CAPTURING ODE KNOWLEDGE FOR SCS

CAPTURING ORDINARY DIFFERENTIAL EQUATIONS KNOWLEDGE FOR SCS IN DRASIL

BY

DONG CHEN, M.Eng.

A REPORT

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF ENGINEERING

Master of Engineering (2022)	McMaster University
(Department of Computing and Software)	Hamilton, Ontario, Canada

TITLE:	Capturing Ordinary Differential Equations Knowledge for SCS in Drasil
AUTHOR:	Dong Chen M.Eng. in (Systems Engineering), Boston University, Massachusetts, USA
SUPERVISOR:	Spencer Smith and Jacques Carette
NUMBER OF PAGES:	xii, 23

Lay Abstract

A lay abstract of not more 150 words must be included explaining the key goals and contributions of the thesis in lay terms that is accessible to the general public.

Abstract

Abstract here (no more than 300 words)

Your Dedication

Optional second line

Acknowledgements

Acknowledgements go here.

Contents

List of Figures

List of Tables

Notation, Definitions, and Abbreviations

Notation

$A \leq B$	A is less than or equal to B

Definitions

Challenge With respect to video games, a challenge is a set of goals presented to the player that they are tasks with completing; challenges can test a variety of player skills, including accuracy, logical reasoning, and creative problem solving

Abbreviations

AI Artificial intelligence

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

Chapter 0

Software Automation

From the Industrial Revolution (1760-1840) to the mass production of automobiles that we have today, human beings never lack innovation to improve the process. In the Industrial Revolution, we start to use machines to replace human labour. Today, we have been building assembly lines and robots in the automobile industry to reach a scale of massive production. Hardware automation has been relatively successful in the past one hundred years, and they have been producing mass products for people at a relatively low cost. With the success story of automating hardware, could software be the next one? Nowadays, the software is used every day in our daily life. Most software still requires a human being to write them. Programmers usually write software in a specific language and produce other byproducts during development time. Whether in an enterprise or research intuition, manually creating software prone to errors, and it is not as efficient as a code generator. In the long term, a stable code generator usually beats programmers in performance. They will eventually bring the cost down because of the labour cost reduction. Perhaps this is why human beings consistently seek to automate work. History demonstrates that we successfully automate hardware. With fairly well-understood knowledge of software, creating a comprehensive system to produce software is not impossible. Can you imagine that programmers no longer programming in the future world? In the future world, code generators will generate software. There will be a role called ”code alchemist” who is responsible write the recipe for the code generator. The recipe will dictate what kind of software people want. In other words, the recipe is also a software requirement document that the code generator can understand. The recipe can exist in the form of a high-end programming language. Once the code generator receives the recipe, it will automatically produce software artifacts. The code generator exists in the form of a compiler. The described above is revolutionary if there is such a code generator, and the Drasil framework could be it.

Chapter 1

Introduction

The Drasil is a framework that generates software, including code, documentation, software requirement specification, user manual, axillary files, and so on. We call those artifacts software artifacts. By now, the Drasil framework targets generating software to overcome scientific problems. Recently, the Drasil team has been interested in expanding its knowledge to solve a high-order ordinary differential equation (ODE) through external libraries. There are two main reasons why we want to do that.

1. Scientists and researchers frequently use ODE as a research model in scientific problems, and this model describes the nature phenomenons. Building a research model in software is relatively common, and the software that the Drasil framework generates can solve scientific problems. Thus, expanding the Drasil framework's potential to solve all ODE would solve many scientific problems. Currently, the Drasil can only solve first-order ODEs.

2. Many external libraries are hard to write and embody much knowledge, so the Drasil team wants to re-use them instead of reproducing them. Among many external libraries, libraries that solve ODEs are probably the most important ones.

Another reason is that the Drasil team is interested in how the Drasil framework interacts with external libraries. Once the team understands how to interact between the Drasil framework and external libraries, they will start to add more external libraries. In this way, it would unlock the potential to allow the Drasil framework to solve more scientific problems than before.

However, the Drasil framework neither captures ODE knowledge nor solves high-order ordinary differential equations. The previous researcher researched to solve a first-order ODE, but it only covers a small area of the knowledge of ordinary differential equations. Adding high-order linear ODEs into the Drasil framework will expand the area where it has never reached before. Therefore, my research will incorporate high-order linear ODEs in a complex knowledge-based and generative environment that can link to externally provided libraries.

To solve a high-order linear ODE, we have to represent ODEs in the Drasil database. On the one hand, users can input an ODE as naturally as writing an ODE in mathematical expressions. On the other hand, they can display the ODE in conventional mathematical expressions. The data represented will preserve the relationship between each element in the equations. Then, we will analyze the commonality and variability of selected four external libraries. This analysis will lead us to know how external libraries solve ODEs, what their capabilities are, what options we have, and what interfaces look like. Last, we need to bridge the gap between the Drasil ODE data representation and external libraries. The Drasil ODE data representation can not directly communicate with external libraries. Each library has its standard in terms of solving ODEs. The existing gap requires a transformation from Drasil ODE data representation to a generic data form before solving ODE in each programming language. Finally, users can run software artifacts to get the numerical solution of the ODE.

Before conducting my research, the Drasil framework can solve explicit equations and numerically solve a first-order ODE. After my research, the Drasil framework will have full capability to solve a high-order linear ODE numerically. In addition, we will explore the possibility of solving a system of ODE numerically. We will introduce a new case study, the double pendulum. It contains an example that solves a system of high-order non-linear ODE.

Chapter 1 will cover how to represent the data of linear ODE in Drasil. Then, in Chapter 2, we will analyze external libraries. In Chapter 3, we will explore how to connect the Drasil ODE data representation with external libraries. Last, we will discuss a user's choice to solve ODE differently in the Drasil framework.

Chapter 2

ODE Data Represent

In the Drasil framework, there is a single data structure containing all the information for all products, and we call it System Information. The giant System Information collects a multitude of pieces of information; whenever we need it, we extract the information from the System Information. In previous research, we store all ordinary differential equations (ODEs) information in the System Information. However, that information existed in the form of plain text. In other words, we explicitly wrote ODEs in the text without any advanced data structure. Although this method maintains the relationship of ODEs, it restricts any transformation of ODEs. Therefore, the Drasil team decide to create a new data structure to store ODEs information. This chapter will describe how to store an ODE in the Drasil framework.

2.1 Matrix Form

In general, an equation contains a left-hand expression, a right-hand expression, and an equal sign. The left-hand and right-hand expressions connect by an equal sign.

A linear ODE also has its left-hand and right-hand sides. Each side has its unique shape. We can write a linear ODE in the shape of

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2.1.1}$$

On the left-hand side, A is an m * n matrix, and x is an n-vector. On the right-hand side, b is an m-vector. The A is commonly known as the coefficient matrix, b is the constant vector, and x is the unknown vector.

Given the following example 2.1.2,

$$y_t'' + (1 + K_d)y_t' + (20 + K_p)y_t = r_t K_p \tag{2.1.2}$$

y_t is the dependent variable, and K_d, K_p, and r_t are constant variables. We can write this equation as follows.

$$\begin{bmatrix} 1, & 1 + K_d, & 20 + K_p \end{bmatrix} \cdot \begin{bmatrix} y_t'' \\ y_t' \\ y_t \end{bmatrix} = \begin{bmatrix} r_t K_p \end{bmatrix} \tag{2.1.3}$$

The relationship between the matrix form 2.1.1 and the example 2.1.3 is not hard to find. Firstly, the coefficient matrix A is a 1 * 3 matrix that consists of 1, $1 + K_d$, ane $20 + K_p$. Secondly, the unknown vector is a 3 * 1 vector with y_t'', y_t', and y_t. Last, the constant vector is a 1 * 1 vector with $r_t K_p$. The matrix form 2.1.1 very well captures all the knowledge we need to present an ODE. Therefore, we decided to create a datatype called `DifferentialModel` to preserve ODEs information. The `DifferentialModel` has six records, and here is the representing code for

DifferentialModel.

```
data DifferentialModel = SystemOfLinearODEs {
        _indepVar :: UnitalChunk,
        _depVar :: ConstrConcept,
        _coefficients :: [[Expr]],
        _unknowns :: [Unknown],
        _dmConstants :: [Expr],
        _dmconc :: ConceptChunk
}
```

Previous to this research, UnitalChunk, ConstrConcept, Expr, and ConceptChunk already existed in Drasil. We created an `Unknown` type for this experiment. Their semantics will show up in Appendix A A.1

The `_indepVar` represents the independent variable in an ODE, and it is usually time. The `_depVar` represents the dependent variable in an ODE. The `_coefficients` is a list of lists `Expr`, and it represents the coefficient matrix in an ODE. The `_unknowns` is a list of `Unknown`, and each `Unknown` indicates an nth order of derivative of the dependent variable. The `_dmConstants` is a list of `Expr`, and it represents the constant vector. Last, the `_dmconc` contains metadata of this model. To represent example 2.1.3 in `DifferentialModel`, `_indepVar` is time, `_depVar` is y_t, `_coefficients` is the 1 *3 matrix, `_unknowns` is the 3 * 1 vector, `_dmConstants` is the 1 * 1 vector, and `_dmconc` is `ConceptChunk` that describes what this model is.

Currently, the `DifferentialModel` only captures the knowledge of ODEs with one dependent variable, and it is a special case of the family of linear ODEs. Studying this special case will help the Drasil team better understand how to capture the knowledge of all ODEs and eventually lead to solving a system of linear ODE with multiple dependent variables.

2.2 Input Language

There are many reasons why we want to provide an input language for users to input ODE equations. One major reason is that it could be over complicated for users to input a single ODE in a matrix form, so introducing an input language could simplify inputting a single ODE. What will this input language look like? The example 2.1.2 represents a linear second-order ODE, and how to represent a linear nth-order ODE? Based on Paul's Online Notes (1), we can write all linear ODEs in the shape of

$$a_n(t)y^n(t) + a_{n-1}(t)y^{n-1}(t) + \cdots + a_1(t)y'(t) + a_0(t)y(t) = g(t) \tag{2.2.1}$$

On the left-hand side of the linear equation 2.2.1, the expression is a collection of terms. Each term consists of a coefficient and a derivative of the dependent variable. With ideas of term, coefficient, and derivative, we create new data types to mimic the mathematical expression of a linear ODE. The following is the detail of the code for new data types and operators.

```
type Unknown = Integer
data Term = T{
        _coeff :: Expr,
        _unk :: Unknown
}
type LHS = [Term]

($^^) :: ConstrConcept -> Integer -> Unknown
($^^) _ unk' = unk'

($*) :: Expr -> Unknown -> Term
($*) = T

($+) :: [Term] -> Term -> LHS
($+) xs x  = xs ++ [x]
```

For new types, the `LHS`, the short name for the left-hand side, is a list of `Term`. Each `Term` has an `Expr` and `Unknown`. For new operators, they are inspired by the linear equation 2.2.1. The `$^^` operator connects a dummy dependent variable and the order of its derivative. We call it a dummy dependent variable because it is a placeholder to increase users' readability. The `$*` operator creates a term by combining a coefficient matrix and an unknown variable. Last, the `$+` operator connects all terms. Let's write pseudo code for the example matrix form 2.1.2 in the newly introduced input language. The full detail of the input language for the PD_Controller example will show up in Appendix A.

```
lhs = [1 $* (y_t $^^ 2)] --
        $+ (1 + K_d) $* (y_t $^^ 1)
        $+ (20 + K_p) $* (y_t $^^ 0)
rhs = r_t K_p
```

2.3 Two Constructors

There are two constructors to create a `DifferentialModel`, which they use for different design purposes. The first constructor is `makeASystemDE`. A user can set the coefficient matrix, unknown vector, and constant vector by explicitly giving `[[Expr]]`, `[Unknown]`, and `[Expr]`. There will be several guards to check whether inputs are well-formed.

1. The coefficient matrix and constant vector dimension need to match. The `_coefficients` is an m * n matrix, and `_dmConstants` is an m vector. This guard makes sure they have the same m dimension. If an error says "Length of coefficients matrix should equal to the length of the constant vector", it means `_coefficients` and `_dmConstants` has different m dimension, violating mathematical rules.

2. The dimension of each row in the coefficient matrix and unknown vector need to match. The `_coefficients` use a list of lists to represent an m * n matrix. It means each list in `_coefficients` will have the same length n, and `_unknowns` is an n-vector. Therefore, the length of each row in the `_coefficients` should equal the length of `_unknowns`. If an error says, "The length of each row vector in coefficients need to equal to the length of unknowns vector.", it means `_coefficients`and `_unknowns`

violate mathematical rules.

3. The order of the unknown vector needs to be descending due to design decisions. We have no control over what users will give to us, and there are infinite ways to represent a linear equation in the matrix form 2.1.1. We strictly ask users to input the unknown vector descending, so we can maintain the shape of a normal form of linear ODE 2.2.1. This design choice will simplify the implementation for solving a linear ODE numerically in Chapter 3. If an error says, "The order of giving unknowns needs to be descending.", it means the order of unknown vector is not descending.

The following pseudo-code shows how to directly set the example 2.1.2's coefficient matrix, unknown vector, and constant vector. The full detail of how to directly set the coefficient matrix, unknown vector, and constant vector for the PD_Contoller example will show up in the Appendix A.

```
coefficent = [[1, 1 + K_d, 20 + K_p]]
unknowns   = [2, 1, 0]
constants  = [r_t K_p]
```

The second constructor is called `makeASingleDE`. This constructor uses the input language 2.2 to simplify the input of a single ODE. In `makeASingleDE`, we create the coefficient matrix, unknown vector, and constant vector based on inputs the input language restricts. In other words, users no longer set the data by directly giving values. The `DifferentialModel` will generate all data for the coefficient matrix, unknown vector, and constant vector accordingly. The constructor first creates a descending unknown vector base on the highest number of its derivatives. For example, if the highest order of its derivative on the left-hand side of the equation is 2, we will generate a list that contains 2, 1 and 0. Then, we will create the coefficient matrix by finding its related coefficient based on the descending order of the unknown vector. One advantage of this design choice is that it increases the readability while inputting a single ODE. The second advantage is that the `DifferentialModel` will no longer require users to input the unknown vector in descending order. Any order of the unknown vector will be acceptable because we will generate the data of `DifferentialModel`.

2.4 Display Matrix

After a `DifferentialModel` obtains ODE information, we want to display them in the software requirements specification (SRS). Previously, we mentioned the Drasil framework able to generate software artifacts, and SRS is a part of them. This section will discuss two ways to display ODEs in the SRS.

1. We can display ODEs in a matrix form. The matrix form 2.1.3 demonstrates how the ODE will appear in a matrix form in the SRS. In the `DifferentialModel`, the coefficient matrix is a list of lists expression, the unknown vector is a list of integers, and the constant vector is a list of expressions. It should be fairly straightforward for the Drasil printer to display them by printing each part sequentially.

2. We also can display ODEs in a shape of a linear equation. The example 2.1.2 demonstrates how the ODE will show up in the shape of a linear equation in the SRS. Display an ODE in a linear equation is a special case. When there is only a single ODE, it would be over complicated to display it in a matrix form. This is the same reason we want to create an input language to manage the input of a single ODE better.

In the future, the Drasil team wants to explore more variability in displaying an ODE. One topic highlighted in the discussion is showing an ODE in a canonical form. However, many mathematicians have different opinions on a canonical form, and the name of canonical form has been used differently, such as normal form or standard form. More research on this part would help us better understand the knowledge of ODE.

Chapter 3

Your Chapter Title

This is a sample chapter

If you need to use quotes, type it "like this".

3.1 Referencing

These are some sample references to GAMYGDALA (3) from the `references.bib` file and state effects of cognition (2) from the `references_another.bib` file. These references are not in the same .bib file.

3.2 Figures

This is a single image figure (Figure 3.1):

This is a multi-image figure with a top (Figure 3.2a) and bottom (Figure 3.2b) aligned subfigures:

Figure 3.1: This is a single figure environment

3.3 Tables

Here is a sample table (Table 3.1):

A	$\longleftrightarrow$	B
C	$\longleftrightarrow$	D

Table 3.1: A sample table

3.3.1 Long Tables

A sample long table is shown in Appendix B.

3.4 Equations

Here is a sample equation (Equation 3.4.1):

$$y = mx + b \tag{3.4.1}$$

(a) Figure 1

(b) Figure 2

Figure 3.2: A Multi-Figure Environment

Chapter 4

Conclusion

Every thesis also needs a concluding chapter

Appendix A

Your Appendix

This appendix provides detailed explanations of various parts of DifferentialModel.

Type	**Semantics**
`UnitalChunk`	concepts with quantities that must have a unit definition.
`ConstrConcept`	conceptual symbolic quantities with Constraints and maybe a reasonable value.
`Expr`	a type encode mathematical expression.
`ConceptChunk`	a concept that contains an idea, a definition, and an associated domain of knowledge
`Unknown`	synonym of Integer

Table A.1: Type use in DifferentialModel

A.1 Type in DifferentialModel

```
-- $K_d$ is qdDerivGain
-- $y_t$ is opProcessVariable
-- $K_p$ is qdPropGain
-- $r_t$ is qdSetPointTD
lhs = [exactDbl 1 $* (opProcessVariable $^^ 2)]
        $+ (exactDbl 1 `addRe` sy qdDerivGain $* (opProcessVariable $^^ 1))
        $+ (exactDbl 20 `addRe` sy qdPropGain $* (opProcessVariable $^^ 0))
rhs = sy qdSetPointTD `mulRe` sy qdPropGain
```

Appendix B

Long Tables

This appendix demonstrates the use of a long table that spans multiple pages.

Col A	Col B	Col C	Col D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

Continued on the next page

Continued from previous page

Col A	**Col B**	**Col C**	**Col D**
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

Bibliography

[1] Dawkins, P. Paul's Online Notes linear differential equations. `https://tutorial.math.lamar.edu/Classes/DE/Definitions.aspx`. Accessed: 2022-08-01.

[2] Hudlicka, E. This time with feeling: Integrated model of trait and state effects on cognition and behavior. *Applied Artificial Intelligence 16*, 7-8 (2002), 611–641.

[3] Popescu, A., Broekens, J., and van Someren, M. GAMYGDALA: An emotion engine for games. *IEEE Transactions on Affective Computing 5*, 1 (2014), 32–44.