Department of Computer Science

Submitted in part fulfilment for the degree of MEng.

Integrating Theorem Proving with Computational Algebra Systems

Christian Pardillo Laursen

7th May 2020

Supervisor: Simon Foster

Contents

Executive Summary **vi**

1 Introduction **1**

2 State of the Art **3**
2.1 Formal verification 3
2.2 Current approaches used for CPS verification 4
2.3 Isabelle and its implementation 4
2.4 Isabelle for CPS verification 5

3 Preliminaries **6**
3.1 Isabelle/ML 6
3.2 Lambda terms 6
3.3 Isabelle representation of ODEs 8
3.4 Verifying solutions to ODEs 9

4 Methods **10**
4.1 Overview 10
4.2 Testing 10

5 Implementation **13**
5.1 Translation to Mathematica 13
5.1.1 Interfacing with the Wolfram Engine 14
5.2 Conversion to AST 15
5.2.1 Lexical analysis 15
5.2.2 Parsing 16
5.3 Importing to Isabelle 17
5.3.1 Domain retrieval 17
5.3.2 Translation to term 17
5.4 Certifying the solution 18

6 Analysis **19**
6.1 Evaluation 19
6.2 Reflection 21

7 Conclusions **22**
7.1 Further work 22

A Formal Languages for Lexing and Parsing **24**
A.1 Token regular expressions 24
A.2 BNF grammar for expressions 24

List of Figures

3.1 Type synonym for systems of ODEs in Isabelle 8
3.2 Representation of equation 3.1 as an Isabelle term 8
3.3 Solution of equation 3.1 as an Isabelle term 9
3.4 Definition of solves_ode 9

4.1 Plugin workflow 11

5.1 ODE representation in Isabelle 13
5.2 ODE solution in Isabelle 18
5.3 Verifying the ODE solution 18
5.4 Isabelle accepting the lemma as true 18

List of Tables

3.1 Meaning of each term constructor 7

4.1 ODE test set 12

6.1 Test results 20

Executive Summary

The objective of this project is to produce a program which will integrate the Isabelle proof assistant with Mathematica, a Computational Algebra System, with the aim of producing solutions to ordinary differential equations (ODEs).

A proof assistant is a piece of software which is used for verifying that a piece of mathematics is correct. It finds use not only in formalizing mathematics, but also in proving properties about real systems. A Computational Algebra System, or CAS, is mathematical software whose purpose it is to manipulate and solve mathematical expressions. Examples include MATLAB and Mathematica.

This project will help improve the ability of Isabelle to verify Cyber-Physical Systems (CPS). These are systems that combine discrete control with continuous dynamics, e.g. robots. The dynamics of such a system can be specified via ODEs, and the solutions to the ODEs indicate how the system will behave. This information can be used to prove many critical properties; a hypothetical example of what can be done is the proof that upon pulling the emergency break lever on a train, it will always stop within a set distance. Currently, Isabelle is not capable of producing solutions to ODEs, although when given a solution it is capable of reasoning about it. Mathematica is capable of producing these solutions; hence an integration between it and Isabelle would be beneficial.

The program is written in Isabelle/ML, an extension of the Standard ML programming language which is deeply integrated with the Isabelle system. All the code is written with the main objectives of readability and maintainability, following the guidelines set out by the Isabelle development team. One of the aims is to provide a well documented real-world example of how one may write software which integrates with Isabelle.

The plugin workflow for solving an ODE is to first translate it from Isabelle to Mathematica. Then, the translated expression is sent over an interface to Mathematica for solving, and the Mathematica expression representing the solution is obtained. The domain of the expression is also retrieved from Mathematica, and both the solution and domain are translated back to Isabelle and then combined to form a lemma that proves that the obtained expression solves the original ODE.

The outcome of this project is therefore a plugin for Isabelle which provides

an interface to Mathematica and is able to generate and verify solutions to ODEs within its environment. A test set of ODEs was devised, with the aim of testing the plugin's its functionality in a variety of cases. The plugin performs well on the tests, producing the correct response to most elements of the test set; there were two failures, both of which due to issues unrelated to the plugin.

There is much potential for future work in this area, as there is more functionality in Mathematica that may prove useful, such as quantifier elimination. In addition, the tool is currently not very user-friendly, and an IDE integration would aid its usability.

1 Introduction

Cyber-physical systems (CPS) are engineered systems that are built from, and depend upon, the seamless integration of computation and physical components[1]. They arise everywhere, and range from the minuscule, such as pacemakers, to large scale, such as aircrafts or the power grid[2]. The safe operation of such systems is often critical, as a software error could lead to accidents and loss of life. This is why it is very important to be able to certify that the design of a CPS is correct.

There are many ways of doing this verification, but deductive verification is perhaps the one that provides the strongest guarantee. In this approach, formal proofs are produced about a mathematical model of a system[3]. They are developed using a proof assistant, a software tool which is capable of verifying that a proof is valid. Proof assistants also provide many tools to enhance automation. By mathematically proving properties about a hybrid system, it is guaranteed that these properties will always hold in the real world, as long as the assumptions made when producing the model hold too.

A difficult part of reasoning about CPS is the fact that they interact with real world physics, difficult to model due to their continuous nature. Ordinary differential equations (ODEs) provide a useful way of describing continuous dynamics by specifying how the physical system will evolve over time in any given state[4]. A solution to an ODE describes the actual behaviour of the system modelled by said ODE, which is straightforward to use in a proof.

The proof assistant that will be considered in this report is Isabelle. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus[5]. Its main application is the formalization of mathematical proofs and in particular formal verification, which includes proving correctness of computer hardware or software and properties of computer languages and protocols, such as the seL4 microkernel[6]. Isabelle is capable of verifying that a given function solves an ODE, but it cannot generate these solutions therefore relies on the user to supply them. This hampers its ability to verify CPS.

To generate the solutions, we can instead use a Computer Algebra System (CAS). These are pieces of software designed to manipulate mathematical expressions in analytical form[7], and many of them support

symbolically solving differential equations. Examples include MATLAB, Wolfram Mathematica and SageMath.

The aim of this project is to produce a plug-in that integrates Mathematica with Isabelle to allow reasoning about the symbolic solutions of ordinary differential equations. This will extend Isabelle's ability to reason about cyber-physical systems, as it will remove the need for the user to supply the solutions to ODEs by hand. This plug-in should be written with extensibility in mind, as there are other features of Mathematica that can also be used in CPS verification.

2 State of the Art

2.1 Formal verification

There are many ways in which software engineers try to ascertain that their software functions correctly. The most common ones are informal specification and testing, but for many applications they are not sufficient. They cannot guarantee that the program will always function correctly, only that it sometimes will. A more robust approach can be found with formal methods. They are defined by Woodcock et al. as “mathematical techniques, often supported by tools, for developing software and hardware systems”[8].

Formal methods can be applied at every stage of the software engineering process, from specifying requirements to verifying that the implementation meets those requirements. They can be applied in a wide variety of fields, although they are most popular with safety-critical software where correct operation is of utmost importance.

For example, there are swathes of techniques for system specification, such as the Alloy language[9], inspired by the Z notation. This language is targeted at the creation of models, which can be explored via model-checking. In this paradigm, a transition model for a system is constructed and fully explored. This provides a total guarantee that the model operates correctly, which also proves that the system being modelled is correct given that it conforms to the model. The major downside of this approach is that the computational expense of checking the model scales exponentially with size, which restricts its ability for verifying real software.

The approach for system verification discussed in this report is deductive verification, where a designer interacts with an interactive theorem prover to generate proofs of correctness of systems[10]. These proofs can guarantee properties about systems with a very large or continuous state space. However, they tend to require more user interaction and domain knowledge than model checking, which is automatic.

2.2 Current approaches used for CPS verification

Hybrid systems are a mathematical model of dynamic systems that combine discrete control with continuous dynamics. A familiar example is a robot, which is operated by a computer processor - discrete control - and interacts with the real world - continuous dynamics. From a mathematical perspective cyber-physical systems are hybrid systems[4, p. 6]. There are many approaches to formally verifying hybrid systems, such as modelling and reachability analysis, and also deductive verification.

Differential dynamic logic(d$\mathcal{L}$) is a program notation for hybrid systems, which has been used in practice to model and verify cyber-physical systems. It provides a way of representing continuous evolution of a dynamic system by using ODEs, as well as representing discrete state transitions as instantaneous assignments.

Currently, the most popular theorem prover for d$\mathcal{L}$ is KeYmaera X[11], which uses integration with the Mathematica CAS to obtain solutions to ODEs. It has been used in real case studies, such as in [12] for verifying a control algorithm for a robot that will never be responsible for a crash, and in [13] for proving that the ACAS X aircraft collision avoidance system provides safe guidance under a set of assumptions.

There are downsides to KeYmaera, such as its limitation to only reasoning about d$\mathcal{L}$ programs. This means that software being executed by a CPS cannot be verified by KeYmaera alone; a solution to this is presented in the VeriPhy pipeline which generates executable software from a verified program[14]. However, it would be better if a program could be proven to meet the specification given by the hybrid system so that it may be optimized by non-verified tools or humans, instead of relying on code generated directly from the specification.

2.3 Isabelle and its implementation

A way of verifying that code meets a specification is by using Isabelle, a more general alternative to KeYmaera. This can for example be done for the C programming language by using Isabelle/C[15]. It enables deductive verification of C programs within the Isabelle environment by adding a C environment to Isabelle/jEdit, the default IDE, which can be augmented with annotations specifying invariants or pre-conditions.

Isabelle is a general theorem prover. It has a meta-logic, Isabelle/Pure, upon which other logics are founded. Pure is based on the typed λ-

calculus[16] and is written in the Poly/ML[17] implementation of the Standard ML language. The main logic used to prove theorems is Isabelle/HOL, which implements Higher-Order logic, although there are also implementations of Zermelo-Fraenkel set theory and First-Order logic[18], amongst others. Formal developments in Isabelle are available from the Archive of Formal Proofs (AFP), a collection of Isabelle-checked proof libraries[19].

Isabelle/Pure marks the point where the raw ML toplevel is superseded by Isabelle/ML[20]. Isabelle/ML is an extension of ML that is deeply integrated within the Isabelle environment, and is used for writing code that interacts with the proofs. ML can be integrated in proof scripts via the `ML` and `ML_file` commands. ML scripts can be used to transform the proof state via tactics, which can be included in formal proofs and prove useful for a range of applications.

2.4 Isabelle for CPS verification

There are several options available for CPS verification in Isabelle. One of them is to use Isabelle/UTP[21], an implementation of the Unifying Theories of Programming[22]. It presents a framework for embedding a wide range of logics into Isabelle/HOL, which can then be reasoned about. These are shallow embeddings, which attempt to make use of the available facilities in the host language to enhance efficiency as opposed to deep embeddings which implement the language from scratch. The approach to CPS verification outlined in [23] is to develop an appropriate logic within Isabelle/UTP, and it presents the hybrid relational calculus for this purpose. d$\mathcal{L}$ has also been implemented in Isabelle/UTP, as well as a wide variety of other logics; this testifies to the generality of the framework.

An alternative approach is presented in [24], which introduces a differential Hoare logic and a differential refinement calculus and proves that they are as applicable for verifying simple hybrid programs as d$\mathcal{L}$.

There has also been much work done towards analysis of ODEs in Isabelle, too. Most of it has been directed toward approximating numerical solutions, but theorems about their properties have also been developed to verify the correctness of the solutions[25], [26]. This includes ways of symbolically specifying an ODE and its solution. It is possible to symbolically compute the derivative of a function within Isabelle/HOL, which allows Isabelle to certify solutions even though it does not have the ability to produce them.

3 Preliminaries

3.1 Isabelle/ML

Standard ML (SML) is a statically typed, functional language. It is defined by a formal specification, given in [27]. Any implementation of the language must follow this specification, and may add additional features; Poly/ML is no exception, and it adds parallel programming facilities which are used to improve the performance of Isabelle.

SML has features that are common to modern functional languages, such as pattern matching, partial function application and algebraic data structures. It also features a structure system which aids in organizing large projects.

Isabelle/ML extends SML by introducing antiquotations, which provide a way of interacting with the Isabelle environment. Antiquotations are expressions of the form `@{name args}`, and they allow to refer to formal entities in a robust manner[20]. For example, a commonly used antiquotation is term, which produces an Isabelle expression that can then be manipulated and returned by an ML function to be used within Isabelle.

3.2 Lambda terms

Isabelle/Pure constitutes the meta-logic upon which other logics, such as Isabelle/HOL or Isabelle/ZF - which implements Zermelo-Fraenkel set theory, are founded. It is based on the typed λ-calculus[16, p. 4], in which λ-terms, henceforth referred to as terms, hold a central role.

In Pure, terms are represented as the following algebraic datatype:

```
datatype term =
    Const of string * typ |
    Free of string * typ |
    Var of indexname * typ |
    Bound of int |
    Abs of string * typ * term |
    $ of term * term
```

Where each applicable term is annotated with its respective type (typ). An explanation of what each term constructor represents can be found in table 3.1.

`Const`	Constant terms. They are not limited to values, as functions are also regarded as constant
`Free`	Free variables, which have not been bound by any enclosing abstraction.
`Var`	Schematic variables, used in theorems.
`Bound`	De-Brujin index of a variable bound by an abstraction. The index represents how many abstractions one has to skip to reach the binding one.
`Abs`	Abstraction binding a variable to a term.
`$`	Application of an abstraction to another term.

Table 3.1: Meaning of each term constructor

For example, the term `λx y. x+y` is parsed by Isabelle as

```
Abs ("x", "'a",
     Abs ("y", "'a",
       Const ("Groups.plus_class.plus",
              "'a ⇒ 'a ⇒ 'a")
        $ Bound 1
        $ Bound 0))
```

There are a few things to note. First, the two argument abstraction `λx y` is just syntactic sugar for `λx. λy`. Pre-defined functions are treated simply as constant terms, as evidenced by the `Const` constructor used for the plus function. Functions of multiple arguments are implemented through currying. This is why the plus function is applied twice, first to `Bound 1` (x), and then to `Bound 0` (y).

The inferred type for the expression is `'a ⇒ 'a ⇒ 'a`. `'a` is a type variable which can be instantiated to any type for which the operations are defined. In this case, the term will typecheck when `'a` is instantiated to any group. For example, it can be specified to be a function of real numbers by adding a type annotation: `λ(x::real) y.x+y`. This provides enough information for Isabelle to infer the type of the whole expression:

```
Abs ("x", "real",
    Abs ("y", "real",
            Const ("Groups.plus_class.plus",
                   "real ⇒ real ⇒ real")
                $ Bound 1
                $ Bound 0))
```

3.3 Isabelle representation of ODEs

Following [25], a system of ODEs is represented as a term which maps each parameter in the system to its derivative at a given point in time. Throughout the report the type synonym shown in figure 3.1 will be used for ODEs, where `'c` can be instantiated to a vector of any length. This representation only admits first-order differential equations, but any higher-order derivatives can be reduced to a first-order system by introducing additional variables.

```
type_synonym 'c ODE = "real ⇒ 'c ⇒ 'c"
```

Figure 3.1: Type synonym for systems of ODEs in Isabelle

For example, the ODE $\frac{dx^3}{d^3t} = 0$ can be represented as the first-order system in equation 3.1. This can then be directly translated to Isabelle, as shown in figure 3.2

$$x' = v, v' = a, a' = 0 \tag{3.1}$$

```
value "(λ t (x,v,a). (v,a,0)) :: (real × real × real) ODE"
```

Figure 3.2: Representation of equation 3.1 as an Isabelle term

The solution to a system of ODEs is a term of type $\mathbb{R}^n \Rightarrow \mathbb{R} \Rightarrow \mathbb{R}^n$. The order of the arguments has been swapped due to the convention of canonical argument order, in which the arguments that vary less are moved further to the left than those that vary more. This allows partial application on those which are invariant without having to alter their order. The arguments to a solution are initial values for the parameters and a value for time; it makes sense to assume that time will vary more often. The solution for 3.1 is shown in equation 3.2, which is translated to the Isabelle term in figure 3.3.

$$x' = x + tv + \frac{t^2 a}{2}, v' = v + ta, a' = a \tag{3.2}$$

```
value "(λ(x, v, a) t.
  (x + t * v + 1 / 2 * t² * a, v + t * a, a))::
  (real × real × real ⇒ real ⇒ real × real × real)"
```

Figure 3.3: Solution of equation 3.1 as an Isabelle term

3.4 Verifying solutions to ODEs

For Isabelle to be able to use the solution to a system of ODEs in a proof it needs to first ascertain that the given equation in fact solves the system. This is not trivial, as one might see by trying to verify a solution by hand. It requires differentiation and simplification, but thankfully Isabelle is capable of doing both. The predicate which indicates this is `solves_ode`, defined in the `Ordinary_Differential_Equations` AFP entry.

Its definition is shown in figure 3.4. From its type signature, it can be observed that its arguments are the solution to the ODE, the ODE itself, the domain in which it is defined, and its range. It is defined in terms of the predicate `has_vderiv_on`, which stands for “has vector derivative in domain”.

```
definition
  solves_ode :: "(real ⇒ 'a::real_normed_vector) ⇒ (real ⇒ 'a ⇒ 'a) ⇒ real set ⇒ 'a set ⇒ bool"
  (infix "(solves'_ode)" 50)
where
  "(y solves_ode f) T X ⟷ (y has_vderiv_on (λt. f t (y t))) T ∧ y ∈ T → X"
```

Figure 3.4: Definition of solves_ode

The ode_cert tactic provides a way of doing proofs which involve solves_ode. It uses derivative introduction rules, which allow the derivative of a function to be used in proof, together with simplification, to show that the derivative of the solution is equivalent to the original ODE.

4 Methods

4.1 Overview

The CAS that will be used for translation is the Wolfram Engine, which provides the backend for Mathematica and is openly available for development purposes. It will be referred to it as Mathematica throughout the report for simplicity.

First, the Isabelle term that represents the ODE to be solved is translated to an equivalent Mathematica term and is then wrapped in the DSolve function, which is Mathematica's differential equation solver. The interface with Mathematica is provided by the WolframScript command line utility; it is accessed by creating a bash process which executes the call to Mathematica and returns the result.

The solution produced is then converted to an abstract syntax tree (AST) via lexing and parsing. If there are no errors, the domain of the solution is retrieved, and the AST is then interpreted as an Isabelle term. Finally, the plugin combines the domain, the solution, and the original ODE, into a predicate which states that the solution solves the ODE. This is wrapped in a lemma and printed out for the user to paste into a proof session. This workflow is illustrated in figure 4.1, where each node represents data, and each arrow can be interpreted as a function of the data.

There are no legal, social, ethical or professional issues with this project. Isabelle is open-source software, and the Wolfram Engine is freely available for non-production use[28]. There are issues with software verification itself either, as it is strictly preferable to non-verified software due to increased safety.

4.2 Testing

A set of ODEs is used to test the functionality of the program. The objective of this test set is to provide a complete picture of how well the solver is performing; it should be capable of producing the correct response to each ODE if and only if it is fully featured. This is done by considering a wide range of ODEs with different kinds of solutions. Harris presents

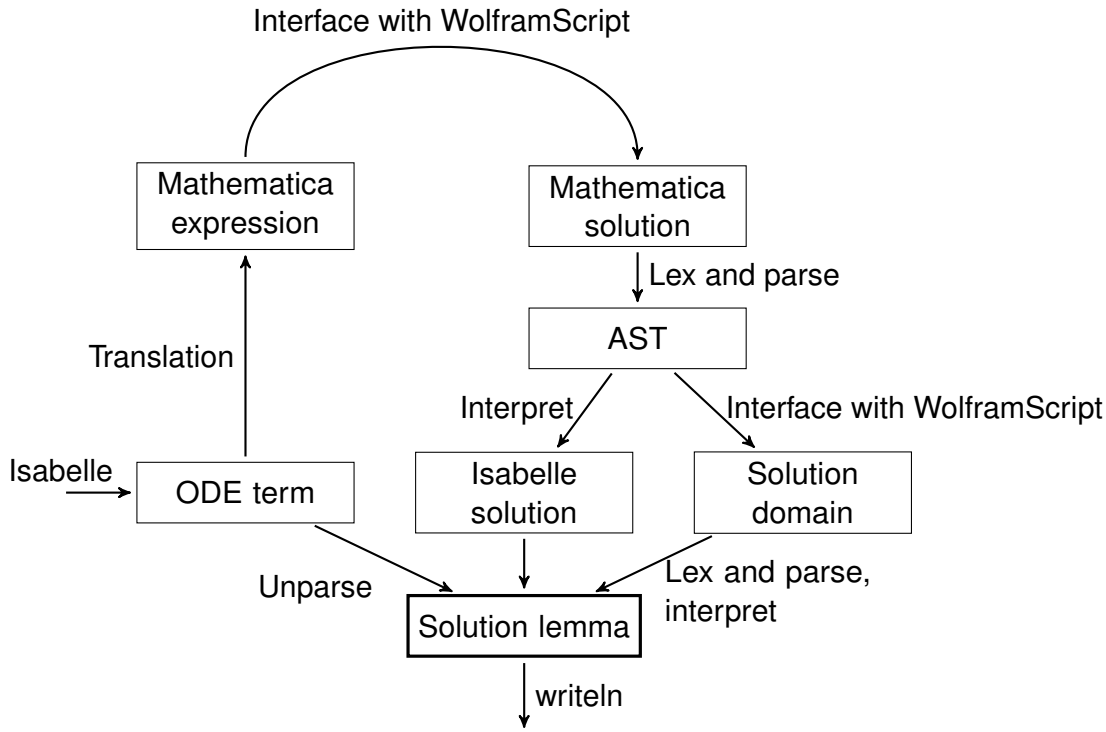

Figure 4.1: Plugin workflow

an exploration of different integral classes and their solvability by various CASs, together with a simple test set [29]; this provides a starting point for the one presented here. Unsolvable equations are included to test how failure is handled, and ODEs with partial solutions test domain retrieval. In addition, material is used from the robot verification case-study[12] to simulate modelling a real CPS.

A plugin that is capable of producing the correct response to every element of the test set would be well suited for real-world usage, as the equations test a wide array of non-trivial cases. However, a test set cannot provide a complete guarantee, as some edge cases could have been missed. The test set can be found in table 4.1.

Very simple ODE $x' = x$
Represents turning - simple trigonometric solution $x' = -y, y' = x$
Non-integer exponent $x' = \sqrt{t}$
Object in free fall $x' = v, v' = 9.81$
Object with fixed acceleration $a' = 0, v' = a, p' = v$
Simple system that includes t $x' = t + 2, y' = x + 3$
Example from [12, p.10]. d and p are two dimensional vectors indicating orientation and position respectively. $d^{\perp}$ is the orthogonal complement. $p' = sd, d' = -\omega d^{\perp}, s' = a, \omega' = \frac{a}{r}$
Error function - cannot be represented in Isabelle $x' = y, y' = e^{t^2}$
No symbolic solution $x' = \frac{\sin(x)}{\log(x)}$
Solution only in the positive reals $x' = \ln(t), y' = x$
Solution not continuous $x' = x^2$

Table 4.1: ODE test set

5 Implementation

To illustrate the workings of the implementation, the internal representation at each stage of the process will be shown for the following example system of ODEs:

$$\begin{aligned} x' &= y \\ y' &= -x \end{aligned} \tag{5.1}$$

In this equation the derivatives are given with respect to time. It has solution

$$\begin{aligned} x &= x_0 \cos(t) + y_0 \sin(t) \\ y &= y_0 \cos(t) - x_0 \sin(t) \end{aligned} \tag{5.2}$$

This system has been chosen with the purpose of illustrating all aspects of the solving process without hampering understanding by being too large. We begin with its representation in Isabelle:

```
abbreviation ode :: "(real × real) ODE" where
"ode ≡ λ t (x,y). (y,-x)"
```

Figure 5.1: ODE representation in Isabelle

5.1 Translation to Mathematica

The program given by Harris in [29] implements a part of this translation, and was used as a starting point for this implementation. It did, however, have issues which required it to be re-written completely.

The general approach for conversion is to first generate an alphabetically ordered list of variable names to be used in place of the ones in the Isabelle term. This is done because Mathematica orders its solutions in alphabetical order, and it makes converting them back into Isabelle much easier if they

do not have to be sorted after solving. In addition, it also prevents any name clashes with Mathematica functions. To do this, the program also collects a list of the original variable names, which are restored after the conversion.

Then, the right-hand side of the term is recursively converted to Mathematica expressions via pattern-matching, and the whole term is rewritten to fit Mathematica's description of a system of ODEs.

Finally, Mathematica produces a solution by using the DSolve function, which is capable of solving a wide array of differential equations.

$$\text{DSolve}[\{eqn_1, eqn_2, \dots\}, \{u_1, u_2, \dots\}, t] \tag{5.3}$$

The function, as used in the plugin, takes a list of equations, a list of variables being solved for and an independent variable.

It returns a list of rules, which work similarly to functions.

To represent system 5.1, the variable mapping $t \to a, x \to b, y \to c$ is applied. It is then translated to the following:

```
DSolve[
    {b'[a]==Minus[c[a]], c'[a]==b[a]},
    {b[a],c[a]},
    a
]
```

5.1.1 Interfacing with the Wolfram Engine

WolframScript is the command-line interface to the Wolfram Engine, which is provided by Wolfram as a way of integrating Mathematica into applications.

Isabelle can interface with external programs using `bash_output`, which has the type signature given below. This function creates a new bash process, which executes the argument string and returns its output, together with its return value. The WolframScript interface uses it as follows:

```
val bash_output: string -> string * int

fun mathematica_output(text : string) : string =
  fst (Isabelle_System.bash_output
      ("echo \"OutputForm @ Quiet[FullForm[" ^ text ^
              "]]\" |wolframscript -noprompt"))
```

This function outputs a Mathematica expression into wolframscript, adding

the following options to ensure that the output is as easy to parse as possible.

- Quiet - suppress any warnings.
- FullForm - remove any formatting and syntactic sugar, representing every function in the same uniform format, which makes parsing easier.
- OutputForm - force Mathematica to apply the specified formatting.
- noprompt - omit command prompt.

The Mathematica solution to 5.1 is

```
{{b[a] -> C[1] Cos[a] - C[2] Sin[a],
  c[a] -> C[2] Cos[a] + C[1] Sin[a]}}
```

which in FullForm is

```
List[List[
    Rule[b[a],
        Plus[Times[C[1], Cos[a]], Times[-1, C[2], Sin[a]]]],
    Rule[c[a],
        Plus[Times[C[2], Cos[a]], Times[C[1], Sin[a]]]]
]]
```

This solution is a function of time and constants `C[1]` and `C[2]`, which are the initial values for each of the variables in the system, indexed by the order in which they appear; i.e. `C[1]`$= x_0$ and `C[2]`$= y_0$.

5.2 Conversion to AST

Having obtained an output from Mathematica, the first step is to convert the string into an abstract syntax tree (AST). This makes it much easier to manipulate the expression, as it abstracts away from its string representation by encoding the important information in the tree structure, while getting rid of extraneous characters.

5.2.1 Lexical analysis

Lexical analysis, or lexing, is the first stage of analysing the input string. It consists of partitioning the input into tokens defined by regular expressions. The main reason for doing this is that it will improve the parsing code's readability and efficiency. The language for parsing is defined as a context-free grammar, which is more expressive but also more computationally

expensive than regular expressions. Therefore, by minimizing the size of the parser's input by doing some preliminary work the program will run faster.

The lexer is adapted from the one presented in [30, Chapter 9]. It defines a token datatype which has four constructors: `Id`, which stores identifiers, Int for integers, Real for real numbers and Punct for any punctuation and syntactical delimiters. The regular expressions for each are presented in appendix A.

Lexing 5.1.1 returns a list of tokens, of which the first few are:

```
[Id "List", Punct "[", Id "List", Punct "[", Id "Rule",
 Punct "[", Id "aa", Punct "[", Id "a", Punct "]",...]
```

5.2.2 Parsing

Parsing was implemented using the parsing combinators defined in the Isabelle/ML source. A viable alternative would have been to use the mllex and mlyacc parser generators, but there are incompatibilities between the mlton implementation of SML used to write them and Poly/ML.

The main idea behind parsing combinators is to combine very simple primitive parsers in various ways to represent complex behaviour. For example, parsing a Mathematica expression involves (recursively) parsing a list of expressions. The function for this combines the primitive `punct` parser, which consumes a specified `Punct` token, with the expression parser, which is defined recursively as follows:

```
(punct "[" |--
    (repeat (expression --| punct ","))
    @@@ single expression)
    --| punct "]")
```

Here, `|--` discards the left (resp. right for `--|`) element after parsing, which is used when the element discarded is a syntactic delimiter. `@@@` combines two lists of elements. It is used to combine the list produced by `repeat`, which is equivalent to the * operator in regular expressions, with the one produced by `single`, which parses the element and wraps it in a one-element list. This ensures that the parsed list is non-empty. The BNF grammars for the parser are presented in appendix A.

Parsing the token list into an AST yields the following:

```
Fun ("List",
```

```
[Fun ("List",
      [Fun ("Rule", [Fun ("b", [Id "a"]),
                     Fun ("Plus", …
       Fun ("Rule", [Fun ("c", [Id "a"]),
                     Fun ("Plus", …
```

5.3 Importing to Isabelle

5.3.1 Domain retrieval

Mathematica returns a list of solutions to the ODE, each one being a list of rules that map each variable to its corresponding differential. Rules are Mathematica's way of transforming expressions.

An issue with the solutions provided by the Wolfram Engine is that they may be partial functions, while the solution proof needs the domain stated explicitly and computing arbitrary domains is not trivial. This is resolved by obtaining the solution domains via a second query to the Mathematica.

This is done by first defining the following datatype, which stores a single Mathematica rule:

```
datatype rule = Rule of string * ParseMathematica.expr
```

I can then transform the AST storing the solutions into a `rule list list` without losing information. This data structure is a list of solutions, each of which is itself a list of rules. To retrieve the domain, the Mathematica `Domain` function is used.

Because equation 5.1 is total, the call to `Domain` returns t - it is defined at any value of t. Partial domains are represented as predicates on t, which are false when the function is not defined.

5.3.2 Translation to term

Converting the Mathematica AST back into an Isabelle term is similar to the inverse, the main difference being that the original variable names are restored. This is done by creating a translation table that maps the new variables to the original ones. Having this, the expression can be recursively translated through pattern matching.

Converting the AST produces the term shown in figure 5.2.

```
abbreviation ode_sol :: "(real × real) ⇒ real ⇒ (real × real)" where
"ode_sol ≡ λ (x,y) t. (((x * cos(t)) + (y * sin(t))),((y * cos(t)) + (-1 * x * sin(t))))"
```

Figure 5.2: ODE solution in Isabelle

5.4 Certifying the solution

The final part of solving the ODE is to generate a lemma that proves that our proposed solution is correct. This is done using the `ode_cert` tactic presented in Section 3.4. In order to certify the solution, the following proposition is constructed.

```
((solution (variables)) solves_ode ode) domain UNIV
```

The solution is partially applied to the inital values of the variables, to yield a function from time to variable values. These initial values are arbitrary, indicating that the solution is valid regardless of which are chosen. Equally, UNIV stands for the universal set, and indicates the set of possible initial values.

The domain D retrieved from Mathematica is given as a predicate. It is turned into a set via the set comprehension `{t. D}` — the set of all values of t which fulfil the predicate that defines the domain.

The solution to equation 5.1 is presented in figure 5.3, where the identifiers abbreviate their corresponding expression as presented in figures 5.1 and 5.2:

```
lemma "(ode_sol (x,y) solves_ode ode) T UNIV"
  by ode_cert
```

Figure 5.3: Verifying the ODE solution

This is accepted by Isabelle, as shown in figure 5.4. The identifiers prefixed by question marks (`?x,?y,?T`) are schematic variables, which can be replaced by any value and yield a true instantiation of the lemma.

```
theorem (ode_sol (?x, ?y) solves_ode ode) ?T UNIV
```

Figure 5.4: Isabelle accepting the lemma as true

6 Analysis

6.1 Evaluation

We use the test set presented in table 4.1 to verify that the implementation has been successful as follows:

1. Write the ODE as an Isabelle term. This is straightforward and should not introduce any errors.
2. Call the `Solve_ODE.solution_lemma` function on the term and copy the output - this is a string which holds a lemma that verifies the solution.
3. Paste the lemma into an Isabelle session that has the appropriate libraries loaded and check whether it is accepted by Isabelle.

The results are presented in table 6.1. They are overall satisfactory but there are some caveats. Firstly, `ode_cert` has issues with solutions that use non-integer powers. This is due to the fact that their derivative is defined using the logarithm, which has a domain restriction. The other issue arises with the equation that was taken from [12]. Wolfram fails to solve it, and it is not exactly clear why. Apart from that, all the other results are satisfactory, as they are either correctly verified or they return the correct error message, indicating that the solution produced cannot be represented in Isabelle. All test set terms are available from the `test_set.thy` file.

Overall, the test results indicate that the implementation has achieved its goals, since all errors arise from sources external to the plugin. Code quality cannot be similarly tested, but the code for the plugin follows the coding practices outlined for Isabelle source, and is written to be extensible and easily maintainable, so it is reasonable to conclude that this goal has been met as well.

The source code is very modular, and it can be extended to provide a general interface with Mathematica. The code for lexing and parsing is fully independent of the rest of the system, as is the translation from Isabelle to Mathematica; this is evidenced in how the domain retrieval is able to use the same functions as the solver with no adaptations.

$x' = x$	✓
$x' = -y, y' = x$	✓
$x' = \sqrt{t}$	ode_cert fails
$x' = v, v' = 9.81$	✓
$a' = 0, v' = a, p' = v$	✓
$x' = t + 2, y' = x + 3$	✓
$t' = 1, p' = sd, d' = -\omega d^{\perp}, s' = a, \omega' = \frac{a}{r}$	Wolfram fails to solve it
$x' = x^2 - t$	“Invalid function - BesselJ”
$x' = y, y' = e^{t^2}$	“Invalid function - Erfi”
$x' = \frac{\sin(x)}{\ln(x)}$	“Invalid function - InverseFunction”
$x' = \ln(t), y' = x$	✓
$x' = x^2$	✓

Table 6.1: Test results

6.2 Reflection

The most difficult part of this project was learning how to write code that integrates with the Isabelle system. Although some documentation is available in [20] and [31], much of the learning process involves reading the source code to see what functions are available and how they are used. For example, most of the parsing combinators in the Scan structure only have their type signature as documentation.

The code underwent a few minor revisions throughout the development process, but the general structure remained the same. Most notable is the inclusion of domain retrieval, which was not initally planned. This was only added after the verification of some of the ODEs in the test set failed due to them being partial.

Overall, the project was undertaken well. The result is functional and meets the objectives.

7 Conclusions

This report presents a plugin for the Isabelle proof assistant which produces symbolic solutions to systems of ODEs via an integration with the Mathematica CAS. The plugin source code is extensively documented and highly modular. A set of ODEs designed to test the functionality of the plugin is also presented. The performance of the solver on this set was satisfactory, with only minor issues in the verification of the solutions, none of which are due to design flaws in the plugin itself.

The main purpose of the plugin is to aid the verification of cyber-physical systems, since solving ODEs is often required but could not before be done within Isabelle and the solutions had to be provided by hand. There exist more specialized tools for CPS verification, but by using Isabelle we gain the benefit of generality.

Isabelle/ML is not very well documented, so this plugin can act as a small, self contained reference to how software may be written to integrate with the Isabelle environment. The emphasis on readability is also important for any future work that may build on this project, as expanding the integration with Mathematica or even adding other tools could add functionality that would improve the Isabelle user experience.

7.1 Further work

Even though the plugin successfully implements the solver, it has usability issues. Because it is implemented as an ML function, any development that makes use of it must import the relevant ML files locally, and obtain the solutions via a function call, from which the user has to copy the output into the proof script. An improvement would be to add integration with Isabelle/jEdit similar to what other Isabelle tools provide, which would make the tool more usable and intuitive.

This project presents a flexible way of interfacing Isabelle with Mathematica, which can be used in further work to exploit more of Mathematica's capabilities. This includes quantifier elimination.

Quantifier elimination is the process of removing existential or universal quantifiers from a formula. Quantified formulas can be difficult to use in

proofs, as they require all values of a variable to be checked (in the case of $\forall$) or to find exactly the correct value for a variable (in the case of $\exists$)[4, p.583]. Mathematica implements algorithms for quantifier elimination of real, boolean and complex formulas[32], which could be leveraged in Isabelle through an extension of my integration.

Currently the interface with Mathematica is limited to translating functions which are used for representing ODEs. A future project could endeavour to produce a complete translation tool that is capable of translating any Mathematica expression to Isabelle. This would require some changes to the structure of the current code for translation, which would have to be optimized for handling a large library of functions.

A Formal Languages for Lexing and Parsing

A.1 Token regular expressions

`identifier=` `[a-z][0-9, a-z]*`

`integer=` `[0-9]+`

`real=` `[0-9]+'.'[0-9]+`

`punct=` anything but `[a-z,0-9]`

A.2 BNF grammar for expressions

⟨*expression*⟩ ::= identifier | ⟨*number*⟩ | ⟨*function*⟩

⟨*number*⟩ ::= integer | real

⟨*function*⟩ ::= identifier '[' [⟨*expression*⟩ ',']* ⟨*expression*⟩ ']'

⟨*curryfun*⟩ ::= identifier ['[' [⟨*expression*⟩ ',']* ⟨*expression*⟩']']+

Bibliography

[1] National Science Foundation, *Cyber-physical systems*. [Online]. Available: https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503286 (visited on 20/01/2020).

[2] S. K. Khaitan and J. D. McCalley, 'Design techniques and applications of cyberphysical systems: A survey', *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, Jun. 2015, ISSN: 2373-7816. DOI: 10.1109/JSYST.2014.2322503.

[3] J. Filliâtre, 'Deductive software verification', *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 397, 20th Aug. 2011.

[4] A. Platzer, *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

[5] Isabelle team, *Isabelle overview*. [Online]. Available: http://isabelle.in.tum.de/overview.html (visited on 21/01/2020).

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood, 'SeL4: Formal verification of an OS kernel', in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP 09, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220, ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. [Online]. Available: https://doi.org/10.1145/1629575.1629596.

[7] (). MathWorks - computer algebra system, MathWorks, [Online]. Available: uk.mathworks.com/discovery/computer-algebra-system.html (visited on 05/05/2020).

[8] J. Woodcock, P. Larsen, J. Bicarregui and J. Fitzgerald, 'Formal methods: Practice and experience', *ACM Computing Surveys*, vol. 41, Oct. 2009. DOI: 10.1145/1592434.1592436.

[9] AlloyTools. (2019). Alloy home page, [Online]. Available: https://alloytools.org (visited on 10/04/2020).

[10] R. Alur, 'Formal verification of hybrid systems', in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT 11, Taipei, Taiwan: Association for Computing Machinery, 2011, pp. 273–278, ISBN: 9781450307147. DOI: 10.1145/2038642.2038685. [Online]. Available: https://doi.org/10.1145/2038642.2038685.

[11] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp and A. Platzer, 'KeYmaera X: An axiomatic tactical theorem prover for hybrid systems', in *CADE*, A. P. Felty and A. Middeldorp, Eds., ser. LNCS, vol. 9195, Springer, 2015, pp. 527–538. DOI: 10.1007/978-3-319-21401-6_36.

[12] S. Mitsch, K. Ghorbal, D. Vogelbacher and A. Platzer, 'Formal verification of obstacle avoidance and navigation of ground robots', *The International Journal of Robotics Research*, vol. 36, no. 12, pp. 1312–1340, 2017. DOI: 10.1177/0278364917733549. eprint: https://doi.org/10.1177/0278364917733549. [Online]. Available: https://doi.org/10.1177/0278364917733549.

[13] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, A. Schmidt, R. Gardner, S. Mitsch and A. Platzer, 'A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system', *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 6, pp. 717–741, 1st Nov. 2017, ISSN: 1433-2787. DOI: 10.1007/s10009-016-0434-1. [Online]. Available: https://doi.org/10.1007/s10009-016-0434-1.

[14] B. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen and A. Platzer, 'VeriPhy: Verified controller executables from verified cyber-physical system models', *SIGPLAN Not.*, vol. 53, no. 4, pp. 617–630, 2018, ISSN: 0362-1340. DOI: 10.1145/3296979.3192406. [Online]. Available: https://doi.org/10.1145/3296979.3192406.

[15] F. Tuong and B. Wolff, 'Deeply integrating C11 code support into Isabelle/PIDE', *Electronic Proceedings in Theoretical Computer Science*, vol. 310, pp. 13–28, Dec. 2019, ISSN: 2075-2180. DOI: 10.4204/eptcs.310.3. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.310.3.

[16] L. C. Paulson, 'The foundation of a generic theorem prover', University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-130, Mar. 1988. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-130.pdf.

[17] D. Matthews. (17th Mar. 2019). The Poly/ML implementation of standard ML, [Online]. Available: polyml.org (visited on 04/08/2020).

[18] L. C. Paulson. (5th Apr. 2020). Isabelle's logics: FOL and ZF, [Online]. Available: https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2020/doc/logics-ZF.pdf (visited on 23/04/2020).

[19] Isabelle team, *Archive of formal proofs*. [Online]. Available: https://www.isa-afp.org/about.html (visited on 30/01/2020).

[20] M. Wenzel. (9th Jun. 2019). The Isabelle/Isar implementation, [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2019/doc/implementation.pdf (visited on 13/11/2019).

[21] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock and F. Zeyda, 'Unifying semantic foundations for automated verification tools in Isabelle/UTP', *CoRR*, vol. abs/1905.05500, 2019. arXiv: 1905.05500. [Online]. Available: http://arxiv.org/abs/1905.05500.

[22] C. A. R. Hoare and H. Jifeng, *Unifying Theories of Programming*. Prentice Hall, 1998.

[23] S. Foster and J. Woodcock, 'Towards verification of cyber-physical systems with UTP and Isabelle/HOL', English, in *Concurrency, Security, and Puzzles*, vol. 10160, Lecture Notes in Computer Science, Jan. 2017, pp. 39–64, ISBN: 978-3-319-51045-3.

[24] S. Foster, J. Munive and G. Struth, 'Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/hol', English, in *18th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2020)*. 16th Dec. 2019.

[25] F. Immler and J. Hölzl, 'Numerical analysis of ordinary differential equations in Isabelle/HOL', in *Interactive Theorem Proving*, L. Beringer and A. Felty, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 377–392, ISBN: 978-3-642-32347-8.

[26] F. Immler, 'A verified ODE solver and the Lorenz attractor', *Journal of Automated Reasoning*, vol. 61, no. 1, pp. 73–111, 1st Jun. 2018, ISSN: 1573-0670. DOI: 10.1007/s10817-017-9448-y. [Online]. Available: https://doi.org/10.1007/s10817-017-9448-y.

[27] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML, revised*. MIT press.

[28] (). Free wolfram engine for developers - faq, Wolfram Research, Inc., [Online]. Available: https://www.wolfram.com/engine/faq/ (visited on 05/05/2020).

[29] P. Harris, 'Solving differential equations for the formal verification of hybrid systems in Isabelle/UTP', Bachelor's Thesis, University of York, 30th Apr. 2019.

[30] L. C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1996.

[31] C. Urban. (16th Jul. 2019). The Isabelle cookbook, [Online]. Available: http://talisker.inf.kcl.ac.uk/cgi-bin/repos.cgi/isabelle-cookbook/rawfile/tip/progtutorial.pdf (visited on 22/11/2019).

[32] *Some notes on internal implementation - Wolfram language*, Wolfram Research, Inc. [Online]. Available: https://reference.wolfram.com/language/tutorial/SomeNotesOnInternalImplementation.html.