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, 19

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 1

Introduction

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.

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

Intro

2.1 Matrix Form

In general, a equation contains a left hand expression, a right hand expression, and a equal sign. The left hand expression and the right hand expression connect with each other by equal sign. We can write a linger ODE in a shape of

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

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

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

To take an example, in above equation, y_t is the dependent variable, and K_d, K_p, and r_t are constant variables. We can write this equation in a form of (matrix form 2.1.1)

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

Base on this analysis, we decide to create a datatype **called** `DifferentialModel` to capture the ODE knowledge. The `DifferentialModel` consists of the independent variable, the dependent variable, the coefficient matrix, the unknown vector, the constant vector, and its meta data.

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

Currently, the `DifferentialModel` only capture knowledge of ODEs with one dependent variable. This 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 the ODEs, and eventually lead to solve a system of linear ODE with multiple dependent variables.

Variable	Semantics
`_indepVar`	represent the independent variable in ODE, and it is usually time. UnitalChunk is a Quantity(concept with a symbolic representation) with a Unit
`_depVar`	represent the dependent variable in ODE
`_coefficients`	represent the coefficient of the ODE, it is A in (matrix form 2.1.1). A list of lists represent a matrix. `Expr` is a type encode mathematical expression.
`_unknowns`	represent the unknown vector, it is x in (matrix form 2.1.1). The `Unknown` is synonym of Integer, it indicates nth order of derivative of the dependent variable
`_dmConstants`	represent the constant vector, it is b in (matrix form 2.1.1). It is a list of `Expr`
`_dmconc`	ConceptChunk records a concept !!citation in here!!

Table 2.1: A sample table

2.2 Input Language

We introduced an input language to simplify the input a single ODE, because it could be over complicated for users to input a single ODE in to a matrix form. The structure of the input language will mimic the mathematical expression of linear differential equation that was bases on GAMYGDALA (2)

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

The left hand side of equations is a collection of terms. A term consist a coefficient and a derivative of the dependent variable. The following is the detail of code for new type introduced

```
type Unknown = Integer
data Term = T{
```

Variable	Semantics
LHS	reflects to left hand side, and consist of a sequence of Term.
Term	contains a Unknown, and a Expr.
Unknown	is a integer, indicate nth derivative of the dependent variable.

Table 2.2: A sample table

```
        _coeff :: Expr,
        _unk :: Unknown
}
type LHS = [Term]
```

The following are new operators introduced

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

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

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

To take (matrix form 2.1.2) as an example. Replacing symbols with equivalent variable in the Drasil framework.

we can write them as the following

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

Operator	Semantics
`$^^`	take a `ConstrConcept` and `Integer` to form a `Unknown`. The `ConstrConcept` is the dependent variable and `Integer` is the order of nth derivative. Eg, depVar `$^^` d, means nth derivative of depVar.
`$*`	take a `Expr` and `Unknown` to form a `Term`. The `Expr` is the coefficient and `Unknown` is the nth derivative.
`$+`	take a `[Term]` and `Term` to form a new `[Term]` by appending `Term` to `[Term]`.

Table 2.3: A sample table

Variable	Equivalent in Drasil
K_d	qdDerivGain
y_t	opProcessVariable
K_p	qdPropGain
r_t	qdSetPointTD

Table 2.4: A sample table

lhs represents the left hand side and rhs represents the right hand side. lhs is a type of LHS, and LHS is synonym for `[Term]`. rhs is just a `Expr`. In line 1, the whole syntax create a single list of `Term`. In side of this single `Term`, everything before the `$*` is the coefficient, and everything after the `$*` is the `Unknown`. The coefficient is `1 + Kd`, and the `Unknown` is the first derivative of y_t. In line 2, everything after the `$+` is a `Term`. By using `$+`, we append a `Term` into a list of `Term`.

2.3 Two Constructors

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 wether inputs is well-formed.

1. matrix and constant vector: `_coefficients` is a m * n matrix and `_dmConstants` is a m vector. This guard make sure they have the same m dimension. If there is error says "Length of coefficients matrix should equal to the length of the constant vector", it means `_coefficients` and `_dmConstants` has different m dimension, and it violates mathematical rules.

2. each row in matrix and unknown vector: `_coefficients` use list of list to represent a m * n matrix. It means each list in `_coefficients` will have the same length n, and `_unknowns` is n vector. Therefore, the length of each row in the `_coefficients` should be equal to the length of `_unknowns`. The following code will create a math expression to indicate wether inputs for `_coefficients` and `_unknowns` are valid.

```
isCoeffsMatchUnknowns :: [[Expr]] -> [Unknown] -> Bool
isCoeffsMatchUnknowns [] _ = error "Coefficients matrix can not be
   empty"
isCoeffsMatchUnknowns _ [] = error "Unknowns column vector can not
   be empty"
isCoeffsMatchUnknowns coeffs unks = foldr (\ x -> (&&) (length x ==
   length unks)) True coeffs
```

If there is 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 unknown vector: this guard is designed for simplifying implementation. We have no control on what users give to us. There are infinite ways to represent a linear equation in the form of **Ax** = **b**, so we strictly ask users put the order of the unknown vector descending. This design choice will help to simply the later transformation from **Ax** = **b** to **X**' = **AX** + **c** (link

here). If there is error says ”The order of giving unknowns need to be descending”, it means the order of unknown vector is not descending.

The following is the example how to directly set the coefficient matrix, unknown vector, and constant vector

```
coeffs = [[exactDbl 1, recip_ (sy tauW)]]
unknowns = [1, 0]
constants = [recip_ (sy tauW) `mulRe` sy tempC]
```

In `makeASystemDE`, it requires users to input the unknown descending.

The second constructor is called `makeASingleDE`. This constructor uses the input language to simplify the input of a single ODE. The following is the example how to use input language to create a `DifferentialModel`.

reference ??

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

In `makeASingleDE`, users don't necessary to input the unknown vector in descending order. Any order will be fine, because we will sort out the unknown vector base on its derivative and build the coefficient matrix accordingly. Once users enter the ODE equations by using the input language, there are several steps to generate its data represent.

First, we create a descending unknown vector base on the highest order of the derivatives. For example, the highest order in the lhs is 2, then we will generate 2, 1 and 0 derivative respectively.

```
unks = createAllUnknowns(findHighestOrder lhs ^. unk) depVar''

findHighestOrder :: LHS -> Term
findHighestOrder = foldr1 (\x y -> if x ^. unk >= y ^. unk then x
    else y)

createAllUnknowns :: Unknown -> ConstrConcept -> [Unknown]
createAllUnknowns highestUnk depv
| highestUnk  == 0  = [highestUnk]
| otherwise = highestUnk : createAllUnknowns (highestUnk - 1) depv
```

Second, we will create the coefficient matrix base on the descending unknown vector. For each derivative, we search it coefficient base on the order of derivative.

```
coeffs = [createCoefficients lhs unks]

createCoefficients :: LHS -> [Unknown] -> [Expr]
createCoefficients [] _ = error "Left hand side is an empty list"
createCoefficients _ [] = []
createCoefficients lhs (x:xs) = genCoefficient (findCoefficient x
    lhs) : createCoefficients lhs xs
```

It would be tedious to require users to input a matrix form for a single ODE. Therefore, we create an input language to make it easier to input such equations.

2.4 Display Matrix

After users enter ODE equations, we display them in SRS. There are two different ways to display equations, matrix and linear equations.

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 (2) from the `references.bib` file and state effects of cognition (1) 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

Your appendix goes here.

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] 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.

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