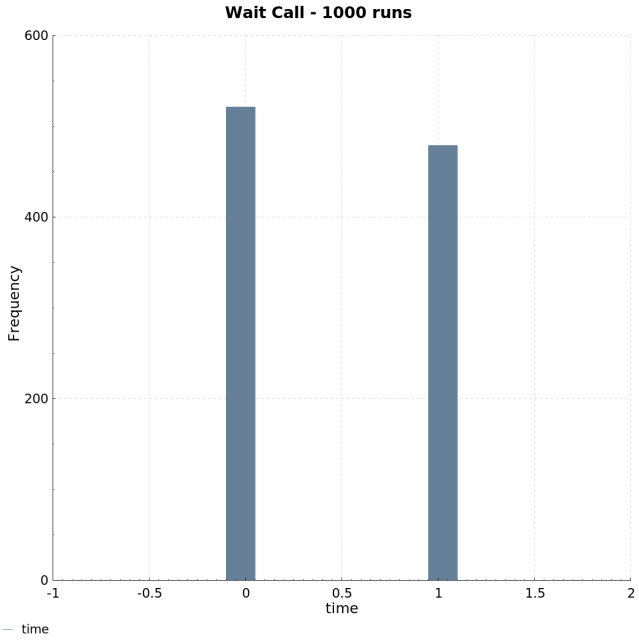


Figure 6: wait call for 1000 runs

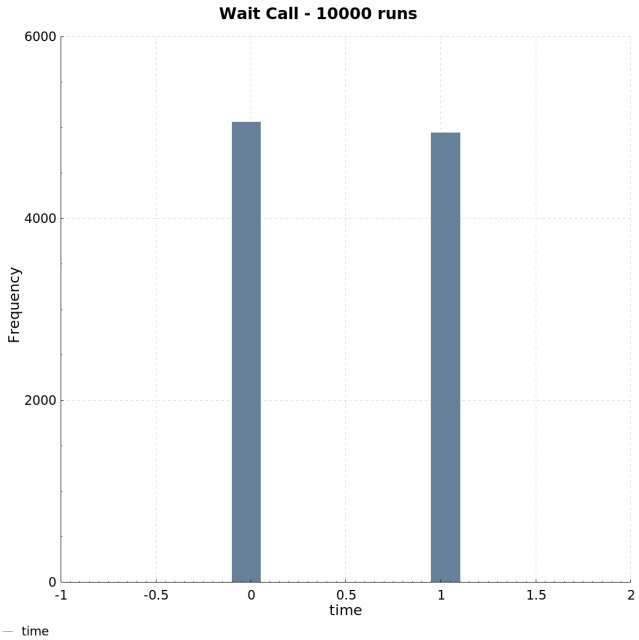


Figure 7: wait call for 10000 runs

6.1.2 Negative exponential distribution

The next example we want to analyse was obtained from [18]. The objective of this example is to see how many coin tosses are needed to obtain the value 1:

```
var x := 0;
```

```
var r := 0;

while (x == 0) do {
    x := coin();
    r := r + 1
}
```

Listing 6.2: Program that flips a coin until the value is zero

For each toss (cycle iteration) the probability of obtain the value 0 or 1 is $\frac{1}{2}$. The probability of performing two coin flips is $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$, so the probability of halting after n executions is 2^{-n} [18]. The following histograms are the result of the execution of this program for 100, 500 and 1000 runs. In these, we can observe that they all generate a negative exponential distribution (as expected) and in the majority of cases the program does perform only two coin flips until obtaining the value 1. Similarly to the previous example, the distribution becomes clearer once we increase the number of executions:

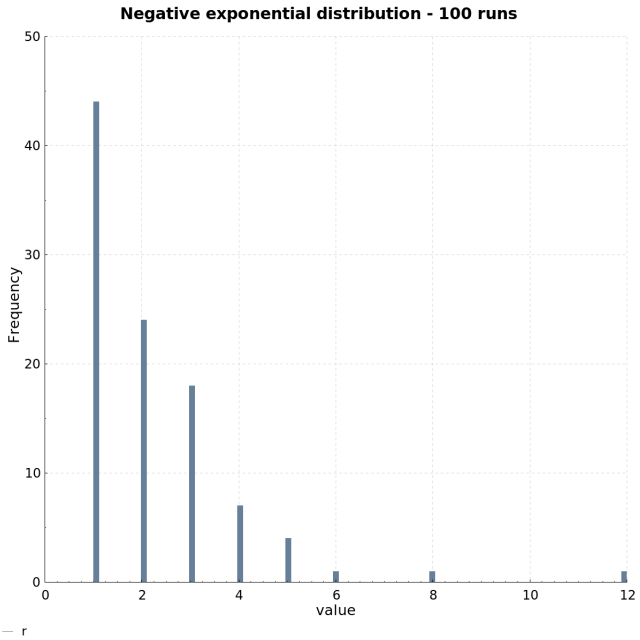


Figure 8: Negative exponential program execution result for 100 runs

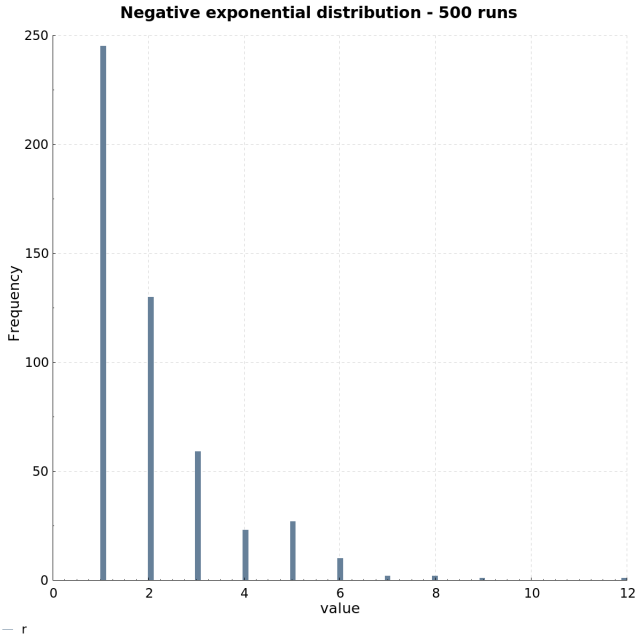


Figure 9: Negative exponential program execution result for 500 runs

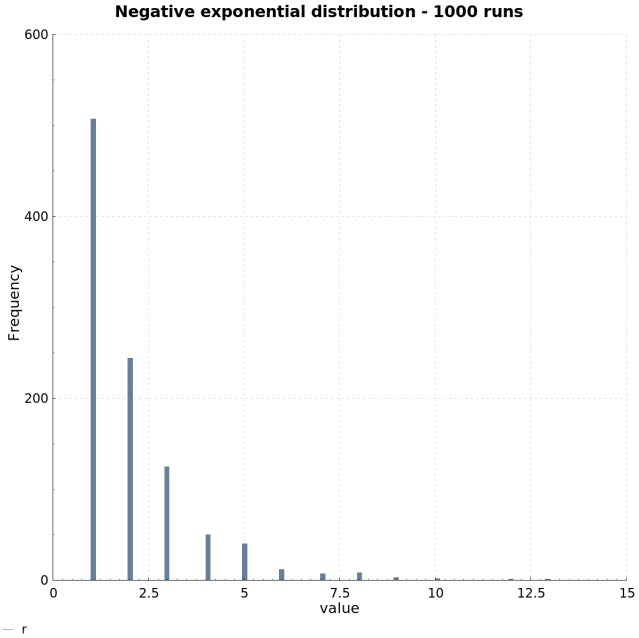


Figure 10: Negative exponential program execution result for 1000 runs

6.1.3 Normal distribution

One of the more interesting examples present in [18] is the random walk program. The program consists in a series of steps in a 2D grid where in each one of those we perform two coin tosses which decide the direction the next step is going to be. Before moving on to our implementation of the 2D random

walk we present a simpler version of the problem by considering only one dimension. Thus the program starts in the origin (x variable equal to 0) and we throw a random coin which will determine if either we increment the value of x or decrement it (moving along the x axis). Another adaptation we had to perform to the example in [18] is the restriction in the number of steps we let the program do. This is because the program could be infinite, and by adding this constraint we guarantee that it will halt.

```
var s := 1;
var steps := 10;
var x := coin();
if (x == 0) then {x := -1} else {skip};
var position := x;

while (s < steps) do {
    x := coin();
    if (x == 0) then {x := -1} else {skip};
    position := position + x;
    s := s + 1
}
```

Listing 6.3: 1D random walk implementation

Now, if we evaluate this program we should verify that value of the position follows a normal distribution around the initial position (origin of the grid). The following figures show the results we obtained for 100, 500 and 1000 evaluations, respectively.

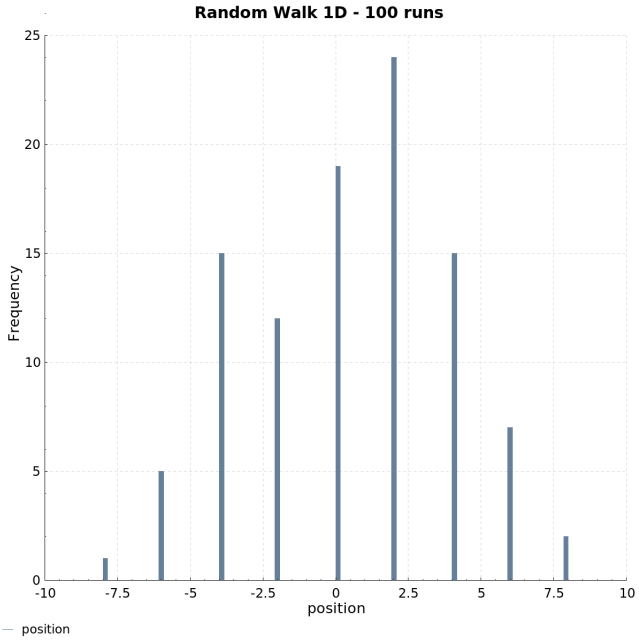


Figure 11: One dimensional random walk execution result for 100 runs

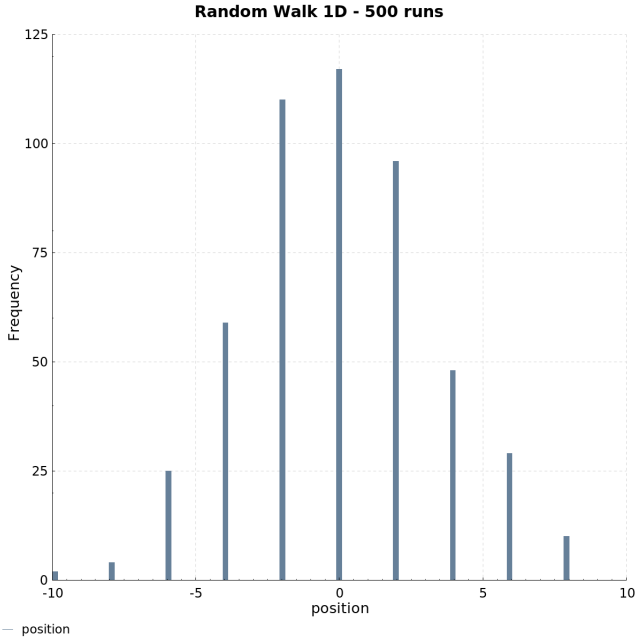


Figure 12: One dimensional random walk execution result for 500 runs

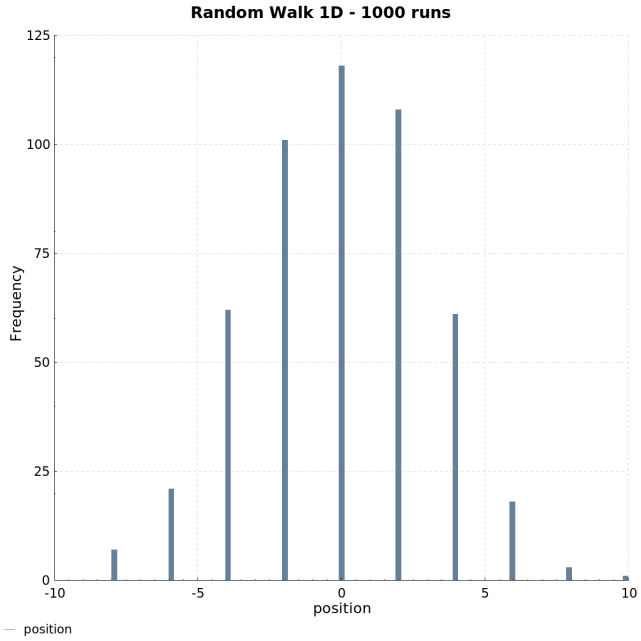


Figure 13: One dimensional random walk execution result for 1000 runs

Returning to the original 2D random walk presented in [18], not only we had to limit the number of steps we also had to introduce a new variable to collect information about the variables `x` and `y`. Specifically, we introduced a new variable to measure the distance between the final position of the random walk and the origin of the grid, which then allows us to generate an informative histogram based only on one variable:

```
var s := 1;
var x := coin();
var y := coin();
var u := (x-y);
var v := (x+y-1);

var steps := 100;
var distance := 0;

while (s < steps) do {
    x := coin();
    y := coin();
    u := u + (x-y);
    v := v + (x+y-1);
    s := s + 1
};

if (u < 0) then {u := -u} else {skip};
```

```
if (v < 0) then {v := -v} else {skip};

distance := u + v
```

Listing 6.4: 2D random walk implementation

We evaluated this program for 100, 1000 and 10000 runs. One thing we observe is that the difference between the distributions were much more pronounced as we increased the number of executions. Only in the final result (for 10000 runs) we obtained a distribution which resembles a normal one. Note that it does have its center around 0 as we are dealing with distances, which can not be negative. This shifts the curve of the distribution.

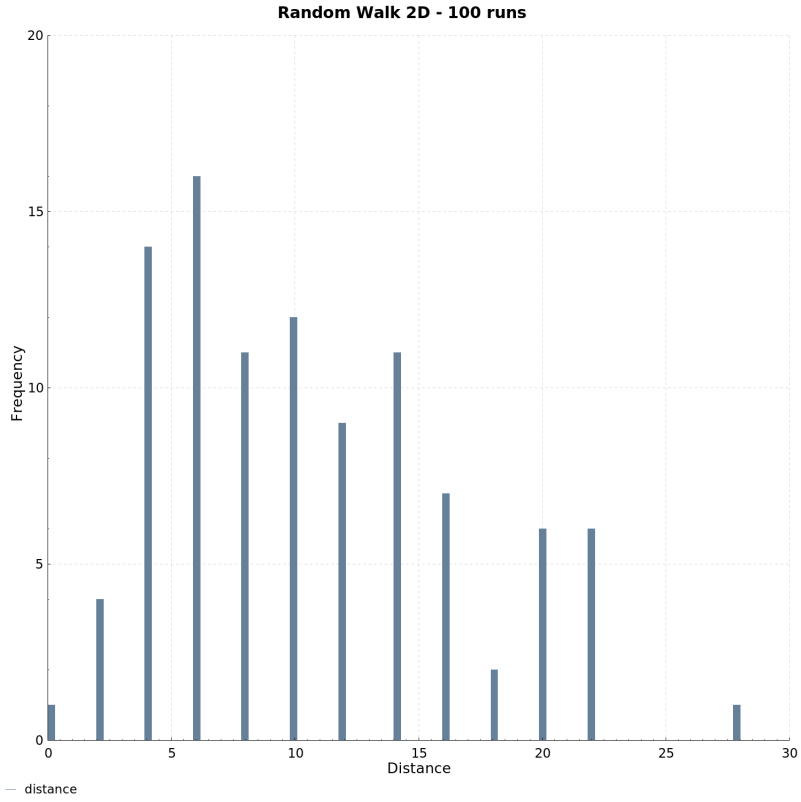


Figure 14: Random walk execution result for 100 runs

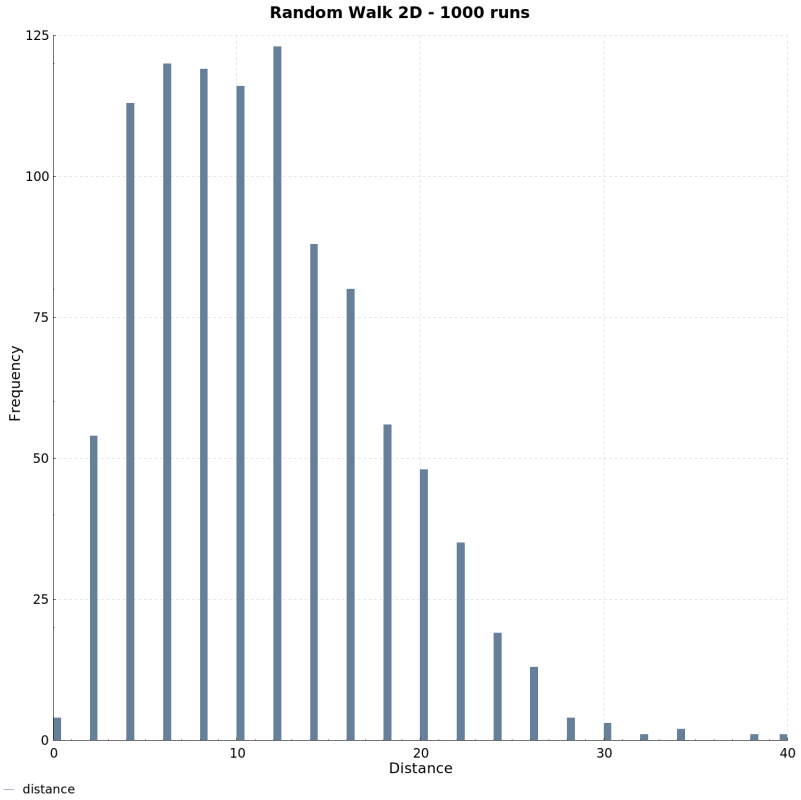


Figure 15: Random walk execution result for 1000 runs

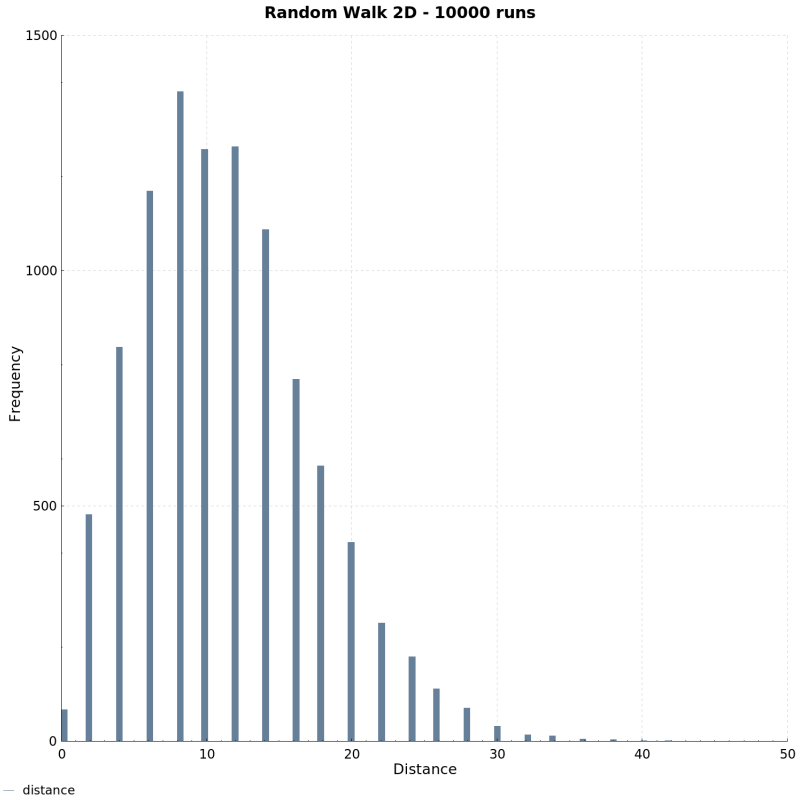


Figure 16: Random walk execution result for 10000 runs

6.2 Incorporating probabilistic behaviour in hybrid programs

In this section we present the results we obtained when adding probabilistic elements to hybrid programs which presented no elements of such kind. Some of these programs refer to cruise control systems or landing systems which we can now model with margins of error, creating a range of possible trajectories instead of just one.

6.2.1 Basic composition

We start with a simple hybrid program. In this example we perform a sequence of differential statements for a total of 4 seconds (the example is adapted from [5]):

```
var v:=0;
v'=1 for 2; v'=3 for 2
```

Listing 6.5: Basic composition of two trajectories

Figure 17 shows the corresponding trajectory:

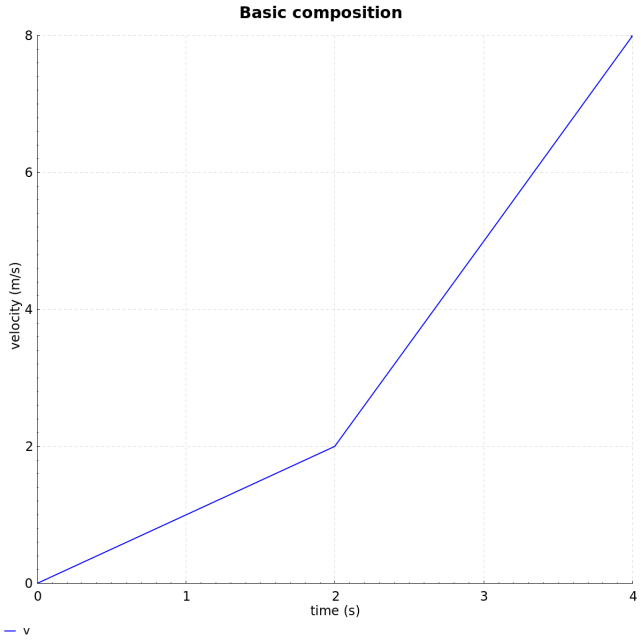


Figure 17: Basic hybrid composition

We can now include probabilistic elements, such as adding a margin of error to the duration of both trajectories; and indeed, see below that we altered the duration of both statements to follow a normal distribution with the same mean but with different standard deviations:

```
var v:=0;
v'=1 for normal(2.0,0.6); v'=3 for normal(2,0.2)
```

Listing 6.6: Basic hybrid computation with normal distributions

The result we obtain is very different from the original. By evaluating this program and then sampling from the resulting distribution of possible trajectories we can observe a set of trajectories (each corresponding to a sample) in the following plot:

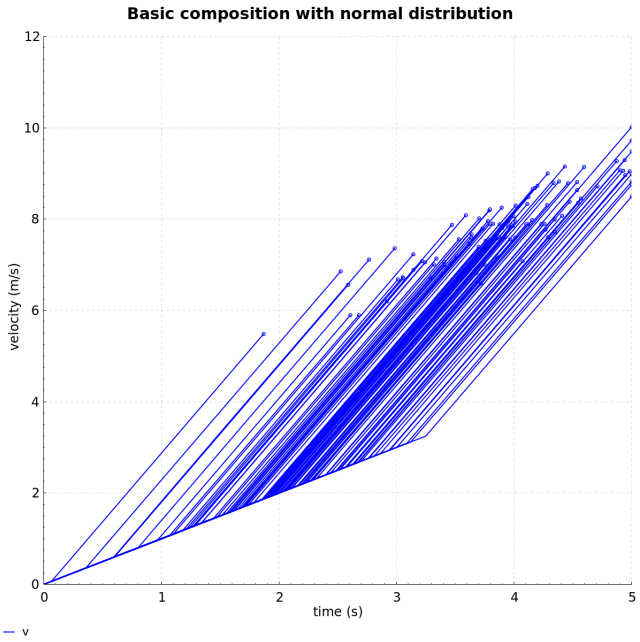


Figure 18: Probabilistic basic hybrid composition (Normal distribution)

We also explored a case where instead of using normal distributions, we would use continuous uniform distributions with a range equal to $[1, 3]$.

```
var v:=0;
v'=1 for rand() * 2 + 1; v'=3 for rand() * 2 + 1
```

Listing 6.7: Basic hybrid computation with uniform distribution

The main difference between both examples is that the figure 18 has a higher density of trajectories around the middle of the figure, becoming less dense the more we move away from that zone. The second approach (figure 19) reveals a more evenly spaced out set of trajectories as intended:

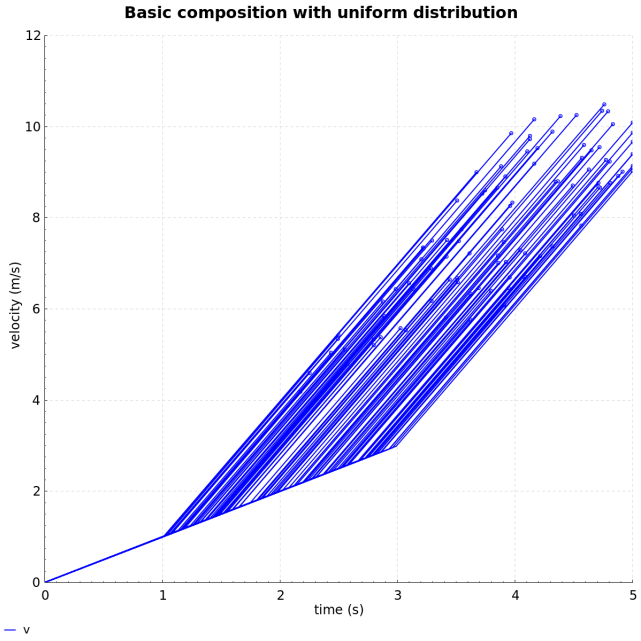


Figure 19: Probabilistic basic hybrid composition (Uniform distribution)

6.2.2 Bouncing Ball

The next example corresponds to a ball that is launched from 10 m high with a velocity of 5 m/s. The ball is pulled down by the gravitational acceleration of the Earth (9.8 m/s^2) until it hits the floor and bounces. In order to determine whether the ball has hit the floor, the condition in the while-loop below is checked every 0.01 seconds and while true the ball continues to fall with the aforementioned acceleration. The variable "c" makes the program stop after four bounces (this example is adapted from [5]).

```
var v:= 5; var p:= 10; var c:= 0;

while (c < 4) do {
    while (!(p < 0 && v < 0)) do {v'=-9.8,p'=v for 0.01};
    v:=-0.5*v; c:=c+1
}
```

Listing 6.8: Bouncing ball implementation

The results we obtain (figure 20) are the exactly same as the ones shown in [5], which is an extra indicator of the correctness of our implementation.

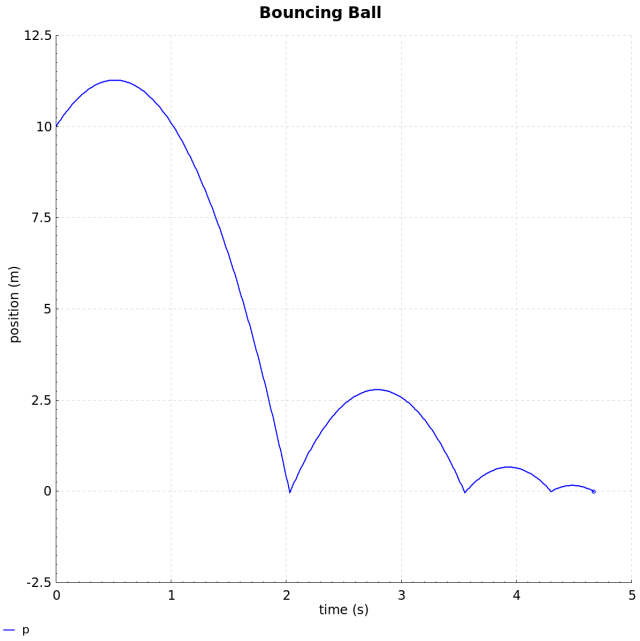


Figure 20: Bouncing ball simulation

After modelling this system, we again added probabilistic elements to it. For this case we took into account the fact that it is impossible to determine with exact precision the elasticity of the ball – which is reflected in a degree of uncertainty w.r.t. the dampening factor associated to hitting the ground. If we specify this factor as a normal distribution around the value of 0.5 we obtain the following program:

```
var v:= 5; var p:= 10; var c:= 0;
var factor := normal(0.5,0.02);

while (c < 4) do {
    while (!(p < 0 && v < 0)) do {v'=-9.8,p'=1*v for 0.01};
    v:=-factor*v; c:=c+1
}
```

Listing 6.9: Probabilistic bouncing ball implementation

The result we then obtain when sampling from the resulting distribution is very different from the original one. In some cases the ball does not bounce as high, which results in less time to performs the four bounces. In other cases, we see the exact opposite.

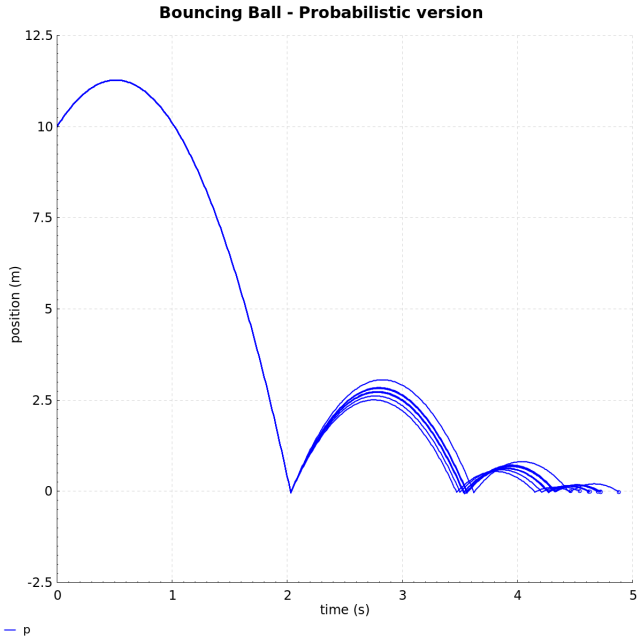


Figure 21: Probabilistic Bouncing ball evaluation

6.2.3 Cruise Control system

The next example corresponds to a specific cruise control system. Its objective is to maintain the vehicle's speed around a determined value (the example is adapted from [5]).

```
var p:=0; var v:=2;

while true do {
  if (v <10 || v == 10)
  then {p':=v,v':=5 for 1}
  else {p':=v,v':=-2 for 1}
}
```

Listing 6.10: Cruise control system

The figure 22 shows the variations in the velocity of the vehicle and how it is kept around the chosen value. We also observe the progression of its movement by analysing the line in blue which indicates its positioning.

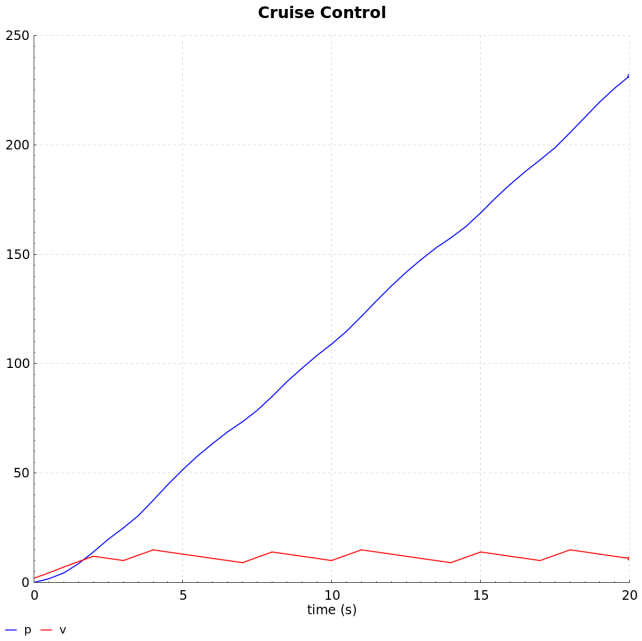


Figure 22: Cruise control system

One could point out that the system is well implemented as it keeps the vehicle moving at a controlled pace with the velocity around a fixed value. However, one of the main issues with this type of system is that one cannot know with exact precision for how long the vehicle will accelerate or decelerate – some margins of error have to be considered, as mentioned before in this dissertation. We thus update the program to include some kind of error associated with how long we run the systems of differential equations, resulting in the following program:

```
var p:=0; var v:=2;

while true do {
  if (v < 10 || v == 10)
  then {p'=v,v'=5 for rand() + 1}
  else {p'=v,v'=-2 for rand() + 2}
}
```

Listing 6.11: Probabilistic cruise control system

After evaluating this program and obtaining a few samples, we observed several possible trajectories that this vehicle could perform.

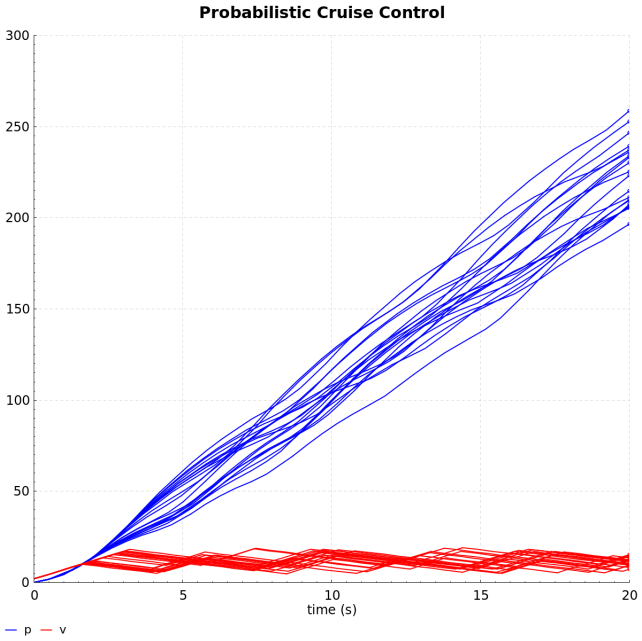


Figure 23: Probabilistic cruise control system

If we look at the original program (figure 22) and corresponding trajectory we see that around the 5 second mark the position is around the value of 50. In the new probabilistic model (figure 23), we observe several trajectories where the value for the position is higher than 50 for the exact same time instant. This could indicate the existence of a problem with the system if we consider that 50 was the limit in terms of safety for the value of the position. For example, any value higher than that could entail that the vehicle would hit an obstacle or a car ahead of it.

This safety concern becomes clearer in the next example where a program of another cruise control system was implemented. This time we have two different vehicles, a leader and a follower. The idea is that the latter should follow the former as close as possible but without crashing (the example is adapted from [5] and further details can be consulted there).

```
var p:= 0; var v := 2;
var pl:= 50; var vl := 10;

while true do {
    if ((p + v + 2.5 < pl + 10) && (((v-5)*(v-5) + 4*(p + v + 2.5 - pl -
    10)) < 0))
    then {p'=v,v'=5,pl'=10 for 1}
    else {p'=v,v'=-2,pl'=10 for 1}
}
```

Listing 6.12: Adaptive cruise control

By evaluating the previous model we obtain the following result, which tells that indeed a close but safe distance is being kept.

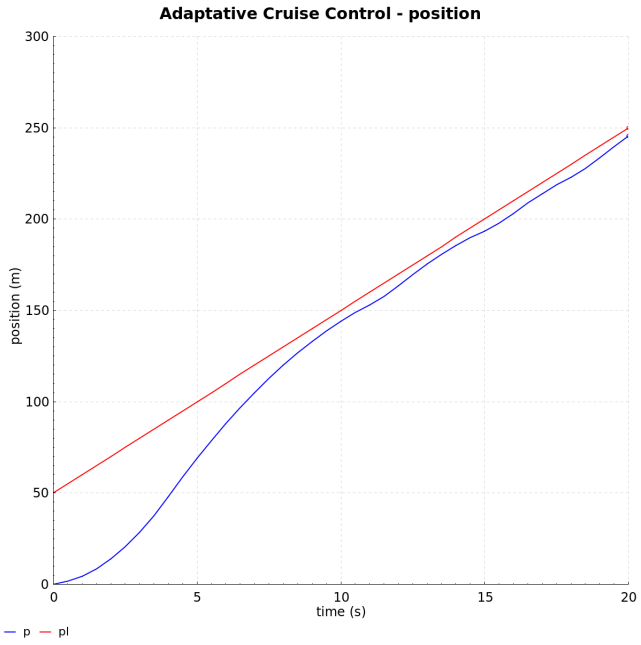


Figure 24: Adaptive cruise control system

However, if we add a margin of error to the duration of acceleration and decelaration we obtain a very different result.

```
var p:= 0; var v := 2;
var pl:= 50; var vl := 10;

while true do {
    if ((p + v + 2.5 < pl + 10) && (((v-5)*(v-5) + 4*(p + v + 2.5 - pl -
    10)) < 0))
    then {p'=v,v'=5,pl'=10 for rand() + 1}
    else {p'=v,v'=-2,pl:=10 for rand() + 2}
}
```

Listing 6.13: Probabilistic adaptive cruise control

In fact the evaluation of the new program (figure 25) now provides possible trajectories in which the follower clearly hits the leader. The margins of error that were added might not be realistic but show how important it is to acknowledge their existence and the types of mistakes we can catch when we are able to include them in our programs.

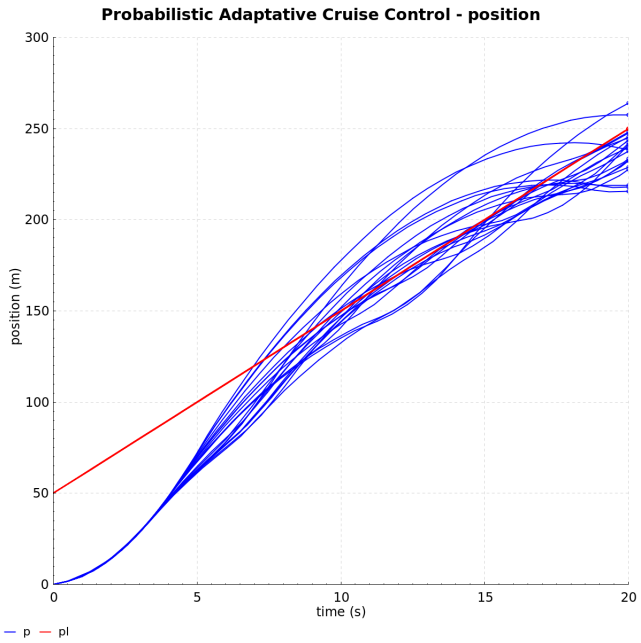


Figure 25: Probabilistic adaptive cruise control

Yet another change we could implement in our system is to use a smaller value for the acceleration. Updating the model with these adjustments and evaluating with the same margins of error produced a situation where all of the follower's trajectories do not intersect the leader's trajectory.

```
var p:= 0; var v := 2;
var pl:= 50; var vl := 10;

while true do {
    if ((p + v + 2.5 < pl + 10) && (((v-5)*(v-5) + 4*(p + v + 2.5 - pl -
   10)) < 0))
    then {p'=0.65*v,v'=5,pl'=10 for rand() + 1}
    else {p'=0.65*v,v'=-2,pl'=10 for rand() + 2}
}
```

Listing 6.14: Probabilistic adaptive cruise control

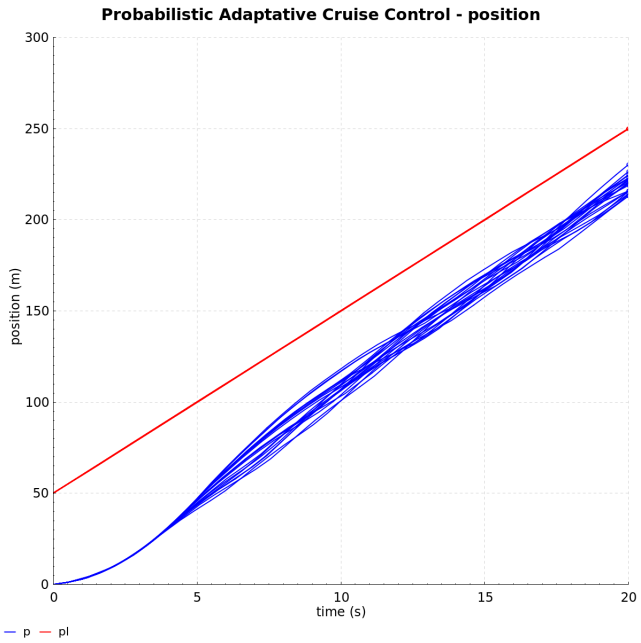


Figure 26: Probabilistic adaptive cruise control with acceleration constraints

6.2.4 Landing System

The following hybrid program describes a landing system of an spacecraft. In order to perform a soft landing the landing process was divided into three stages corresponding to strong, medium, and soft deceleration (the example is adapted from [5]).

```
var y := 10000; var v := -1000; var a:= 0; var g:= 10;
while (y > 1000 || y == 1000) do {
  if (v < -100 || v == -100) then a := (100 - g)
              else a:= -g;
    y' = 1*v, v' = 1*a for 1
};
while (y > 25 || y == 25) do {
  if (v < -20 || v == -20) then a := (20 - g)
             else a:= -g;
    y' = 1*v, v' = 1*a  for 1
};
while (y > 1 || y == 1) do {
  if (v < -1 || v == -1) then a := (15 - g)
            else a:= -g;
    y' = 1*v, v' = 1*a  for 1
}
```

Listing 6.15: Landing system

The result we obtained for this system is the same as the one observed in [5], where the aircraft performs a smooth landing and reaches the ground around the 140 second mark.

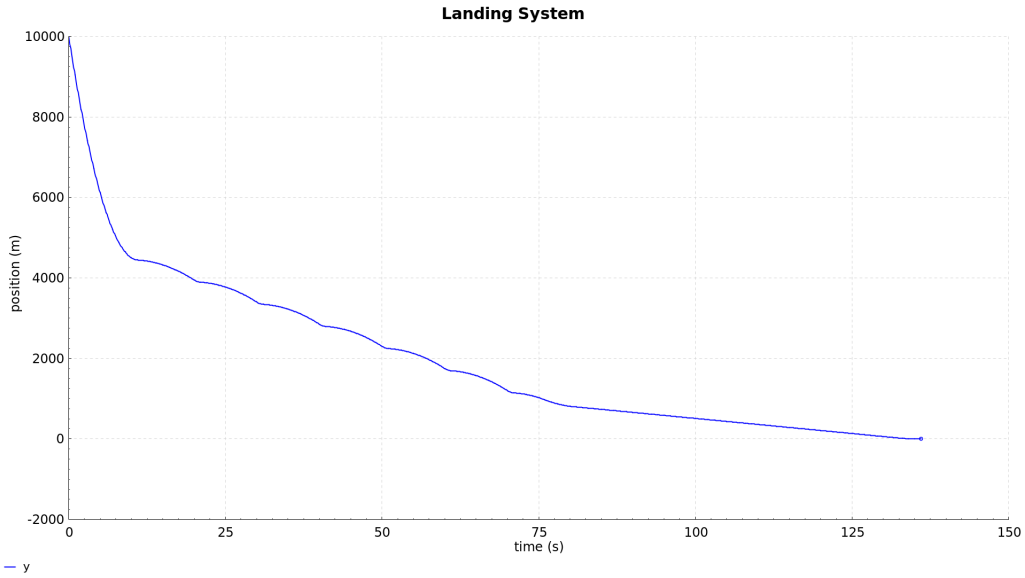


Figure 27: Landing System

In the program above we are again 'running' the system of differential equations for an exact amount of time. But as pointed out previously, we cannot know with exact precision for how long they will actually run. So we can enhance our model to include margins of error, even if they are really small. By doing so we obtain the following:

```
var y := 10000; var v := -1000; var a:= 0; var g:= 10;

while (y > 1000 || y == 1000) do {
  if (v < -100 || v == -100) then a := (100 - g)
              else a:= -g;
    y' := v, v' := a for 1 + rand()*0.25
};

while (y > 25 || y == 25) do {
  if (v < -20 || v == -20) then a := (20 - g)
             else a:= -g;
    y' := v, v' := a  for 1 + coin()*0.25
};
```

```
while (y > 1 || y == 1) do {
  if (v < -1 || v == -1) then a := (15 - g)
            else a:= -g;
    y' := v, v' := a  for 0.01 + rand() * 0.1
}
```

Listing 6.16: Probabilistic landing system

The result we get shows different trajectories, some of them revealing the possibility of the spacecraft landing much earlier than in the original model. This could become hazardous as the fall might be too fast for the crew to handle, or the materials of the aircraft might not be prepared for such an abrupt descent.

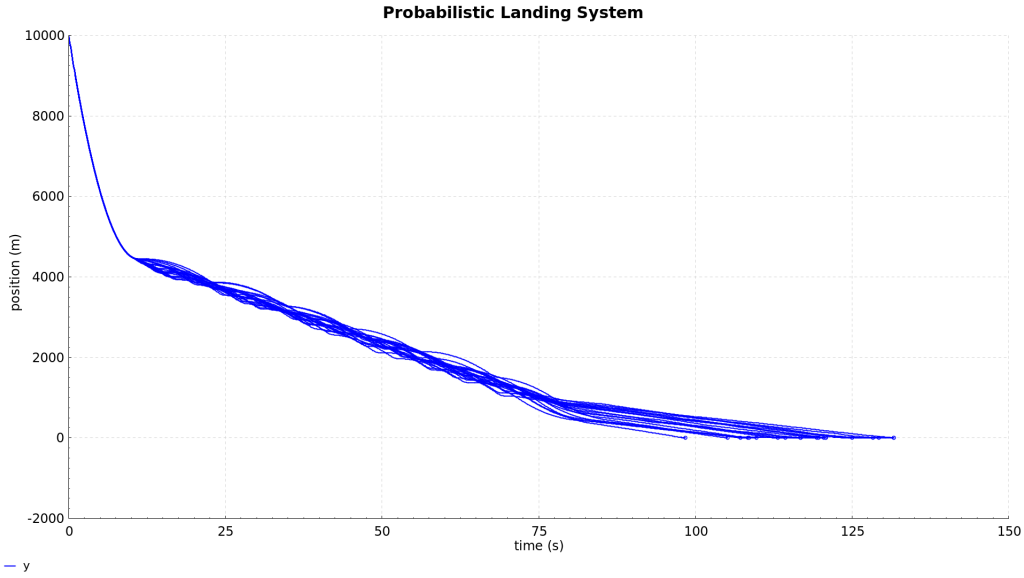


Figure 28: Probabilistic Landing System

6.2.5 Thermostat

Thermostats can also be written down as hybrid programs. These systems are present in many places ranging from rooms in regular houses to industrial freezers which keep large quantities of food or other materials with a constant temperature. So an error in the implementation of these devices can lead to a great loss in terms of materials and money [31]. We have decided to model a specific thermostat by analysing the thermostat system present in [31]. However, differently from our previous case-studies we used our DSL to implement the system. We used the DSL instead of the while-language to show its capabilities and also to illustrate how to use this tool when modelling hybrid systems.

The following code block shows some of the constants which were used such as the initial temperature,

the maximum and minimum temperature of the room and the power of the heater when turned on.

```
module Thermostat where

import ProbTimeDSL

initialTemp :: Double
initialTemp = 20

heaterPower :: Double
heaterPower = 50

tempMax :: Double
tempMax = 25

tempMin :: Double
tempMin = 18
```

Listing 6.17: Initial values of the thermostat implementation

After defining these constants we add the variable `temperature` to the model which we intend to simulate, and define the conditions that dictate when the heater should turn on and off. When the current temperature surpasses the value of the maximum temperature we turn off the heater and turn it back on as soon as the temperature drops below the minimum temperature.

```
initial :: PreM ()
initial = addVar' "temperature" initialTemp

condOn :: PreM Bool
condOn = do
        temperature <- readVar' "temperature"
        return $ temperature < tempMin

condOff :: PreM Bool
condOff = do
        temperature <- readVar' "temperature"
        return $ temperature > tempMax
```

Listing 6.18: Thermostat on and off conditions

Next we define the discrete events which are essentially the systems of differential equations dictating how

the temperature changes when the heater is on or off. The `heatUp` and `coolDown` functions are responsible for running these events, respectively, and check the previously mentioned conditions every 0.01 seconds.

```
turnOn :: DEvent
turnOn = DiffSystem ["temperature"] (\t [temperature] -> [0.1*(heaterPower
    - temperature)])

turnOff :: DEvent
turnOff = DiffSystem ["temperature"] (\t [temperature] -> [negate $ 0.1 *
    temperature])

heatUp :: PreM ()
heatUp = untilC' (preDEvent turnOn 0.01) condOff

coolDown :: PreM ()
coolDown = untilC' (preDEvent turnOff 0.01) condOn
```

Listing 6.19: Thermostat turn off and on events

Finally, the thermostat model will use the `infinite'` combinator to allow the cooling and warming up of the room for an infinite amount of time. As we are using the `sequential'` combinator we know that the simulation will stop once the evaluations hits its maximum duration (value passed as input of the evaluation).

```
thermostat :: PreM ()
thermostat = sequential' initial (infinite' (sequential' coolDown heatUp))
```

Listing 6.20: The thermostat function

The previous model when simulated for 20 seconds returns the following trajectory. Observe that the value of the temperature is always in between the desired temperatures.

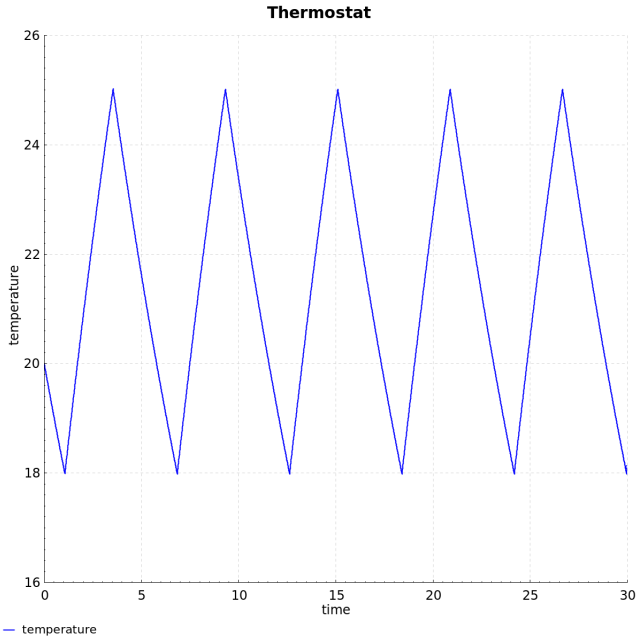


Figure 29: Thermostat

As in the previous examples, we can change this model to include probabilistic elements. In this case, we have some uncertainty associated with the time it takes between the moment the thermostat signals the heater to turn off or on, and the physical moment it happens. Thus we change both the `heatUp` and `coolDown` functions to run the systems of differential equations for a duration which follows an uniform continuous distribution between 0 and 1; and obtain the following code:

```
heatUp :: PreM ()
heatUp = rand' >>= (\v -> sequential' (untilC' (preDEvent turnOn 0.1)
    condOff) (preDEvent turnOn v))

coolDown :: PreM ()
coolDown = rand' >>= (\v -> sequential' (untilC' (preDEvent turnOff 0.1)
    condOn) (preDEvent turnOff v))
```

Listing 6.21: Adding probabilities to the thermostat events

The results now show us some trajectories where the temperature goes either above the maximum or under the minimum.

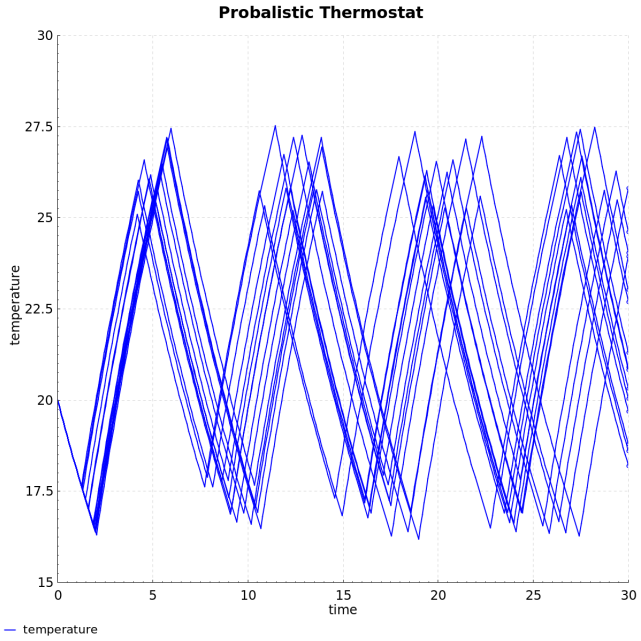


Figure 30: Probabilistic thermostat

Again the incorporation of uncertainty in a program leads to changes in the behaviour that can pose significant safety risks.

6.2.6 Water Tank

The last example we will show corresponds to a water tank system. This example was described in [31] using an hybrid automaton. As in the previous example we will use our DSL to write down an hybrid program that corresponds to this system.

The following code reveals a water level variable, which changes over time, as well as the conditions under which a water pump is turned on or off. The first condition happens corresponds to when the water level falls below 5 cm while the latter occurs when the level goes above 10 cm.

```
module WaterTank where

import ProbTimeDSL

initial :: PreM ()
initial = addVar' "waterLevel" 0

condOpen :: PreM Bool
condOpen = do
        l <- readVar' "waterLevel"
```

```
        return $ l < 5

condClose :: PreM Bool
condClose = do
        l <- readVar' "waterLevel"
        return $ l > 10
```

Listing 6.22: Water tank open and close events

Next, we have the discrete events which change how the system evolves over time. In this case we have the `signalOpen` and `signalClose` which dictate how fast the level of the water increases or decreases after the signal to open and close the pump is sent. The other two events are related to the variation in the water level when the pump is activated and deactivated. In this case, we wish to model a water tank which fills up at a rate of 1 cm s^{-1} and empties at a rate of 2 cm s^{-1}.

```
signalOpen :: DEvent
signalOpen = DiffSystem ["waterLevel"] (\t [level] -> [-2])

signalClose :: DEvent
signalClose = DiffSystem ["waterLevel"] (\t [level] -> [1])

activatePump :: DEvent
activatePump = DiffSystem ["waterLevel"] (\t [level] -> [1])

deactivatePump :: DEvent
deactivatePump = DiffSystem ["waterLevel"] (\t [level] -> [-2])
```

Listing 6.23: Water tank signal events and pump activation/deactivation

The following code block shows the durations for which these events will be simulated. While the tank is filling up the condition that triggers the closing of the pump is checked every 0.1 s. When it is true there is a delay related to the amount of time it takes between the sending of the signal to close the pump and the actual moment where it physically closes. The contrary happens when the tank is being emptied. Again we use the `infinite'` combinator to run these computation until we reach the time specified as an argument.

```
fillTank :: PreM ()
fillTank = sequential' (untilC' (preDEvent activatePump 0.1) condClose) (
    preDEvent signalClose 2)
```

```
emptyTank :: PreM ()
emptyTank = sequential' (until1C' (preDEvent deactivatePump 0.1) condOpen) (
    preDEvent signalOpen 2)

waterTank :: PreM ()
waterTank = sequential' initial (infinite' (sequential' fillTank emptyTank)
    )
```

Listing 6.24: main events of the water tank implementation

The result we obtained showed that for these settings the tank is always in between the level of 1 cm and 12 cm, as we expected.

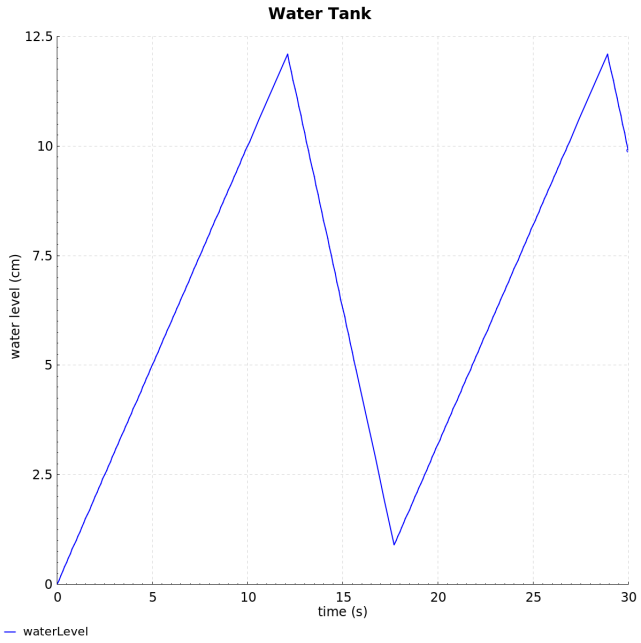


Figure 31: Water tank

In this model it is possible to verify that a delay between the signal and the action is taken into consideration. However, the value is deterministic, which is not realistic. Because of this we decided to turn this delay into a probabilistic delay, following an uniform distribution with a range between 2 and 2.5 seconds.

```
signalDuration :: PreM ()
signalDuration = rand' >>= (\v -> addVar' "signalDuration" (v*0.5 + 2))
```

Listing 6.25: Making the delay probabilistic

This update produced situations where the water level is above 12 cm or below 1 cm. If we expanded this problem and thought of the water tank as a big water reservoir used in a water distribution grid to

companies or the general population, we could have situations where no water was being distributed, which could lead to major problems.

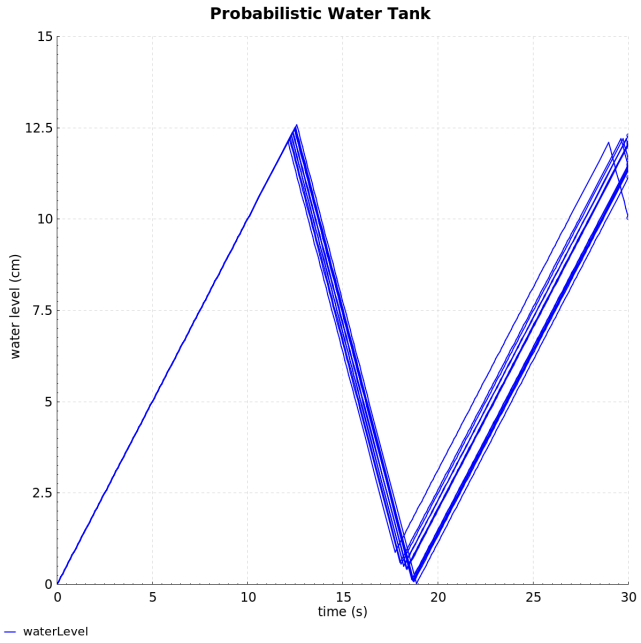


Figure 32: Probabilistic water tank

Chapter 7

Conclusions and future work

In this dissertation we started by covering some of the core concepts of monads and discussed their connection to programming. We also saw some techniques to combine them, such as distributive laws and monad transformers. Then via some well-known examples, such as monadic interpreters, we observed that we can use individual monads to capture specific computational effects (*e.g.* probabilistic behaviour) and subsequently combine them, through the aforementioned methods, to obtain a composite monad capable of handling all the effects at hand.

We then moved to the notion of a hybrid system and verified the ample amount of examples they cover in today's world. We gave a special emphasis to the ones classified as critical, where any type of failure may cause a catastrophe. That lead us to review some of the more well-known theories used to design and analyse hybrid systems rigorously. In this context, we also saw that hybrid effects can be captured by a monad, and briefly discussed the benefits of studying hybrid systems from a monadic view – among other things, we gave particular emphasis to the benefit of studying hybrid computation combined with other effects under the lenses of monad composition.

We then initiated the process of specifying and implementing a while-language capable of handling both probabilistic and hybrid behaviours. We began by reviewing the syntax and operational semantics of a language which deals with wait calls only and probabilistic choice (using a random number generator). Subsequently we devised two operational semantics for probabilistic and full hybrid behaviour (*i.e.* not just wait calls). We also created two complementary denotational semantics, revolving around two different monads, to provide a mathematical foundation to the language.

Afterwards we presented our interpreter of a probabilistic hybrid while-language, whose core engine consists of the two aforementioned operational semantics for probabilistic and full hybrid behaviour. Recall as well that both semantics were implemented by recurring to monads and monad transformers in Haskell, which had the benefit of providing a high level of modularity to our implementation. We also created a visualisation module which allowed to produce histograms and/or trajectories so one can properly analyse

the behaviour of hybrid programs. All in all, this tool facilitates the design and analysis of hybrid programs, promoting an early testing of ideas and concepts and thus giving indications that the software being developed will behave as expected.

Not only this, we also devised an embedded DSL which in essence offers a more expressive language to the hybrid programmer – with all features present in Haskell and with a palette of combinators created specifically for the hybrid paradigm.

Finally, we used both probLince and the DSL to design and analyse different hybrid programs. First we evaluated deterministic versions of these programs from which we could verify totally safe trajectories. We then introduced uncertainty to these models, usually associated with the amount of time the differential system 'runs' for, and checked whether or not the results presented differences when compared to the original versions. In the majority of cases we found that just a little amount of uncertainty would result in vastly different outcomes, with many safety related concerns. A prime example is the adaptive cruise controller: the deterministic version showed no collisions between two vehicles, but the addition of subtle probabilistic elements introduced possibilities of collision.

We thus conclude that uncertainty plays a big part in how hybrid systems should be designed and implemented, as it can quickly lead to undesired outcomes, not always easy to observe. Furthermore, the existence of hybrid modelling tools such as probLince becomes increasingly important as they allow us to be more certain about the design of hybrid systems, which will continue to be more and more present in our daily lives.

After the recap of what was achieved in this dissertation and the conclusions we obtained, we list some ideas to be explored in the future:

- We intend to complete our denotational semantics by adding a definition of the interpretation of while loops. A possible way of doing this is to prove that the generalised writer monad combined with the distribution monad is Elgot.

- Another interesting line of future work is to add new functionalities to the operational semantics, such as the ability to catch and handle exceptions. This would allow us to handle problematic operations, such as division by 0, in a consistent manner.

- It would also be interesting to explore the addition of new probabilistic elements. As an example, both the sample and score functions could be implemented as they are central in many probabilistic languages. This would provide new ways of writing down probabilistic hybrid programs, allowing us to specify more complex systems.

- Soundness and adequacy are very well-known properties that connect denotational and operational semantics [23, 25]. In a nutshell, these properties entail that both styles of semantics agree with each other. It would therefore be useful to establish these properties for the operational and denotational semantics that we devised.

- Finally our tool allows to evaluate hybrid programs in a multitude of scenarios but only a finite amount of them. To be able to cover all scenarios, and thus achieve full verification of a hybrid program, we would need to use for example logical machinery as it happens in [10]. It would therefore be very useful to add some form of logical machinery to probLince so that we have a way of fully verifying hybrid programs. It would be particularly interesting to specify logical formulas with temporal elements, allowing us to ask questions like: "Is there a moment where the vehicle stops after 50 meters?".

Bibliography

[1] Cabal. URL https://cabal.readthedocs.io/en/3.4/index.html. [Online] Last accessed on 2023-07-20.

[2] Chart. URL https://hackage.haskell.org/package/Chart. [Online] Last accessed on 2023-01-14.

[3] Ordinary Differential Equations — GSL 2.7 documentation. URL https://www.gnu.org/software/gsl/doc/html/ode-initval.html. [Online] Last accessed on 2023-07-31.

[4] hmatrix-gsl. URL //hackage.haskell.org/package/hmatrix-gsl. [Online] Last accessed on 2023-08-5.

[5] lince. URL http://arcatools.org/assets/lince.html#fulllince. [Online] Last accessed on 2023-07-20.

[6] Data map. URL https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html. [Online] Last accessed on 2023-08-08.

[7] monad-bayes, . URL https://hackage.haskell.org/package/monad-bayes. [Online] Last accessed on 2023-07-31.

[8] Control.Monad.Random.Lazy, . URL https://hackage.haskell.org/package/MonadRandom-0.5.3/docs/Control-Monad-Random-Lazy.html. [Online] Last accessed on 2023-01-14.

[9] parsec. URL https://hackage.haskell.org/package/parsec. [Online] Last acessed 2023-01-14.

[10] Uppaal. URL https://uppaal.org/. [Online] Last accessed on 2023-07-31.

[11] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage DSL embedded in Haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 225–228, New York, NY, USA, September 2008. Association for Computing Machinery. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411236. URL https://dl.acm.org/doi/10.1145/1411204.1411236.

[12] Marcello M. Bonsangue, Helle Hvid Hansen, Alexander Kurz, and Jurriaan Rot. Presenting Distributive Laws. *Logical Methods in Computer Science*, Volume 11, Issue 3, August 2015. doi: 10.2168/LMCS-11(3:2)2015. URL https://lmcs.episciences.org/1578. Online version: https://arxiv.org/abs/1503.02447.

[13] Marco Devesas Campos and L. S. Barbosa. Implementation of an Orchestration Language as a Haskell Domain Specific Language. *Electronic Notes in Theoretical Computer Science*, 255:45–64, November 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.10.024. URL https://www.sciencedirect.com/science/article/pii/S1571066109004447.

[14] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 7–18, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137583. doi: 10.1145/871895.871897. URL https://doi.org/10.1145/871895.871897.

[15] P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. *The Journal of Logic and Algebraic Programming*, 62(2):191–245, 2005. ISSN 1567-8326. doi: https://doi.org/10.1016/j.jlap.2004.02.001. URL https://www.sciencedirect.com/science/article/pii/S1567832604000232.

[16] Fredrik Dahlqvist and Renato Neves. Compositional semantics for new paradigms: probabilistic, hybrid and beyond, April 2018. URL http://arxiv.org/abs/1804.04145. arXiv:1804.04145 [cs].

[17] Fredrik Dahlqvist, Louis Parlant, and Alexandra Silva. Layer by layer – combining monads. In Bernd Fischer and Tarmo Uustalu, editors, *Theoretical Aspects of Computing – ICTAC 2018*, pages 153–172, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02508-3. Online version: https://arxiv.org/abs/1712.01113.

[18] Fredrik Dahlqvist, Alexandra Silva, and Dexter Kozen. Semantics of Probabilistic Programming: A Gentle Introduction. In Gilles Barthe, Joost-Pieter Katoen, and Alexandra

Silva, editors, *Foundations of Probabilistic Programming*, pages 1–42. Cambridge University Press, 1 edition, December 2020. ISBN 978-1-108-77075-0 978-1-108-48851-8. doi: 10.1017/9781108770750.002. URL https://www.cambridge.org/core/product/identifier/9781108770750%23c1/type/book_part.

[19] EW Dijkstra. *Notes on structured programming*. 1970.

[20] Martin Erwig and Steve Kollmansberger. Kollmansberger, S.: Functional pearls: Probabilistic functional programming in Haskell. J. Funct. Program. 16(1), 21-34. *J. Funct. Program.*, 16:21–34, January 2006. doi: 10.1017/S0956796805005721.

[21] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347, Gothenburg Sweden, August 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628138. URL https://dl.acm.org/doi/10.1145/2628136.2628138.

[22] Sergey Goncharov. Uniform elgot iteration in foundations. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPICS.ICALP.2021.131. URL https://drops.dagstuhl.de/opus/volltexte/2021/14200/. Online version: https://arxiv.org/abs/2102.11828.

[23] Sergey Goncharov and Renato Neves. An adequate while-language for hybrid computation. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372497. doi: 10.1145/3354166.3354176. URL https://doi.org/10.1145/3354166.3354176. Online version: https://arxiv.org/abs/1902.07684.

[24] Sergey Goncharov, Julian Jakob, and Renato Neves. A semantics for hybrid iteration. 2018. doi: 10.4230/LIPICS.CONCUR.2018.22. URL http://drops.dagstuhl.de/opus/volltexte/2018/9560/. Online version: https://arxiv.org/abs/1807.01053.

[25] Sergey Goncharov, Renato Neves, and José Proença. Implementing hybrid semantics: From functional to imperative. In Violet Ka I. Pun, Volker Stolz, and Adenilso Simao, editors, *Theoretical Aspects of Computing – ICTAC 2020*, pages 262–282, Cham, 2020. Springer International Publishing. ISBN 978-3-030-64276-1. Online version: https://arxiv.org/abs/2009.14322.

[26] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, page 220–229, Arlington, Virginia, USA, 2008. AUAI Press. ISBN 0974903949. Online version: https://arxiv.org/abs/1206.3255.

[27] T.A. Henzinger. The theory of hybrid automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, July 1996. doi: 10.1109/LICS.1996.561342. ISSN: 1043-6871.

[28] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. *Arrows, Robots, and Functional Reactive Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-44833-4. doi: 10.1007/978-3-540-44833-4_6. URL https://doi.org/10.1007/978-3-540-44833-4_6.

[29] Peter Höfner and Bernhard Möller. An algebra of hybrid systems. *The Journal of Logic and Algebraic Programming*, 78(2):74–97, January 2009. ISSN 1567-8326. doi: 10.1016/j.jlap.2008.08.005. URL https://www.sciencedirect.com/science/article/pii/S1567832608000799.

[30] Stamatis Karnouskos. Cyber-Physical Systems in the SmartGrid. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 20–23, Lisbon, Portugal, July 2011. IEEE. ISBN 978-1-4577-0435-2. doi: 10.1109/INDIN.2011.6034829. URL http://ieeexplore.ieee.org/document/6034829/.

[31] Tomas Krilavičius. Bestiarium of Hybrid Systems. January 2005.

[32] J. E. Labra Gayo, J. M. Cueva Lovelle, M. C. Luengo Díez, and A. Cernuda del Río. Specification of Logic Programming Languages from Reusable Semantic Building Blocks. *Electronic Notes in Theoretical Computer Science*, 64:220–233, September 2002. ISSN 1571-0661. doi: 10.1016/S1571-0661(04)80352-9. URL https://www.sciencedirect.com/science/article/pii/S1571066104803529.

[33] Daan Leijen and Erik Meijer. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science*, 41(1):1–20, 2001.

[34] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming lan-*

guages, POPL '95, pages 333–343, New York, NY, USA, January 1995. Association for Computing Machinery. ISBN 978-0-89791-692-9. doi: 10.1145/199448.199528. URL https://doi.org/10.1145/199448.199528.

[35] Christoph Lüth and Neil Ghani. Composing monads using coproducts. *ACM SIGPLAN Notices*, 37 (9):133–144, September 2002. ISSN 0362-1340. doi: 10.1145/583852.581492. URL https://dl.acm.org/doi/10.1145/583852.581492.

[36] Eugenio Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, June 1989. doi: 10.1109/LICS.1989.39155.

[37] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(91)90052-4. URL https://www.sciencedirect.com/science/article/pii/0890540191900524. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[38] Renato Neves. *Hybrid programs*. PhD thesis, Minho University, 2018.

[39] José Nuno Oliveira. Program design by calculation. *Draft of textbook in preparation*, 2018.

[40] Hugo Paquet and Sam Staton. LazyPPL: laziness and types in non-parametric probabilistic programs. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*, 2021. URL https://openreview.net/forum?id=yHox9OyegeX.

[41] Daniela Petrişan and Ralph Sarkis. Semialgebras and Weak Distributive Laws. *Electronic Proceedings in Theoretical Computer Science*, 351:218–241, December 2021. ISSN 2075-2180. doi: 10.4204/EPTCS.351.14. URL http://arxiv.org/abs/2106.13489v3.

[42] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991. ISBN 9780262660716.

[43] Maciej Piróg and Sam Staton. Backtracking with cut via a distributive law and left-zero monoids. *Journal of Functional Programming*, 27:e17, 2017. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796817000077. URL https://www.cambridge.org/core/product/identifier/S0956796817000077/type/journal_article.

[44] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. ISBN 978-3-642-14508-7.

[45] André Platzer. *Logical foundations of cyber-physical systems*, volume 662. Springer, 2018. ISBN 978-3-319-63587-3. doi: https://doi.org/10.1007/978-3-319-63588-0.

[46] André Platzer. Differential Dynamic Logic for Hybrid Systems. *Journal of Automated Reasoning*, 41 (2):143–189, August 2008. ISSN 1573-0670. doi: 10.1007/s10817-008-9103-8. URL https://doi.org/10.1007/s10817-008-9103-8.

[47] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of Cyber-Physical Systems. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, Nanjing, China, November 2011. IEEE. ISBN 978-1-4577-1010-0 978-1-4577-1009-4 978-1-4577-1007-0 978-1-4577-1008-7. doi: 10.1109/WCSP.2011.6096958. URL http://ieeexplore.ieee.org/document/6096958/.

[48] Juliana Souza and Renato Neves. Personal Communication, 2022.

[49] Guy L. Steele. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 472–492, New York, NY, USA, February 1994. Association for Computing Machinery. ISBN 978-0-89791-636-3. doi: 10.1145/174675.178068. URL https://doi.org/10.1145/174675.178068.

[50] David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. *arXiv preprint arXiv:1608.05263*, 2016.

[51] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA, May 1990. Association for Computing Machinery. ISBN 978-0-89791-368-3. doi: 10.1145/91556.91592. URL https://doi.org/10.1145/91556.91592.

[52] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, February 1992. Association for Computing Machinery. ISBN 978-0-89791-453-6. doi: 10.1145/143165.143169. URL https://doi.org/10.1145/143165.143169.

[53] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993. ISBN 9780262731034.

[54] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

[55] Maaike Zwart and Dan Marsden. No-Go Theorems for Distributive Laws. *Logical Methods in Computer Science*, Volume 18, Issue 1:6253, January 2022. ISSN 1860-5974. doi: 10.46298/lmcs-18(1:13)2022. URL https://lmcs.episciences.org/6253. Online version: https://arxiv.org/abs/2003.12531.

[56] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(ICFP):83:1–83:29, July 2018. doi: 10.1145/3236778. URL https://doi.org/10.1145/3236778.

Appendix A
Parsing grammar

```
DTerm = DTerm - DTermExpr
      | DTerm - DTermExpr

DTermExpr = DTermExpr '*' DTermFactor
          | DTermExpr '/' DTermFactor

DTermFactor = '(' DTerm ')'
            | -DTerm
            | var String
            | Double

Term = Term '-' TermFactor
      | Term '+' TermFactor

TermFactor = -Term
           | (Term)
           | DTerm
           | coin()
           | normal(Term, Term)
           | rand()

BTerm = BTerm && BFactor
      | BTerm || BFactor
      | !BFactor

BFactor = DTerm == DTerm
        | DTerm '<' DTerm
        | DTerm '>' DTerm
        | true
        | false

commentBlock = /* MultipleComment */

MultipleComment = MultipleComment comment
                | empty

comment = comment '\n'
        | comment '\r\n'
        | comment char
        | empty

commentLine = // comment

ProgramSeq = ProgramsSeq ';' Program
```

```
Program = skip
        | varAssign ':=' Term
        | varAtrib ':=' Term
        | if '(' BTerm ')' then '{' Program '}' else '{' Program '}'
        | while '(' BTerm ')' do '{' Program '}'
        | wait '(' Term ')' do '{' Program '}'
        | diffSystem for Term

VarAssign = var String

varAtrib = String

diffSystem = diffSystem ',' diffEquation

diffEquation = String '=' diffExpr

diffExpr = diffExpr '+' diffTerm
         | diffExpr '-' diffTerm

diffTerm = diffTerm '*' diffFactor
         = diffTerm '-' diffFactor

diffFactor = -diffExpr
           | (diffExpr)
           | Double
```

Appendix B
Installation and user guide

B.1 Interpreter

To install this tool the user needs to first download the folder "probLince-WL" present in the following github repository: https://github.com/RuiC10/probLince. Then moving into the folder and using the command `cabal install` the project will be built, and the executable will be generated. The location of the executable might depend on the user's cabal configuration or operating system, so it is better to check cabal's documentation [1]. Note that before doing the `cabal install` command it might be necessary to install the BLAS and LAPACK packages, as well as the GNU scientific library as the ODE solver uses it to calculate systems of differential equations. The steps for installing these libraries highly depend on the operating system, but they can be check online.

We present below the user manual, with all the flags necessary to run the interpreter as well as the explanation of those commands:

COMMANDS

```
probLince-WL EXECUTION_MODE PATH -h CONF_PATH NUM_RUNS MAX_DUR
probLince-WL EXECUTION_MODE PATH -t CONF_PATH NUM_RUNS MAX_DUR
probLince-WL EXECUTION_MODE PATH -trj CONF_PATH NUM_RUNS PRECISION MAX_DUR
```

DESCRIPTION

probLince is a tool which allows the modelling of hybrid systems with probabilistic behaviour.

OVERVIEW

This tool integrates a parser and interpreter for a while-language as well as a visualisation module which creates histograms or graphics. To use this tool the user needs to specify the execution mode (see below) and the path of the file (PATH). Moreover, there are three different options for the visualisation mode:

- `-h`: Creates an histogram which shows the frequency of the last value each variable presented in every execution.
- `-t`: Creates an histogram which shows the frequency of the durations of each execution.

- `-trj`: Creates a graphic which shows the trajectory of the variables for every execution.

Next, the user needs to specify the CONF_PATH which corresponds to the configuration file of either an histogram or a graphic (see below). In every case the user needs to specify the amount of executions the interpreter will make (NUM_RUNS). When dealing with histograms the user only needs to specify the maximum duration (MAX_DUR, in seconds) for each execution. When creating graphics of trajectories, the user needs to specify the precision (PRECISION, in seconds) of the trajectory.

Execution modes

- `-rte`: Performs the execution of hybrid programs using real-time wait calls and random number generators for the probabilistic elements.

- `-pre`: Performs the execution of hybrid programs without using real-time wait and uses distributions instead of RNG for the probabilistic elements. The end result consists in the sampling of values from the distributions.

Configuration files

The configuration files for both histogram and graphics correspond to `JSON` files. The following fragment shows an example for an histogram configuration file,

```
{
  "imageSize": [1500,1500]
  , "fileName": "randomWalk2D-wait-500.png"
  , "title":"Random Walk 2D with wait call - 500 runs"
  , "range": null
  , "showZeroBins": true
  , "numBins": 100
  , "x_axis_title": "Time"
  , "y_axis_title": "Frequency"
  , "var": "time"
  , "varColor": [0.4, 0.5, 0.6]
  , "y_axisSize": 24
  , "x_axisSize": 24
  , "y_labelSize": 24
  , "x_labelSize": 24
  , "titleSize": 30
  , "legendSize": 22
}
```

Listing B.1: Histogram configuration file example for -t flag

Note that the histogram configuration above is only for obtaining histograms related to the durations of the evaluations (`-t` flag). If the user wishes to obtain a histogram related to a specific variable (`-h` flag), then the following configuration is the right one (notice the “var” field):

```
{
```

```
    "imageSize": [1500,1500]
    , "fileName": "randomWalk2D-10000.png"
    , "title":"Random Walk 2D - 10000 runs"
    , "range": null
    , "showZeroBins": true
    , "numBins": 100
    , "x_axis_title": "Distance"
    , "y_axis_title": "Frequency"
    , "var": "distance"
    , "varColor": [0.4, 0.5, 0.6]
    , "y_axisSize": 24
    , "x_axisSize": 24
    , "y_labelSize": 24
    , "x_labelSize": 24
    , "titleSize": 30
    , "legendSize": 22
}
```

Listing B.2: Histogram configuration file example for -h flag

In the fragment below we can observe the content for a graphic configuration file:

```
{
    "imageSize": [1200,1200]
    , "fileName": "BouncingBall.png"
    , "title": "Bouncing Ball"
    , "vars": [["p", [0,0,1]]]
    , "axisTitle": ["time (s)", "position (m)"]
    , "y_axisSize": 24
    , "x_axisSize": 24
    , "y_labelSize": 24
    , "x_labelSize": 24
    , "titleSize": 30
    , "legendSize": 22
}
```

Listing B.3: Graphic configuration file example

Inside the "probLince-WL" folder there is another folder called "tests" with several instances of hybrid systems. If we apply the following command to the cruise controller system, we obtain 20 trajectories, each with a maximum duration of 100 seconds being every trajectory evaluated in intervals of 0.5 seconds:

```
probLince -pre "tests/CruiseControl/ProbCruiseControl" -trj "tests/
    CruiseControl/probGraph.json" 20 0.5 100
```

Listing B.4: probLince test command

The figure 33 show the result for the command above:

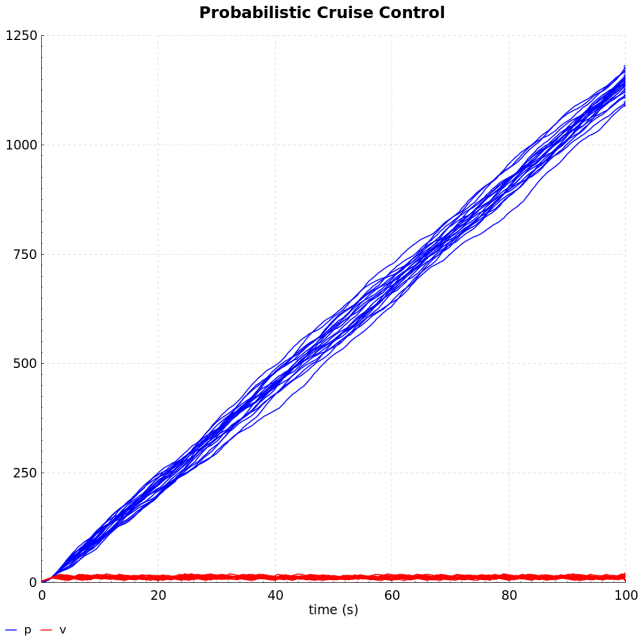


Figure 33: ProbLince command test

B.2 Domain specific language

The DSL is specified as a Haskell library, which allows it to be used by other projects. First clone the repository and then move the folder `probLince-DSL` to the desired location (*e.g.* local dependency folder of another cabal project). To make the package available to the `.cabal` file, we first need to update the `cabal.project` file. If no file exists, one should be created. Then add the following line:

```
packages: {path to the folder}/probLinceDSL.cabal ./*.cabal
```

Next change the `build-depends` section of the `cabal` file to include the library:

```
build-depends:    base ^>=4.16.4.0,
                  probLinceDSL
```

By doing these steps, we can now import the library inside a Haskell project and use its features. To know more details about the usage of cabal to create an Haskell project visit the documentation in [1].

Appendix C
Proofs

C.1 Proof of theorem 1

Proof. In order to prove that the following function is indeed a distributive law between a generic monad T and the generalised writer monad, we need to prove that the four diagrams commute:

$$\lambda \;=\; [T\ i_1 \cdot \tau, \eta^T \cdot i_2]$$

We will start by proving that the following equation holds: $[T\ i_1 \cdot \tau, \eta^T \cdot i_2] \cdot W[\mathbb{M}]\eta^T = \eta^T$. By developing the left side of the equation we obtain,

$$\begin{aligned}
&[Ti_1 \cdot \tau, \eta^T \cdot i2] \cdot W[\mathbb{M}]\eta^T \\
&= \{\ \text{definition of } W[\mathbb{M}]\ \} \\
&[Ti_1 \cdot tau, \eta^T \cdot i2] \cdot (\eta^T \times id + id) \\
&= \{\ \text{coproduct laws}\ \} \\
&[Ti_1 \cdot \tau \cdot (\eta^T \times id), \eta^T \cdot i_2] \\
&= \{\ \text{strong monad laws}\ \} \\
&[Ti_1 \cdot \eta^T, \eta^T \cdot i_2] \\
&= \{\ \text{naturality of } \eta\ \} \\
&[\eta \cdot i_1, \eta^T \cdot i_2] \\
&= \{\ [f \cdot g, f \cdot k] = f \cdot [g, k] \text{ and } [i_1, i_2] = id\ \} \\
&\eta^T
\end{aligned}$$

Which matches the right side of the equation. Thus the diagram commutes and we can move on to the next equation: $[T\ i_1 \cdot \tau, \eta^T \cdot i_2] \cdot \eta_T^{W[\mathbb{M}]} = T\ \eta^{W[\mathbb{M}]}$. Again, starting with the left-hand side of the equation we get,

$$\begin{aligned}
&[Ti_1 \cdot \tau, \eta^T \cdot i_2] \cdot \eta^{W[\mathbb{M}]} \\
&= \{\ \text{def. } \eta^{W[\mathbb{M}]}\ \} \\
&[Ti_1 \cdot \tau, \eta^T \cdot i_2] \cdot i_1 \cdot \langle id, \underline{i} \rangle \\
&= \{\ \text{coproduct laws}\ \} \\
&Ti_1 \cdot \tau \cdot \langle id, \underline{i} \rangle \\
&= \{\ \text{strong functor laws}\ \} \\
&Ti_1 \cdot T\langle id, \underline{i} \rangle
\end{aligned}$$

$$= \{\, T\ f \cdot T\ g = T\ (f \cdot g)\,\}$$
$$T\eta^{W[\mathbb{M}]}$$

It is possible to observe that we also obtain the same value as the one in the right side of the equation. So this diagram also commutes, which takes us to the third proof: $\lambda \cdot W[\mathbb{M}]\mu^T = \mu^T_{W[\mathbb{M}]} \cdot T\lambda \cdot \lambda_T$.

$$\begin{aligned}
&\lambda \cdot W[\mathbb{M}]\mu^T \\
&= \{ \text{ def. } \lambda, \text{ def. } W[\mathbb{M}] \} \\
&[Ti_1 \cdot \tau, \eta \cdot i_2] \cdot ((\mu^T \times id) + id) \\
&= \{ \text{ coproduct laws } \} \\
&[Ti_1 \cdot \tau \cdot (\mu^T \times id), \eta^T \cdot i_2] \\
&= \{ \text{ strong monad laws } \} \\
&[Ti_1 \cdot \mu^T \cdot T\tau \cdot \tau, \eta^T \cdot i_2] \\
&= \{ \text{ naturality of } \mu^T \} \\
&[\mu^T \cdot TTi_1 \cdot T\tau \cdot \tau, \eta^T \cdot i_2] \\
&= \{ \text{ monad laws } \} \\
&[\mu^T \cdot TTi_1 \cdot T\tau \cdot \tau, \mu^T \cdot T\eta^T \cdot \eta^T \cdot i_2] \\
&= \{\text{coproduct laws}\} \\
&\mu^T \cdot [TTi_1 \cdot T\tau \cdot \tau, T\eta^T \cdot \eta^T \cdot i_2] \\
&= \{ \text{ naturality of } \eta \} \\
&\mu^T \cdot [TTi_1 \cdot T\tau \cdot \tau, T\eta^T \cdot Ti_2 \cdot \eta^T] \\
&= \{ \text{ absorption + } \} \\
&\mu^T \cdot [TTi_1 \cdot T\tau, T\eta^T \cdot Ti_2] \cdot (\tau + \eta^T) \\
&= \{ \text{ coproduct and functorial laws } \} \\
&\mu^T \cdot T[Ti_1 \cdot \tau, \eta \cdot i_2] \cdot [Ti_1 \cdot \tau, Ti_2 \cdot \eta^T] \\
&= \{ \text{ naturality of } \eta \} \\
&\mu^T \cdot T[Ti_1 \cdot \tau, \eta \cdot i_2] \cdot [Ti_1 \cdot \tau, \eta^T \cdot i_2] \\
&= \{ \lambda \text{ definition } \} \\
&\mu^T \cdot T\lambda \cdot \lambda
\end{aligned}$$

This ends the proof as both sides of the equation match, which means the diagram commutes.

Lastly, we need to verify if the equation $\lambda \cdot \mu_T^{W[\mathbb{M}]} = T\mu^{W[\mathbb{M}]} \cdot \lambda_{W[\mathbb{M}]} \cdot \mathbb{W}^{\mathbb{DE}}\lambda$ holds. So, by developing the left side of the equation we obtain:

$$\begin{aligned}
&T\mu \cdot \lambda \cdot W[\mathbb{M}]\lambda \\
&= \{\text{def. } \lambda, \text{ fusion-+, def functor } W[\mathbb{M}]\} \\
&[T\mu \cdot Ti_1 \cdot \tau, T\mu \cdot \eta \cdot i_2] \cdot (\lambda \times id + id) \\
&= \{\text{absorption + }\} \\
&[T\mu \cdot Ti_1 \cdot \tau \cdot (\lambda \times id), T\mu \cdot \eta \cdot i_2] \\
&= \{\text{def } \mu, \text{ natural-}\eta\} \\
&[T[((id \times (\cdot)) \cdot assocr + \triangleright) \cdot distl, i_2] \cdot Ti_1 \cdot \tau \cdot (\lambda \times id), T\mu \cdot Ti_2 \cdot \eta]
\end{aligned}$$

$= \{\text{Functor laws, coproduct cancellation }\}$

$[T(((id \times (\cdot)) \cdot assocr + \triangleright) \cdot distl) \cdot \tau \cdot (\lambda \times id), T\mu \cdot Ti_2 \cdot \eta]$

$= \{\text{def } \mu\text{, functor laws, coproduct cancellation }\}$

$[T(((id \times (\cdot)) \cdot assocr + \triangleright) \cdot distl) \cdot \tau \cdot (\lambda \times id), Ti_2 \cdot \eta]$

$= \{\text{ strong functor laws, natural-}\eta\}$

$[T(((id \times (\cdot)) \cdot assocr + \triangleright) \cdot distl) \cdot Tundistl \cdot [Ti_1 \cdot \tau, Ti_2 \cdot \eta] \cdot (\tau \times id + id) \cdot distl, \eta \cdot i_2]$

$= \{\text{ Functor laws, } distl \cdot undistl = id\}$

$[T((id \times (\cdot)) \cdot assocr + \triangleright) \cdot [Ti_1 \cdot \tau, Ti_2 \cdot \eta] \cdot (\tau \times id + id) \cdot distl, \eta \cdot i_2]$

$= \{\text{ Fusion + }\}$

$[[T((id \times (\cdot)) \cdot assocr + \triangleright) \cdot Ti_1 \cdot \tau, T((id \times (\cdot)) \cdot assocr + \triangleright) \cdot Ti_2 \cdot \eta] \cdot (\tau \times id + id) \cdot distl, \eta \cdot i_2]$

$= \{\text{ Absorption + }\}$

$[[T((id \times (\cdot)) \cdot assocr + \triangleright) \cdot Ti_1 \cdot \tau \cdot (\tau \times id), T((id \times (\cdot)) \cdot assocr + \triangleright) \cdot Ti_2 \cdot \eta] \cdot distl, \eta \cdot i_2]$

$= \{\text{ Strong functor laws }\}$

$[[Ti_1 \cdot \tau \cdot ((id \times (\cdot)) \cdot assocr), T((id \times (\cdot)) \cdot assocr + \triangleright) \cdot Ti_2 \cdot \eta] \cdot distl, \eta \cdot i_2]$

$= \{\text{ Functor laws, Natural-}i_2\text{ }\}$

$[[Ti_1 \cdot \tau \cdot ((id \times (\cdot)) \cdot assocr), T(i_2 \cdot \triangleright) \cdot \eta] \cdot distl, \eta \cdot i_2]$

$= \{\text{ Natural-}\eta\text{ }\}$

$[[Ti_1 \cdot \tau \cdot ((id \times (\cdot)) \cdot assocr), \eta \cdot i_2 \cdot \triangleright] \cdot distl, \eta \cdot i_2]$

$= \{\text{ Absorption + }\}$

$[[Ti_1 \cdot \tau, \eta \cdot i_2] \cdot ((id \times (\cdot)) \cdot assocr + \triangleright) \cdot distl, \eta \cdot i_2]$

$= \{\text{ } \lambda \text{ def }\}$

$[\lambda \cdot ((id \times (\cdot)) \cdot assocr + \triangleright) \cdot distl, \lambda \cdot i_2]$

$= \{\text{ Fusion + }\}$

$\lambda \cdot [((id \times (\cdot)) \cdot assocr + \triangleright) \cdot distl, i_2]$

$= \{\text{ def } \mu\text{ }\}$

$\lambda \cdot \mu$

This allows us to verify that the last equation also holds and so the last diagram also commutes, ending the proof of the theorem.