



January 2020

Automation And Visualization Of Program Correctness For Automatically Generating Code

Jason Michael Hicks

[How does access to this work benefit you? Let us know!](#)

Follow this and additional works at: <https://commons.und.edu/theses>

Recommended Citation

Hicks, Jason Michael, "Automation And Visualization Of Program Correctness For Automatically Generating Code" (2020). *Theses and Dissertations*. 3376.
<https://commons.und.edu/theses/3376>

This Thesis is brought to you for free and open access by the Theses, Dissertations, and Senior Projects at UND Scholarly Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UND Scholarly Commons. For more information, please contact und.commons@library.und.edu.

AUTOMATION AND VISUALIZATION OF PROGRAM CORRECTNESS FOR
AUTOMATICALLY GENERATING CODE

by

Jason Michael Hicks

Doctor of Philosophy, Chemistry, University of North Dakota, 2018
Master of Science, Chemistry, University of North Dakota, 2013
Bachelor of Science, Chemistry, University of North Dakota, 2010
Associate of Arts, Northland Community and Technical College, 2008

A Thesis

Submitted to the Graduate Faculty

of the

University of North Dakota

in partial fulfillment of the requirements

for the degree of

Master of Science

Grand Forks, North Dakota

December

2020

Copyright 2020 Jason Hicks

Name: Jason Hicks

Degree: Master of Science

This document, submitted in partial fulfillment of the requirements for the degree from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done and is hereby approved.

DocuSigned by:
Emanuel S. Grant
10A2B20E6TAE418...
Emanuel S. Grant, Chairperson

DocuSigned by:
Hassan Reza
7DBBFCC1BD6F4BE...
Hassan Reza

DocuSigned by:
Wen-Chen Hu
EE71D886B1C04D0...
Wen-Chen Hu

This document is being submitted by the appointed advisory committee as having met all the requirements of the School of Graduate Studies at the University of North Dakota and is hereby approved.

DocuSigned by:
Chris Nelson
2E0A7088C735403...
Chris Nelson
Dean of the School of Graduate Studies

12/10/2020

Date

PERMISSION

Title	Automation and Visualization of Program Correctness for Automatically Generating Code
Department	School of Electrical Engineering and Computer Science
Degree	Master of Science

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work or, in his absence, by the chairperson of the department or the dean of the School of Graduate Studies. It is understood that any copying or publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use which may be made of any material in my thesis.

Jason M. Hicks
9/10/2020

Contents

List of Figures	vii
List of Tables	ix
ACKNOWLEDGEMENTS	x
ABSTRACT	xii
1 INTRODUCTION	1
1.1 Problem Statement	1
1.2 Research Objectives and Plan	2
1.3 NASA Relevance	4
1.4 Scope and Expected Outcome	6
1.5 Structure of Thesis	7
2 BACKGROUND AND RELATED WORK	9
2.1 Theoretical Background	9
2.1.1 Unified Modeling Language	9
2.1.2 Formal Methods	10
2.1.3 Object Constraint Language	11
2.1.4 Description of AUTOBAYES	11
2.1.5 Gaussian Distribution	14
2.2 AUTOBAYES Applications	16
2.3 Related Work	18
3 METHODOLOGY	22
3.1 Introduction	22
3.2 Description of Methods	22
3.2.1 Code Generator Input and Output Model Development	23
3.2.2 Code Generator Constraints Definition and Formal Specification	24
3.2.3 Code Generator Model Analysis	24
4 CASE STUDY: AUTOBAYES CODE GENERATOR SYSTEM	26
4.1 Introduction	26
4.2 AUTOBAYES Input and Output Model Development	27
4.3 AUTOBAYES Constraints Definition and Formal Specification	29
4.4 AUTOBAYES Model Analysis With the USE Tool	31

5	RESULTS AND DISCUSSION	32
5.1	Introduction	32
5.2	Results	33
5.2.1	UML Class Diagrams	33
5.2.2	Constraints	43
5.2.3	USE Analysis	45
5.3	Discussion	50
6	CONCLUSIONS AND FUTURE WORK	53
6.1	Conclusions	53
6.2	Future Work	54
7	FUNDING ACKNOWLEDGEMENT	56
A	APPENDICES	57
A.1	AUTOBAYES Input Files	57
A.2	AUTOBAYES Output C++ Code	61
A.3	List of Constraints	112
A.4	Input File for USE	115
A.5	User Interface Screens and Output From USE	129
	REFERENCES	137

List of Figures

2.1	AUTOBAYES system architecture [1].	13
2.2	Left image: Hyperspectral image cube (MODIS). Right image: Clustering result for hyperspectral data and 5 classes as produced by AUTOBAYES [1].	17
3.1	Diagrammatic representation of approach to demonstrate program correctness.	23
4.1	Diagrammatic representation of approach to demonstrate program correctness applied to AUTOBAYES.	27
4.2	Examples of classes representing potential input into AUTOBAYES.	28
4.3	The Gaussian Model Class of the AUTOBAYES output code.	29
5.1	Input CD for AUTOBAYES when a Normal distribution of data is assumed and thus the Gaussian equation is used.	33
5.2	Output CD.	34
5.3	Output CD - Matrices.	35
5.4	Output CD - Normal Distribution and Transformations.	36
5.5	Output CD - Gaussian and Retval.	36
5.6	Output CD - Discrete EM.	37
5.7	Output CD - Initialization Phase.	38
5.8	Output CD - Hidden Variable Extraction.	38
5.9	Output CD - Convergence Phase.	39
5.10	Output CD - EM Loop.	40
5.11	Output CD - Current To Old.	41
5.12	Output CD - Maximization Step.	41
5.13	Output CD - Expectation Step.	42
5.14	The USE Class Invariant View.	46
5.15	The USE class diagram.	47
5.16	Iteration 1, "Dry run", output.	48
5.17	Iteration 2 output after applying the the Pragma class fix.	48
5.18	Iteration 3 output after fixing all the object counts in the configuration.	48
5.19	Iteration 4 output.	49
5.20	Iteration 5 output.	49
5.21	Iteration 6 output.	49
5.22	Iteration 7 output.	50
A.1	The simplest input file that assumes a normal distribution of data.	57
A.2	A slight modification to the input file in Figure A.1 in which a square-normal transformation was used.	58

A.3	A slight modification to the input file in Figure A.1 in which a log-normal transformation was used.	58
A.4	An input file in which a basic clustering example is given with a mixture of Gaussians.	59
A.5	An input file in which a more complex clustering example is given with a multivariate mixture of Gaussians.	60
A.6	The input file used as an example in [1] designed to be used with the Fisher Iris flower multivariate data set.	61
A.7	The USE tool user interface after loading my input file.	129
A.8	The initial USE Create Configuration and Validator Readout.	130
A.9	The USE Model Validator Configuration - Basic Types and Options tab.	130
A.10	The USE Model Validator Configuration - Classes and Associations tab.	131
A.11	The USE Model Validator Configuration - Invariants tab.	132
A.12	The USE Model Validator Configuration - Classes and Associations tab with fix for the Pragma Class error.	133
A.13	The USE Model Validator Configuration - Classes and Associations tab with fix for each of the Classes that can have a multiplicity of 0.	134
A.14	The USE Model Validator Configuration - Classes and Associations tab with fix for the Goal Class error.	135
A.15	The USE Model Validator Configuration - Classes and Associations tab with fix for the Mean Class error.	135
A.16	The USE Model Validator Configuration - Classes and Associations tab with fix for the Gaussian Class error.	136
A.17	The USE Model Validator Configuration - Classes and Associations tab with fix for the GaussianModel Class error.	136

List of Tables

2.1	ADAPTED TABLE FROM [1] PRESENTING DIFFERENT DISTRIBUTIONS FOR MIXTURE MODELS OF AUTOBAYES. REMARKS: (1) AUTOBAYES HAS TO BE CALLED WITH -PRAGMA SCHEMA CONTROL ARBITRARY INIT VALUES=TRUE TO OBTAIN ITERATIVE SOLUTION. (2) PATCHED VERSION OF AUTOBAYES NECESSARY. (3) SOLUTION REQUIRES A CUSTOMIZED SCHEMA.	14
5.1	A FEW OF THE CONSTRAINTS DEVELOPED FOR THE INPUT, OUTPUT, AND THE RELATIONSHIP BETWEEN THEM.	44
A.1	CONSTRAINTS DEVELOPED FOR THE INPUT, OUTPUT, AND THE RELATIONSHIP BETWEEN THEM.	112
A.2	CONSTRAINTS DEVELOPED FOR THE OUTPUT.	113
A.3	CONSTRAINTS DEVELOPED FOR THE RELATIONSHIP BETWEEN THEM.	114

Acknowledgments

First, I would like to thank my advisor, Prof. Emanuel Grant for his continual guidance in my education and research. He has always been a delight to work under. I have had the privilege of taking multiple classes from him, and it can clearly be seen that he genuinely cares for his students. I have thoroughly enjoyed each time I have had a meeting with Dr. Grant, he is always personable, kind, and willing to help. I especially wanted to thank him for connecting me with this NASA-relevant work and for aiding in the successful application for the summer NASA graduate research fellowship two years in a row. I consider myself very fortunate to have had the privilege of being advised by him and look forward to staying in contact.

I would also like to thank Prof. Hassan Reza and Prof. Wen-Chen Hu for all of their help and for agreeing to serve on my committee. I have had the privilege of taking courses from both Dr. Reza and Dr. Hu during my time here, and I thoroughly enjoyed them all. They have given me so much of their time and guidance, my time here would have been nowhere near as fruitful and enjoyable without their counsel. Dr. Reza was the first person I talked to at the computer science department and he played a large role in getting me set up and connected. I always enjoyed our discussions. Concerning Dr. Hu, I had the pleasure of taking one of my first computer science

classes from him while I was finishing up my Computational Chemistry PhD, and he played a huge role in sparking and strengthening my interest in pursuing a masters in computer science. I owe him a great deal as well.

I would like to thank all of my fellow graduate students! I will not soon forget all "fun" all night coding, studying, and writing sessions (more coffee anyone)! I would also like to thank my other friends and family who have so instrumental to my life. My mom and dad deserve an extraordinary amount of thanks for all of their continual support and prayers.

I would especially like to thank my wonderful wife Erica Hicks for all of her love and encouragement throughout my stay here. She has so lovingly put up with all the late nights I have put in finishing this thesis. There have been so many times that her loving encouragement has given me energy and drive to push beyond anything I thought I could do. My world would be so much darker without her lovely smile and selfless support to help me through. I am so incredibly fortunate to be married to my best friend. It has certainly been a great adventure together, and now with two little ones, Jase (nearly 3) and Ellie (3 months), I am excited to continue on to this next of life together!

Last, but certainly not least, I would like to thank God for all of his guidance, love, presence and wisdom. He has been there from the beginning pushing me to step outside my comfort zone and giving me the wisdom, strength and motivation I need to make it through. I am deeply grateful for all of He has done and is still doing for me. There were so many times throughout my graduate work, when things were not going well, that He filled me with and indescribable peace through His Holy Spirit or miraculously came through and helped me in ways I did not even think possible. He is always drawing me closer to Him and I really have no way to properly describe how incredibly awesome that is.

Abstract

Program synthesis systems can be highly advantageous in that users can automatically generate code to fit a wide variety of applications from high-level specifications without needing any low-level programming skills or knowledge of which type of data structures and algorithms should be used. NASA has developed and uses two of these systems, AUTOFILTER and AUTOBAYES. Though much is gained in terms of time and cost efficiency in the use of these systems, they suffer from an issue that is inherent in all code generator systems, the verifiability of the correctness of the generated code against the input specifications. Many times, this verification process can take just as long, if not longer than manually developing and testing the code would have been. Because of this, much work has been done by NASA and others to develop methods for automatic certification that can be produced along with the program and are easy to use. However, there is still more work to be done in this area, especially in the area of automatic visual verification (e.g., by using UML diagrams to provide visual aid in the verification of the generated code). Work has been done by Grant et al. in collaboration with NASA to develop a rigorous approach to system correctness verification that uses domain-specific graphical meta-models of the expected input/output systems with identified constraints on the input/output and their relationships. Though this approach has been applied to AUTOFILTER, it has yet to be applied to other domains. In this work, Grant's approach is extended to the data analysis domain by being applied to AUTOBAYES. A model of the input specification for AUTOBAYES was obtained for the case in which a normal distribution of

data is assumed. This model, derived from the AUTOBAYES input files, the n-dimensional Gaussian equation, and allowed priors, is a UML class diagram (CD). Similarly, a UML CD model of the AUTOBAYES program output was derived. These CD's were then used to develop 30 constraints on the input, the output, and the relationship between them. These constraints were then transformed into the OCL formal specification language and analyzed with the USE tool, along with the derived comprehensive CD (i.e., a combination of the input CD, output CD, and the relationships between each other). These models and constraints were used to successfully check that all of the developed constraints were satisfied with the model representing AUTOBAYES. Unfortunately, a configuration for a full validation with USE was not obtained, after several iterations, due to project time restrictions. However, the results obtained adequately demonstrate that this method can be extended to the domain of AUTOBAYES. This work was motivated both due to its relevance to NASA in the chosen case study of AUTOBAYES as well to show that Grant's approach can be extended to other domains beyond AUTOFILTER.

1 INTRODUCTION

1.1 Problem Statement

Since the work of Alonzo Church in 1957 [2], and the idea of an automatic programmer, first explored in the 1960s [3], there have been many great developments in the area of program synthesis over the years [4–7], with a surge of recent advancements in which Artificial Intelligence and Machine Learning have been used [6, 8–12]. In program synthesis, executable code is generated from high-level specifications. This approach of generating tailored software for a specific domain from parameterized templates and schemas and/or existing libraries of program components is one of several approaches and can save considerable time and money for developers [13, 14]. Two program synthesis systems, developed by NASA researchers, are used for state estimation, i.e., the AUTOFILTER system [14–18], and used for data analysis, i.e., the AUTOBAYES [1, 19–25] system. Like many program synthesis systems, AUTOFILTER and AUTOBAYES have the advantage of being fully automatic, easy to use, quick, and requires no low-level programming skills. Therefore, there is no need for the user to decide on what algorithms or data structures to use, or how to call all the necessary library functions.

Though algorithms for program synthesis have continued to improve, the practical use of program synthesis systems in many domains are limited. This is due in part to the fundamental issue of the lack of a testing environment to ensure the generated output code correctly implements the input specification. This is because they are usually complex artifacts that make use of advanced software engineering techniques. Furthermore, the way program synthesis systems are designed and used requires that they can correctly implement output from an extensive assortment of potentially unforeseen inputs. Therefore, it can be exceedingly difficult to check the relationship between their input and output.

1.2 Research Objectives and Plan

In the work presented in this thesis, a technique developed by Grant et al. [26, 27] is used to check the input/output relationship of NASA's AUTOBAYES system. Grant's approach was developed with NASA researchers at the NASA AMES Research Center and uses domain-specific graphical meta-models of the expected input/output systems with identified constraints on the input/output and their relationships. This allows for the rigorous analysis, in the form of mathematical expressions, of these constraints against specific instances of input/output.

Another advantage of this approach is that there is no need for regression testing. Code generators will be modified and expanded to solve new problems in the problem domain. With the method used in this work, there is no need to refer to old test data, run the system through the same tests, and hope to get the same results. This is known as regression testing and it can be tedious and challenging. With this work, if the input matches the constraints on the input side, the output

matches the constraints on the output side, and the constraints on the relationships between them match as well, then we know the code generating system is working. If something violates those constraints, then we know the modification to the system broke the code generator. This is because while in traditional testing, test cases check only a single example, in this method the input/output constraints are defined at the domain modeling level. Therefore, they are valid for all instances generated by the program synthesis system.

This method of checking constraints is lightweight and goes beyond traditional testing methods yet does not involve formal verification [26,27]. Grant's approach was successfully applied to AUTOFILTER but has yet to be applied to AUTOBAYES. Though NASA researchers have worked toward the verification of program synthesis systems, termed certifiable synthesis [14, 16, 28–59], the benefit of doing this work is that Grant's rigorous analysis may provide verification of program correctness beyond other known testing strategies.

Moreover, additional work must be done to determine the suitability of this approach in other problem domains, beyond that of AUTOFILTER, where program synthesis systems are used. Therefore, this verification of program correctness strategy is applied to AUTOBAYES in this work to demonstrate its applicability to other domains such as for safety-critical systems [60,61], which is especially relevant for NASA, the space industry, and aviation (e.g., the Boeing 737 MAX Maneuvering Characteristics Augmentation System (MCAS)).

Lastly, this work is meant to showcase the effectiveness of formal methods in software engineering and testing. Formal methods can be used to make the ambiguous, informal object-oriented semantics precise. These methods are mathematically rigorous techniques that are used in the specification, development and verification of software. The correct use of formal methods can contribute greatly to the reliability and robustness of a system. This can be accomplished through

the use of mathematical analysis of a formal specification written in a formal language [62].

1.3 NASA Relevance

A large part of the motivation for this work stems from the relevance to NASA. The specific areas that this work relates to the published strategies, plans, and technological taxonomy are presented in this section. This includes both the previously published 2015 NASA Technology Roadmaps document [63] along with the recently published NASA's Technology Taxonomy TX11 in Software, Modeling, Simulation, and Information Processing [64], NASA's Strategic Technology Investment Plan's Advanced Information Systems [65], and NASA's 2018 Strategic Plan's Strategic Object 4.3: Assure Safety and Mission Success [66].

This work aligns with NASA's Technology Roadmaps TA11 in Modeling, Simulation, Information Technology, and Processing [63] and with NASA's Technology Taxonomy TX11 in Software, Modeling, Simulation, and Information Processing [64]. In the area of computing, a verification procedure could aid in the trust in AUTOBAYES generated flight software to support autonomous data triage at the point of data collection and aid in software development capabilities. In the area of modeling, this work could help develop trusted autonomous, integrated, and interoperable approaches for models and model development. It would increase productivity and manage risk by improving autonomy and integration in modeling for NASA's future missions. In the area of information processing it could aid in the develop software frameworks and toolsets that efficiently and reliably manage greatly increased volume, variety, and velocity of data across the science, engineering, and mission data lifecycle. It could also help increase system and crew

autonomy through advanced software [63]. In the area of Software Development, Engineering, and Integrity, a verification procedure could aid in the trust in AUTOBAYES generated flight software to support autonomous data triage at the point of data collection and aid in software development capabilities. AUTOBAYES also fits right into the area of information processing [64].

This work will also align with NASA's Strategic Technology Investment Plan's Advanced Information Systems [65]. It can aid in NASA's Critical flight computing technologies by increasing autonomy for onboard operations. AUTOBAYES could be trusted to generate code for on-board processing of larger volumes of data. It could also support work requiring Big Data processing and advanced analytics. Verification would fall under the safety and mission success (SMS) programs which protects "the health and safety of the NASA workforce and improve the likelihood that NASA's programs, projects, and operations are completed safely and successfully" [65].

Lastly, our work aligns with NASA's 2018 Strategic Plan's Strategic Object 4.3: Assure Safety and Mission Success [66]. This work is highly applicable to safety critical systems, which falls well into this strategic objective. NASA states that "Objective Overview SMS programs include programs that provide technical excellence, mission assurance, and technical authority" [66]. This work has the potential of meeting each of those criteria. Furthermore, our work could help assure that directives and requirements are appropriately implemented, and a way to aid in the performance of independent technical analysis of safety and mission critical software products. Our work could help provide independent assessments of the mission critical generated software products. It would also relate to one of the key indications to support SMS strategies for success, i.e., "the ability to independently verify and validate critical software safety and mission assurance Capabilities" [66]. NASA states elsewhere in the strategic objective that "SMS programs are charged with understanding and assuring that the Agency mitigates, to an acceptable level, all

safety, health, and technical risks to NASA missions” [66]. Our work would relate to how NASA accomplishes this by evaluating software aspects to identify hazards, including the impacts of new requirements and departures from existing requirements [66].

1.4 Scope and Expected Outcome

The scope of this work involved a few areas. The primary deliverable was to give a proof of concept that the method used in this work, developed by Grant et al. [26,27], could be extended to other problem domains that involved code generator systems. The code generator AUTOBAYES was chosen for that purpose, being that it is a program synthesis system for the statistical data analysis domain rather than the Kalman Filter domain for AUTOFILTER. Furthermore, AUTOBAYES was selected since it was also developed by NASA, thus being a natural extension of previous work.

Within AUTOBOYES, we are looking at one specific example, the case in which a normal distribution of data is assumed. While there are many statistical models that AUTOBAYES can be used with, applying Grant’s method to allow for all possible statistical models would be highly time consuming, unnecessary for the proof of concept the work presented in this thesis is after, and thus out of the scope of this work. Similarly, only one pragma was tested and the code was always generated for use with the OCTAVE environment, which can be seen in each the code listing in the Appendices, rather than for the MATLABTM environment. These steps were also done, to limit the scope of this work to focusing on a proof concept rather than an exhaustive application.

This work also involved the use of the Unified Modeling Language (UML) [67], but we limited its use to UML class diagrams (CD). The formal specification language Object Constraint

Language (OCL) [67] was chosen for this work rather than another formal specification language (e.g., Z notation) due to three reasons: (i) OCL was developed to work UML, (ii) I have personal experience working with OCL, and (iii) I have experience working with the USE tool [68] which is designed for OCL.

The specific deliverables needed for the primary deliverable listed above, are as follows: (i) a CD modeling the possible input given to AUTOBAYES, (ii) a CD modeling the possible output code produced by AUTOBAYES, (iii) several identified constraints on the input, (iv) several identified constraints on the output, (v) several identified constraints on the relationship between the input and the output (n.b., again, only a subset of constraints were necessary for the goal for this work, obtaining all possible constraints would be out of scope), (vi) transforming the textual description on the constraints to precise mathematical representation (i.e., in OCL for this work), and (vii) an analysis with the USE tool to show the process of identifying and correcting any deficiencies in the CD's and/or constraints.

From my initial investigation, the expected outcome of this work was that Grant's method would be extensible to AUTOBAYES. This would then show that that method is, in fact, extensible to other problem domains.

1.5 Structure of Thesis

This thesis continues with Chapter 2 giving the background of this work, starting with the theoretical background of UML, formal methods, and OCL, a brief description of the AUTOBAYES program synthesis system, and an introduction to Gaussian or normal distributions, followed by

several highlights of AUTOBAYES applications, and it finished off with a survey of publications related to the work presented in this thesis. In Chapter 3, the methodology used in this work is described when applied to a general code generator. Next, this method applied to a case study will be presented in Chapter 4, where the verification of the correctness of automatically generated code from a NASA-developed program synthesis system, AUTOBAYES, was conducted. The results and discussion are then given in Chapter 5. Next, the conclusions and future work are given in Chapter 6 followed by Chapter 7 the funding source of this project is recognized. Lastly, an Appendix is given, followed by the references for this work.

2 BACKGROUND AND RELATED WORK

2.1 Theoretical Background

There are several areas in the work presented in this thesis that the reader may not be familiar with or need a refresher in. This section is meant to give a brief refresher or a working knowledge of these areas with the intention of giving the reader the tools they need to understand the content of this thesis. This chapter gives an introduction to (i) the Unified Modeling Language (UML), (ii) formal methods, (iii) Object Constraint Language (OCL), (iv) AUTOBAYES, and (v) Gaussian or normal distributions. Next this chapter familiarized the reader with the various applications that AUTOBAYES has been used for. Lastly, a collection of published work related to the work presented in this thesis is summarized.

2.1.1 Unified Modeling Language

The work presented in this thesis used what is known as the Unified Modeling Language (UML) [67]. UML is a collection of notations and models used in software engineering to model

software designs and specifications. It provide a standard way to visualize the design of a system. Originally conceived for object-oriented (OO) systems, UML represents systems in terms of objects and methods. UML was adopted by the object management group (OMG) in 1997 and is currently managed by them [67].

UML has many types of diagrams, but they can be grouped into two categories of diagrams, structure diagrams and behavior diagrams. it is worth mentioning that another, well known category, interaction diagrams, are actually a subset of behavior diagrams. The most well-known model of UML is a member of the structure diagrams, the class diagram (CD). A CD is a diagram that relates the classes or entities in the specification [69]. CDs are used extensive in the work presented in this paper.

2.1.2 Formal Methods

Formal methods can be a powerful tool. They are specification and verification methods and have formal (i.e., mathematical) semantics, must be unambiguous, and facilitate proofs of correctness. Though formal methods are based on mathematics, it does not require in-depth mathematical understanding and some of the work is even done in an informal way to reduce complexity. Though formal methods have been in use since the late 1970s they still see limited use. Globally, they see a lot more use in Europe than the United States, but their use is growing. Some examples of formal methods include deduction verification, model checking and testing. There are many different formal method languages, e.g., Z, OCL, and VDM. In the work presented in the work presented in this thesis, Object Constraint Language (OCL) [67] is used.

2.1.3 Object Constraint Language

Another important part of this work uses the formal specification language, Object Constraint Language (OCL), which is part of the UML standard [67]. OCL was designed as a constraint language meant to be easy for nonmathematicians to understand and use yet still maintain mathematical precision. OCL was developed specifically with the expression of constraints on UML object models (e.g., CDs) in mind, since UML, though very helpful, is not enough when high levels of precision is required due to its ambiguous nature. OCL also introduces language constructs for dealing with collections of objects, for using association paths to navigate from one object to another, and for expressing queries on object types [69]. In the work presented in this thesis, OCL is used to express constraints on CDs related with AUTOBAYES input and output, their relationships to each other, and when combined with the relevant classes and associations, it was used in an analysis of AUTOBAYES program correctness with the USE tool.

2.1.4 Description of AUTOBAYES

The AUTOBAYES program synthesis system automatically generates customized algorithms for the statistical data analysis domain. It constructs efficient executable code from high-level declarative specifications, which can be seen below in Figure 2.1, to solve parameter estimation problems in this domain. Data analysis is an important task whenever useful information needs to be obtained from raw data.

AUTOBAYES takes an input of a parameterized statistical model (i.e., a probability distribu-

tion which specifies the properties for each problem variable and its dependencies) and a goal that is a probability term involving parameters and the associated input data. It then outputs optimized, fully-documented C/C++ code for the specified data analysis application which computes values for those parameters that maximize the probability term. In this way, AUTOBAYES can be readily used in the context of describing clustering, change point detection, and parameter estimation type statistical analysis problems. The output code from AUTOBAYES can also be dynamically linked to MATLABTM and Octave environments [1, 19–25].

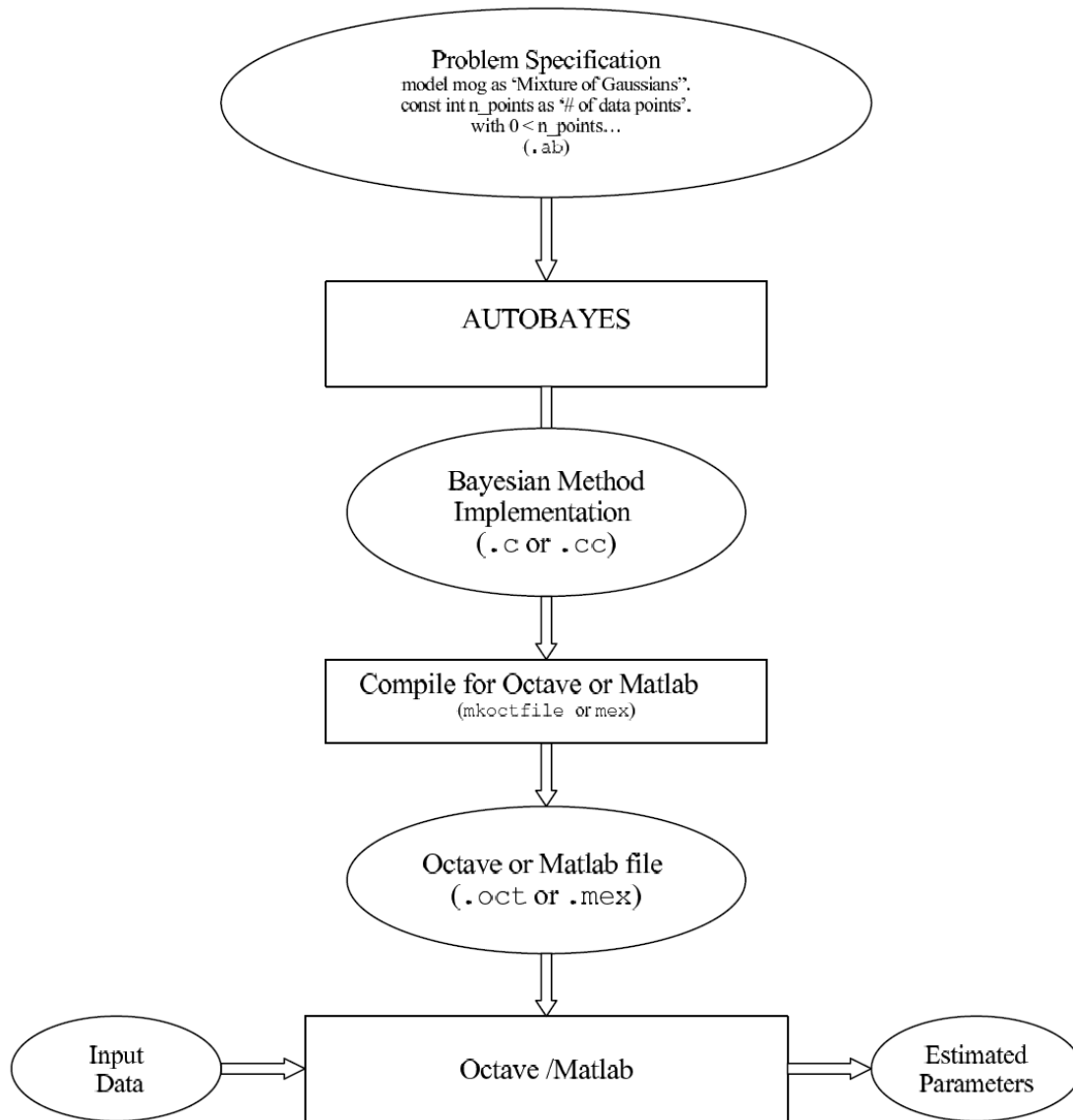


Figure 2.1. AUTOBAYES system architecture [1].

AUTOBAYES has a wide variety of allowed input equations compared to that of AUTOFILTER, which uses a static or dynamic Kalman filter input equation. All available statistical models that AUTOBAYES can be used with are given below in in Table 2.1. It is also worth noting that AUTOBAYES allows for mixtures of those distributions to be used. For some distributions displayed in Table 2.1, closed form solutions are found by AUTOBAYES, and denoted with a "Y", for

others a closed form solution is not found, denoted with an "N".

Table 2.1. ADAPTED TABLE FROM [1] PRESENTING DIFFERENT DISTRIBUTIONS FOR MIXTURE MODELS OF AUTOBAYES. REMARKS: (1) AUTOBAYES HAS TO BE CALLED WITH -PRAGMA SCHEMA CONTROL ARBITRARY INIT VALUES=TRUE TO OBTAIN ITERATIVE SOLUTION. (2) PATCHED VERSION OF AUTOBAYES NECESSARY. (3) SOLUTION REQUIRES A CUSTOMIZED SCHEMA.

Name	Notation	Closed Form	Remarks
Bernoulli	$x \sim \text{bernoulli}(p)$	Y	
Beta	$x \sim \text{beta}(\alpha, \beta)$	N	1
Binomial	$x \sim \text{binomial}(n, p)$	Y	2
Cauchy	$x \sim \text{cauchy}(x, y)$	N	1
Exponential	$x \sim \text{exp}(\lambda)$	Y	
Gamma	$x \sim \text{gamma}(k, \theta)$	Y	k known
Gamma	$x \sim \text{gamma}(k, \theta)$	N	1
Gauss	$x \sim \text{gauss}(\mu, \sigma^2)$	Y	
Poisson	$x \sim \text{poisson}(\lambda)$	Y	
vonMises	$x \sim \text{vonmises}(\mu, k)$	Y	3
Weibull	$x \sim \text{weibull}(\alpha, \beta)$	N	1

2.1.5 Gaussian Distribution

Due to the many possible statistical models that AUTOBAYES can be invoked upon, and because AUTOBAYES allows for mixtures of those distributions to be used, a full model describing all possible input would be needlessly time consuming. This is because, the scope of this study just requires a proof of concept in the checking the extensibility of Grant's method to another problem domain. Therefore, the most commonly used statistical model, which assumes a Gaussian or normal distribution of the data was modeled. When developing the input CD, the form of the equation used needed to be considered. The consideration included the 1D Gaussian through the n-

dimensional Gaussian. Therefore this sub section gives the Gaussian equations from 1D Gaussian through its n -dimensional form.

A normal distribution or Gaussian-like distribution has seen a great deal of use in data analysis, probability theory, statistics, physical sciences, and humanities. It is a type of continuous probability distribution for a real-valued random variable. A normal distribution typically has two parameters, the mean, represented by μ , and the standard deviation, represented by σ . It is worth noting that σ^2 is called the variance of the distribution. If a random variable has a Gaussian distribution, it is said that it is normally distributed.

The 1D Gaussian equation is

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \quad (2.1)$$

For the n -dimensional Gaussian equation, it is typically given in matrix form. For an n -dimensional $x = (x_1, x_2, \dots, x_n)$, let $x \sim N_n(\mu, \Sigma)$ where

$$\Sigma = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^2 \end{bmatrix} \quad (2.2)$$

only has diagonal nonzero elements. Then $\det(\Sigma) = \sigma_1^2 \sigma_2^2 \dots \sigma_n^2$ and the matrix form of the n -dimensional Gaussian equation is given by

$$f(x|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp\left\{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right\} \quad (2.3)$$

However, in order to represent the case of an n -dimensional Gaussian, the non-matrix form was also helpful. Thus, the n -dimensional Gaussian equation was converted to the non-matrix form, given as

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{(2\pi)^n \sigma_1^2 \sigma_2^2 \cdots \sigma_n^2}} \exp \left\{ -\frac{1}{2} \left(\frac{(x_1 - \mu_1)^2}{2\sigma_1^2} + \frac{(x_2 - \mu_2)^2}{2\sigma_2^2} + \cdots + \frac{(x_n - \mu_n)^2}{2\sigma_n^2} \right) \right\} \quad (2.4)$$

It is worth noting here that these equations were analyzed to aid in the construction of the AUTOBAYES input CD.

2.2 AUTOBAYES Applications

AUTOBAYES has enjoyed a great deal of successful applications at NASA. Several of its NASA-relevant applications are now briefly described to give a better understanding of the AUTOBAYES system's applicability, capabilities, and importance. First, it has been used for data analysis on large software simulations, e.g., analyzing abort and re-entry scenarios for Orion. It has also seen use in this way for small-satellite guidance, navigation, and control systems [1].

Secondly, AUTOBAYES has been used for data analysis for air traffic control data, where it was used in a study which it took actual aircraft trajectories and performed data mining on those trajectories [1].

Thirdly, AUTOBAYES has been used in the application of shape analysis of planetary nebulae. Here, statistical data analysis models that estimate the center and elliptical extent and ori-

entation of the nebula were required to automatically analyze this data. AUTOBAYES successfully filled that role and was able to provide estimates for the center and extent of a nebula [1].

Fourthly, AUTOBAYES was used for clustering for Sloan Digital Galaxy Survey. From a description of an ensemble approach to building what is known as Mercer Kernels with prior information, AUTOBAYES was used to estimate the parameters for the kernels, an efficient customized variate of the EM algorithm, and automatically generate and typeset the mathematical derivation [1].

Fifthly, AUTOBAYES has been used for hyperspectral clustering of earth science data. More specifically, it was used to take data blocks called hyperspectral cubes obtained from earth-observing satellites (e.g., MODIS), and develop a simple multivariate mixture model to cluster the data into most probably class assignments for each pixel [1]. This can be seen below in Figure 2.2.

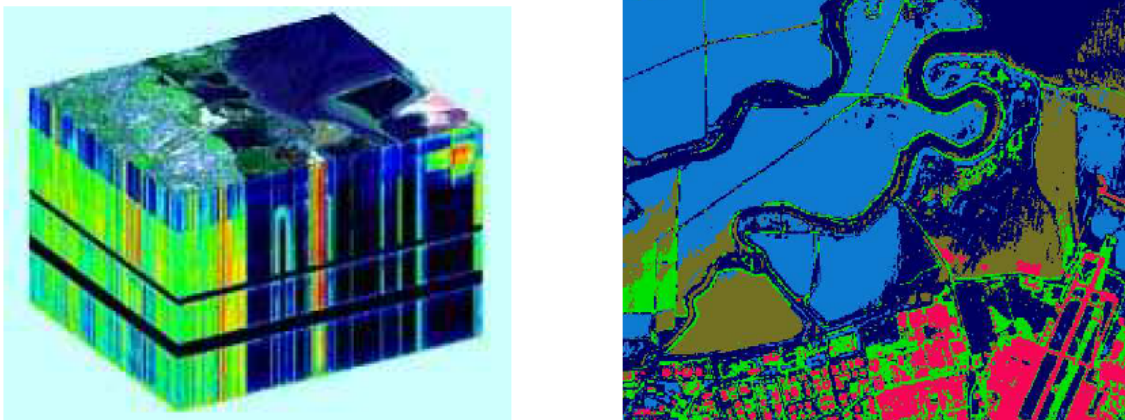


Figure 2.2. Left image: Hyperspectral image cube (MODIS). Right image: Clustering result for hyperspectral data and 5 classes as produced by AUTOBAYES [1].

Sixthly, AUTOBAYES was used in the application of clustering and mapping of geospatial data. In this example application, census data, which is typically highly multivariate (e.g., for each household, many variables are present, like age, income, size of household, etc.), was considered and related to the ZIP code. In a similar fashion as the previous example, AUTOBAYES was used to develop a simple multivariate clustering model, thereby allowing the data to be far more easily processed and even visualized [1].

Lastly, AUTOBAYES has been used in the context of detection of gamma-ray spikes. It was shown that a simple AUTOBAYES model can be used to detect and isolate intense gamma-ray burst events. If the inter-arrival time of photons is assumed to be exponentially distributed, a detector for a switchpoint can be readily specified in the specification language input used by AUTOBAYES to generate a program that has been shown to successfully isolate recognized bursts from the rest of the data [1].

2.3 Related Work

As mentioned in the introduction, NASA researchers have worked to develop means in which to certify the correctness of the code generated from their program synthesis systems. This work is termed as certifiable synthesis [14, 16, 28–59]. In this subsection, a few key papers will be reviewed that are most relevant to our work presented here.

In Grant’s approach [27], the role of modeling is emphasized to bring out what is to be checked during the constraint checking of the input specification, the output code, and the relationship between them. They used uses domain-specific

models of the expected input, the output, and the derived constraints between them. The domain-specific models used were created through the integration of an informal modeling notation (i.e., UML) with a formal specification language (i.e., OCL). It was shown that this technique of constraint-checking can provide a high degree of confidence in the correctness of the outputted code and the use of the code generator system it is applied to. Grant's lightweight verification method is a type of product-oriented verification. In product-oriented verification, the result of the program synthesis system is verified rather than the system itself. There has been some other work on product-oriented certification.

One of these product-oriented certification techniques was published in a paper by Whalen et al. in 2002 [28], a type of product-oriented certification is used to check AUTOBAYES for simple safety property violations (e.g., safe array bounds, or absence of division by zero). The presented approach generates the verification conditions to be proven by a theorem prover. This is accomplished through using a set of rules (i.e. a safety policy) which are obtained by encoding the safety properties.

Also in 2002, a similar product-oriented certification approach, applied to AUTOFILTER, was presented by Rosu et al. [14, 16]. In these papers, the authors used term rewriting to check AUTOFILTER's functional properties. It is from the ideas of proof-carrying code (PCC) [70] that the product-oriented approach is derived from. In PCC, which allows for the verification of properties using a formal proof, a compiler is augmented with certificates of partial correctness of the object code generated. Related to this is an approach called run-time result-checking [71]. In this method, rather than checking the correctness of the software system itself, the correctness of a particular run is checked during runtime.

In later work published in 2004 [37], researchers from the NASA Ames Research Center

used formal certification, which involves the use of mathematical proofs to show formally that certain properties of a given program is certified to be correct. Though these proofs are typically not of much use to the average engineer, it was shown to be possible to use the information contained in them to produce an easier to use textual justification of correctness. NASA researchers described an approach to generate textual explanations from automatically generated formal mathematical proofs of program safety. This was done in the context of ensuring the proofs are in compliance with an explicit safety policy that can be varied as application varies. These researchers described a tool which implements this strategy to certify automatically generated code from their AUTOFILTER and AUTOBAYES program synthesis systems [37].

Lastly, in a paper published in 2006 [50], NASA researchers described a generic post-generation annotation inference algorithm that bypasses some of the problems inherent in certifiable code generation. Typically, code generators for realistic application domains have been difficult to directly verify in practice. In the approach of certifiable code generation, fully automated program proofs of various safety properties can be obtained from the generator by extending it to not just generate the program, but also generate logical annotations (i.e., pre- and postconditions and loop invariants). In practice, however, this is can be challenging to implement and maintain because of the annotations are cross-cutting concerns at the object-level in the generated code and on the meta-level in the generator. Another added complication is that the certifiable code generation approach requires access to the generator sources. The NASA researchers were able to circumvent these problems by exploiting the highly idiomatic nature of the output of the code generator, thus patterns could be used to describe all code constructs that required annotations. Though the algorithm used by the NASA researchers is generic, it was shown to work well on the patterns that are specific to the idioms of the code generator they studied and to the specific safety property shown.

Their algorithm is based on a pattern matcher and a graph traversal. The pattern matcher is used to identify instances of the idioms and build a property specific abstracted control flow graph. The algorithm's graph traversal follows the paths from the use nodes backwards to all the corresponding definitions and annotates the statements along those paths. This approach was illustrated by being successfully applied to NASA's AUTOFILTER and AUTOBAYES by automatically certifying initialization safety for a variety of programs generated from both of those systems [50].

3 METHODOLOGY

3.1 Introduction

In this work, our goal was to apply Grant's approach to program synthesis system input/output verification. This chapter gives a brief introduction to this method when applied to a generic code generator.

3.2 Description of Methods

The method used in this work involves the identification of suitable graphical model representations, use of formal specification notation, and availability of associated formal analysis tools. This approach models the input specifications, the output code, and the relationships between them using UML CD's and OCL constraints [26, 27]. The steps of the approach is as follows:

1. Identify key components of the program synthesis system input and output.
2. Identify relationships between its input and output.
3. Identify necessary attributes of the components.

4. Identify the activity and constraints on the problem.

5. Transform the textual description on the constraints to precise mathematical representation using a formal specification language.

The mathematical representation of the problem is then analyzed to identify any deficiencies. These deficiencies could be: (i) omission, (ii) conflict in constraints, and/or (iii) incomplete constraints.

A diagram is given below, in Figure 3.1, to describe the overall process of the approach applied to a code generator, treated as a black box. It is worth noting that if the steps within the dotted line in Figure 3.1 are conducted, there is no longer a need for regression testing.

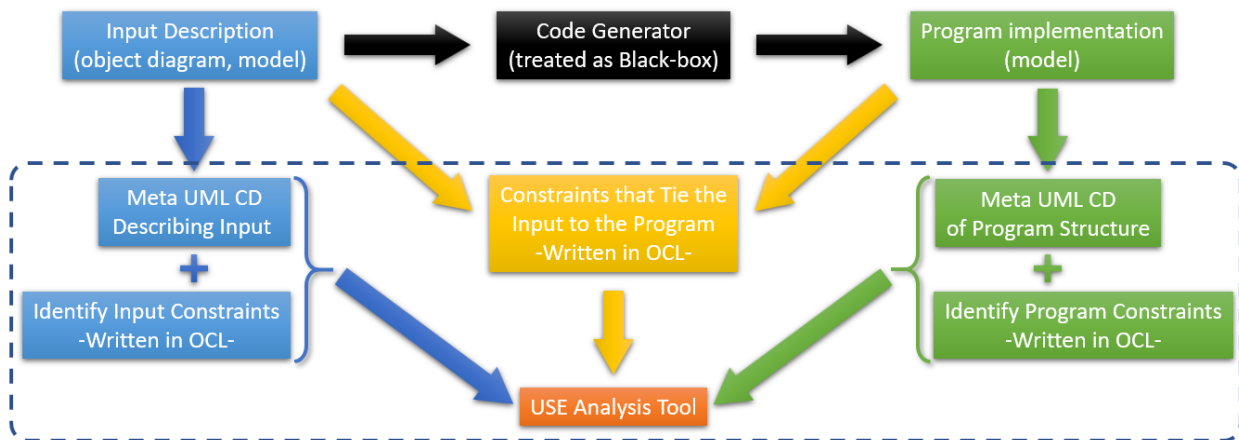


Figure 3.1. Diagrammatic representation of approach to demonstrate program correctness.

3.2.1 Code Generator Input and Output Model Development

First, an equation description of all input (i.e., base equations), along with all other inputs, must be obtained. This can then be used to generate the input CD. Specifically, the comprehensive input description of a code generator must be obtained. It should be noted that with this approach

there is no need to know what the program synthesis is doing or how it does it. We just need to know if it's doing it correctly. Therefore, it can be treated as a black-box.

Next, the program implementation (i.e., the output code) must be used to generate the output CD. When generating the output CD with the intention of obtaining a full verification, every possible category of input should be used to generate all possible categories of classes. This is because the output CD is meant to be a super set of all possible correct output just as the input CD should account for all possible meaningful input.

3.2.2 Code Generator Constraints Definition and Formal Specification

It is from the input and output CD's that the equation (input) and program (output) constraints must be identified, respectively. Next, the constraints that tie the input equation to the program must be identified. This can be constraints on the classes, the attributes of those classes, the operations of those classes, and on the multiplicities between the classes.

The input constraints, output constraints, and the constraints that connect the input and output are then transformed using formal methods into a formal specification language for a precise, mathematically rigorous and testable form.

3.2.3 Code Generator Model Analysis

Finally, the verification of input/output must be carried out. In some cases, this can be done in a semi-automatic fashion by means of an analysis tool. Typically, to do an analysis, both the

input and output CD are combined to form a comprehensive CD. Lastly, the constraints, written in a formal specification language, are checked against the CD to identify any errors.

Both the syntactic and semantic constraints expressed in the UML models should be verified. Typically this can be done through the use of a description of a UML CD model with constraints written in a formal specification language along with (ii) an object diagram description for its specification. Then the verification of that object diagram description against the CD model and constraints can be conducted.

There are three tasks required when checking a code generators input equation specification against its output code: (i) syntactic checking by inspecting the input specification against the input model, (ii) syntactic checking by inspecting the output code against the output model, and (iii) semantic checking by mutual inspecting of the input and output semantic constraints.

Next, in Chapter 4, this methodology is applied to specifically to AUTOBAYES using the OCL formal specification language and the USE analysis tool.

4 CASE STUDY: AUTOBAYES CODE GENERATOR SYSTEM

4.1 Introduction

This chapter will discuss how the general methodology presented in Chapter 3 was applied to NASA's code generator, AUTOBAYES, which is used to generate programs for statistical data analysis. This approach to system correctness verification using UML models of the expected input/output systems with identified constraints on the input/output and their relationships was developed by Grant [26, 27]. CDs and OCL constraints were developed for the domain of AUTOBAYES where a normal distribution (i.e., a Gaussian distribution) of data is assumed.

The diagram from Chapter 3 is given again below, in Figure 4.1, for the readers convenience as well as to give the overall process of the approach as specifically applied to AUTOBAYES. As was the case for a generic code generator, there is no need for regression testing when the steps within the dotted line in Figure 4.1 are conducted.

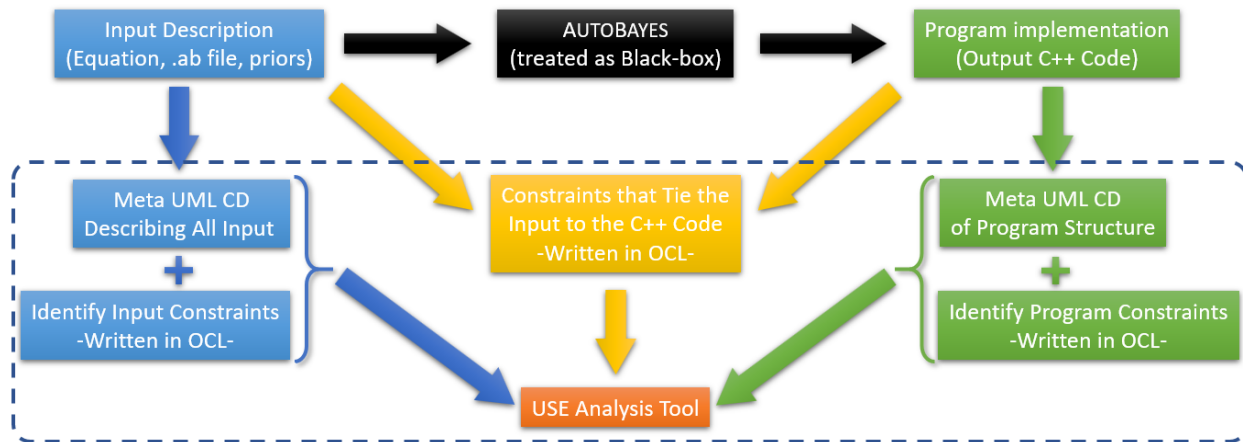


Figure 4.1. Diagrammatic representation of approach to demonstrate program correctness applied to AUTOBAYES.

4.2 AUTOBAYES Input and Output Model Development

First, the key components of AUTOBAYES input and output needed to be identified. The input CD was obtained through an equation description of all input (i.e., base equations), along with all other inputs. Specifically, the comprehensive input description of AUTOBAYES needed to be obtained. It should be noted that with this approach there is no need to know what AUTOBAYES is doing or how it does it. Again, we just need to know if it's doing it correctly can safely treat it as a black-box.

Next, the outputted program code must be used to generate the output CD. When working on developing the output CD for AUTOBAYES, if a full code generator verification was the goal, every possible input would have to be used to generate every type of program implementation possible. However, that was out of the scope of this work, so only a subset of input was used. This was sufficient for the purposes of this study.

While the input CD and output CD was being developed, the identification of necessary attributes and operations of the components was essential to capture a realist picture. Below in Figure 4.2 two example classes derived for AUTOBAYES's input are given. Here the Gaussian class is a generalization of the Statistical Model class. In AUTOBAYES, there are many different models that can be used, but as mentioned earlier, we are only considering the case where a normal distribution of data is assumed. Therefore, only the Gaussian generalization is included in the input CD.



Figure 4.2. Examples of classes representing potential input into AUTOBAYES.

Some classes require a large amount of attributes and operations to realistically model a portion of AUTOBAYES's output. Below, in Figure 4.3, is given. Here, the Gaussian Model class is given, in which there needed to be many attributes and operations. Granted, constructing the input and output CD is a highly creative process, and 10 different researchers could construct 10 different CD's. However, it is important the the content is fully accounted for.

Gaussian Model
+name: STRING +input_args: LIST OF OCTAVE_VALUES +output_args: INTEGER +arg_data: OCTAVE_VALUE +arg_n_classes: OCTAVE_VALUE +arg_tolerance: OCTAVE_VALUE +arg_maxiteration: OCTAVE_VALUE +n_points: INTEGER +n_variables: INTEGER +n_classes: INTEGER +input_data: MATRIX +data_set_name: STRING +tolerance: REAL +sum_up_the_Diffs: REAL +maxiteration: INTEGER +loopcounter: INTEGER
+check_in_and_out_args(): BOOLEAN +return_retval(): OCTAVE_VALUE_LIST +check_data_format(): BOOLEAN +check_n_classes_format(): BOOLEAN +check_tolerance_format(): BOOLEAN +check_maxiteration_format(): BOOLEAN +get_data(): MATRIX +get_n_classes(): INTEGER +get_tolerance(): REAL +get_maxiteration(): INTEGER +get_n_points(): INTEGER

Figure 4.3. The Gaussian Model Class of the AUTOBAYES output code.

4.3 AUTOBAYES Constraints Definition and Formal Specification

It is from the input and output CD's that the AUTOBAYES's constraints were identified. An example of a constraint on the input is

"The n_points attribute of the class Model Parameters must be greater than zero".

An example of a constraint on the output CD is

"The column size attribute (i.e., col_size) of the Memoized Common Subexpression must be equal to one".

Next, the constraints that tie the input equations to the programs were identified from the relationships between AUTOBAYES's input files and output code. An example of a constraint that ties the input to the output is

"If the equation attribute of the Statistical Model class is equal to 'x(_) ~ gauss(mu, sqrt(sigma_sq))', then the Normal Distribution class must be used".

After a sufficient amount of constraints on the input, output and the input/output relationship were derived, the textual description of the constraints were transformed into the formal specification language of OCL. Example of this are

```
"context ModelParameters inv ModParamSize: self.n_points >= 1",
```

```
"context MemoizedCommonSubexpression inv MemoComSubSize: self.col_size = 1",
```

and

```
"context StatisticalModel inv StatModNormDist: self.equation = 'x(_) ~ gauss(mu, sqrt(sigma_sq)).' implies (self.gaussianModel.normalDistribution->size() = 1"
```

to match the above given textual descriptions, respectively.

4.4 AUTOBAYES Model Analysis With the USE Tool

Finally, the verification of input CD, the output CD, their respective associations and constraints, as well as the constraints on the relationship between the input and output can be carried out in a semi-automatic fashion by means of an analysis tool for UML called USE [68]. In USE both the input and output CD are combined to form a comprehensive CD. Lastly, the OCL constraints are input into USE and checked against the CD to identify any errors. The mathematical representation of the problem (i.e., CDs, associations with their respective multiplicities, and OCL constraints) is then analyzed to identify any deficiencies. These deficiencies could be: (i) omission, (ii) conflict in constraints, and/or (iii) incomplete constraints.

USE can be utilized to verify both the syntactic and semantic constraints expressed in the UML models. USE was originally developed as a PhD project as a UML OCL verifier by applying Dijkstra's algorithm for proof. It uses (i) a description of a UML CD model with OCL constraints along with (ii) an object diagram description for its specification (i.e., specifications of instances of AUTOBAYES's input and output). USE can then verify that object diagram description against the CD model and constraints.

There are three tasks required when checking AUTOBAYES's input equation specification against its output code: (i) syntactic checking by inspecting the input specification against the input model, (ii) syntactic checking by inspecting the output code against the output model, and (iii) semantic checking by mutual inspecting of the input and output semantic constraints.

5 RESULTS AND DISCUSSION

5.1 Introduction

This chapter will first present the results that have been obtained along with the various challenges and errors that were encountered and how they were overcome and corrected. Secondly, this chapter will give discussion related to the insights gained from those results. This will include discussion on whether or not this methodology was successful when applied to a new problem domain, what was learned in the process, and did this methodology show promise for a broader applicability for code generators across all problem domains.

These results presented next, for the case study of AUTOBAYES, were obtained by applying Grant's approach to system correctness verification using UML models of the expected input/output systems with identified constraints on the input/output and their relationships [26, 27]. CDs and OCL constraints were developed for the domain of AUTOBAYES for the case where a normal distribution (i.e., a Gaussian distribution) of data is assumed.

5.2 Results

5.2.1 UML Class Diagrams

The derived CD from the n-dimensional Gaussian equation is given in Figure 5.1. This was obtained by carefully considering both the possible input when assuming a normal distribution of data and the structure of a Gaussian equation. This can be seen by comparing the various sections of the AUTOBAYES input files given in the appendix (i.e., Figures A.1 - A.6) to the input CD.

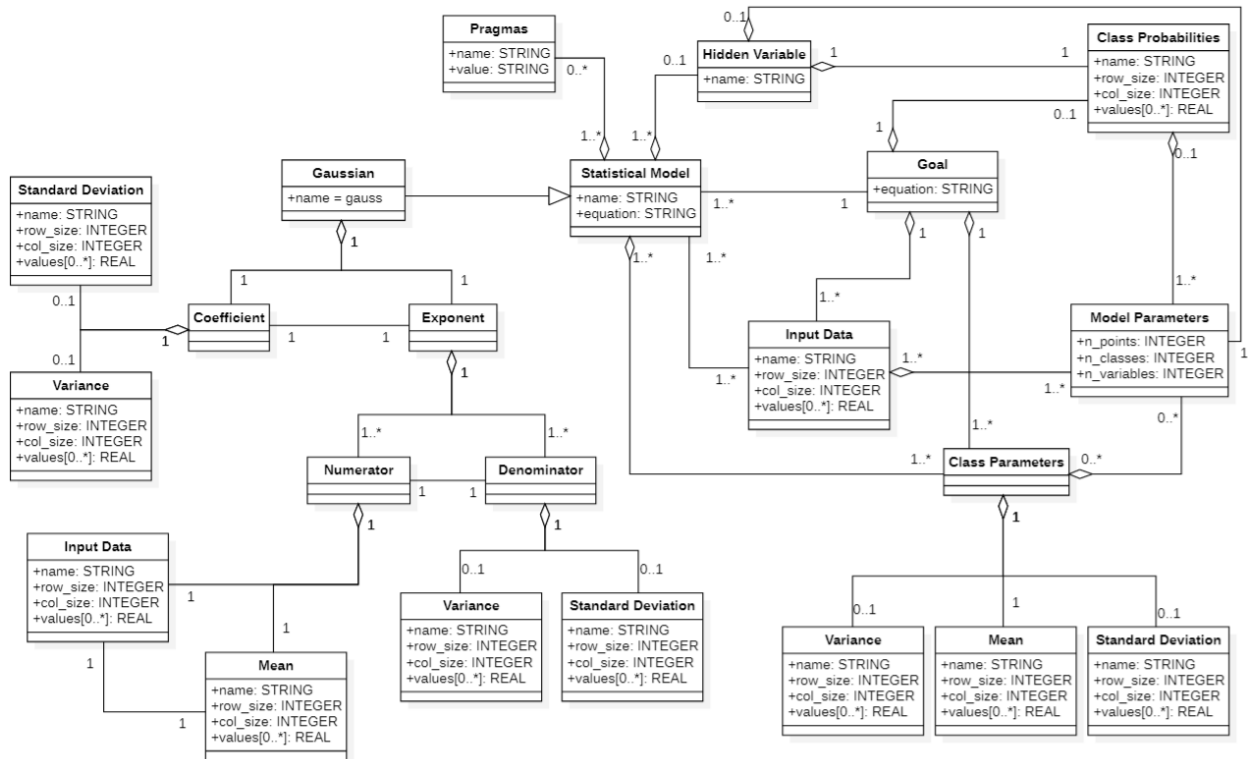


Figure 5.1. Input CD for AUTOBAYES when a Normal distribution of data is assumed and thus the Gaussian equation is used.

The output CD is given below in Figure 5.2. This CD derived from the code that was output from of the considered input options. This CD is quite large, so I will break down each section

(row_size=1, col_size=1), vectors (row_size=n, col_size=1) and matrices (row_size=n, col_size=m)

to the output code.

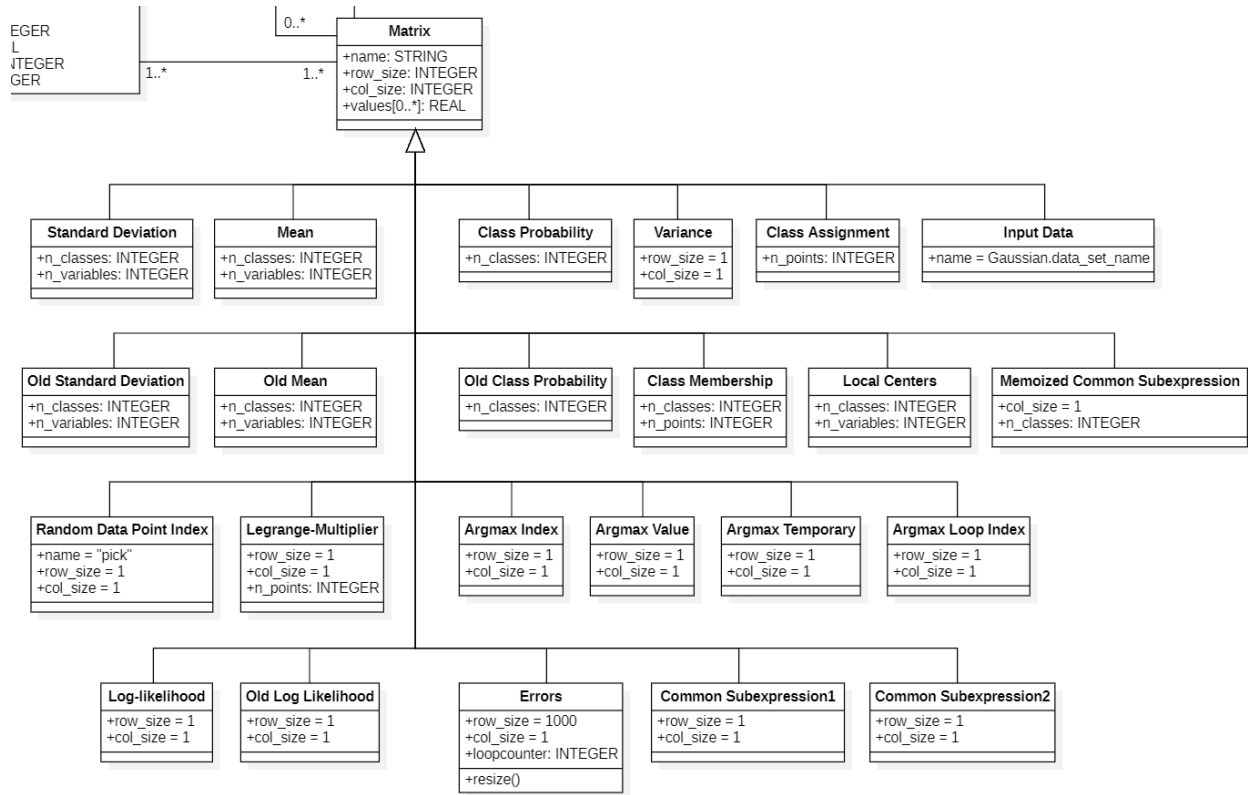


Figure 5.3. Output CD - Matrices.

The Normal Distributions and Transformation are given below in Figure 5.4. Due to both time restrictions and that more work is not needed for a the proof-of-concept work presented in this thesis, only output for normal distributions with a 1D Gaussian and transformations on 1D Gaussians were considered for the constraint and USE analysis.

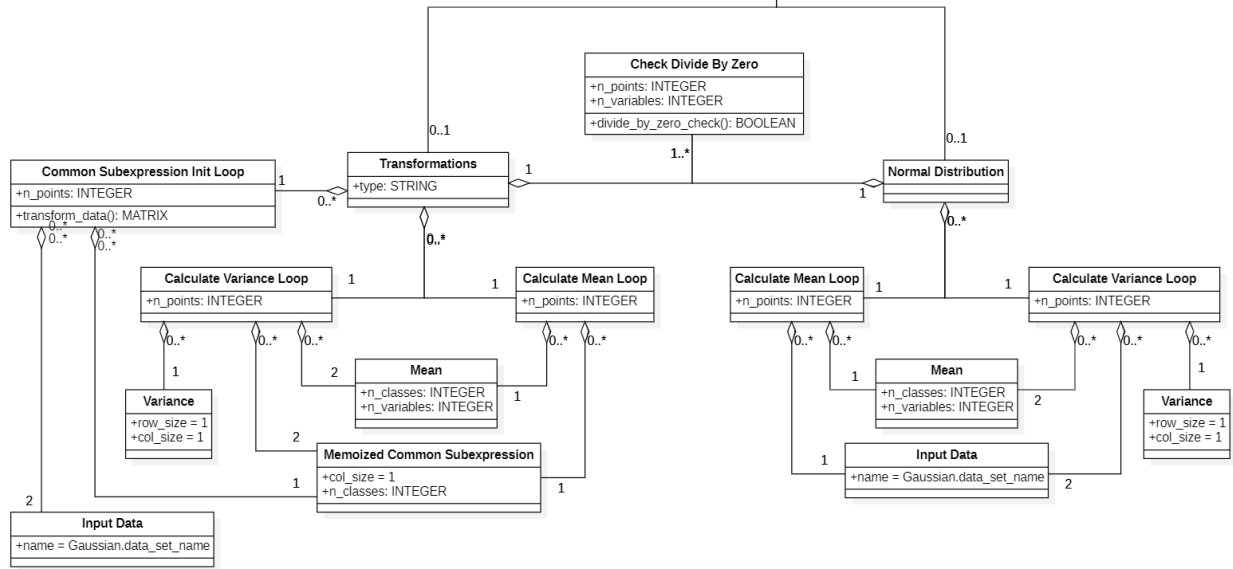


Figure 5.4. Output CD - Normal Distribution and Transformations.

The next sub-CD given a zoomed in view of the Gaussian Model Class along with the Declaration and Initialization classes. Also, the Retval class is used to represent the "retval" construct given in each output code. Here retval stands for "return value" and it stores and returns the values of interest at the end of the calculations present in the code.

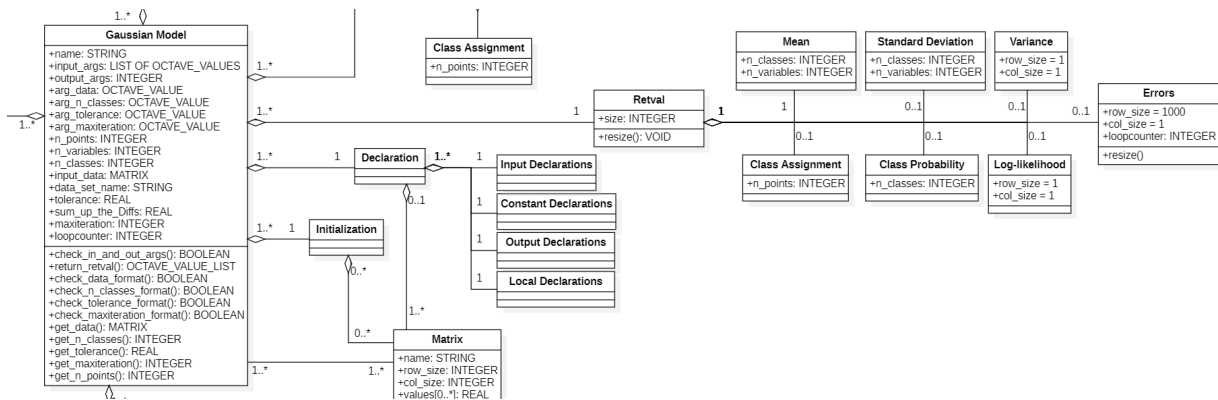


Figure 5.5. Output CD - Gaussian and Retval.

Next, the Discrete EM-algorithm Class is focused upon. This appears in the code when a mixture of Gaussians or multivariate Gaussians are used in the AUTOBAYES input files (e.g.,

Figure 5.7. Here the code for both the random initialization of center values and the calculation for local distributions were present.

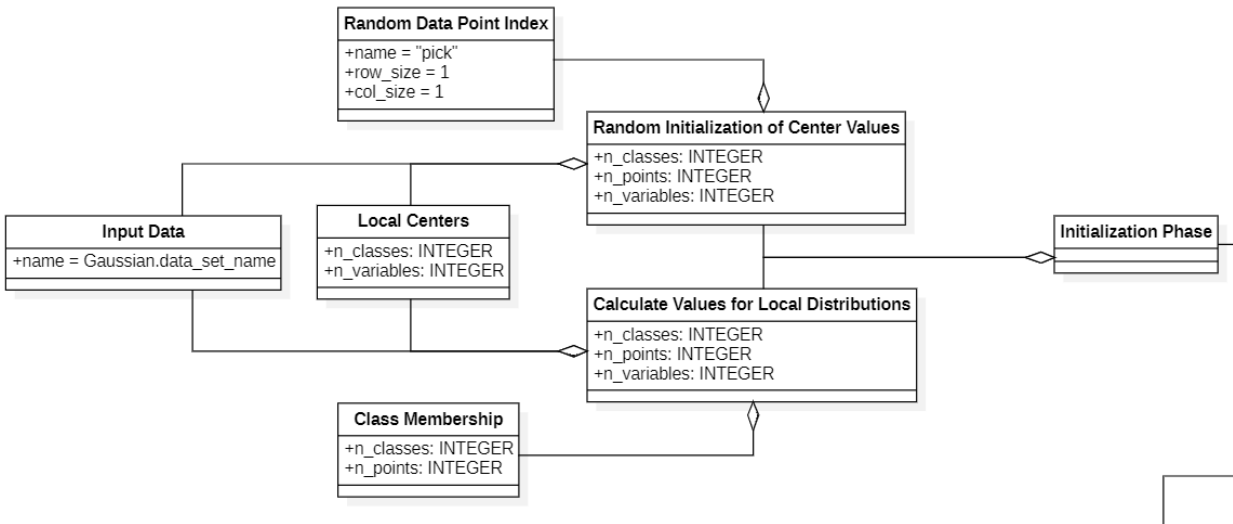


Figure 5.7. Output CD - Initialization Phase.

Second, the Hidden Variable Extraction class representing it's respective phase in the code is given below in Figure 5.8.

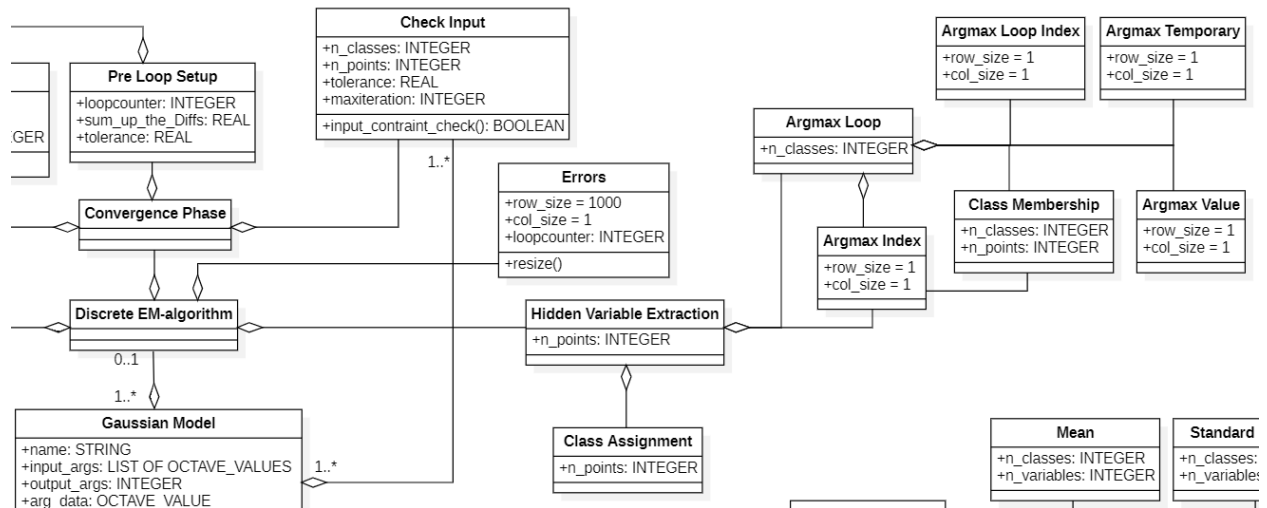


Figure 5.8. Output CD - Hidden Variable Extraction.

Lastly, the Convergence Phase Class is focused on below in Figure 5.9. In the code, the

convergence phase consists of checking the input, a pre loop setup and finally the main body of the actual computational section of the code, the EM-Loop.

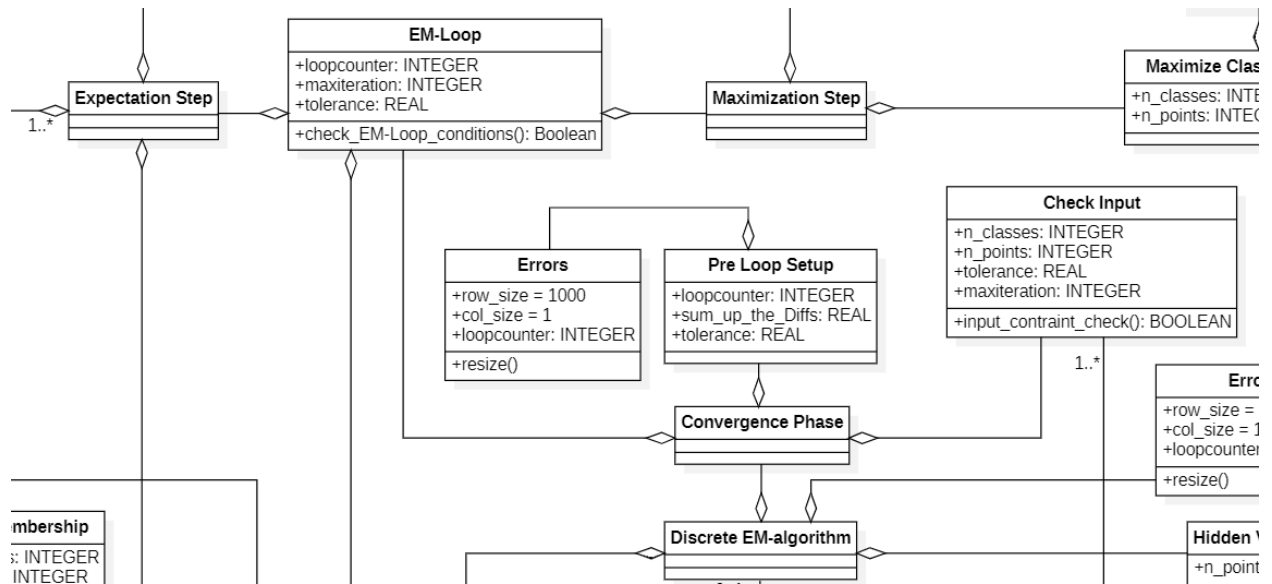


Figure 5.9. Output CD - Convergence Phase.

The EM Loop is made up of an Expectation Step (the "E" in "EM"), a Maximization Step (the "M" in "EM"), and when finished with an iteration, storing the current values of relevant variables as "old" values, later used to test convergence. A zoomed in construction of the CD centered on the EM-Loop class is given below in Figure 5.10.

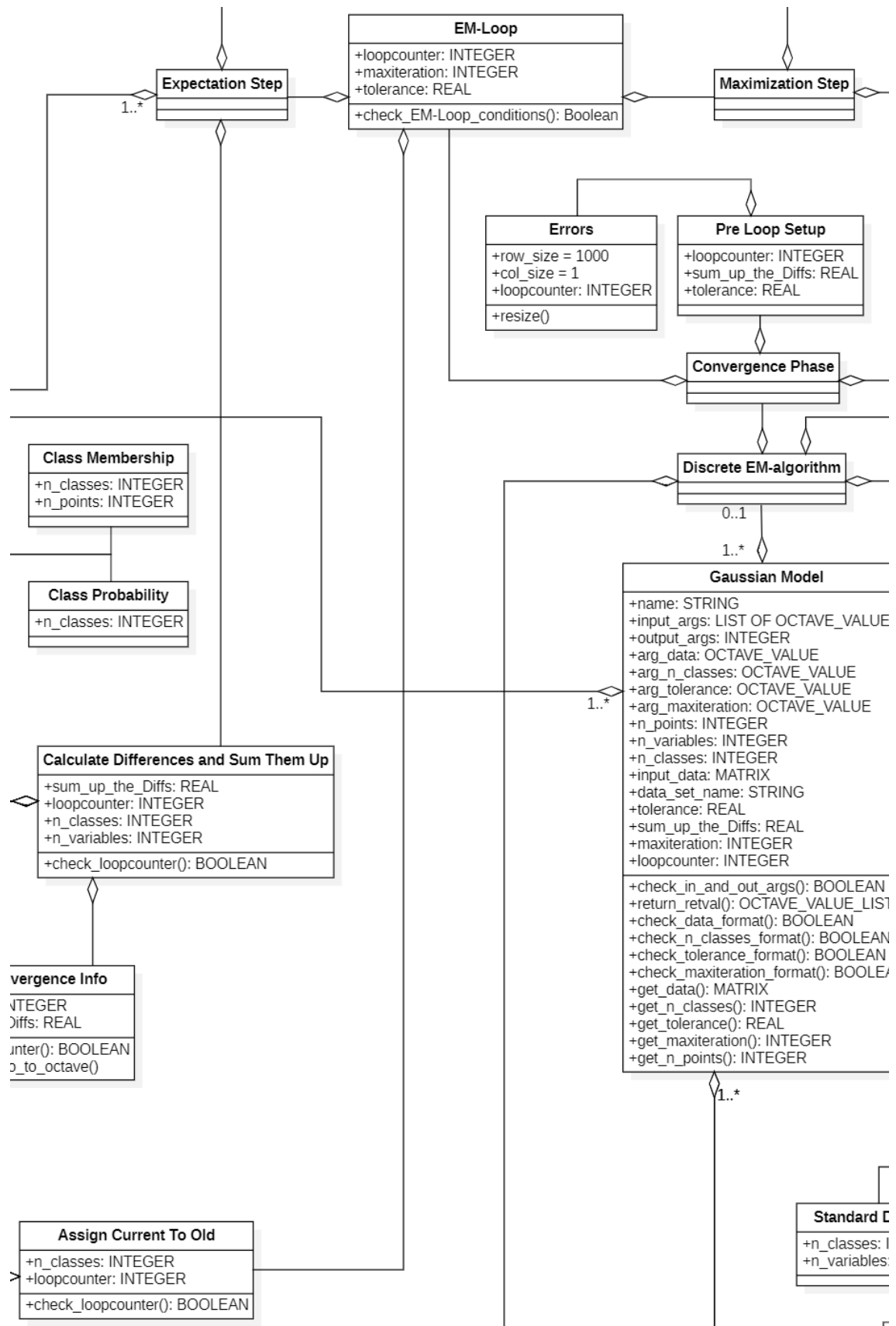


Figure 5.10. Output CD - EM Loop.

First, the Assign Current To Old class will be focused on, given below in Figure 5.11. In this part, the sub-output CD for where the code stores the current values and labels them as old values for comparison against later is given. This gives a list for each of the possible values of interest that can be stored for later use.

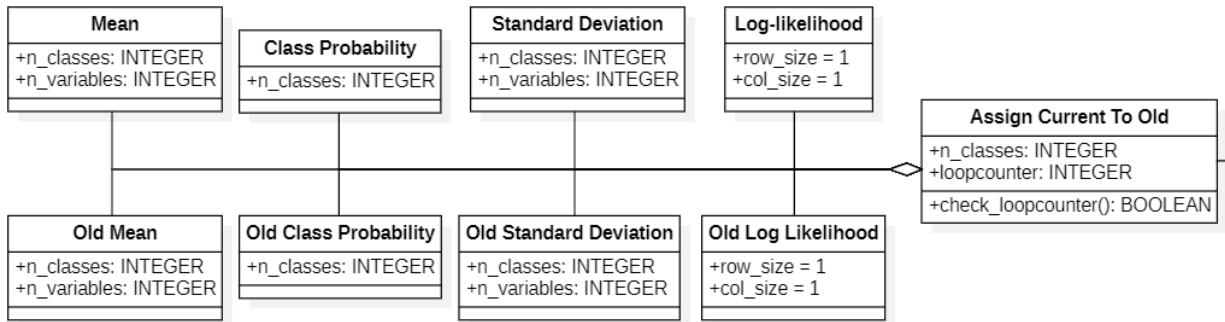


Figure 5.11. Output CD - Current To Old.

Next, the Maximization Step Class is zoomed in on below in Figure 5.12. Here both the Mean and the Standard Deviation is adjusted each iteration. The Class Probability is also Maximized.

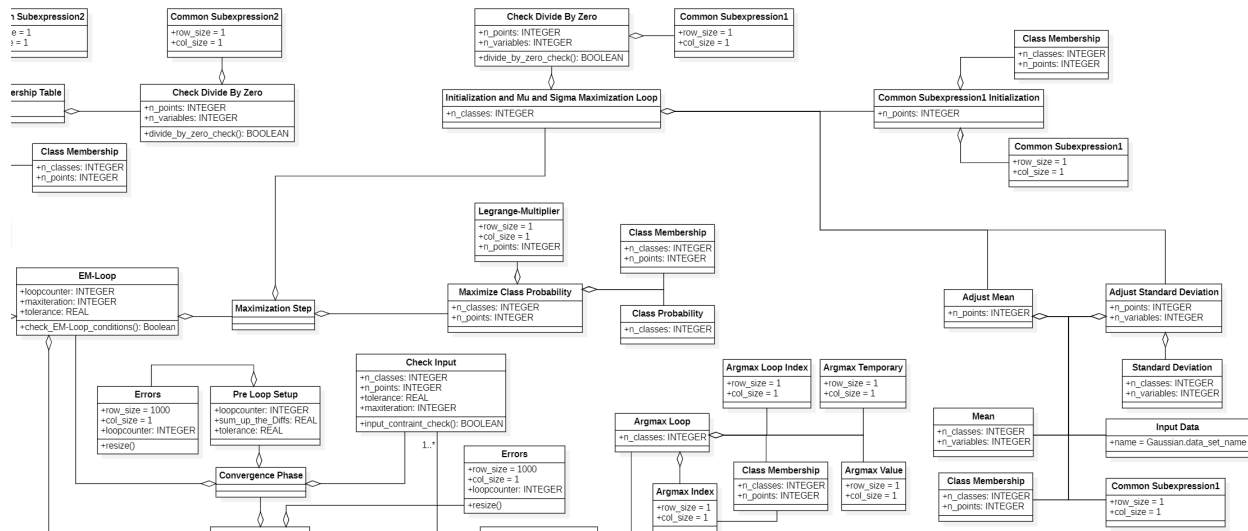


Figure 5.12. Output CD - Maximization Step.

5.2.2 Constraints

The next step was to define the constraint statements for the input CD, the output CD, and the relationship between them. Using domain knowledge, along with the input and output CD models, the necessary relationships between input and output can be specified. Several of the written out constraints that were developed for the input CD, the output CD, and the relationship between them are given below in Table 5.1. For a full list of example constraints reference the appendix (i.e., Table A.1 for the input CD, Table A.2 for the output CD, and Table A.3 for the constraints on the relationship between them). Several of these constraints, as well as others, were then transformed using formal methods into the formal specification language OCL for the validation step described below. It is worth noting, that because of time restrictions and since the scope of this work is a proof-of-concept, only output for normal distributions with a 1D Gaussian and transformations on 1D Gaussians were considered for the constraints that were put into OCL and inputted to USE for analysis. All of the OCL constraints are included in the appendix in Listing A.4.

Table 5.1. A FEW OF THE CONSTRAINTS DEVELOPED FOR THE INPUT, OUTPUT, AND THE RELATIONSHIP BETWEEN THEM.

Number	Constraint
Input	
1	IF Variance is used in ClassParameters, StandardDeviation is not used and vice versa.
2	IF Variance is used in Denominator, StandardDeviation is not used and vice versa.
3	IF Variance is used in Coefficient, StandardDeviation is not used and vice versa.
4	ModelParameters.n_classes > 0
5	ModelParameters.n_variables > 0
6	ModelParameters.n_points > 0
7	Mean.name must be specified.
8	Mean.row_size > 0 and Mean.col_size > 0
9	Variance.name must be specified.
10	Variance.row_size = Variance.col_size = 1.
11	InputData.name must be specified.
Output	
1	Variance.row_size = 1 AND Variance.col_size = 1
2	IF NormalDistribution is used THEN Mean.row_size = Mean.col_size = 1
3	IF Transformations is used THEN Mean.row_size = Mean.col_size = 1
4	IF NormalDistribution is used THEN Transformations is NOT used.
5	IF Transformations is used THEN NormalDistribution is NOT used.
6	The value of variance must always be > 0
7	There must always be a Declaration and an Initialization in the output code.
8	MemoizedCommonSubexpression.col_size = 1
In-Out	
1	IF StatisticalModel.name = gauss AND sqrt() is used in the StatisticalModel.equation (e.g., gauss(mu, sqrt(sigma_sq))), THEN the Variance class must be used.
2	IF StatisticalModel.equation = 'x(_) ~ gauss(mu, sqrt(sigma_sq))' THEN the NormalDistribution class must be used.
3	Input Mean.row_size must equal output Mean.row_size AND input Mean.col_size must equal output Mean.col_size.
4	Input Variance.row_size must equal output Variance.row_size AND input Variance.col_size must equal output Variance.col_size.
5	The input InputData.name must equal the output InputData.name.
6	The input Mean.name must equal the output Mean.name.
7	The input Variance.name must equal the output Variance.name.

5.2.3 USE Analysis

Lastly USE was utilized to verify program correctness. In order to do this, the CD given in Figure 5.1 was used, along with the CD for the output, given in Figure 5.2, the constraints on the input, the constraints on the output, and the constraints on the relationships between them. The CDs, along with the constraints had to be converted to the USE format. Then, both syntactic and semantic checking was automatically conducted from the inputted USE file.

Each of my Class Invariants (OCL constraints) from the USE input were shown to be satisfied in the checks against the inputted model. This USE tool output is shown below in Figure 5.14. To see the full listing of the USE tool input file, which contains the classes, the associations between them, and the OCL constraints, see Listing A.4 in the appendix. However, there are several relevant snippets included here for pedagogical purposes.

Class invariants	
Invariant	Satisfied
ClassParameters::VarStdDevCP	true
ClassParameters::inv4	true
Coefficient::VarStdDevCoeff	true
Declaration::inv2	true
Denominator::VarStdDevDenom	true
Gaussian::GaussName	true
GaussianModel::NormDistOrTransform	true
GaussianModel::NormMeanSize	true
GaussianModel::NormOutInDataCalcMCalcV	true
GaussianModel::TransformCSInitMemoCS	true
GaussianModel::TransformMeanSize	true
GaussianModel::inv1	true
Goal::inv5	true
InputData::InDataOutData	true
InputData::InputDataName	true
Mean::InMeanOutMean	true
Mean::MeanSize	true
MemoizedCommonSubexpression::MemoComSubSize	true
ModelParameters::ModParamSize	true
NormalDistribution::inv6	true
OutputCodeMean::OCMeanSize	true
OutputCodeVariance::OCVarSize	true
OutputCodeVariance::OCVarValues	true
StatisticalModel::StatModNormDist	true
StatisticalModel::StatModTransformLog	true
StatisticalModel::StatModTransformSquare	true
StatisticalModel::inv3	true
Transformations::inv7	true
Variance::InVarOutVar	true
Variance::VarSize	true

Cnstrs. OK. (31ms) 100%

Figure 5.14. The USE Class Invariant View.

Below, in Figure 5.15, the generated CD from USE is given. It was constructed by com-

binning relevant portions of the input CD and output CD along with the relationships between the input and output CD. Note, that only a subset of the output CD diagram is used, since that is all is needed for the proof-of-concept work presented in this thesis.

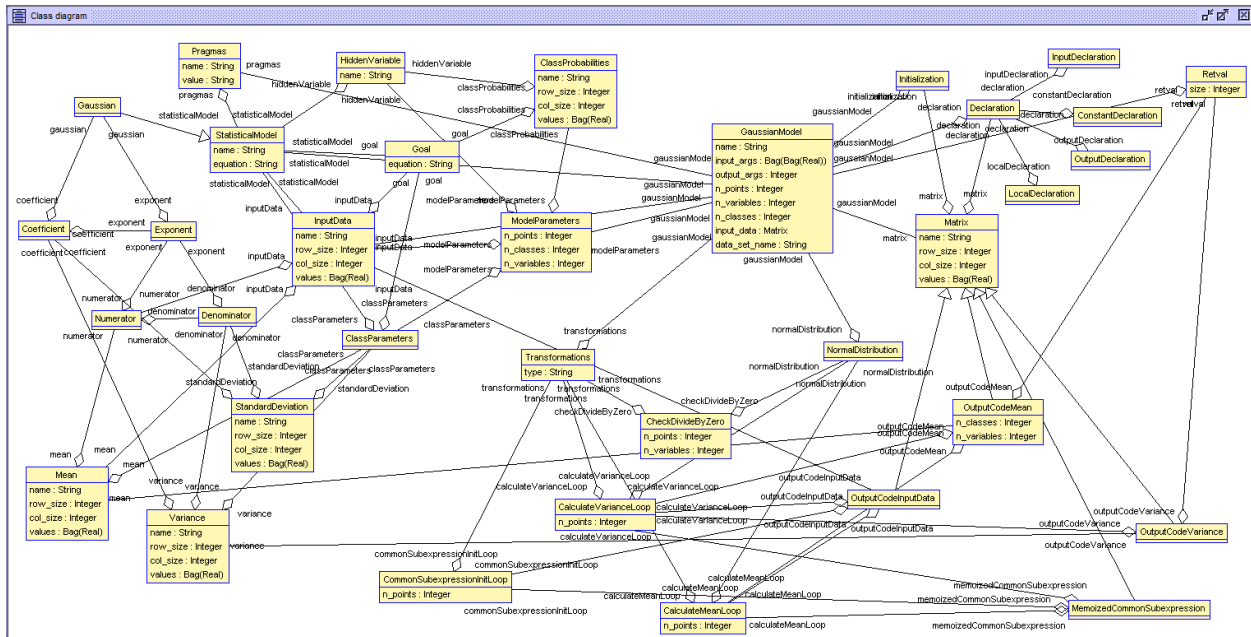


Figure 5.15. The USE class diagram.

Next, the output of the first seven iterations of running the USE model validator are given. The user interface screens, initial configurations, and any changes to the configurations to overcome any errors are given in the appendix in Figures A.7 - A.17. The first iteration was a dry run, i.e., a run just using the default configurations. The output warnings and errors from that iteration are given below in Figure 5.16.

```

INFO: Model configuration successful
INFO: Searching solution with SatSolver `MiniSat' and bitwidth 8...
INFO: TRIVIALY_UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 162 ms; Solving time: 0 ms
INFO: Unsatisfiable proof:
< node: (all c: one Pragma | one (c . Pragma_name)), literal: 5, env: {}>
< node: !(Undefined in (univ . Pragma_name)), literal: -5, env: {}>

```

Figure 5.16. Iteration 1, "Dry run", output.

The second iteration came about after changing the configuration to successfully correct the error related to the Pragma class. The output from that run is given below in Figure 5.17.

```

INFO: Model configuration successful
INFO: Searching solution with SatSolver `MiniSat' and bitwidth 8...
INFO: TRIVIALY_UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 48 ms; Solving time: 0 ms
INFO: Unsatisfiable proof:
< node: (all c: one HiddenVariable | one (c . HiddenVariable_name)), literal: 8, env: {}>
< node: !(Undefined in (univ . HiddenVariable_name)), literal: -8, env: {}>

```

Figure 5.17. Iteration 2 output after applying the the Pragma class fix.

Next, in iteration 3, the configuration change that fixed the error associated with the Pragma class was applied all classes that had the option of having zero objects in their CD multiplicities (e.g., 0..1 or 0..*). The output that came after those changes is given below in Figure 5.18.

```

INFO: Model configuration successful
INFO: Searching solution with SatSolver `MiniSat' and bitwidth 8...
INFO: TRIVIALY_UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 83 ms; Solving time: 0 ms
INFO: Unsatisfiable proof:
< node: (all c: one Goal | one (c . Goal_equation)), literal: 58, env: {}>
< node: !(Undefined in (univ . Goal_equation)), literal: -58, env: {}>

```

Figure 5.18. Iteration 3 output after fixing all the object counts in the configuration.

In iteration 4, the option of having a zero object count for the Goal class added to overcome the error message given in Figure 5.18. The new error after this fix is given below in Figure 5.19.

```
INFO: Model configuration successful
INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
INFO: TRIVIALY_UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 82 ms; Solving time: 0 ms
INFO: Unsatisfiable proof:
< node: (all c: one Mean | one (c . Mean_name)), literal: 246, env: {}>
< node: !(Undefined in (univ . Mean_name)), literal: -246, env: {}>
```

Figure 5.19. Iteration 4 output.

Similar to iteration 4, in iteration 5, the configuration was modified to allow for a zero object count for mean to overcome the error message given in Figure 5.19. The new error after this fix is given below in Figure 5.20.

```
INFO: Model configuration successful
INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
INFO: TRIVIALY_UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 42 ms; Solving time: 0 ms
INFO: Unsatisfiable proof:
< node: (all self: one Gaussian | ((if (self = Undefined) then Undefined else (self . StatisticalModel_name)) = String_gauss)), literal: -2147483647, env: {}>
```

Figure 5.20. Iteration 5 output.

Next, in iteration 6, the Min. Object Quantity for the Gaussian class was set to 0, this corrected the error message given above in Figure 5.20. After this change a new error was produced and is given below in Figure 5.21.

```
INFO: Model configuration successful
INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
INFO: TRIVIALY_UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 32 ms; Solving time: 0 ms
INFO: Unsatisfiable proof:
< node: (all c: one GaussianModel | one (c . GaussianModel_data_set_name)), literal: 326, env: {}>
< node: !(Undefined in (univ . GaussianModel_data_set_name)), literal: -326, env: {}>
```

Figure 5.21. Iteration 6 output.

For the seventh iteration, the configuration in Classes and Associations was again modified to overcome this error. This was accomplished by allowing the validator to test with a zero object quantity minimum for the GaussianModel class. This fixed the error, given above in Figure 5.21, and produced the the output given below in Figure 5.22.

```
INFO: Model configuration successful
INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
INFO: UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 113 ms; Solving time: 0 ms
```

Figure 5.22. Iteration 7 output.

5.3 Discussion

In this section, the results will be discussed along with key learning experiences and the potential benefits of this this work.

The development of the input CD had gone through several iterations, each time a deeper and fuller understanding of the process of this methodology was obtained. It is my hope that this CD, along with the AUTOBAYES input files given in the appendix in Figures A.1 - A.6, others will have the examples they need to apply this method to other code generators.

The development of the output CD proved to be significantly more intricate and complex than the previous application of this methodology to AUTOFILTER due to both the large amount of generated code for a given input, and the large variation in the outputted code based on small changes to the input. The output code is given in the appendix in Listings A.2 - A.2. This proved to be an excellent learning opportunity, since it required careful reading through the output code and

careful consideration on correct and efficient representation of code sections with classes in the CD. Through this work, ideas on automation of the output CD development from outputted code have made themselves apparent. This could save future users of this methodology a significant amount of time.

The process of designing each of the constraints listed in Table A.1 for the input CD, Table A.2 for the output CD, and Table A.3 for the constraints on the relationship between them in the appendix as well as the full list of constraints that were transformed into OCL and included at the end of the USE input file given in the appendix, Listing A.4 was highly successfully and educational. One of the best advantages of formal methods is that it forces the user to think deeply about the system under study. This was certainly the case, and it brought about multiple revisions to the input and output CD's. This process displayed in the work presented in this thesis serves as a great proof-of-concept of the usefulness of formal specification languages when a deep understanding of a system is needed and a high degree of confidence is required (e.g., in safety-critical systems).

For the USE tool for analysis of the model, though it successfully tested all of the class invariant and the constraints were satisfied in the model, it was not able to be successfully utilized for full validation of a given configuration for the input and output at this time. However, the developed model with the classes, associations, and constraints can be of use for others to test a specific configurations. It should be noted that obtaining a validated configuration, the objectives of this work was still met (i) it was shown that this methodology is extensible beyond the state estimation domain (ii) and a practical example of the use of formal methods was given. Though we have yet to get a fully validated configuration, this portion of the project has been an excellent learning experience. This is true for both generating input (i.e., constructing one CD from an input

and output CD, determining multiplicities, and writing the developed constraints in the format needed for USE input), and for running the various analysis capabilities present in USE (i.e., the Class Invariant tester and the model validation against a specific configuration). This ground work will be useful for both my own potential future work as well as other students using Grant's procedure for verification of program synthesis systems.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Over the years, there have been great advances in the area of program synthesis and it has been shown to have many advantageous uses [2–6, 6–12]. However, certifying the correctness of the generated code from the input specifications can be a difficult procedure. Therefore, much work has been devoted to this by NASA [14, 16, 28–59] in the context of their program synthesis systems, AUTOFILTER [14–18] and AUTOBAYES [1, 19–25]. The approach presented by Grant et al. in collaboration with NASA researchers contributed to this effort for AUTOFILTER [26, 27] providing an automatically generated verification. This approach uses domain-specific graphical meta-models of the expected input/output systems with identified constraints on the input/output and their relationships which allows for a rigorous analysis of these constraints against specific instances of input/output using mathematical expressions [26, 27]. However, this verification procedure had not yet been applied to AUTOBAYES. In the work presented in this paper, Grant’s approach is applied to AUTOBAYES and initial results have been obtained. The CD representing the input specification for the case in which a normal distribution of data is assumed was successfully obtained. In

this case, the n -dimensional Gaussian equation is used, where n is the number of dimensions of the data considered. The output CD was successfully derived from several instances of outputted code from running AUTOBAYES on multiple input files. Constraints on the input CD, output CD, and the relationship between the input and output were developed. Several of these constraints were then transformed into OCL and input into USE for analysis, along with the relevant classes from the input and output CDs. The constraints were found to satisfy the model. Unfortunately, a configuration that could be fully validated was not obtained due to time limitations and the high level of complexity of AUTOBAYES. Though AUTOBAYES was shown to be far more complex than initially thought, the success of applying Grant's approach to AUTOBAYES, shows the potential that it is ready to be applied to a wide variety of domains. One of our main research goals was to investigate the applicability of Grant's approach to other domains, e.g. in the safety-critical system domain, which is especially relevant for NASA and an interesting future direction for us.

6.2 Future Work

There are several future directions of this work. The most natural being to determine a configuration that can be fully validated. Another future direction after that would be to complete a full analysis of AUTOBAYES all possible input and output, allowing for all the mentioned input equations in Table 2.1 and the mixture of those inputs. Though this would be a highly time intensive task, it may be useful depending on if AUTOBAYES sees a resurgence of use in the future.

Another direction would be to work on methods to fully automate this process. By far, the most time intensive part of this work, was developing the input and output CD's. Now that I have

gone through the process manually for both the input and output CD development, several ideas have come up. The transformation the output code into a CD should be quite straight forward, and there are even some tools currently available that may be able to help with that, the development of an input CD from a general program synthesis system from its specification documentation is a non-trivial, creative task. However, with the recent advancements in Machine Learning, it may be possible to capitalize on these advances to derive either a starting input CD or even a complete CD from the specification documents.

Lastly, since this work produced an updated description of the program correctness verification methodology based on lessons learned from its application in the AUTOFILTER case study, this method may be now applied to other domains. It would be of great interest to us to validate the extension of this to safety critical systems. If successful, this research can have broad impacts reaching beyond the AUTOFILTER and AUTOBAYES domains and may be applied to other program synthesis systems and even adapted to non-program synthesis systems. One well known domain, in which the use of a strategy like the one presented in this thesis could prove advantageous, is in a system similar to the Boeing 737 MAX MCAS avionic system.

7 FUNDING ACKNOWLEDGEMENT

I would like to acknowledge and thank the North Dakota Space Grant Consortium (NDSGC) for providing the funding for this research through two summers of a Graduate Student Research Fellowship. They generously supported this research in both summer 2019 and summer 2020.

I would also like to acknowledge and thank Dr. Johann Schumann from NASA's AMES Research Center for providing us with the most updated version of AUTOBAYES as well as instructions on how to navigate the installation process with the necessary legacy versions of SWI-Prolog.

A APPENDICES

A.1 AUTOBAYES Input Files

This section of the appendices will give all of the AUTOBAYES input files used in this work. Each of these files aided in the derivation of the of the input CD. AUTOBAYES input files all must end in ".ab".

```
1  model normal as 'Normal Distributed Data'.
2
3  const nat n as 'NUMBER OF DATA POINTS'.
4
5  double mu as 'UNKNOWN MEAN'.
6  double sigma_sq as 'UNKNOWN VARIANCE'.
7  |   where  $\theta < \text{sigma\_sq}$ .
8
9  data double x(0..n-1) as 'GIVEN DATA POINTS'.
10
11 x(_) ~ gauss(mu, sqrt(sigma_sq)).
12
13 max pr( x | {mu, sigma_sq} ) for {mu, sigma_sq}.
```

Figure A.1. The simplest input file that assumes a normal distribution of data.

```

1  model square_normal as 'SQURE-NORMAL MODEL'.
2
3  const nat n as 'NUMBER OF DATA POINTS'.
4
5  double mu as 'UNKNOWN MEAN'.
6  double sigma_sq as 'UNKNOWN VARIANCE'.
7  |
8  |   where  $\theta < \text{sigma\_sq}$ .
9
10 data double x(0..n-1) as 'CURRENT DATA POINTS (KNOWN)'.
11 x(_)**2 ~ gauss(mu, sqrt(sigma_sq)).
12 max pr( x | {mu, sigma_sq} ) for {mu, sigma_sq}.

```

Figure A.2. A slight modification to the input file in Figure A.1 in which a square-normal transformation was used.

```

1  model log_normal as 'LOG-NORMAL MODEL'.
2
3  const nat n as 'NUMBER OF DATA POINTS'.
4
5  double mu as 'UNKNOWN MEAN'.
6  double sigma_sq as 'UNKNOWN VARIANCE'.
7  |
8  |   where  $\theta < \text{sigma\_sq}$ .
9
10 data double x(0..n-1) as 'CURRENT DATA POINTS (KNOWN)'.
11 log(x(_)) ~ gauss(mu, sqrt(sigma_sq)).
12 max pr( x | {mu, sigma_sq} ) for {mu, sigma_sq}.

```

Figure A.3. A slight modification to the input file in Figure A.1 in which a log-normal transformation was used.


```

1  model mog as 'MIXTURE OF GAUSSIANS'.
2
3      % MODEL PARAMETERS
4  const nat n_points as 'NUMBER OF DATA POINTS'.
5      where  $\theta < n\_points$ .
6  const nat n_classes as 'NUMBER OF CLASSES'.
7      where  $\theta < n\_classes$ .
8      where  $n\_classes \ll n\_points$ .
9
10     % CLASS PROBABILITIES
11  double phi( $\theta..n\_classes-1$ ) as 'CLASS PROBABILITY VECTOR.'.
12     where  $\theta = \text{sum}(I := \theta .. n\_classes-1, \text{phi}(I))-1$ .
13
14     % CLASS PARAMETERS
15  double mu( $\theta..n\_classes-1$ ) as 'COLUMN VECTOR OF MEANS'.
16  double sigma( $\theta..n\_classes-1$ ) as 'COLUMN VECTOR OF STD DEVS'.
17     where  $\theta < \text{sigma}(\_)$ .
18
19     % HIDDEN VARIABLE
20  output nat c( $\theta..n\_points-1$ ) as 'CLASS ASSIGNMENT VECTOR'.
21  c( $\_$ ) ~ discrete(vector( $I := \theta .. n\_classes-1, \text{phi}(I)$ )).
22
23     % DATA
24  data double x( $\theta..n\_points-1$ ).
25  x( $I$ ) ~ gauss(mu(c( $I$ )), sigma(c( $I$ ))).
26
27  max pr( x | {sigma, mu, phi} ) for {sigma, mu, phi}.

```

Figure A.4. An input file in which a basic clustering example is given with a mixture of Gaussians.

```

1  model mult_cluster as 'SIMPLE MULTIVARIATE CLUSTERING MODEL'.
2
3      % MODEL PARAMETERS
4  const nat n_variables as 'NUMBER OF VARIABLES'.
5  const nat n_points as 'NUMBER OF DATA POINTS'.
6  const nat n_classes as 'NUMBER OF CLASSES'.
7      where  $\theta < n\_classes$ .
8      where  $n\_classes \ll n\_points$ .
9
10     % CLASS PROBABILITIES
11  double phi( $\theta..n\_classes-1$ ) as 'CLASS PROBABILITY VECTOR.'.
12     where  $\theta = \text{sum}(I := \theta .. n\_classes-1, \text{phi}(I))-1$ .
13
14     % CLASS PARAMETERS
15  double mu( $\theta..n\_variables-1, \theta..n\_classes-1$ ) as 'COLUMN VECTOR OF MEANS'.
16  double sigma( $\theta..n\_variables-1, \theta..n\_classes-1$ ) as 'COLUMN VECTOR OF STD DEVS'.
17     where  $\theta < \text{sigma}(\_,\_)$ .
18
19     % HIDDEN VARIABLE
20  output nat class_assignment( $\theta..n\_points-1$ ) as 'HIDDEN VARIABLE'.
21  class_assignment(_) ~ discrete(vector( $I := \theta .. n\_classes-1, \text{phi}(I)$ )).
22
23     % DATA
24  data double sim_data( $\theta..n\_variables-1, \theta..n\_points-1$ ).
25  sim_data(C,I) ~ gauss(mu(C,class_assignment(I)), sigma(C,class_assignment(I))).
26
27     % GOAL
28  max pr( {sim_data} | {phi, mu, sigma} ) for {phi, mu, sigma}.

```

Figure A.5. An input file in which a more complex clustering example is given with a multivariate mixture of Gaussians.

```

1  model iris as
2  |   'SIMPLE MULTIVARIATE CLUSTERING MODEL FOR CLASSICAL IRIS FLOWER EXAMPLE'.
3
4  const nat n_variables as 'NUMBER OF FEATURES'.
5  const nat n_points as 'NUMBER OF DATA POINTS'.
6  const nat n_classes as 'NUMBER OF CLASSES'.
7  |   where  $\theta < n\_classes$ .
8  |   where  $n\_classes \ll n\_points$ .
9
10 double phi( $\theta..n\_classes-1$ ) as 'CLASS PROBABILITY VECTOR.'.
11 |   where  $\text{sum}(I := \theta .. n\_classes-1, \text{phi}(I)) = 1$ .
12
13 double mu( $\theta..n\_variables-1, \theta..n\_classes-1$ ) as 'MATRIX OF MEANS'.
14 double sigma( $\theta..n\_variables-1, \theta..n\_classes-1$ ) as 'MATRIX OF STD DEVS'.
15 |   where  $\theta < \text{sigma}(\_, \_)$ .
16
17 output nat class_assignment( $\theta..n\_points-1$ ) as 'CLASS OF EACH POINT'.
18 class_assignment(_) ~ discrete(vector(I :=  $\theta .. n\_classes-1, \text{phi}(I)$ )).
19
20 data double iris_data( $\theta..n\_variables-1, \theta..n\_points-1$ ).
21 iris_data(C,I) ~ gauss(mu(C, class_assignment(I)), sigma(C, class_assignment(I))).
22
23 max pr( {iris_data} | {phi, mu, sigma} ) for {phi, mu, sigma}.

```

Figure A.6. The input file used as an example in [1] designed to be used with the Fisher Iris flower multivariate data set.

A.2 AUTOBAYES Output C++ Code

In this section the programs generated by invoking AUTOBAYES on each of the above listed input files are given. As mentioned in Chapter 1, the code was always generated for use with the OCTAVE environment, which can be seen in each the code listing below, rather than for the MATLABTM environment. Each of the output files given below from AUTOBAYES end with a ".cc" extension. The reason this code is included here in the appendix of this thesis is to aid in the understanding of reader. The reader is encouraged to make connections between these output files and the output CD given early in Chapter 5 in Figure 5.2.

```

1
2 //-----
3 // Code file generated by AutoBayes V0.9.9
4 // AutoBayes(c) 2008-2011 United States Government as represented by
5 // the Administrator of NASA. AutoBayes is distributed under the NASA
6 // Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
7 // details.
8 // Problem:   Normal Distributed Data
9 // Source:    normal.ab
10 // Command:
11 //
12 //   PROLOG_VAR
13 //
14
15 //           -designdoc
16 //           normal.ab
17 // Generated: Fri Jun 26 12:08:31 2020
18 //-----
19
20 #include "autobayes.h"
21
22
23
24 //-----
25 // Octave Function: normal
26 //-----
27
28 DEFUN_DLD(normal, input_args, output_args,
29           "usage: [double mu,double sigma_sq] = normal(vector x)\n\n"
30           )
31 {
32   octave_value_list retval;
33   if (input_args.length () != 1 || output_args != 2 ){
34     octave_stdout << "usage: [double mu,double sigma_sq] = normal(vector x)\n\n";
35     return retval;
36   }
37
38   //-- Input declarations -----
39
40   // GIVEN DATA POINTS
41   octave_value arg_x = input_args(0);
42   if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
43     gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
44     return retval;
45   }
46   ColumnVector x = (ColumnVector)(arg_x.vector_value());
47
48   //-- Constant declarations -----
49
50   // NUMBER OF DATA POINTS
51   int n = arg_x.rows();
52
53   //-- Output declarations -----

```

```

54
55 // UNKNOWN MEAN
56 double mu;
57 // UNKNOWN VARIANCE
58 double sigma_sq;
59
60 //-- Local declarations -----
61
62 // Summation accumulator
63 // sum([pv1 := 0 .. -1 + n], x(pv1))
64 double pv3;
65
66 int pv1;
67
68 // Summation accumulator
69 // sum([pv2 := 0 .. -1 + n], (-1 * mu + x(pv2)) ** 2)
70 double pv5;
71
72 int pv2;
73
74
75 // The conditional probability pr(x | {mu,sigma_sq}) is under the
76 // dependencies given in the model equivalent to
77 //
78 // prod([pv0 := 0 .. -1 + n], pr(x(pv0) | {mu,sigma_sq}))
79 //
80 // The probability occurring here is atomic and can thus be replaced by the
81 // respective probability density function given in the model. This yields
82 // the log-likelihood function
83 //
84 // log(prod([pv0 := 0 .. -1 + n],
85 //          exp(-1 / 2 * (x(pv0) - mu) ** 2 / (sigma_sq ** (1 / 2)) ** 2)
86 //          *
87 //          (1 / (sqrt(2 * pi) * sigma_sq ** (1 / 2)))))
88 // which can be simplified to
89 //
90 // -1 / 2 * n * log(2) + -1 / 2 * n * log(pi) + -1 / 2 * n * log(sigma_sq)
91 // +
92 // -1 / 2 * sigma_sq ** -1 *
93 // sum([pv0 := 0 .. -1 + n], (-1 * mu + x(pv0)) ** 2)
94 // This function is then optimized w.r.t. the goal variables mu and sigma_sq
95 //
96 // The summands
97 //
98 // -1 / 2 * n * log(2)
99 // -1 / 2 * n * log(pi)
100 //
101 // are constant with respect to the goal variables mu and sigma_sq and can
102 // thus be ignored for maximization.
103 //
104 // The factor

```

```

105 //
106 // 1 / 2
107 //
108 // is non-negative and constant with respect to the goal variables mu and
109 // sigma_sq and can thus be ignored for maximization.
110 //
111 // The function
112 //
113 // -1 * n * log(sigma_sq) +
114 // -1 * sigma_sq ** -1 * sum([pv0 := 0 .. -1 + n], (-1 * mu + x(pv0)) **
115 // 2)
116 // is then symbolically maximized w.r.t. the goal variables mu and sigma_sq.
117 // The partial differentials
118 //
119 // df / d_mu ==
120 // -2 * mu * n * sigma_sq ** -1 +
121 // 2 * sigma_sq ** -1 * sum([pv0 := 0 .. -1 + n], x(pv0))
122 // df / d_sigma_sq ==
123 // -1 * n * sigma_sq ** -1 +
124 // sigma_sq ** -2 * sum([pv0 := 0 .. -1 + n], (-1 * mu + x(pv0)) ** 2)
125 //
126 // are set to zero; these equations yield the solutions
127 //
128 // mu ==
129 // cond(0 == n, fail(division_by_zero),
130 // n ** -1 * sum([pv1 := 0 .. -1 + n], x(pv1)))
131 // sigma_sq ==
132 // cond(0 == n, fail(division_by_zero),
133 // n ** -1 * sum([pv2 := 0 .. -1 + n], (-1 * mu + x(pv2)) ** 2))
134 //
135 if ( 0 == n )
136 { ab_error( division_by_zero ); }
137 else
138 {
139     pv3 = 0.0;
140     for( pv1 = 0;pv1 <= n - 1;pv1++ )
141         pv3 += x(pv1);
142     mu = pv3 / (double)(n);
143 }
144 if ( 0 == n )
145 { ab_error( division_by_zero ); }
146 else
147 {
148     pv5 = 0.0;
149     for( pv2 = 0;pv2 <= n - 1;pv2++ )
150         pv5 += (x(pv2) - mu) * (x(pv2) - mu);
151     sigma_sq = pv5 / (double)(n);
152 }
153
154 retval.resize(2);
155 retval(0) = mu;
156 retval(1) = sigma_sq;
157

```

```

158     return retval;
159 }
160 //-- End of code

```

Listing A.1. The C++ code AUTOBAYES generated from the input file given in Figure A.1.

```

1
2 //-----
3 // Code file generated by AutoBayes V0.9.9
4 // AutoBayes(c) 2008-2011 United States Government as represented by
5 // the Administrator of NASA. AutoBayes is distributed under the NASA
6 // Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
7 // details.
8 // Problem:    SQUARE-NORMAL MODEL
9 // Source:     square_normal.ab
10 // Command:
11 //
12 //    PROLOG_VAR
13 //
14
15 //          -designndoc
16 //          square_normal.ab
17 // Generated: Fri Jun 26 13:43:50 2020
18 //-----
19
20 #include "autobayes.h"
21
22
23
24 //-----
25 // Octave Function: square_normal
26 //-----
27
28 DEFUN_DLD(square_normal, input_args, output_args,
29           "usage: [double mu,double sigma_sq] = square_normal(vector x)\n\n"
30           )
31 {
32     octave_value_list retval;
33     if (input_args.length () != 1 || output_args != 2 ){
34         octave_stdout << "usage: [double mu,double sigma_sq] = square_normal(
35             vector x)\n\n";
36         return retval;
37     }
38     //-- Input declarations -----
39
40     // CURRENT DATA POINTS (KNOWN)
41     octave_value arg_x = input_args(0);
42     if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
43         gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
44         return retval;
45     }

```

```

46 ColumnVector x = (ColumnVector)(arg_x.vector_value());
47
48 //-- Constant declarations -----
49
50 // NUMBER OF DATA POINTS
51 int n = arg_x.rows();
52
53 //-- Output declarations -----
54
55 // UNKNOWN MEAN
56 double mu;
57 // UNKNOWN VARIANCE
58 double sigma_sq;
59
60 //-- Local declarations -----
61
62 // Memoized common subexpression
63 // x(pv2) ** 2
64 ColumnVector pv4(n);
65
66 // Summation accumulator
67 // sum([pv2 := 0 .. -1 + n], pv4(pv2))
68 double pv7;
69
70 // Loop variable
71 int pv2;
72
73 // Summation accumulator
74 // sum([pv3 := 0 .. -1 + n], (-mu + pv4(pv3)) ** 2)
75 double pv8;
76
77 int pv3;
78
79
80 // The conditional probability pr(x | {mu,sigma_sq}) is under the
81 // dependencies given in the model equivalent to
82 //
83 // prod([pv1 := 0 .. -1 + n], pr(x(pv1) | {mu,sigma_sq}))
84 //
85 // The probability occurring here is atomic and can thus be replaced by the
86 // respective probability density function given in the model. This yields
87 // the log-likelihood function
88 //
89 // log(prod([pv1 := 0 .. -1 + n],
90 //          abs(deriv(x(pv1) ** 2, x(pv1))) *
91 //          exp(-1 / 2 * (x(pv1) ** 2 - mu) ** 2 /
92 //          (sigma_sq ** (1 / 2)) ** 2) *
93 //          (1 / (sqrt(2 * pi) * sigma_sq ** (1 / 2)))))
94 //
95 // which can be simplified to
96 //
97 // -1 / 2 * n * log(2) + -1 / 2 * n * log(pi) + -1 / 2 * n * log(sigma_sq)
98 // +
99 // -1 / 2 * sigma_sq ** -1 *

```



```

99 //      sum([pv1 := 0 .. -1 + n], (-1 * mu + x(pv1) ** 2) ** 2) +
100 //      sum([pv1 := 0 .. -1 + n], log(abs(2 * x(pv1))))
101 //
102 // This function is then optimized w.r.t. the goal variables mu and sigma_sq
103 //
104 // The summands
105 //
106 //      -1 / 2 * n * log(2)
107 //      -1 / 2 * n * log(pi)
108 //      sum([pv1 := 0 .. -1 + n], log(abs(2 * x(pv1))))
109 //
110 // are constant with respect to the goal variables mu and sigma_sq and can
111 // thus be ignored for maximization.
112 //
113 // The factor
114 //
115 //      1 / 2
116 //
117 // is non-negative and constant with respect to the goal variables mu and
118 // sigma_sq and can thus be ignored for maximization.
119 //
120 // The function
121 //
122 //      -1 * n * log(sigma_sq) +
123 //      -1 * sigma_sq ** -1 *
124 //      sum([pv1 := 0 .. -1 + n], (-1 * mu + x(pv1) ** 2) ** 2)
125 //
126 // is then symbolically maximized w.r.t. the goal variables mu and sigma_sq.
127 // The partial differentials
128 //
129 //      df / d_mu ==
130 //      -2 * mu * n * sigma_sq ** -1 +
131 //      2 * sigma_sq ** -1 * sum([pv1 := 0 .. -1 + n], x(pv1) ** 2)
132 //      df / d_sigma_sq ==
133 //      -1 * n * sigma_sq ** -1 +
134 //      sigma_sq ** -2 *
135 //      sum([pv1 := 0 .. -1 + n], (-1 * mu + x(pv1) ** 2) ** 2)
136 //
137 // are set to zero; these equations yield the solutions
138 //
139 //      mu ==
140 //      cond(0 == n, fail(division_by_zero),
141 //          n ** -1 * sum([pv2 := 0 .. -1 + n], x(pv2) ** 2))
142 //      sigma_sq ==
143 //      cond(0 == n, fail(division_by_zero),
144 //          n ** -1 * sum([pv3 := 0 .. -1 + n], (-1 * mu + x(pv3) ** 2) ** 2)
145 //      )
146 //
147 // Initialization of common subexpression
148 for( pv2 = 0;pv2 <= n - 1;pv2++ )
149     pv4(pv2) = x(pv2) * x(pv2);
150

```

```

151  if ( 0 == n )
152      { ab_error( division_by_zero ); }
153  else
154      {
155      pv7 = 0.0;
156      for( pv2 = 0;pv2 <= n - 1;pv2++ )
157          pv7 += pv4(pv2);
158      mu = pv7 * ((double)(1) / (double)(n));
159      }
160  if ( 0 == n )
161      { ab_error( division_by_zero ); }
162  else
163      {
164      pv8 = 0.0;
165      for( pv3 = 0;pv3 <= n - 1;pv3++ )
166          pv8 += (pv4(pv3) - mu) * (pv4(pv3) - mu);
167      sigma_sq = pv8 * ((double)(1) / (double)(n));
168      }
169
170  retval.resize(2);
171  retval(0) = mu;
172  retval(1) = sigma_sq;
173
174  return retval;
175 }
176 //-- End of code

```

Listing A.2. The C++ code AUTOBAYES generated from the input file given in Figure A.2.

```

1
2 //-----
3 // Code file generated by AutoBayes V0.9.9
4 // AutoBayes(c) 2008-2011 United States Government as represented by
5 // the Administrator of NASA. AutoBayes is distributed under the NASA
6 // Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
7 // details.
8 // Problem:    LOG-NORMAL MODEL
9 // Source:     log_normal.ab
10 // Command:
11 //
12 //    PROLOG_VAR
13 //
14
15 //          -designndoc
16 //          log_normal.ab
17 // Generated: Fri Jun 26 13:38:58 2020
18 //-----
19
20 #include "autobayes.h"
21
22
23

```

```

24 //-----
25 // Octave Function: log_normal
26 //-----
27
28 DEFUN_DLD(log_normal, input_args, output_args,
29           "usage: [double mu, double sigma_sq] = log_normal(vector x)\n\n"
30           )
31 {
32   octave_value_list retval;
33   if (input_args.length () != 1 || output_args != 2 ){
34     octave_stdout << "usage: [double mu, double sigma_sq] = log_normal(vector x
35     )\n\n";
36     return retval;
37   }
38   //-- Input declarations -----
39
40   // CURRENT DATA POINTS (KNOWN)
41   octave_value arg_x = input_args(0);
42   if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
43     gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
44     return retval;
45   }
46   ColumnVector x = (ColumnVector)(arg_x.vector_value());
47
48   //-- Constant declarations -----
49
50   // NUMBER OF DATA POINTS
51   int n = arg_x.rows();
52
53   //-- Output declarations -----
54
55   // UNKNOWN MEAN
56   double mu;
57   // UNKNOWN VARIANCE
58   double sigma_sq;
59
60   //-- Local declarations -----
61
62   // Memoized common subexpression
63   // log(x(pv2))
64   ColumnVector pv4(n);
65
66   // Summation accumulator
67   // sum([pv2 := 0 .. -1 + n], pv4(pv2))
68   double pv7;
69
70   // Loop variable
71   int pv2;
72
73   // Summation accumulator
74   // sum([pv3 := 0 .. -1 + n], (-mu + pv4(pv3)) ** 2)
75   double pv8;
76

```

```

77 int pv3;
78
79
80 // The conditional probability pr(x | {mu,sigma_sq}) is under the
81 // dependencies given in the model equivalent to
82 //
83 // prod([pv1 := 0 .. -1 + n], pr(x(pv1) | {mu,sigma_sq}))
84 //
85 // The probability occurring here is atomic and can thus be replaced by the
86 // respective probability density function given in the model. This yields
87 // the log-likelihood function
88 //
89 // log(prod([pv1 := 0 .. -1 + n],
90 //          abs(deriv(log(x(pv1))), x(pv1))) *
91 //          exp(-1 / 2 * (log(x(pv1)) - mu) ** 2 /
92 //              (sigma_sq ** (1 / 2)) ** 2) *
93 //          (1 / (sqrt(2 * pi) * sigma_sq ** (1 / 2)))))
94 //
95 // which can be simplified to
96 //
97 // -1 / 2 * n * log(2) + -1 / 2 * n * log(pi) + -1 / 2 * n * log(sigma_sq)
98 // +
99 // -1 / 2 * sigma_sq ** -1 *
100 // sum([pv1 := 0 .. -1 + n], (-1 * mu + log(x(pv1))) ** 2) +
101 // sum([pv1 := 0 .. -1 + n], log(abs(x(pv1)) ** -1))
102 // This function is then optimized w.r.t. the goal variables mu and sigma_sq
103 //
104 // The summands
105 //
106 // -1 / 2 * n * log(2)
107 // -1 / 2 * n * log(pi)
108 // sum([pv1 := 0 .. -1 + n], log(abs(x(pv1)) ** -1))
109 //
110 // are constant with respect to the goal variables mu and sigma_sq and can
111 // thus be ignored for maximization.
112 //
113 // The factor
114 //
115 // 1 / 2
116 //
117 // is non-negative and constant with respect to the goal variables mu and
118 // sigma_sq and can thus be ignored for maximization.
119 //
120 // The function
121 //
122 // -1 * n * log(sigma_sq) +
123 // -1 * sigma_sq ** -1 *
124 // sum([pv1 := 0 .. -1 + n], (-1 * mu + log(x(pv1))) ** 2)
125 //
126 // is then symbolically maximized w.r.t. the goal variables mu and sigma_sq.
127 // The partial differentials
128 //

```

```

129 // df / d_mu ==
130 //   -2 * mu * n * sigma_sq ** -1 +
131 //     2 * sigma_sq ** -1 * sum([pv1 := 0 .. -1 + n], log(x(pv1)))
132 // df / d_sigma_sq ==
133 //   -1 * n * sigma_sq ** -1 +
134 //     sigma_sq ** -2 *
135 //       sum([pv1 := 0 .. -1 + n], (-1 * mu + log(x(pv1))) ** 2)
136 //
137 // are set to zero; these equations yield the solutions
138 //
139 // mu ==
140 //   cond(0 == n, fail(division_by_zero),
141 //     n ** -1 * sum([pv2 := 0 .. -1 + n], log(x(pv2))))
142 // sigma_sq ==
143 //   cond(0 == n, fail(division_by_zero),
144 //     n ** -1 * sum([pv3 := 0 .. -1 + n], (-1 * mu + log(x(pv3))) ** 2)
145 //   )
146 //
147 // Initialization of common subexpression
148 for( pv2 = 0;pv2 <= n - 1;pv2++ )
149   pv4(pv2) = safelog(x(pv2));
150
151 if ( 0 == n )
152   { ab_error( division_by_zero ); }
153 else
154   {
155     pv7 = 0.0;
156     for( pv2 = 0;pv2 <= n - 1;pv2++ )
157       pv7 += pv4(pv2);
158     mu = pv7 * ((double)(1) / (double)(n));
159   }
160 if ( 0 == n )
161   { ab_error( division_by_zero ); }
162 else
163   {
164     pv8 = 0.0;
165     for( pv3 = 0;pv3 <= n - 1;pv3++ )
166       pv8 += (pv4(pv3) - mu) * (pv4(pv3) - mu);
167     sigma_sq = pv8 * ((double)(1) / (double)(n));
168   }
169
170 retval.resize(2);
171 retval(0) = mu;
172 retval(1) = sigma_sq;
173
174 return retval;
175 }
176 //-- End of code
-----

```

Listing A.3. The C++ code AUTOBAYES generated from the input file given in Figure A.3.

```

1
2 //-----
3 // Code file generated by AutoBayes V0.9.9
4 // AutoBayes(c) 2008-2011 United States Government as represented by
5 // the Administrator of NASA. AutoBayes is distributed under the NASA
6 // Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
7 // details.
8 // Problem:   MIXTURE OF GAUSSIANS
9 // Source:    mog.ab
10 // Command:
11 //
12 //   PROLOG_VAR
13 //
14
15 //           -designndoc
16 //           mog.ab
17 // Generated: Fri Jun 26 14:21:50 2020
18 //-----
19
20 #include "autobayes.h"
21
22
23
24 //-----
25 // Octave Function: mog
26 //-----
27
28 DEFUN_DLD(mog, input_args, output_args,
29           "usage: [vector c,vector mu,vector phi,vector sigma] = mog(int
30             n_classes,vector x,double tolerance,int maxiteration)\n\n"
31           )
32 {
33   octave_value_list retval;
34   if (input_args.length () != 4 || output_args != 4 ){
35     octave_stdout << "usage: [vector c,vector mu,vector phi,vector sigma] =
36       mog(int n_classes,vector x,double tolerance,int maxiteration)\n\n";
37     return retval;
38   }
39
40   //-- Input declarations -----
41
42   // NUMBER OF CLASSES
43   octave_value arg_n_classes = input_args(0);
44   if (!arg_n_classes.is_real_scalar()){
45     gripe_wrong_type_arg("n_classes", (const std::string &)"int expected");
46     return retval;
47   }
48   int n_classes = (int)(arg_n_classes.int_value());
49
50   octave_value arg_x = input_args(1);
51   if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
52     gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
53     return retval;
54   }

```

```

53 ColumnVector x = (ColumnVector)(arg_x.vector_value());
54
55 // Iteration tolerance for convergence loop
56 octave_value arg_tolerance = input_args(2);
57 if (!arg_tolerance.is_real_scalar()){
58     gripe_wrong_type_arg("tolerance", (const std::string &)"double expected");
59     return retval;
60 }
61 double tolerance = (double)(arg_tolerance.double_value());
62
63 // maximal number of iterations
64 octave_value arg_maxiteration = input_args(3);
65 if (!arg_maxiteration.is_real_scalar()){
66     gripe_wrong_type_arg("maxiteration", (const std::string &)"int expected");
67     return retval;
68 }
69 int maxiteration = (int)(arg_maxiteration.int_value());
70
71 //-- Constant declarations -----
72
73 // NUMBER OF DATA POINTS
74 int n_points = arg_x.rows();
75
76 //-- Output declarations -----
77
78 // CLASS ASSIGNMENT VECTOR
79 ColumnVector c(n_points);
80
81 // COLUMN VECTOR OF MEANS
82 ColumnVector mu(n_classes);
83
84 // CLASS PROBABILITY VECTOR.
85 ColumnVector phi(n_classes);
86
87 // COLUMN VECTOR OF STD DEVS
88 ColumnVector sigma(n_classes);
89
90
91 //-- Local declarations -----
92
93 // Label: label0
94 // class membership table used in Discrete EM-algorithm
95 Matrix q(n_points, n_classes);
96
97 // local centers used for center-based initialization
98 Matrix center(n_classes, 1);
99
100 // Random index of data point
101 int pick;
102
103 // Loop variable
104 int pv53;
105
106 // Loop variable

```

```

107 int pv51;
108
109 // Lagrange-multiplier
110 double l;
111
112 // Loop variable
113 int pv32;
114
115 // Loop variable
116 int pv11;
117
118 // Loop variable
119 int pv22;
120
121 // Common subexpression
122 // sum([pv37 := 0 .. -1 + n_points], q(pv37, pv32))
123 double pv40;
124
125 // Memoized common subexpression
126 // exp(-1 / 2 * (x(pv11) - mu(pv42)) ** 2 / sigma(pv42) ** 2) * phi(pv42)
127 // (1 / (sigma(pv42) * sqrt(2 * pi)))
128 ColumnVector pv44(n_classes);
129
130 // Common subexpression
131 // sum([pv41 := 0 .. -1 + n_classes], pv44(pv41))
132 double pv46;
133
134 // Loop variable
135 int pv42;
136
137 // Loop variable
138 int pv61;
139
140 // Summation accumulator
141 // sum([pv54 := 0 .. -1 + n_classes],
142 //      sqrt((center(pv54, 0) - x(pv11)) ** 2))
143 double pv66;
144
145 int pv54;
146
147 ColumnVector muold(n_classes);
148
149 ColumnVector phiold(n_classes);
150
151 ColumnVector sigmaold(n_classes);
152
153 int pv56;
154
155 int pv57;
156
157 int pv58;
158
159 // convergence loop counter

```



```

160  int loopcounter;
161
162  // sum up the Diffs
163  double pv67;
164
165  // Summation accumulator
166  //  sum([pv24 := 0 .. -1 + n_points], q(pv24, pv22))
167  double pv73;
168
169  int pv24;
170
171  // Summation accumulator
172  //  sum([pv37 := 0 .. -1 + n_points], q(pv37, pv32))
173  double pv74;
174
175  int pv37;
176
177  // Summation accumulator
178  //  sum([pv36 := 0 .. -1 + n_points], x(pv36) * q(pv36, pv32))
179  double pv75;
180
181  int pv36;
182
183  // Summation accumulator
184  //  sum([pv38 := 0 .. -1 + n_points],
185  //      (-mu(pv32) + x(pv38)) ** 2 * q(pv38, pv32))
186  double pv76;
187
188  int pv38;
189
190  // Summation accumulator
191  //  sum([pv41 := 0 .. -1 + n_classes], pv44(pv41))
192  double pv81;
193
194  int pv41;
195
196  int pv69;
197
198  // Summation accumulator
199  //  sum([pv69 := 0 .. -1 + n_classes],
200  //      abs(phi(pv69) - phiold(pv69)) / (abs(phi(pv69)) + abs(phiold(pv69)))
201  //      ))
202  double pv83;
203
204  int pv68;
205
206  // Summation accumulator
207  //  sum([pv68 := 0 .. -1 + n_classes],
208  //      abs(mu(pv68) - muold(pv68)) / (abs(mu(pv68)) + abs(muold(pv68))))
209  double pv82;
210
211  // Summation accumulator
212  //  sum([pv70 := 0 .. -1 + n_classes],
213  //      abs(sigma(pv70) - sigmaold(pv70)) /

```

```

213 // (abs(sigma(pv70)) + abs(sigmaold(pv70)))
214 double pv84;
215
216 int pv70;
217
218 // Argmax index
219 int pv85;
220
221 // Argmax value
222 double pv86;
223
224 // Argmax temporary
225 double pv87;
226
227 // Argmax loop index
228 int pv64;
229
230 // Check constraints on inputs
231 ab_assert( 0 < n_classes );
232 ab_assert( 10 * n_classes < n_points );
233 ab_assert( 0 < n_points );
234
235 // Label: label1
236 // Label: label2
237 // Label: label4
238 // Discrete EM-algorithm
239 //
240 // The model describes a discrete latent (or hidden) variable problem with
241 // the latent variable c and the data variable x. The problem to optimize
242 // the conditional probability  $\text{pr}(x \mid \{\mu, \phi, \sigma\})$  w.r.t. the variables
243 // mu, phi, and sigma can thus be solved by an application of the (discrete)
244 // EM-algorithm.
245 // The algorithm maintains as central data structure a class membership
246 // table q (see "label0") such that  $q(\text{pv11}, \text{pv47})$  is the probability that
247 // data point pv11 belongs to class pv47, i.e.,
248 //
249 //  $q(\text{pv11}, \text{pv47}) == \text{pr}([c(\text{pv11}) == \text{pv47}])$ 
250 //
251 // The algorithm consists of an initialization phase for q (see "label2"),
252 // followed by a convergence phase (see "label5"), followed by the
253 // extraction of the hidden variable c (see "label6").
254 //
255 // Initialization
256 //
257 // The initialization is center-based, i.e., for each class (i.e., value of
258 // the hidden variable c) a center value center is chosen first
259 // (see "label4"). Then, the values for the local distribution are
260 // calculated as distances between the data points and these center values
261 // (see "label7").
262 //
263 // Random initialization of the centers center with data points;
264 // note that a data point can be picked as center more than once.
265 for( pv51 = 0; pv51 <= n_classes - 1; pv51++ )
266 {

```

```

267     pick = uniform_int_rnd(n_points - 1);
268     center(pv51, 0) = x(pick);
269 }
270 // Label: label7
271 for( pv11 = 0;pv11 <= n_points - 1;pv11++ )
272     for( pv53 = 0;pv53 <= n_classes - 1;pv53++ )
273     {
274         pv66 = 0.0;
275         for( pv54 = 0;pv54 <= n_classes - 1;pv54++ )
276             pv66 += sqrt((center(pv54, 0) - x(pv11)) *
277                         (center(pv54, 0) - x(pv11)));
278         q(pv11, pv53) = sqrt((center(pv53, 0) - x(pv11)) *
279                             (center(pv53, 0) - x(pv11))) / pv66;
280     }
281
282 // Label: label5
283 // EM-loop
284 //
285 // The EM-loop iterates two steps, expectation (or E-Step) (see "label8"),
286 // and maximization (or M-Step) (see "label9"); however, due to the form of
287 // the initialization used here, the are ordered the other way around. The
288 // loop runs until convergence in the values of the variables mu, phi, and
289 // sigma is achieved.
290 //
291 // Tolerance value must be positive
292 ab_assert( tolerance > 0 );
293 // max nr of iterations must be positive
294 ab_assert( maxiteration > 0 );
295 loopcounter = 0;
296 // repeat at least once
297 pv67 = tolerance;
298 while( ((loopcounter < maxiteration) && (pv67 >= tolerance)) )
299     {
300         loopcounter = 1 + loopcounter;
301         if ( loopcounter > 1 )
302             {
303                 // assign current values to old values
304                 for( pv56 = 0;pv56 <= n_classes - 1;pv56++ )
305                     muold(pv56) = mu(pv56);
306                 // assign current values to old values
307                 for( pv57 = 0;pv57 <= n_classes - 1;pv57++ )
308                     phiold(pv57) = phi(pv57);
309                 // assign current values to old values
310                 for( pv58 = 0;pv58 <= n_classes - 1;pv58++ )
311                     sigmaold(pv58) = sigma(pv58);
312             }
313         else
314             ;
315
316         // Label: label9
317         // Label: label3
318         // M-Step
319         //
320         // Decomposition I

```

```

321 //
322 // The problem to optimize the conditional probability  $\text{pr}(\{c,x\} |$ 
323 //  $\{\mu,\phi,\sigma\})$  w.r.t. the variables  $\mu$ ,  $\phi$ , and  $\sigma$  can under the
324 // given dependencies by Bayes rule be decomposed into two independent
325 // subproblems:
326 //
327 //  $\max \text{pr}(c | \phi)$  for  $\phi$ 
328 //  $\max \text{pr}(x | \{c,\mu,\sigma\})$  for  $\{\mu,\sigma\}$ 
329 //
330 //
331 // The conditional probability  $\text{pr}(c | \phi)$  is under the dependencies
332 // given in the model equivalent to
333 //
334 //  $\text{prod}([pv15 := 0 .. -1 + n\_points], \text{pr}(c(pv15) | \phi))$ 
335 //
336 // The probability occurring here is atomic and can thus be replaced by
337 // the respective probability density function given in the model.
338 // Summing out the expected variable  $c(pv11)$  yields the log-likelihood
339 // function
340 //
341 //  $\text{sum\_domain}([pv11 := 0 .. -1 + n\_points],$ 
342 //  $[pv16 := 0 .. -1 + n\_classes], [c(pv11)], q(pv11, pv16),$ 
343 //  $\log(\text{prod}([pv15 := 0 .. -1 + n\_points], \phi(c(pv15))))$ 
344 //
345 // which can be simplified to
346 //
347 //  $\text{sum}([pv16 := 0 .. -1 + n\_classes],$ 
348 //  $\log(\phi(pv16)) *$ 
349 //  $\text{sum}([pv15 := 0 .. -1 + n\_points], q(pv15, pv16)))$ 
350 //
351 // This function is then optimized w.r.t. the goal variable  $\phi$ .
352 //
353 // The expression
354 //
355 //  $\text{sum}([pv16 := 0 .. -1 + n\_classes],$ 
356 //  $\log(\phi(pv16)) *$ 
357 //  $\text{sum}([pv15 := 0 .. -1 + n\_points], q(pv15, pv16)))$ 
358 //
359 // is maximized w.r.t. the variable  $\phi$  under the constraint
360 //
361 //  $0 == -1 + \text{sum}([pv21 := 0 .. -1 + n\_classes], \phi(pv21))$ 
362 //
363 // using the Lagrange-multiplier  $l$ .
364 l = (double)(n_points);
365 for( pv22 = 0;pv22 <= n_classes - 1;pv22++ )
366 // The summand
367 //
368 //  $l$ 
369 //
370 // is constant with respect to the goal variable  $\phi(pv22)$  and can
371 // thus be ignored for maximization.
372 //
373 // The function
374 //

```

```

375 // -1 * l * sum([pv21 := 0 .. -1 + n_classes], phi(pv21)) +
376 //   sum([pv16 := 0 .. -1 + n_classes],
377 //     log(phi(pv16)) *
378 //     sum([pv15 := 0 .. -1 + n_points], q(pv15, pv16)))
379 //
380 // is then symbolically maximized w.r.t. the goal variable phi(pv22).
381 // The differential
382 //
383 // -1 * l +
384 //   phi(pv22) ** -1 * sum([pv15 := 0 .. -1 + n_points], q(pv15, pv22)
))
385 //
386 // is set to zero; this equation yields the solution
387 //
388 // l ** -1 * sum([pv24 := 0 .. -1 + n_points], q(pv24, pv22))
389 //
390 {
391   pv73 = 0.0;
392   for( pv24 = 0;pv24 <= n_points - 1;pv24++ )
393     pv73 += q(pv24, pv22);
394   phi(pv22) = pv73 / l;
395 }
396
397 // The conditional probability pr(x | {c,mu,sigma}) is under the
398 // dependencies given in the model equivalent to
399 //
400 // prod([pv28 := 0 .. -1 + n_points], pr(x(pv28) | {c(pv28),mu,sigma})
)
401 //
402 // The probability occurring here is atomic and can thus be replaced by
403 // the respective probability density function given in the model.
404 // Summing out the expected variable c(pv11) yields the log-likelihood
405 // function
406 //
407 // sum_domain([pv11 := 0 .. -1 + n_points],
408 //   [pv29 := 0 .. -1 + n_classes], [c(pv11)], q(pv11, pv29),
409 //   log(prod([pv28 := 0 .. -1 + n_points],
410 //     exp(-1 / 2 * (x(pv28) - mu(c(pv28))) ** 2 /
411 //       sigma(c(pv28)) ** 2) *
412 //     (1 / (sigma(c(pv28)) * sqrt(2 * pi))))))
413 //
414 // which can be simplified to
415 //
416 // -1 *
417 //   sum([pv29 := 0 .. -1 + n_classes],
418 //     log(sigma(pv29)) *
419 //     sum([pv28 := 0 .. -1 + n_points], q(pv28, pv29))) +
420 // -1 / 2 * n_points * log(2) + -1 / 2 * n_points * log(pi) +
421 // -1 / 2 *
422 //   sum([pv29 := 0 .. -1 + n_classes],
423 //     sigma(pv29) ** -2 *
424 //     sum([pv28 := 0 .. -1 + n_points],
425 //       (-1 * mu(pv29) + x(pv28)) ** 2 * q(pv28, pv29)))
426 //

```

```

427 // This function is then optimized w.r.t. the goal variables mu and
428 // sigma.
429 //
430 // The summands
431 //
432 // -1 / 2 * n_points * log(2)
433 // -1 / 2 * n_points * log(pi)
434 //
435 // are constant with respect to the goal variables mu and sigma and can
436 // thus be ignored for maximization.
437 //
438 // Index decomposition
439 //
440 // The function
441 //
442 // -1 *
443 //   sum([pv29 := 0 .. -1 + n_classes],
444 //     log(sigma(pv29)) *
445 //     sum([pv28 := 0 .. -1 + n_points], q(pv28, pv29))) +
446 // -1 / 2 *
447 //   sum([pv29 := 0 .. -1 + n_classes],
448 //     sigma(pv29) ** -2 *
449 //     sum([pv28 := 0 .. -1 + n_points],
450 //       (-1 * mu(pv29) + x(pv28)) ** 2 * q(pv28, pv29)))
451 //
452 // can be optimized w.r.t. the variables mu(pv32) and sigma(pv32)
453 // element by element (i.e., along the index variable pv32) because
454 // there are no dependencies along that dimension.
455 for( pv32 = 0;pv32 <= n_classes - 1;pv32++ )
456 // The factor
457 //
458 // n_classes
459 //
460 // is non-negative and constant with respect to the goal variables
461 // mu(pv32) and sigma(pv32) and can thus be ignored for maximization.
462 //
463 // The function
464 //
465 // -1 * log(sigma(pv32)) *
466 //   sum([pv28 := 0 .. -1 + n_points], q(pv28, pv32)) +
467 // -1 / 2 * sigma(pv32) ** -2 *
468 //   sum([pv28 := 0 .. -1 + n_points],
469 //     (-1 * mu(pv32) + x(pv28)) ** 2 * q(pv28, pv32))
470 //
471 // is then symbolically maximized w.r.t. the goal variables mu(pv32)
472 // and sigma(pv32). The partial differentials
473 //
474 // df / d_mu(pvar(32)) ==
475 // -1 * mu(pv32) * sigma(pv32) ** -2 *
476 //   sum([pv28 := 0 .. -1 + n_points], q(pv28, pv32)) +
477 //   sigma(pv32) ** -2 *
478 //   sum([pv28 := 0 .. -1 + n_points], x(pv28) * q(pv28, pv32))
479 // df / d_sigma(pvar(32)) ==
480 // -1 * sigma(pv32) ** -1 *

```

```

481 //      sum([pv28 := 0 .. -1 + n_points], q(pv28, pv32)) +
482 //      sigma(pv32) ** -3 *
483 //      sum([pv28 := 0 .. -1 + n_points],
484 //          (-1 * mu(pv32) + x(pv28)) ** 2 * q(pv28, pv32))
485 //
486 // are set to zero; these equations yield the solutions
487 //
488 //      mu(pv32) ==
489 //      cond(0 == sum([pv34 := 0 .. -1 + n_points], q(pv34, pv32)),
490 //          fail(division_by_zero),
491 //          sum([pv35 := 0 .. -1 + n_points], q(pv35, pv32)) ** -1 *
492 //          sum([pv36 := 0 .. -1 + n_points], x(pv36) * q(pv36, pv32))
493 //
494 //      sigma(pv32) ==
495 //      cond(0 == sum([pv37 := 0 .. -1 + n_points], q(pv37, pv32)),
496 //          fail(division_by_zero),
497 //          sum([pv38 := 0 .. -1 + n_points],
498 //              (-1 * mu(pv32) + x(pv38)) ** 2 * q(pv38, pv32)) **
499 //              (1 / 2) *
500 //              sum([pv39 := 0 .. -1 + n_points], q(pv39, pv32)) **
501 //              (-1 / 2))
502 {
503 // Initialization of common subexpression
504 pv74 = 0.0;
505 for( pv37 = 0;pv37 <= n_points - 1;pv37++ )
506     pv74 += q(pv37, pv32);
507 pv40 = pv74;
508
509 if ( 0 == pv40 )
510     { ab_error( division_by_zero ); }
511 else
512     {
513         pv75 = 0.0;
514         for( pv36 = 0;pv36 <= n_points - 1;pv36++ )
515             pv75 += x(pv36) * q(pv36, pv32);
516         mu(pv32) = pv75 * ((double)(1) / pv40);
517     }
518 if ( 0 == pv40 )
519     { ab_error( division_by_zero ); }
520 else
521     {
522         pv76 = 0.0;
523         for( pv38 = 0;pv38 <= n_points - 1;pv38++ )
524             pv76 += (x(pv38) - mu(pv32)) * (x(pv38) - mu(pv32)) *
525                 q(pv38, pv32);
526         sigma(pv32) = sqrt(pv76) * ((double)(1) / sqrt(pv40));
527     }
528 }
529 // Label: label8
530 // E-Step
531 // Update the current values of the class membership table q.
532 for( pv11 = 0;pv11 <= n_points - 1;pv11++ )
533 {

```

```

534 // Initialization of common subexpression
535 for( pv42 = 0;pv42 <= n_classes - 1;pv42++ )
536     pv44(pv42) = exp(-0.5 * (x(pv11) - mu(pv42)) *
537                    (x(pv11) - mu(pv42)) /
538                    (sigma(pv42) * sigma(pv42))) * phi(pv42) *
539                    ((double)(1) /
540                    (sigma(pv42) * sqrt(M_PI * (double)(2))));
541
542     pv81 = 0.0;
543     for( pv41 = 0;pv41 <= n_classes - 1;pv41++ )
544         pv81 += pv44(pv41);
545     pv46 = pv81;
546     for( pv61 = 0;pv61 <= n_classes - 1;pv61++ )
547         // The denominator pv46 can become zero due to round-off errors.
548         // In that case, each class is considered to be equally likely.
549         if ( pv46 == 0.0 )
550             q(pv11, pv61) = (double)(1) / (double)(n_classes);
551         else
552             q(pv11, pv61) = pv44(pv61) / pv46;
553     }
554     if( loopcounter > 1 )
555     {
556         pv82 = 0.0;
557         for( pv68 = 0;pv68 <= n_classes - 1;pv68++ )
558             pv82 += abs(mu(pv68) - muold(pv68)) /
559                    (abs(mu(pv68)) + abs(muold(pv68)));
560
561         pv83 = 0.0;
562         for( pv69 = 0;pv69 <= n_classes - 1;pv69++ )
563             pv83 += abs(phi(pv69) - phiold(pv69)) /
564                    (abs(phi(pv69)) + abs(phiold(pv69)));
565
566         pv84 = 0.0;
567         for( pv70 = 0;pv70 <= n_classes - 1;pv70++ )
568             pv84 += abs(sigma(pv70) - sigmaold(pv70)) /
569                    (abs(sigma(pv70)) + abs(sigmaold(pv70)));
570         pv67 = pv82 + pv83 + pv84;
571     }
572     else
573         ;
574 }
575 // Label: label6
576 // Extract the most likely values of the hidden variable c(pv11) from the
577 // class membership table q.
578 for( pv11 = 0;pv11 <= n_points - 1;pv11++ )
579     {
580         // Determine the position of the maximum with in the range
581         // 0
582         // ...
583         // -1 + n_classes
584         // by iterating over this range and calculating the value at each point
585         // (argmax).
586         //
587         // Argmax loop

```



```

588     for( pv64 = 0;pv64 <= n_classes - 1;pv64++ )
589     {
590         pv87 = q(pv11, pv64);
591         if ( ((pv64 == 0) || (pv87 > pv86)) )
592             // Save new maximum
593             {
594                 pv86 = pv87;
595                 pv85 = pv64;
596             }
597         else
598             ;
599     }
600     c(pv11) = pv85;
601 }
602
603 retval.resize(4);
604 retval(0) = c;
605 retval(1) = mu;
606 retval(2) = phi;
607 retval(3) = sigma;
608
609 return retval;
610 }
611 //-- End of code

```

Listing A.4. The C++ code AUTOBAYES generated from the input file given in Figure A.4.

```

1
2 //-----
3 // Code file generated by AutoBayes V0.9.9
4 // AutoBayes(c) 2008-2011 United States Government as represented by
5 // the Administrator of NASA. AutoBayes is distributed under the NASA
6 // Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
7 // details.
8 // Problem:   SIMPLE MULTIVARIATE CLUSTERING MODEL
9 // Source:    mult_cluster.ab
10 // Command:
11 //
12 //   PROLOG_VAR
13 //
14
15 //           -designndoc
16 //           -instrument
17 //           -pragma em_log_likelihood_convergence=true
18 //           mult_cluster.ab
19 // Generated: Fri Jun 26 15:51:13 2020
20 //-----
21
22 #include "autobayes.h"
23
24
25

```

```

26 //-----
27 // Octave Function: mult_cluster
28 //-----
29
30 DEFUN_DLD(mult_cluster, input_args, output_args,
31           "usage: [vector class_assignment, matrix mu, vector phi, matrix sigma,
           double loglikelihood, vector errors] = mult_cluster(int n_classes, matrix
           sim_data, double tolerance, int maxiteration)\n\n"
32           )
33 {
34 octave_value_list retval;
35 if (input_args.length () != 4 || output_args != 6 ){
36     octave_stdout << "usage: [vector class_assignment, matrix mu, vector phi,
           matrix sigma, double loglikelihood, vector errors] = mult_cluster(int
           n_classes, matrix sim_data, double tolerance, int maxiteration)\n\n";
37     return retval;
38 }
39
40 //-- Input declarations -----
41
42 // NUMBER OF CLASSES
43 octave_value arg_n_classes = input_args(0);
44 if (!arg_n_classes.is_real_scalar()){
45     gripe_wrong_type_arg("n_classes", (const std::string &)"int expected");
46     return retval;
47 }
48 int n_classes = (int)(arg_n_classes.int_value());
49
50 octave_value arg_sim_data = input_args(1);
51 if (!arg_sim_data.is_real_matrix()){
52     gripe_wrong_type_arg("sim_data", (const std::string &)"Matrix expected");
53     return retval;
54 }
55 Matrix sim_data = (Matrix)(arg_sim_data.matrix_value());
56
57 // Iteration tolerance for convergence loop
58 octave_value arg_tolerance = input_args(2);
59 if (!arg_tolerance.is_real_scalar()){
60     gripe_wrong_type_arg("tolerance", (const std::string &)"double expected");
61     return retval;
62 }
63 double tolerance = (double)(arg_tolerance.double_value());
64
65 // maximal number of iterations
66 octave_value arg_maxiteration = input_args(3);
67 if (!arg_maxiteration.is_real_scalar()){
68     gripe_wrong_type_arg("maxiteration", (const std::string &)"int expected");
69     return retval;
70 }
71 int maxiteration = (int)(arg_maxiteration.int_value());
72
73 //-- Constant declarations -----
74
75 // NUMBER OF DATA POINTS

```

```

76  int n_points = arg_sim_data.columns();
77
78  // NUMBER OF VARIABLES
79  int n_variables = arg_sim_data.rows();
80
81  //-- Output declarations -----
82
83  // HIDDEN VARIABLE
84  ColumnVector class_assignment(n_points);
85
86  // COLUMN VECTOR OF MEANS
87  Matrix mu(n_variables, n_classes);
88
89  // CLASS PROBABILITY VECTOR.
90  ColumnVector phi(n_classes);
91
92  // COLUMN VECTOR OF STD DEVS
93  Matrix sigma(n_variables, n_classes);
94
95  // log likelihood
96  double loglikelihood;
97  // instrumentation: assembly of convergence data
98  ColumnVector errors(1000);
99
100
101  //-- Local declarations -----
102
103  // Label: label0
104  // class membership table used in Discrete EM-algorithm
105  Matrix q(n_points, n_classes);
106
107  // local centers used for center-based initialization
108  Matrix center(n_classes, n_variables);
109
110  // Random index of data point
111  int pick;
112
113  // Loop variable
114  int pv74;
115
116  // Loop variable
117  int pv71;
118
119  // Loop variable
120  int pv72;
121
122  // Lagrange-multiplier
123  double l;
124
125  // Loop variable
126  int pv43;
127
128  // Loop variable
129  int pv44;

```

```

130
131 // Loop variable
132 int pv14;
133
134 // Loop variable
135 int pv25;
136
137 // Common subexpression
138 // sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))
139 double pv53;
140
141 // Memoized common subexpression
142 // phi(pv57) *
143 // prod([pv54 := 0 .. -1 + n_variables],
144 //      exp(-1 / 2 * (sim_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
145 //          sigma(pv54, pv57) ** 2) *
146 //          (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
147 ColumnVector pv59(n_classes);
148
149 // Common subexpression
150 // sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
151 double pv61;
152
153 // Loop variable
154 int pv57;
155
156 // Loop variable
157 int pv81;
158
159 int pv76;
160
161 // Summation accumulator
162 // sum([pv76 := 0 .. -1 + n_classes], sqrt(pv87))
163 double pv88;
164
165 // Summation accumulator
166 // sum([pv75 := 0 .. -1 + n_variables],
167 //      (center(pv74, pv75) - sim_data(pv75, pv14)) ** 2)
168 double pv86;
169
170 int pv75;
171
172 // Summation accumulator
173 // sum([pv77 := 0 .. -1 + n_variables],
174 //      (center(pv76, pv77) - sim_data(pv77, pv14)) ** 2)
175 double pv87;
176
177 int pv77;
178
179 double pv89;
180
181 // convergence loop counter
182 int loopcounter;
183

```

```

184 // sum up the Diffs
185 double pv90;
186
187 // Summation accumulator
188 // sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
189 double pv93;
190
191 int pv27;
192
193 // Summation accumulator
194 // sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))
195 double pv94;
196
197 int pv50;
198
199 // Summation accumulator
200 // sum([pv49 := 0 .. -1 + n_points], q(pv49, pv44) * sim_data(pv43, pv49))
201 double pv95;
202
203 int pv49;
204
205 // Summation accumulator
206 // sum([pv51 := 0 .. -1 + n_points],
207 //      (-mu(pv43, pv44) + sim_data(pv43, pv51)) ** 2 * q(pv51, pv44))
208 double pv96;
209
210 int pv51;
211
212 // Product accumulator
213 // sum([pv54 := 0 .. -1 + n_variables],
214 //      exp(-1 / 2 * (sim_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
215 //          sigma(pv54, pv57) ** 2) *
216 //      (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
217 double pv102;
218
219 int pv54;
220
221 // Summation accumulator
222 // sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
223 double pv103;
224
225 int pv56;
226
227 // Summation accumulator
228 // sum([pv19 := 0 .. -1 + n_classes], log(phi(pv19)) * pv109)
229 double pv110;
230
231 // Summation accumulator
232 // sum([pv35 := 0 .. -1 + n_classes], pv104 * pv105)
233 double pv106;
234
235 // Summation accumulator
236 // sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
237 //      pv107 * q(pv34, pv35))

```

```

238 double pv108;
239
240 // Summation accumulator
241 // sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35)))
242 double pv104;
243
244 // Summation accumulator
245 // sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))
246 double pv105;
247
248 // Summation accumulator
249 // sum([pv33 := 0 .. -1 + n_variables],
250 //      (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
251 //      sigma(pv33, pv35) ** -2)
252 double pv107;
253
254 // Summation accumulator
255 // sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19))
256 double pv109;
257
258 // Argmax index
259 int pv111;
260
261 // Argmax value
262 double pv112;
263
264 // Argmax temporary
265 double pv113;
266
267 // Argmax loop index
268 int pv84;
269
270 int pv19;
271
272 // Summation accumulator
273 // sum([pv19 := 0 .. -1 + n_classes], log(phi(pv19)) * pv119)
274 double pv120;
275
276 // Summation accumulator
277 // sum([pv35 := 0 .. -1 + n_classes], pv114 * pv115)
278 double pv116;
279
280 // Summation accumulator
281 // sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
282 //      pv117 * q(pv34, pv35))
283 double pv118;
284
285 int pv35;
286
287 // Summation accumulator
288 // sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35)))
289 double pv114;
290
291 // Summation accumulator

```

```

292 // sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))
293 double pv115;
294
295 int pv34;
296
297 // Summation accumulator
298 // sum([pv33 := 0 .. -1 + n_variables],
299 //      (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
300 //      sigma(pv33, pv35) ** -2)
301 double pv117;
302
303 int pv33;
304
305 // Summation accumulator
306 // sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19))
307 double pv119;
308
309 int pv18;
310
311 // Check constraints on inputs
312 ab_assert( 0 < n_classes );
313 ab_assert( 10 * n_classes < n_points );
314
315 // Label: label0
316 // Label: label0
317 // Label: label0
318 // Discrete EM-algorithm
319 //
320 // The model describes a discrete latent (or hidden) variable problem with
321 // the latent variable class_assignment and the data variable sim_data. The
322 // problem to optimize the conditional probability pr(sim_data |
323 // {mu,phi,sigma}) w.r.t. the variables mu, phi, and sigma can thus be
324 // solved by an application of the (discrete) EM-algorithm.
325 // The algorithm maintains as central data structure a class membership
326 // table q (see "label0") such that q(pv14,pv67) is the probability that
327 // data point pv14 belongs to class pv67, i.e.,
328 //
329 // q(pv14, pv67) == pr([class_assignment(pv14) == pv67])
330 //
331 // The algorithm consists of an initialization phase for q (see "label0"),
332 // followed by a convergence phase (see "label0"), followed by the
333 // extraction of the hidden variable class_assignment (see "label0").
334 //
335 // Initialization
336 //
337 // The initialization is center-based, i.e., for each class (i.e., value of
338 // the hidden variable class_assignment) a center value center is chosen
339 // first (see "label0"). Then, the values for the local distribution are
340 // calculated as distances between the data points and these center values
341 // (see "label0").
342 //
343 // Random initialization of the centers center with data points;
344 // note that a data point can be picked as center more than once.
345 for( pv71 = 0;pv71 <= n_classes - 1;pv71++ )

```

```

346     {
347         pick = uniform_int_rnd(n_points - 1);
348         for( pv72 = 0;pv72 <= n_variables - 1;pv72++ )
349             center(pv71, pv72) = sim_data(pv72, pick);
350     }
351     // Label: label0
352     for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
353         for( pv74 = 0;pv74 <= n_classes - 1;pv74++ )
354             {
355                 pv86 = 0.0;
356                 for( pv75 = 0;pv75 <= n_variables - 1;pv75++ )
357                     pv86 += (center(pv74, pv75) - sim_data(pv75, pv14)) *
358                             (center(pv74, pv75) - sim_data(pv75, pv14));
359
360                 pv88 = 0.0;
361                 for( pv76 = 0;pv76 <= n_classes - 1;pv76++ )
362                     {
363                         pv87 = 0.0;
364                         for( pv77 = 0;pv77 <= n_variables - 1;pv77++ )
365                             pv87 += (center(pv76, pv77) - sim_data(pv77, pv14)) *
366                                     (center(pv76, pv77) - sim_data(pv77, pv14));
367                         pv88 += sqrt(pv87);
368                     }
369                 q(pv14, pv74) = sqrt(pv86) / pv88;
370             }
371     // resize vector to maximal size
372     errors.resize(1000);
373     // initialize convergence output
374     for( loopcounter = 0;loopcounter <= 999;loopcounter++ )
375         errors(loopcounter) = 0;
376     // Tolerance value must be positive
377     ab_assert( tolerance > 0 );
378     // max nr of iterations must be positive
379     ab_assert( maxiteration > 0 );
380     loopcounter = 0;
381     // repeat at least once
382     pv90 = tolerance;
383     while( ((loopcounter < maxiteration) && (pv90 >= tolerance)) )
384         {
385             loopcounter = 1 + loopcounter;
386             if ( loopcounter > 1 )
387                 // assign current values to old values
388                 pv89 = loglikelihood;
389             else
390                 ;
391
392             // Label: label0
393             // Label: label0
394             // M-Step
395             //
396             // Decomposition I
397             //
398             // The problem to optimize the conditional probability
399             // pr({class_assignment, sim_data} | {mu, phi, sigma}) w.r.t. the variables

```



```

400 // mu, phi, and sigma can under the given dependencies by Bayes rule be
401 // decomposed into two independent subproblems:
402 //
403 // max pr(class_assignment | phi) for phi
404 // max pr(sim_data | {class_assignment,mu,sigma}) for {mu,sigma}
405 //
406 //
407 // The conditional probability pr(class_assignment | phi) is under the
408 // dependencies given in the model equivalent to
409 //
410 // prod([pv18 := 0 .. -1 + n_points], pr(class_assignment(pv18) | phi)
)
411 //
412 // The probability occurring here is atomic and can thus be replaced by
413 // the respective probability density function given in the model.
414 // Summing out the expected variable class_assignment(pv14) yields the
415 // log-likelihood function
416 //
417 // sum_domain([pv14 := 0 .. -1 + n_points],
418 //            [pv19 := 0 .. -1 + n_classes], [class_assignment(pv14)],
419 //            q(pv14, pv19),
420 //            log(prod([pv18 := 0 .. -1 + n_points],
421 //                    phi(class_assignment(pv18)))))
422 //
423 // which can be simplified to
424 //
425 // sum([pv19 := 0 .. -1 + n_classes],
426 //      log(phi(pv19)) *
427 //      sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
428 //
429 // This function is then optimized w.r.t. the goal variable phi.
430 //
431 // The expression
432 //
433 // sum([pv19 := 0 .. -1 + n_classes],
434 //      log(phi(pv19)) *
435 //      sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
436 //
437 // is maximized w.r.t. the variable phi under the constraint
438 //
439 // 0 == -1 + sum([pv24 := 0 .. -1 + n_classes], phi(pv24))
440 //
441 // using the Lagrange-multiplier l.
442 l = (double)(n_points);
443 for( pv25 = 0;pv25 <= n_classes - 1;pv25++ )
444 // The summand
445 //
446 // l
447 //
448 // is constant with respect to the goal variable phi(pv25) and can
449 // thus be ignored for maximization.
450 //
451 // The function
452 //

```

```

453 // -1 * l * sum([pv24 := 0 .. -1 + n_classes], phi(pv24)) +
454 //   sum([pv19 := 0 .. -1 + n_classes],
455 //     log(phi(pv19)) *
456 //     sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
457 //
458 // is then symbolically maximized w.r.t. the goal variable phi(pv25).
459 // The differential
460 //
461 // -1 * l +
462 //   phi(pv25) ** -1 * sum([pv18 := 0 .. -1 + n_points], q(pv18, pv25
))
463 //
464 // is set to zero; this equation yields the solution
465 //
466 // l ** -1 * sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
467 //
468 {
469   pv93 = 0.0;
470   for( pv27 = 0;pv27 <= n_points - 1;pv27++ )
471     pv93 += q(pv27, pv25);
472   phi(pv25) = pv93 / l;
473 }
474
475 // The conditional probability pr(sim_data |
476 // {class_assignment,mu,sigma}) is under the dependencies given in the
477 // model equivalent to
478 //
479 // prod([pv33 := 0 .. -1 + n_variables, pv34 := 0 .. -1 + n_points],
480 //   pr(sim_data(pv33,pv34) | {class_assignment(pv34),mu(pv33,*),
sigma(pv33,*)}))
481 //
482 // The probability occurring here is atomic and can thus be replaced by
483 // the respective probability density function given in the model.
484 // Summing out the expected variable class_assignment(pv14) yields the
485 // log-likelihood function
486 //
487 //   sum_domain([pv14 := 0 .. -1 + n_points],
488 //     [pv35 := 0 .. -1 + n_classes], [class_assignment(pv14)],
489 //     q(pv14, pv35),
490 //     log(prod([pv33 := 0 .. -1 + n_variables,
491 //       pv34 := 0 .. -1 + n_points],
492 //         exp(-1 / 2 *
493 //           (sim_data(pv33, pv34) -
494 //             mu(pv33, class_assignment(pv34))) ** 2 /
495 //             sigma(pv33, class_assignment(pv34)) ** 2)
*
496 //           (1 /
497 //             (sqrt(2 * pi) *
498 //               sigma(pv33, class_assignment(pv34)))))))))
499 //
500 // which can be simplified to
501 //
502 // -1 *
503 //   sum([pv35 := 0 .. -1 + n_classes],

```

```

504 //      sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35))) *
505 //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
506 //      -1 / 2 * n_points * n_variables * log(2) +
507 //      -1 / 2 * n_points * n_variables * log(pi) +
508 //      -1 / 2 *
509 //      sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
510 //      q(pv34, pv35) *
511 //      sum([pv33 := 0 .. -1 + n_variables],
512 //      (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
513 //      sigma(pv33, pv35) ** -2))
514 //
515 // This function is then optimized w.r.t. the goal variables mu and
516 // sigma.
517 //
518 // The summands
519 //
520 //      -1 / 2 * n_points * n_variables * log(2)
521 //      -1 / 2 * n_points * n_variables * log(pi)
522 //
523 // are constant with respect to the goal variables mu and sigma and can
524 // thus be ignored for maximization.
525 //
526 // Index decomposition
527 //
528 // The function
529 //
530 //      -1 *
531 //      sum([pv35 := 0 .. -1 + n_classes],
532 //      sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35))) *
533 //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
534 //      -1 / 2 *
535 //      sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
536 //      q(pv34, pv35) *
537 //      sum([pv33 := 0 .. -1 + n_variables],
538 //      (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
539 //      sigma(pv33, pv35) ** -2))
540 //
541 // can be optimized w.r.t. the variables mu(pv43,pv44) and
542 // sigma(pv43,pv44) element by element (i.e., along the index variables
543 // pv43 and pv44) because there are no dependencies along those
544 // dimensions.
545 for( pv43 = 0;pv43 <= n_variables - 1;pv43++ )
546   for( pv44 = 0;pv44 <= n_classes - 1;pv44++ )
547     // The factor
548     //
549     //      n_classes
550     //
551     // is non-negative and constant with respect to the goal variables
552     // mu(pv43,pv44) and sigma(pv43,pv44) and can thus be ignored for
553     // maximization.
554     //
555     // The function
556     //
557     //      -1 * n_variables * log(sigma(pv43, pv44)) *

```

```

558 //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
559 //      -1 / 2 * n_variables * sigma(pv43, pv44) ** -2 *
560 //      sum([pv34 := 0 .. -1 + n_points],
561 //          (-1 * mu(pv43, pv44) + sim_data(pv43, pv34)) ** 2 *
562 //          q(pv34, pv44))
563 //
564 // is then symbolically maximized w.r.t. the goal variables
565 // mu(pv43,pv44) and sigma(pv43,pv44). The partial differentials
566 //
567 // df / d_mu(pvar(43),pvar(44)) ==
568 //      -1 * n_variables * sigma(pv43, pv44) ** -2 * mu(pv43, pv44) *
569 //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
570 //      n_variables * sigma(pv43, pv44) ** -2 *
571 //      sum([pv34 := 0 .. -1 + n_points],
572 //          q(pv34, pv44) * sim_data(pv43, pv34))
573 // df / d_sigma(pvar(43),pvar(44)) ==
574 //      -1 * n_variables * sigma(pv43, pv44) ** -1 *
575 //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
576 //      n_variables * sigma(pv43, pv44) ** -3 *
577 //      sum([pv34 := 0 .. -1 + n_points],
578 //          (-1 * mu(pv43, pv44) + sim_data(pv43, pv34)) ** 2 *
579 //          q(pv34, pv44))
580 //
581 // are set to zero; these equations yield the solutions
582 //
583 // mu(pv43, pv44) ==
584 //      cond(0 == n_variables or
585 //          0 == sum([pv47 := 0 .. -1 + n_points], q(pv47, pv44)),
586 //          fail(division_by_zero),
587 //          sum([pv48 := 0 .. -1 + n_points], q(pv48, pv44)) ** -1 *
588 //          sum([pv49 := 0 .. -1 + n_points],
589 //              q(pv49, pv44) * sim_data(pv43, pv49)))
590 // sigma(pv43, pv44) ==
591 //      cond(0 == n_variables or
592 //          0 == sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44)),
593 //          fail(division_by_zero),
594 //          abs(n_variables) * n_variables ** -1 *
595 //          sum([pv51 := 0 .. -1 + n_points],
596 //              (-1 * mu(pv43, pv44) + sim_data(pv43, pv51)) ** 2 *
597 //              q(pv51, pv44)) ** (1 / 2) *
598 //          sum([pv52 := 0 .. -1 + n_points], q(pv52, pv44)) **
599 //          (-1 / 2))
600 //
601 {
602 // Initialization of common subexpression
603 pv94 = 0.0;
604 for( pv50 = 0;pv50 <= n_points - 1;pv50++ )
605     pv94 += q(pv50, pv44);
606 pv53 = pv94;
607
608 if ( ((0 == n_variables) || (0 == pv53)) )
609     { ab_error( division_by_zero ); }
610 else
611     {

```

```

612         pv95 = 0.0;
613         for( pv49 = 0;pv49 <= n_points - 1;pv49++ )
614             pv95 += q(pv49, pv44) * sim_data(pv43, pv49);
615         mu(pv43, pv44) = pv95 * ((double)(1) / pv53);
616     }
617     if ( ((0 == n_variables) || (0 == pv53)) )
618         { ab_error( division_by_zero ); }
619     else
620         {
621             pv96 = 0.0;
622             for( pv51 = 0;pv51 <= n_points - 1;pv51++ )
623                 pv96 += (sim_data(pv43, pv51) - mu(pv43, pv44)) *
624                     (sim_data(pv43, pv51) - mu(pv43, pv44)) *
625                     q(pv51, pv44);
626             sigma(pv43, pv44) = abs(n_variables) * sqrt(pv96) *
627                 ((double)(1) / (double)(n_variables)) *
628                 ((double)(1) / sqrt(pv53));
629         }
630     }
631     // Label: label0
632     // E-Step
633     // Update the current values of the class membership table q.
634     for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
635     {
636         // Initialization of common subexpression
637         for( pv57 = 0;pv57 <= n_classes - 1;pv57++ )
638         {
639             pv102 = 1.0;
640             for( pv54 = 0;pv54 <= n_variables - 1;pv54++ )
641                 pv102 *= exp(-0.5 * (sim_data(pv54, pv14) - mu(pv54, pv57)) *
642                     (sim_data(pv54, pv14) - mu(pv54, pv57)) /
643                     (sigma(pv54, pv57) * sigma(pv54, pv57))) *
644                     ((double)(1) /
645                     (sqrt(M_PI * (double)(2)) * sigma(pv54, pv57)));
646             pv59(pv57) = phi(pv57) * pv102;
647         }
648
649         pv103 = 0.0;
650         for( pv56 = 0;pv56 <= n_classes - 1;pv56++ )
651             pv103 += pv59(pv56);
652         pv61 = pv103;
653         for( pv81 = 0;pv81 <= n_classes - 1;pv81++ )
654             // The denominator pv61 can become zero due to round-off errors.
655             // In that case, each class is considered to be equally likely.
656             if ( pv61 == 0.0 )
657                 q(pv14, pv81) = (double)(1) / (double)(n_classes);
658             else
659                 q(pv14, pv81) = pv59(pv81) / pv61;
660     }
661
662     // Calculate the Log-likelihood as a convergence measure
663     pv106 = 0.0;
664     for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
665     {

```

```

666     pv104 = 0.0;
667     for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
668         pv104 += safelog(sigma(pv33, pv35));
669
670     pv105 = 0.0;
671     for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
672         pv105 += q(pv34, pv35);
673     pv106 += pv104 * pv105;
674 }
675
676 pv108 = 0.0;
677 for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
678     for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
679     {
680         pv107 = 0.0;
681         for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
682             pv107 += (sim_data(pv33, pv34) - mu(pv33, pv35)) *
683                 (sim_data(pv33, pv34) - mu(pv33, pv35)) /
684                 (sigma(pv33, pv35) * sigma(pv33, pv35));
685         pv108 += pv107 * q(pv34, pv35);
686     }
687
688 pv110 = 0.0;
689 for( pv19 = 0;pv19 <= n_classes - 1;pv19++ )
690     {
691         pv109 = 0.0;
692         for( pv18 = 0;pv18 <= n_points - 1;pv18++ )
693             pv109 += q(pv18, pv19);
694         pv110 += pv109 * safelog(phi(pv19));
695     }
696 loglikelihood = -0.5 *
697                 (n_points * n_variables *
698                 (safelog(2) + safelog(M_PI)) + pv108) + pv110 -
699                 pv106;
700 if ( loopcounter > 1 )
701     {
702         pv90 = abs(loglikelihood - pv89) / (abs(loglikelihood) + abs(pv89));
703
704         if ( loopcounter <= 1000 )
705             // collect convergence info
706             errors(loopcounter - 2) = pv90;
707         else
708             ;
709         octave_stdout << " pvar(90) = " << pv90 << endl;
710     }
711 else
712     ;
713 }
714 errors.resize(loopcounter);
715 // Label: label0
716 // Extract the most likely values of the hidden variable
717 // class_assignment(pv14) from the class membership table q.
718 for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
719     {

```

```

720 // Determine the position of the maximum with in the range
721 // 0
722 // ...
723 // -1 + n_classes
724 // by iterating over this range and calculating the value at each point
725 // (argmax).
726 //
727 // Argmax loop
728 for( pv84 = 0;pv84 <= n_classes - 1;pv84++ )
729 {
730     pv113 = q(pv14, pv84);
731     if ( ((pv84 == 0) || (pv113 > pv112)) )
732         // Save new maximum
733         {
734             pv112 = pv113;
735             pv111 = pv84;
736         }
737     else
738         ;
739 }
740 class_assignment(pv14) = pv111;
741 }
742
743 // Calculation of Log-likelihood
744 pv116 = 0.0;
745 for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
746 {
747     pv114 = 0.0;
748     for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
749         pv114 += safelog(sigma(pv33, pv35));
750
751     pv115 = 0.0;
752     for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
753         pv115 += q(pv34, pv35);
754     pv116 += pv114 * pv115;
755 }
756
757 pv118 = 0.0;
758 for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
759     for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
760     {
761         pv117 = 0.0;
762         for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
763             pv117 += (sim_data(pv33, pv34) - mu(pv33, pv35)) *
764                 (sim_data(pv33, pv34) - mu(pv33, pv35)) /
765                 (sigma(pv33, pv35) * sigma(pv33, pv35));
766         pv118 += pv117 * q(pv34, pv35);
767     }
768
769 pv120 = 0.0;
770 for( pv19 = 0;pv19 <= n_classes - 1;pv19++ )
771 {
772     pv119 = 0.0;
773     for( pv18 = 0;pv18 <= n_points - 1;pv18++ )

```

```

774     pv119 += q(pv18, pv19);
775     pv120 += pv119 * safelog(phi(pv19));
776 }
777 loglikelihood = -0.5 *
778                 (n_points * n_variables * (safelog(2) + safelog(M_PI)) +
779                 pv118) + pv120 - pv116;
780
781 retval.resize(6);
782 retval(0) = class_assignment;
783 retval(1) = mu;
784 retval(2) = phi;
785 retval(3) = sigma;
786 retval(4) = loglikelihood;
787 retval(5) = errors;
788
789 return retval;
790 }
791 //-- End of code
-----

```

Listing A.5. The C++ code AUTOBAYES generated from the input file given in Figure A.5.

```

1
2 //-----
3 // Code file generated by AutoBayes V0.9.9
4 // AutoBayes(c) 2008-2011 United States Government as represented by
5 // the Administrator of NASA. AutoBayes is distributed under the NASA
6 // Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
7 // details.
8 // Problem:   SIMPLE MULTIVARIATE CLUSTERING MODEL FOR CLASSICAL IRIS FLOWER
9 // EXAMPLE
10 // Source:   iris.ab
11 // Command:
12 //
13 //   PROLOG_VAR
14 //
15
16 //           -instrument
17 //           iris.ab
18 // Generated: Thu Jun  4 11:08:35 2020
19 //-----
20
21 #include "autobayes.h"
22
23
24
25 //-----
26 // Octave Function: iris
27 //-----
28
29 DEFUN_DLD(iris, input_args, output_args,
30           "usage: [vector class_assignment, matrix mu, vector phi, matrix sigma,
           vector errors] = iris(matrix iris_data, int n_classes, double tolerance, int

```



```

maxiteration)\n\n"
31     )
32 {
33 octave_value_list retval;
34 if (input_args.length () != 4 || output_args != 5 ){
35     octave_stdout << "usage: [vector class_assignment,matrix mu,vector phi,
matrix sigma,vector errors] = iris(matrix iris_data,int n_classes,double
tolerance,int maxiteration)\n\n";
36     return retval;
37 }
38
39 //-- Input declarations -----
40
41 octave_value arg_iris_data = input_args(0);
42 if (!arg_iris_data.is_real_matrix()){
43     gripe_wrong_type_arg("iris_data", (const std::string &)"Matrix expected");
44     return retval;
45 }
46 Matrix iris_data = (Matrix)(arg_iris_data.matrix_value());
47
48 // NUMBER OF CLASSES
49 octave_value arg_n_classes = input_args(1);
50 if (!arg_n_classes.is_real_scalar()){
51     gripe_wrong_type_arg("n_classes", (const std::string &)"int expected");
52     return retval;
53 }
54 int n_classes = (int)(arg_n_classes.int_value());
55
56 // Iteration tolerance for convergence loop
57 octave_value arg_tolerance = input_args(2);
58 if (!arg_tolerance.is_real_scalar()){
59     gripe_wrong_type_arg("tolerance", (const std::string &)"double expected");
60     return retval;
61 }
62 double tolerance = (double)(arg_tolerance.double_value());
63
64 // maximal number of iterations
65 octave_value arg_maxiteration = input_args(3);
66 if (!arg_maxiteration.is_real_scalar()){
67     gripe_wrong_type_arg("maxiteration", (const std::string &)"int expected");
68     return retval;
69 }
70 int maxiteration = (int)(arg_maxiteration.int_value());
71
72 //-- Constant declarations -----
73
74 // NUMBER OF DATA POINTS
75 int n_points = arg_iris_data.columns();
76
77 // NUMBER OF FEATURES
78 int n_variables = arg_iris_data.rows();
79
80 //-- Output declarations -----
81

```

```

82 // CLASS OF EACH POINT
83 ColumnVector class_assignment(n_points);
84
85 // MATRIX OF MEANS
86 Matrix mu(n_variables, n_classes);
87
88 // CLASS PROBABILITY VECTOR.
89 ColumnVector phi(n_classes);
90
91 // MATRIX OF STD DEVS
92 Matrix sigma(n_variables, n_classes);
93
94 // instrumentation: assembly of convergence data
95 ColumnVector errors(1000);
96
97
98 //-- Local declarations -----
99
100 // Label: label0
101 // class membership table used in Discrete EM-algorithm
102 Matrix q(n_points, n_classes);
103
104 // local centers used for center-based initialization
105 Matrix center(n_classes, n_variables);
106
107 // Random index of data point
108 int pick;
109
110 // Loop variable
111 int pv69;
112
113 // Loop variable
114 int pv66;
115
116 // Loop variable
117 int pv67;
118
119 // Lagrange-multiplier
120 double l;
121
122 // Loop variable
123 int pv43;
124
125 // Loop variable
126 int pv44;
127
128 // Loop variable
129 int pv14;
130
131 // Loop variable
132 int pv25;
133
134 // Common subexpression
135 // sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))

```

```

136 double pv53;
137
138 // Memoized common subexpression
139 //   phi(pv57) *
140 //   prod([pv54 := 0 .. -1 + n_variables],
141 //         exp(-1 / 2 * (iris_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
142 //             sigma(pv54, pv57) ** 2) *
143 //         (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
144 ColumnVector pv59(n_classes);
145
146 // Common subexpression
147 //   sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
148 double pv61;
149
150 // Loop variable
151 int pv57;
152
153 // Loop variable
154 int pv81;
155
156 int pv71;
157
158 // Summation accumulator
159 //   sum([pv71 := 0 .. -1 + n_classes], sqrt(pv87))
160 double pv88;
161
162 // Summation accumulator
163 //   sum([pv70 := 0 .. -1 + n_variables],
164 //         (center(pv69, pv70) - iris_data(pv70, pv14)) ** 2)
165 double pv86;
166
167 int pv70;
168
169 // Summation accumulator
170 //   sum([pv72 := 0 .. -1 + n_variables],
171 //         (center(pv71, pv72) - iris_data(pv72, pv14)) ** 2)
172 double pv87;
173
174 int pv72;
175
176 Matrix muold(n_variables, n_classes);
177
178 ColumnVector phiold(n_classes);
179
180 Matrix sigmaold(n_variables, n_classes);
181
182 int pv74;
183
184 int pv75;
185
186 int pv76;
187
188 int pv77;
189

```

```

190  int pv78;
191
192  // convergence loop counter
193  int loopcounter;
194
195  // sum up the Diffs
196  double pv89;
197
198  // Summation accumulator
199  //  sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
200  double pv97;
201
202  int pv27;
203
204  // Summation accumulator
205  //  sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))
206  double pv98;
207
208  int pv50;
209
210  // Summation accumulator
211  //  sum([pv49 := 0 .. -1 + n_points], iris_data(pv43, pv49) * q(pv49, pv44)
212  //  )
213  double pv99;
214
215  int pv49;
216
217  // Summation accumulator
218  //  sum([pv51 := 0 .. -1 + n_points],
219  //  (-mu(pv43, pv44) + iris_data(pv43, pv51)) ** 2 * q(pv51, pv44))
220  double pv100;
221
222  int pv51;
223
224  // Product accumulator
225  //  sum([pv54 := 0 .. -1 + n_variables],
226  //  exp(-1 / 2 * (iris_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
227  //  sigma(pv54, pv57) ** 2) *
228  //  (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
229  double pv106;
230
231  int pv54;
232
233  // Summation accumulator
234  //  sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
235  double pv107;
236
237  int pv56;
238
239  int pv92;
240
241  // Summation accumulator
242  //  sum([pv92 := 0 .. -1 + n_classes],
243  //  abs(phi(pv92) - phiold(pv92)) / (abs(phi(pv92)) + abs(phiold(pv92)))

```

```

    ))
243 double pv109;
244
245 int pv91;
246
247 int pv90;
248
249 // Summation accumulator
250 // sum([pv90 := 0 .. -1 + n_variables, pv91 := 0 .. -1 + n_classes],
251 //      abs(mu(pv90, pv91) - muold(pv90, pv91)) /
252 //      (abs(mu(pv90, pv91)) + abs(muold(pv90, pv91))))
253 double pv108;
254
255 // Summation accumulator
256 // sum([pv93 := 0 .. -1 + n_variables, pv94 := 0 .. -1 + n_classes],
257 //      abs(sigma(pv93, pv94) - sigmaold(pv93, pv94)) /
258 //      (abs(sigma(pv93, pv94)) + abs(sigmaold(pv93, pv94))))
259 double pv110;
260
261 int pv93;
262
263 int pv94;
264
265 // Argmax index
266 int pv111;
267
268 // Argmax value
269 double pv112;
270
271 // Argmax temporary
272 double pv113;
273
274 // Argmax loop index
275 int pv84;
276
277 // Check constraints on inputs
278 ab_assert( 0 < n_classes );
279 ab_assert( 10 * n_classes < n_points );
280
281 // Label: label1
282 // Label: label2
283 // Label: label4
284 // Discrete EM-algorithm
285 //
286 // The model describes a discrete latent (or hidden) variable problem with
287 // the latent variable class_assignment and the data variable iris_data. The
288 // problem to optimize the conditional probability pr(iris_data |
289 // {mu,phi,sigma}) w.r.t. the variables mu, phi, and sigma can thus be
290 // solved by an application of the (discrete) EM-algorithm.
291 // The algorithm maintains as central data structure a class membership
292 // table q (see "label0") such that q(pv14,pv62) is the probability that
293 // data point pv14 belongs to class pv62, i.e.,
294 //
295 // q(pv14, pv62) == pr([class_assignment(pv14) == pv62])

```

```

296 //
297 // The algorithm consists of an initialization phase for q (see "label2"),
298 // followed by a convergence phase (see "label5"), followed by the
299 // extraction of the hidden variable class_assignment (see "label6").
300 //
301 // Initialization
302 //
303 // The initialization is center-based, i.e., for each class (i.e., value of
304 // the hidden variable class_assignment) a center value center is chosen
305 // first (see "label4"). Then, the values for the local distribution are
306 // calculated as distances between the data points and these center values
307 // (see "label7").
308 //
309 // Random initialization of the centers center with data points;
310 // note that a data point can be picked as center more than once.
311 for( pv66 = 0;pv66 <= n_classes - 1;pv66++ )
312 {
313     pick = uniform_int_rnd(n_points - 1);
314     for( pv67 = 0;pv67 <= n_variables - 1;pv67++ )
315         center(pv66, pv67) = iris_data(pv67, pick);
316 }
317 // Label: label7
318 for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
319     for( pv69 = 0;pv69 <= n_classes - 1;pv69++ )
320     {
321         pv86 = 0.0;
322         for( pv70 = 0;pv70 <= n_variables - 1;pv70++ )
323             pv86 += (center(pv69, pv70) - iris_data(pv70, pv14)) *
324                 (center(pv69, pv70) - iris_data(pv70, pv14));
325
326         pv88 = 0.0;
327         for( pv71 = 0;pv71 <= n_classes - 1;pv71++ )
328         {
329             pv87 = 0.0;
330             for( pv72 = 0;pv72 <= n_variables - 1;pv72++ )
331                 pv87 += (center(pv71, pv72) - iris_data(pv72, pv14)) *
332                     (center(pv71, pv72) - iris_data(pv72, pv14));
333             pv88 += sqrt(pv87);
334         }
335         q(pv14, pv69) = sqrt(pv86) / pv88;
336     }
337 // resize vector to maximal size
338 errors.resize(1000);
339 // initialize convergence output
340 for( loopcounter = 0;loopcounter <= 999;loopcounter++ )
341     errors(loopcounter) = 0;
342 // Tolerance value must be positive
343 ab_assert( tolerance > 0 );
344 // max nr of iterations must be positive
345 ab_assert( maxiteration > 0 );
346 loopcounter = 0;
347 // repeat at least once
348 pv89 = tolerance;
349 while( ((loopcounter < maxiteration) && (pv89 >= tolerance)) )

```

```

350 {
351   loopcounter = 1 + loopcounter;
352   if ( loopcounter > 1 )
353     {
354       // assign current values to old values
355       for( pv74 = 0;pv74 <= n_variables - 1;pv74++ )
356         for( pv75 = 0;pv75 <= n_classes - 1;pv75++ )
357           muold(pv74, pv75) = mu(pv74, pv75);
358       // assign current values to old values
359       for( pv76 = 0;pv76 <= n_classes - 1;pv76++ )
360         phiold(pv76) = phi(pv76);
361       // assign current values to old values
362       for( pv77 = 0;pv77 <= n_variables - 1;pv77++ )
363         for( pv78 = 0;pv78 <= n_classes - 1;pv78++ )
364           sigmaold(pv77, pv78) = sigma(pv77, pv78);
365     }
366   else
367     ;
368
369   // Label: label8
370   // Label: label3
371   // M-Step
372   //
373   // Decomposition I
374   //
375   // The problem to optimize the conditional probability
376   // pr({class_assignment,iris_data} | {mu,phi,sigma}) w.r.t. the
377   // variables mu, phi, and sigma can under the given dependencies by
378   // Bayes rule be decomposed into two independent subproblems:
379   //
380   //   max pr(class_assignment | phi) for phi
381   //   max pr(iris_data | {class_assignment,mu,sigma}) for {mu,sigma}
382   //
383   //
384   // The conditional probability pr(class_assignment | phi) is under the
385   // dependencies given in the model equivalent to
386   //
387   //   prod([pv18 := 0 .. -1 + n_points], pr(class_assignment(pv18) | phi)
388 )
389   //
390   // The probability occurring here is atomic and can thus be replaced by
391   // the respective probability density function given in the model.
392   // Summing out the expected variable class_assignment(pv14) yields the
393   // log-likelihood function
394   //
395   //   sum_domain([pv14 := 0 .. -1 + n_points],
396   //               [pv19 := 0 .. -1 + n_classes], [class_assignment(pv14)],
397   //               q(pv14, pv19),
398   //               log(prod([pv18 := 0 .. -1 + n_points],
399   //                       phi(class_assignment(pv18))))))
400   // which can be simplified to
401   //
402   //   sum([pv19 := 0 .. -1 + n_classes],

```

```

403 //      log(phi(pv19)) *
404 //      sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
405 //
406 // This function is then optimized w.r.t. the goal variable phi.
407 //
408 // The expression
409 //
410 //      sum([pv19 := 0 .. -1 + n_classes],
411 //      log(phi(pv19)) *
412 //      sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
413 //
414 // is maximized w.r.t. the variable phi under the constraint
415 //
416 //      0 == 1 + -1 * sum([pv24 := 0 .. -1 + n_classes], phi(pv24))
417 //
418 // using the Lagrange-multiplier l.
419 l = (double)(-n_points);
420 for( pv25 = 0;pv25 <= n_classes - 1;pv25++ )
421 // The summand
422 //
423 //      -1 * l
424 //
425 // is constant with respect to the goal variable phi(pv25) and can
426 // thus be ignored for maximization.
427 //
428 // The function
429 //
430 //      l * sum([pv24 := 0 .. -1 + n_classes], phi(pv24)) +
431 //      sum([pv19 := 0 .. -1 + n_classes],
432 //      log(phi(pv19)) *
433 //      sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
434 //
435 // is then symbolically maximized w.r.t. the goal variable phi(pv25).
436 // The differential
437 //
438 //      l +
439 //      phi(pv25) ** -1 * sum([pv18 := 0 .. -1 + n_points], q(pv18, pv25
))
440 //
441 // is set to zero; this equation yields the solution
442 //
443 //      -1 * l ** -1 * sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
444 //
445 {
446     pv97 = 0.0;
447     for( pv27 = 0;pv27 <= n_points - 1;pv27++ )
448         pv97 += q(pv27, pv25);
449     phi(pv25) = -pv97 / l;
450 }
451
452 // The conditional probability pr(iris_data |
453 // {class_assignment,mu,sigma}) is under the dependencies given in the
454 // model equivalent to
455 //

```



```

456 // prod([pv33 := 0 .. -1 + n_variables, pv34 := 0 .. -1 + n_points],
457 //      pr(iris_data(pv33,pv34) | {class_assignment(pv34),mu(pv33,*),
sigma(pv33,*)}))
458 //
459 // The probability occurring here is atomic and can thus be replaced by
460 // the respective probability density function given in the model.
461 // Summing out the expected variable class_assignment(pv14) yields the
462 // log-likelihood function
463 //
464 // sum_domain([pv14 := 0 .. -1 + n_points],
465 //            [pv35 := 0 .. -1 + n_classes], [class_assignment(pv14)],
466 //            q(pv14, pv35),
467 //            log(prod([pv33 := 0 .. -1 + n_variables,
468 //                    pv34 := 0 .. -1 + n_points],
469 //                    exp(-1 / 2 *
470 //                        (iris_data(pv33, pv34) -
471 //                          mu(pv33, class_assignment(pv34)))) ** 2 /
472 //                    sigma(pv33, class_assignment(pv34)) ** 2)
*
473 //                                (1 /
474 //                                (sqrt(2 * pi) *
475 //                                sigma(pv33, class_assignment(pv34))))))
476 //
477 // which can be simplified to
478 //
479 // -1 *
480 // sum([pv35 := 0 .. -1 + n_classes],
481 //      sum([pv33 := 0 .. -1 + n_variables, log(sigma(pv33, pv35))] *
482 //          sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
483 //      -1 / 2 * n_points * n_variables * log(2) +
484 //      -1 / 2 * n_points * n_variables * log(pi) +
485 //      -1 / 2 *
486 //      sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
487 //          q(pv34, pv35) *
488 //          sum([pv33 := 0 .. -1 + n_variables],
489 //              (-1 * mu(pv33, pv35) + iris_data(pv33, pv34)) ** 2 *
490 //              sigma(pv33, pv35) ** -2))
491 //
492 // This function is then optimized w.r.t. the goal variables mu and
493 // sigma.
494 //
495 // The summands
496 //
497 // -1 / 2 * n_points * n_variables * log(2)
498 // -1 / 2 * n_points * n_variables * log(pi)
499 //
500 // are constant with respect to the goal variables mu and sigma and can
501 // thus be ignored for maximization.
502 //
503 // Index decomposition
504 //
505 // The function
506 //
507 // -1 *

```

```

508 //      sum([pv35 := 0 .. -1 + n_classes],
509 //          sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35))) *
510 //          sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
511 //      -1 / 2 *
512 //      sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
513 //          q(pv34, pv35) *
514 //          sum([pv33 := 0 .. -1 + n_variables],
515 //              (-1 * mu(pv33, pv35) + iris_data(pv33, pv34)) ** 2 *
516 //              sigma(pv33, pv35) ** -2))
517 //
518 // can be optimized w.r.t. the variables mu(pv43,pv44) and
519 // sigma(pv43,pv44) element by element (i.e., along the index variables
520 // pv43 and pv44) because there are no dependencies along thats
521 // dimensions.
522 for( pv43 = 0;pv43 <= n_variables - 1;pv43++ )
523     for( pv44 = 0;pv44 <= n_classes - 1;pv44++ )
524         // The factor
525         //
526         //      n_classes
527         //
528         // is non-negative and constant with respect to the goal variables
529         // mu(pv43,pv44) and sigma(pv43,pv44) and can thus be ignored for
530         // maximization.
531         //
532         // The function
533         //
534         //      -1 * n_variables * log(sigma(pv43, pv44)) *
535         //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
536         //      -1 / 2 * n_variables * sigma(pv43, pv44) ** -2 *
537         //      sum([pv34 := 0 .. -1 + n_points],
538         //          (-1 * mu(pv43, pv44) + iris_data(pv43, pv34)) ** 2 *
539         //          q(pv34, pv44))
540         //
541         // is then symbolically maximized w.r.t. the goal variables
542         // mu(pv43,pv44) and sigma(pv43,pv44). The partial differentials
543         //
544         //      df / d_mu(pvar(43),pvar(44)) ==
545         //      -1 * n_variables * sigma(pv43, pv44) ** -2 * mu(pv43, pv44) *
546         //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
547         //      n_variables * sigma(pv43, pv44) ** -2 *
548         //      sum([pv34 := 0 .. -1 + n_points],
549         //          iris_data(pv43, pv34) * q(pv34, pv44))
550         //      df / d_sigma(pvar(43),pvar(44)) ==
551         //      -1 * n_variables * sigma(pv43, pv44) ** -1 *
552         //      sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
553         //      n_variables * sigma(pv43, pv44) ** -3 *
554         //      sum([pv34 := 0 .. -1 + n_points],
555         //          (-1 * mu(pv43, pv44) + iris_data(pv43, pv34)) ** 2 *
556         //          q(pv34, pv44))
557         //
558         // are set to zero; these equations yield the solutions
559         //
560         //      mu(pv43, pv44) ==
561         //      cond(0 == n_variables or

```

```

562 //      0 == sum([pv47 := 0 .. -1 + n_points], q(pv47, pv44)),
563 //      fail(division_by_zero),
564 //      sum([pv48 := 0 .. -1 + n_points], q(pv48, pv44)) ** -1 *
565 //      sum([pv49 := 0 .. -1 + n_points],
566 //      iris_data(pv43, pv49) * q(pv49, pv44)))
567 // sigma(pv43, pv44) ==
568 //      cond(0 == n_variables or
569 //      0 == sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44)),
570 //      fail(division_by_zero),
571 //      abs(n_variables) * n_variables ** -1 *
572 //      sum([pv51 := 0 .. -1 + n_points],
573 //      (-1 * mu(pv43, pv44) + iris_data(pv43, pv51)) ** 2 *
574 //      q(pv51, pv44)) ** (1 / 2) *
575 //      sum([pv52 := 0 .. -1 + n_points], q(pv52, pv44)) **
576 //      (-1 / 2))
577 //
578 {
579 // Initialization of common subexpression
580 pv98 = 0.0;
581 for( pv50 = 0;pv50 <= n_points - 1;pv50++ )
582     pv98 += q(pv50, pv44);
583 pv53 = pv98;
584
585 if ( ((0 == n_variables) || (0 == pv53)) )
586     { ab_error( division_by_zero ); }
587 else
588     {
589     pv99 = 0.0;
590     for( pv49 = 0;pv49 <= n_points - 1;pv49++ )
591         pv99 += iris_data(pv43, pv49) * q(pv49, pv44);
592     mu(pv43, pv44) = pv99 * ((double)(1) / pv53);
593     }
594 if ( ((0 == n_variables) || (0 == pv53)) )
595     { ab_error( division_by_zero ); }
596 else
597     {
598     pv100 = 0.0;
599     for( pv51 = 0;pv51 <= n_points - 1;pv51++ )
600         pv100 += (iris_data(pv43, pv51) - mu(pv43, pv44)) *
601                 (iris_data(pv43, pv51) - mu(pv43, pv44)) *
602                 q(pv51, pv44);
603     sigma(pv43, pv44) = abs(n_variables) * sqrt(pv100) *
604                       ((double)(1) / (double)(n_variables)) *
605                       ((double)(1) / sqrt(pv53));
606     }
607 }
608 // Label: label9
609 // E-Step
610 // Update the current values of the class membership table q.
611 for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
612     {
613     // Initialization of common subexpression
614     for( pv57 = 0;pv57 <= n_classes - 1;pv57++ )
615         {

```

```

616     pv106 = 1.0;
617     for( pv54 = 0;pv54 <= n_variables - 1;pv54++ )
618         pv106 *= exp(-0.5 * (iris_data(pv54, pv14) - mu(pv54, pv57)) *
619             (iris_data(pv54, pv14) - mu(pv54, pv57)) /
620             (sigma(pv54, pv57) * sigma(pv54, pv57))) *
621             ((double)(1) /
622             (sqrt(M_PI * (double)(2)) * sigma(pv54, pv57)));
623     pv59(pv57) = phi(pv57) * pv106;
624 }
625
626 pv107 = 0.0;
627 for( pv56 = 0;pv56 <= n_classes - 1;pv56++ )
628     pv107 += pv59(pv56);
629 pv61 = pv107;
630 for( pv81 = 0;pv81 <= n_classes - 1;pv81++ )
631     // The denominator pv61 can become zero due to round-off errors.
632     // In that case, each class is considered to be equally likely.
633     if ( pv61 == 0.0 )
634         q(pv14, pv81) = (double)(1) / (double)(n_classes);
635     else
636         q(pv14, pv81) = pv59(pv81) / pv61;
637 }
638 if ( loopcounter > 1 )
639 {
640     pv108 = 0.0;
641     for( pv90 = 0;pv90 <= n_variables - 1;pv90++ )
642         for( pv91 = 0;pv91 <= n_classes - 1;pv91++ )
643             pv108 += abs(mu(pv90, pv91) - muold(pv90, pv91)) /
644                 (abs(mu(pv90, pv91)) + abs(muold(pv90, pv91)));
645
646     pv109 = 0.0;
647     for( pv92 = 0;pv92 <= n_classes - 1;pv92++ )
648         pv109 += abs(phi(pv92) - phiold(pv92)) /
649             (abs(phi(pv92)) + abs(phiold(pv92)));
650
651     pv110 = 0.0;
652     for( pv93 = 0;pv93 <= n_variables - 1;pv93++ )
653         for( pv94 = 0;pv94 <= n_classes - 1;pv94++ )
654             pv110 += abs(sigma(pv93, pv94) - sigmaold(pv93, pv94)) /
655                 (abs(sigma(pv93, pv94)) + abs(sigmaold(pv93, pv94)));
656     pv89 = pv108 + pv109 + pv110;
657
658     if ( loopcounter <= 1000 )
659         // collect convergence info
660         errors(loopcounter - 2) = pv89;
661     else
662         ;
663     octave_stdout << " pvar(89) = " << pv89 << endl;
664 }
665 else
666     ;
667 }
668 errors.resize(loopcounter);
669 // Label: label6

```

```

670 // Extract the most likely values of the hidden variable
671 // class_assignment(pv14) from the class membership table q.
672 for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
673 {
674     // Determine the position of the maximum with in the range
675     // 0
676     // ...
677     // -1 + n_classes
678     // by iterating over this range and calculating the value at each point
679     // (argmax).
680     //
681     // Argmax loop
682     for( pv84 = 0;pv84 <= n_classes - 1;pv84++ )
683     {
684         pv113 = q(pv14, pv84);
685         if ( ((pv84 == 0) || (pv113 > pv112)) )
686             // Save new maximum
687             {
688                 pv112 = pv113;
689                 pv111 = pv84;
690             }
691         else
692             ;
693     }
694     class_assignment(pv14) = pv111;
695 }
696
697 retval.resize(5);
698 retval(0) = class_assignment;
699 retval(1) = mu;
700 retval(2) = phi;
701 retval(3) = sigma;
702 retval(4) = errors;
703
704 return retval;
705 }
706 //-- End of code
-----

```

Listing A.6. The C++ code AUTOBAYES generated from the input file given in Figure A.6.

A.3 List of Constraints

Table A.1. CONSTRAINTS DEVELOPED FOR THE INPUT, OUTPUT, AND THE RELATIONSHIP BETWEEN THEM.

Number	Constraint
1	The declared InputData.name (e.g., x) must be used in StatisticalModel.equation AND Goal.equation.
2	StatisticalModel.name = 'gauss' THEN the Goal must include Mean.name AND (Variance.name OR StandardDeviation.name).
3	IF Mean.col_size = 1 THEN Mean.row_size = ModelParameters.n_classes.
4	IF ModelParameters.n_variables > 1 THEN Mean.row_size = StandardDeviation.row_size = ModelParameters.n_variables AND Mean.col_size = StandardDeviation.col_size = ModelParameters.n_classes.
5	IF StatisticalModel.name = gauss AND sqrt() is used in the StatisticalModel.equation THEN the Variance is used.
6	IF Variance is used in ClassParameters, StandardDeviation is not used and vice versa.
7	IF Variance is used in Denominator, StandardDeviation is not used and vice versa.
8	IF Variance is used in Coefficient, StandardDeviation is not used and vice versa.
9	IF StatisticalModel.name = gauss THEN one of Variance OR StandardDeviation must be used in the Gaussian Coefficient and Denominator.
10	IF Variance OR StandardDeviation is used in ClassParameters OR StatisticalModel OR Goal OR Coefficient OR Denominator THEN it must be used in all of the others.
11	ModelParameters.n_classes > 0 AND ModelParameters.n_variables > 0 AND ModelParameters.n_points > 0
12	Mean.name must be specified AND Mean.row_size > 0 AND Mean.col_size > 0
13	Variance.name must be specified AND Variance.row_size = Variance.col_size = 1.
14	InputData.name must be specified.
15	StandardDeviation.name must be specified.
16	StandardDeviation.row_size > 0 AND StandardDeviation.col_size > 0

Table A.2. CONSTRAINTS DEVELOPED FOR THE OUTPUT.

Number	Constraint
1	Variance.row_size = 1 AND Variance.col_size = 1
2	IF NormalDistribution is used THEN Mean.row_size = Mean.col_size = 1
3	IF Transformations is used THEN Mean.row_size = Mean.col_size = 1
4	IF NormalDistribution is used THEN Transformations is NOT used.
5	IF Transformations is used THEN NormalDistribution is NOT used.
6	The value of variance must always be > 0
7	There must always be a Declaration and an Initialization in the output code.
8	There must be one Input AND one Constant AND one Output AND one Local Class in the Declaration Class whenever the code is generated.
9	Errors.row_size = 1000
10	IF Transformations is used THEN CommonSubexpressionInitLoop AND MemoizedCommonSubexpression must be used.
11	IF NormalDistribution is used THEN InputData is used in CalculateMean AND CalculateVariance.
12	IF Transformations is used THEN InputData is used to calculate MemoizedCommonSubexpression in the CommonSubexpressionInitializationLoop.
13	Retval.resize is used before any values are stored into it.
14	IF NormalDistribution OR Transformations is used THEN Retval.size must be Retval.resize to 2.
15	IF NormalDistribution OR Transformations is used THEN InputData.col_size = 1.
16	IF Transformations is used THEN MemoizedCommonSubexpression.row_size = InputData.row_size.
17	MemoizedCommonSubexpression.col_size = 1
18	Mean.row_size must equal StandardDeviation.row_size AND Mean.col_size must equal StandardDeviation.col_size.
19	IF NormalDistribution is used, there must be 2 Mean AND 2 InputData in the CalculateVarianceLoop.
20	IF Transformations is used, there must be 2 Mean AND 2 MemoizedCommonSubexpression in the CalculateVarianceLoop.

Table A.3. CONSTRAINTS DEVELOPED FOR THE RELATIONSHIP BETWEEN THEM.

Number	Constraint
1	IF StatisticalModel.name = gauss AND sqrt() is used in the StatisticalModel.equation (e.g., gauss(mu, sqrt(sigma_sq))), THEN the Variance class must be used.
2	IF StatisticalModel.equation = 'x() ~ gauss(mu, sqrt(sigma_sq))' THEN the NormalDistribution class must be used.
3	IF StatisticalModel.equation = 'x()**2 ~ gauss(mu, sqrt(sigma_sq))' THEN the Transformation must be used and the Transformation.type = "square".
4	IF StatisticalModel.equation = 'log(x()) ~ gauss(mu, sqrt(sigma_sq))' THEN the Transformation must be used and the Transformation.type = 'log'.
5	IF Pragma.name = 'em_log_likelihood_convergence' AND Pragma.value = 'true' THEN Log-likelihood AND CalculateLog-likelihood is used.
6	IF input Mean.row_size = 1 AND input Mean.col_size = 1 THEN (NormalDistribution OR Transformations) will be used.
7	Input Mean.row_size must equal output Mean.row_size AND input Mean.col_size must equal output Mean.col_size.
8	Input Variance.row_size must equal output Variance.row_size AND input Variance.col_size must equal output Variance.col_size.
9	Input StandardDeviation.row_size must equal output StandardDeviation.row_size AND input StandardDeviation.col_size must equal output StandardDeviation.col_size.
10	IF ClassProbabilities AND HiddenVariable are used THEN DiscreteEM-algorithm will be used.
11	The input InputData.name must equal the output InputData.name.
12	The input Mean.name must equal the output Mean.name.
13	The input Variance.name must equal the output Variance.name.
14	The input StandardDeviation.name must equal the output StandardDeviation.name.

A.4 Input File for USE

```
1 -- $ProjectHeader: use 5.2.0 Thurs, 15 October 2020-CSci Master Thesis-Jason
   Hicks $
2
3 model AUTOBAYES
4
5 -----
6 -- Classes from input CD -----
7 -----
8
9 class StatisticalModel
10 attributes
11   name : String
12   equation : String
13 end
14
15 class Pragmas
16 attributes
17   name : String
18   value : String
19 end
20
21 class HiddenVariable
22 attributes
23   name : String
24 end
25
26 class ClassProbabilities
27 attributes
28   name : String
29   row_size : Integer
30   col_size : Integer
31   values : Bag(Real)
32 end
33
34 class Goal
35 attributes
36   equation : String
37 end
38
39 class ModelParameters
40 attributes
41   n_points : Integer
42   n_classes : Integer
43   n_variables : Integer
44 end
45
46 class InputData
47 attributes
```

```

48   name : String
49   row_size : Integer
50   col_size : Integer
51   values : Bag(Real)
52 end
53
54 class ClassParameters
55 end
56
57 class StandardDeviation
58 attributes
59   name : String
60   row_size : Integer
61   col_size : Integer
62   values : Bag(Real)
63 end
64
65 class Mean
66 attributes
67   name : String
68   row_size : Integer
69   col_size : Integer
70   values : Bag(Real)
71 end
72
73 class Variance
74 attributes
75   name : String
76   row_size : Integer
77   col_size : Integer
78   values : Bag(Real)
79 end
80
81 class Gaussian < StatisticalModel
82 end
83
84 class Exponent
85 end
86
87 class Coefficient
88 end
89
90 class Numerator
91 end
92
93 class Denominator
94 end
95
96
97 -----
98 -- Classes from subset of output CD -----
99 -----
100
101 class GaussianModel

```

```

102 attributes
103   name : String
104   input_args : Bag(Bag(Real))
105   output_args : Integer
106   n_points : Integer
107   n_variables : Integer
108   n_classes : Integer
109   input_data : Matrix
110   data_set_name : String
111 operations
112   check_in_and_out_args(input_args : Bag(Bag(Real)), output_args : Integer) :
      Boolean
113   check_data_format(input_data : Matrix) : Boolean
114   get_data() : Matrix
115   get_n_points() : Integer
116 end
117
118 class Retval
119 attributes
120   size : Integer
121 operations
122   resize()
123 end
124
125 class Declaration
126 end
127
128 class InputDeclaration
129 end
130
131 class ConstantDeclaration
132 end
133
134 class OutputDeclaration
135 end
136
137 class LocalDeclaration
138 end
139
140 class Initialization
141 end
142
143 class Matrix
144 attributes
145   name : String
146   row_size : Integer
147   col_size : Integer
148   values : Bag(Real)
149 end
150
151 class OutputCodeMean < Matrix
152 attributes
153   n_classes : Integer
154   n_variables : Integer

```

```

155 end
156
157 class OutputCodeVariance < Matrix
158 end
159
160 class OutputCodeInputData < Matrix
161 end
162
163 class MemoizedCommonSubexpression < Matrix
164 end
165
166 class NormalDistribution
167 end
168
169 class CalculateVarianceLoop
170 attributes
171   n_points : Integer
172 end
173
174 class CalculateMeanLoop
175 attributes
176   n_points : Integer
177 end
178
179 class Transformations
180 attributes
181   type : String
182 end
183
184 class CommonSubexpressionInitLoop
185 attributes
186   n_points : Integer
187 operations
188   transform_data(OutputCodeInputData : Matrix) : Matrix
189 end
190
191 class CheckDivideByZero
192 attributes
193   n_point : Integer
194   n_variables : Integer
195 operations
196   divide_by_zero_check(IntToCheck : Integer) : Boolean
197 end
198
199 -----
200 -- Associations within input CD -----
201 -----
202
203 association StatModelGoal between
204   StatisticalModel[1..*]
205   Goal[1]
206 end
207
208 association StatModelInData between

```

```

209     StatisticalModel[1..*]
210     InputData[1..*]
211 end
212
213 aggregation PragStatModel between
214     Pragmas[0..*]
215     StatisticalModel[1..*]
216 end
217
218 aggregation HidVarStatModel between
219     HiddenVariable[0..1]
220     StatisticalModel[1..*]
221 end
222
223 aggregation ClassParaStatModel between
224     ClassParameters[1..*]
225     StatisticalModel[1..*]
226 end
227
228 aggregation ModParaHidVar between
229     ModelParameters[1..*]
230     HiddenVariable[0..1]
231 end
232
233 aggregation ClassProbHidVar between
234     ClassProbabilities[1]
235     HiddenVariable[0..1]
236 end
237
238 aggregation ClassProbGoal between
239     ClassProbabilities[0..1]
240     Goal[1]
241 end
242
243 aggregation ModParaClassProb between
244     ModelParameters[1..*]
245     ClassProbabilities[0..1]
246 end
247
248 aggregation ModParaInData between
249     ModelParameters[1..*]
250     InputData[1..*]
251 end
252
253 aggregation ModParaClassPara between
254     ModelParameters[0..*]
255     ClassParameters[0..*]
256 end
257
258 aggregation ClassParaGoal between
259     ClassParameters[1..*]
260     Goal[1]
261 end
262

```

```

263 aggregation VarClassPara between
264     Variance[0..1]
265     ClassParameters[1]
266 end
267
268 aggregation MeanClassPara between
269     Mean[1]
270     ClassParameters[1]
271 end
272
273 aggregation StandDevClassPara between
274     StandardDeviation[0..1]
275     ClassParameters[1]
276 end
277
278 aggregation InDataGoal between
279     InputData[1..*]
280     Goal[1]
281 end
282
283 ---- From breaking down the Guassian equ ----
284 aggregation CoeffGauss between
285     Coefficient[1]
286     Gaussian[1]
287 end
288
289 aggregation ExpGauss between
290     Exponent[1]
291     Gaussian[1]
292 end
293
294 aggregation CoeffExp between
295     Coefficient[1]
296     Exponent[1]
297 end
298
299 aggregation StandDevCoeff between
300     StandardDeviation[0..1]
301     Coefficient[1]
302 end
303
304 aggregation VarCoeff between
305     Variance[0..1]
306     Coefficient[1]
307 end
308
309 aggregation NumExp between
310     Numerator[1..*]
311     Exponent[1]
312 end
313
314 aggregation DenomfExp between
315     Denominator[1..*]
316     Exponent[1]

```

```

317 end
318
319 aggregation NumDenom between
320   Numerator[1]
321   Denominator[1]
322 end
323
324 aggregation InDataNum between
325   InputData[1]
326   Numerator[1]
327 end
328
329 aggregation MeanNum between
330   Mean[1]
331   Numerator[1]
332 end
333
334 aggregation InDataMean between
335   InputData[1]
336   Mean[1]
337 end
338
339 aggregation VarDenom between
340   Variance[0..1]
341   Denominator[1]
342 end
343
344 aggregation StandDevDenom between
345   StandardDeviation[0..1]
346   Denominator[1]
347 end
348
349
350 -----
351 -- Associations within output CD -----
352 -----
353
354 association GaussModMatrix between
355   GaussianModel[1..*]
356   Matrix[1..*]
357 end
358
359 aggregation MatrixInit between
360   Matrix[1..*]
361   Initialization[0..*]
362 end
363
364 aggregation InitGaussMod between
365   Initialization[1]
366   GaussianModel[1..*]
367 end
368
369 ----- Declarations -----
370 aggregation DeclarGaussMod between

```

```

371 Declaration[1]
372 GaussianModel[1..*]
373 end
374
375 aggregation InDeclar between
376 InputDeclaration[1]
377 Declaration[1..*]
378 end
379
380 aggregation ConstDeclar between
381 ConstantDeclaration[1]
382 Declaration[1..*]
383 end
384
385 aggregation OutDeclar between
386 OutputDeclaration[1]
387 Declaration[1..*]
388 end
389
390 aggregation LocalDeclar between
391 LocalDeclaration[1]
392 Declaration[1..*]
393 end
394
395 aggregation MatrixDeclar between
396 Matrix[1..*]
397 Declaration[0..1]
398 end
399
400 ----- Retval -----
401 aggregation RetvalGaussMod between
402 Retval[1]
403 GaussianModel[1..*]
404 end
405
406 aggregation OutMeanRetval between
407 OutputCodeMean[1]
408 Retval[1]
409 end
410
411 aggregation OutVarRetval between
412 OutputCodeVariance[0..1]
413 Retval[1]
414 end
415
416 ----- Normal -----
417 aggregation NormDistGaussMod between
418 NormalDistribution[0..1]
419 GaussianModel[1..*]
420 end
421
422 aggregation ChDivNormDist between
423 CheckDivideByZero[1..*]
424 NormalDistribution[1]

```



```

425 end
426
427 aggregation CalcMeanNormDist between
428     CalculateMeanLoop[1]
429     NormalDistribution[0..*]
430 end
431
432 aggregation CalcVarNormDist between
433     CalculateVarianceLoop[1]
434     NormalDistribution[0..*]
435 end
436
437 aggregation OutMeanCalcMean between
438     OutputCodeMean[1]
439     CalculateMeanLoop[0..*]
440 end
441
442 aggregation OutInDataCalcMean between
443     OutputCodeInputData[1]
444     CalculateMeanLoop[0..*]
445 end
446
447 aggregation OutMeanCalcVar between
448     OutputCodeMean[2]
449     CalculateVarianceLoop[0..*]
450 end
451
452 aggregation OutInDataCalcVar between
453     OutputCodeInputData[2]
454     CalculateVarianceLoop[0..*]
455 end
456
457 aggregation OutVarCalcVar between
458     OutputCodeVariance[1]
459     CalculateVarianceLoop[0..*]
460 end
461
462 ----- Transform -----
463 aggregation TransformGaussMod between
464     Transformations[0..1]
465     GaussianModel[1..*]
466 end
467
468 aggregation ChDivTransform between
469     CheckDivideByZero[1..*]
470     Transformations[1]
471 end
472
473 aggregation ComSubInitTransform between
474     CommonSubexpressionInitLoop[1]
475     Transformations[0..*]
476 end
477
478 aggregation CalcMeanTransform between

```

```

479 CalculateMeanLoop[1]
480 Transformations[0..*]
481 end
482
483 aggregation CalcVarTransform between
484 CalculateVarianceLoop[1]
485 Transformations[0..*]
486 end
487
488 aggregation MemoComSubCalcMean between
489 MemoizedCommonSubexpression[1]
490 CalculateMeanLoop[0..*]
491 end
492
493 aggregation MemoComSubCalcVar between
494 MemoizedCommonSubexpression[2]
495 CalculateVarianceLoop[0..*]
496 end
497
498 aggregation MemoComSubComSubInit between
499 MemoizedCommonSubexpression[1]
500 CommonSubexpressionInitLoop[0..*]
501 end
502
503 aggregation OutInDataComSubInit between
504 OutputCodeInputData[2]
505 CommonSubexpressionInitLoop[0..*]
506 end
507
508 -----
509 -- Associations input CD and output CD -----
510 -----
511
512 association InInputDataGaussMod between
513 InputData[0..*]
514 GaussianModel[1]
515 end
516
517 association ModParaGaussMod between
518 ModelParameters[1..*]
519 GaussianModel[1]
520 end
521
522 association StatModGaussMod between
523 StatisticalModel[1]
524 GaussianModel[1]
525 end
526
527 association InMeanOutMean between
528 Mean[1]
529 OutputCodeMean[1]
530 end
531
532 association InVarOutVar between

```

```

533     Variance[1]
534     OutputCodeVariance[1]
535 end
536
537 association InInputDataOutInputData between
538     InputData[1]
539     OutputCodeInputData[1]
540 end
541
542 association PragGaussMod between
543     Pragmas[0..*]
544     GaussianModel[1]
545 end
546
547
548 -----
549 -- OCL constraints -----
550 -----
551
552 constraints
553
554 ----- Constraints for input -----
555
556 context Gaussian
557     inv GaussName:
558         self.name = 'gauss'
559
560 context Variance
561     inv VarSize:
562         self.name.size() > 0
563         and self.row_size = 1
564         and self.col_size = 1
565
566 context Mean
567     inv MeanSize:
568         self.name.size() > 0
569         and self.row_size >= 1
570         and self.col_size >= 1
571
572 context ModelParameters
573     inv ModParamSize:
574         self.n_classes >= 1
575         and self.n_variables >= 1
576         and self.n_points >= 1
577
578 context InputData
579     inv InputDataName:
580         self.name.size() > 0
581
582 context ClassParameters
583     inv VarStdDevCP:
584         self.variance->size() = 1 implies self.standardDeviation->size() = 0
585         and self.standardDeviation->size() = 1 implies self.variance->size() = 0
586

```

```

587 context Denominator
588   inv VarStdDevDenom:
589     self.variance->size() = 1 implies self.standardDeviation->size() = 0
590   and self.standardDeviation->size() = 1 implies self.variance->size() = 0
591
592 context Coefficient
593   inv VarStdDevCoeff:
594     self.variance->size() = 1 implies self.standardDeviation->size() = 0
595   and self.standardDeviation->size() = 1 implies self.variance->size() = 0
596
597 ---- Constraints for output ----
598
599 context MemoizedCommonSubexpression
600   inv MemoComSubSize:
601     self.col_size = 1
602
603 context OutputCodeVariance
604   inv OCVarSize:
605     self.name.size() > 0
606     and self.row_size = 1
607     and self.col_size = 1
608
609 context OutputCodeVariance
610   inv OCVarValues:
611     self.values->forAll(v | v > 0)
612
613 context OutputCodeMean
614   inv OCMeanSize:
615     self.name.size() > 0
616     and self.row_size >= 1
617     and self.col_size >= 1
618
619 context GaussianModel
620   inv NormDistOrTransfrom:
621     self.normalDistribution->size() = 1 implies self.transformations->size() =
622     0
623   and self.transformations->size() = 1 implies self.normalDistribution->size()
624     = 0
625
626 context GaussianModel
627   inv NormMeanSize:
628     self.normalDistribution->size() = 1
629     implies self.normalDistribution.calculateMeanLoop.outputCodeMean.row_size
630     = 1
631     and self.normalDistribution.calculateMeanLoop.outputCodeMean.col_size = 1
632
633 context GaussianModel
634   inv TransformMeanSize:
635     self.transformations->size() = 1
636     implies self.transformations.calculateMeanLoop.outputCodeMean.row_size = 1
637     and self.transformations.calculateMeanLoop.outputCodeMean.col_size = 1
638
639 context GaussianModel
640   inv TransformCSInitMemoCS:

```

```

638     self.transformations->size() = 1
639     implies self.transformations.commonSubexpressionInitLoop->size() = 1
640     and self.transformations.commonSubexpressionInitLoop.
        memoizedCommonSubexpression->size() = 1
641
642 context GaussianModel
643     inv NormOutInDataCalcMCalcV:
644         self.normalDistribution->size() = 1
645     implies self.normalDistribution.calculateMeanLoop.outputCodeInputData->size
        () = 1
646     and self.normalDistribution.calculateVarianceLoop.outputCodeInputData->size
        () = 2
647
648
649 ---- Constraints connecting input and output ----
650
651 context Mean
652     inv InMeanOutMean:
653         self.name = self.outputCodeMean.name
654     and self.row_size = self.outputCodeMean.row_size
655     and self.col_size = self.outputCodeMean.col_size
656
657 context Variance
658     inv InVarOutVar:
659         self.name = self.outputCodeVariance.name
660     and self.row_size = self.outputCodeVariance.row_size
661     and self.col_size = self.outputCodeVariance.col_size
662
663 context InputData
664     inv InDataOutData:
665         self.name = self.outputCodeInputData.name
666     and self.row_size = self.outputCodeInputData.row_size
667     and self.col_size = self.outputCodeInputData.col_size
668     and self.values = self.outputCodeInputData.values
669
670 context StatisticalModel
671     inv StatModNormDist:
672         self.equation = 'x(_) ~ gauss(mu, sqrt(sigma_sq)).' implies (self.
        gaussianModel.normalDistribution->size() = 1
673     and self.gaussianModel.transformations->size() = 0
674     and self.classParameters.variance->size() = 1)
675
676 context StatisticalModel
677     inv StatModTransformLog:
678         self.equation = 'log(x(_)) ~ gauss(mu, sqrt(sigma_sq)).' implies (self.
        gaussianModel.transformations->size() = 1
679     and self.gaussianModel.transformations.type = 'log'
680     and self.gaussianModel.normalDistribution->size() = 0
681     and self.classParameters.variance->size() = 1)
682
683 context StatisticalModel
684     inv StatModTransformSquare:
685         self.equation = 'x(_)**2 ~ gauss(mu, sqrt(sigma_sq)).' implies (self.
        gaussianModel.transformations->size() = 1

```

```

686     and self.gaussianModel.transformations.type = 'square'
687     and self.gaussianModel.normalDistribution->size() = 1
688     and self.classParameters.variance->size() = 1)
689
690 ---- The constraints from the multiplicities of the CDs ----
691
692 context GaussianModel inv:
693     self.declaration->size() = 1
694     and self.initialization->size() = 1
695     and self.retval->size() = 1
696     and self.matrix->size() >= 1
697     and (self.normalDistribution->size() = 0 or self.normalDistribution->size()
698         = 1)
699     and (self.transformations->size() = 0 or self.transformations->size() = 1)
700
701 context Declaration inv:
702     self.inputDeclaration->size() = 1
703     and self.constantDeclaration->size() = 1
704     and self.outputDeclaration->size() = 1
705     and self.localDeclaration->size() = 1
706     and self.matrix->size() >= 1
707
708 context StatisticalModel inv:
709     self.pragmas->size() >= 0
710     and (self.hiddenVariable->size() = 0 or self.hiddenVariable->size() = 1)
711     and self.goal->size() = 1
712     and self.inputData->size() >= 1
713     and self.classParameters->size() >= 1
714
715 context ClassParameters inv:
716     self.statisticalModel->size() >= 1
717     and ( (self.variance->size() = 0 and self.standardDeviation->size() = 1)
718         or (self.variance->size() = 1 and self.standardDeviation->size() = 0) )
719     and self.mean->size() = 1
720     and self.modelParameters->size() >= 0
721     and self.goal->size() = 1
722
723 context Goal inv:
724     self.statisticalModel->size() >= 1
725     and (self.classProbabilities->size() = 0 or self.classProbabilities->size()
726         = 1)
727     and self.classParameters->size() >= 1
728     and self.inputData->size() >= 1
729
730 context NormalDistribution inv:
731     self.gaussianModel->size() >= 1
732     and self.calculateMeanLoop->size() = 1
733     and self.calculateVarianceLoop->size() = 1
734     and self.checkDivideByZero->size() >= 1
735
736 context Transformations inv:
737     self.gaussianModel->size() >= 1
738     and self.calculateMeanLoop->size() = 1
739     and self.calculateVarianceLoop->size() = 1

```



```

INFO: Start model transformation for 'AUTOBAYES'
WARN: This approach does not support bags. All collections will be handled like sets.
WARN: This approach does not support bags. All collections will be handled like sets.
WARN: This approach does not support bags. All collections will be handled like sets.
WARN: This approach does not support bags. All collections will be handled like sets.
WARN: This approach does not support bags. All collections will be handled like sets.
WARN: This approach does not support bags. All collections will be handled like sets.
WARN: This approach does not support bags. All collections will be handled like sets.
WARN: This approach does not support bags. All collections will be handled like sets.
ERROR: Cannot transform invariant 'Variance::VarSize'. OCL operation size is not supported.
ERROR: Cannot transform invariant 'Mean::MeanSize'. OCL operation size is not supported.
INFO: Use 'modelvalidator -downloadSolvers' to automatically download and install additional solver libraries.
ERROR: Cannot transform invariant 'InputData::InputDataName'. OCL operation size is not supported.
ERROR: Cannot transform invariant 'OutputCodeVariance::OCVarSize'. OCL operation size is not supported.
ERROR: Cannot transform invariant 'OutputCodeMean::OCMeanSize'. OCL operation size is not supported.
WARN: Collect operation 'self.classParameters->collect($e : ClassParameters | $e.variance)' results in unsupported type 'Bag'. It will be interpreted as 'Set'.
WARN: Collect operation 'self.classParameters->collect($e : ClassParameters | $e.variance)' results in unsupported type 'Bag'. It will be interpreted as 'Set'.
WARN: Collect operation 'self.classParameters->collect($e : ClassParameters | $e.variance)' results in unsupported type 'Bag'. It will be interpreted as 'Set'.
INFO: Invariant transformation successful
INFO: Model transformation successful
INFO: Translation time (USE to Kodkod): 681 ms

```

Figure A.8. The initial USE Create Configuration and Validator Readout.

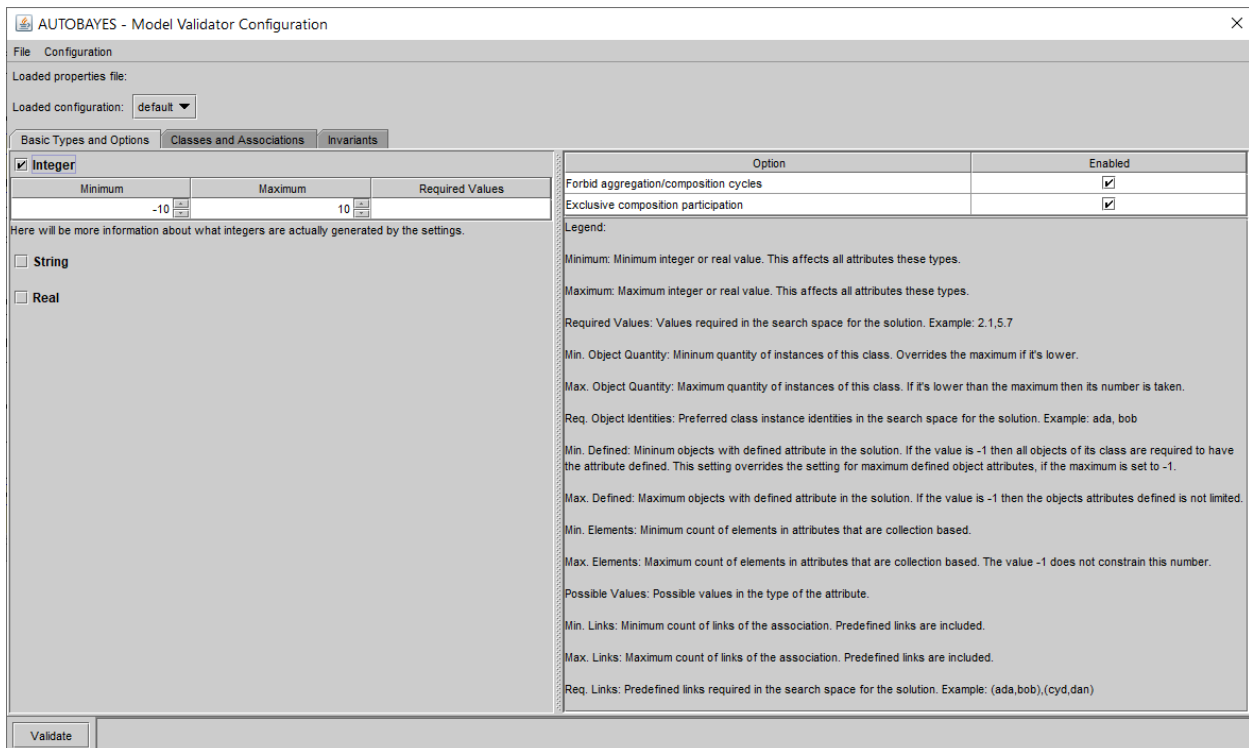


Figure A.9. The USE Model Validator Configuration - Basic Types and Options tab.

AUTOBAYES - Model Validator Configuration

File Configuration

Loaded properties file:

Loaded configuration: default

Basic Types and Options Classes and Associations Invariants

Class	Min. Object Quantity	Max. Object Quantity	Req. Object Identities
StatisticaModel	1	1	
Pragmas	1	1	
HiddenVariable	1	1	
ClassProbabilities	1	1	
Goal	1	1	
ModelParameters	1	1	
InputData	1	1	
ClassParameters	1	1	
StandardDeviation	1	1	
Mean	1	1	
Variance	1	1	
Gaussian	1	1	
Exponent	1	1	
Coefficient	1	1	
Numerator	1	1	
Denominator	1	1	
GaussianModel	1	1	
Retval	1	1	
Declaration	1	1	
InputDeclaration	1	1	
ConstantDeclaration	1	1	
OutputDeclaration	1	1	
LocalDeclaration	1	1	

Attributes of class ModelParameters Show specific bounds

Attribute	Min. Defined	Max. Defined	Min. Elements	Max. Elements	Possible Values
n_classes	*	*			
n_points	*	*			
n_variables	*	*			

Associations of class ModelParameters

Association	Min. Links	Max. Links	Req. Links
ModParaHidVar (modeParame...	1	1	
ModParaClassProb (modePar...	1	1	
ModParaInData (modeParamet...	1	1	
ModParaClassPara (modeIPar...	1	1	
ModParaGaussMod (modeIPar...	1	1	

Abstract Classes:
None.

Validate

Figure A.10. The USE Model Validator Configuration - Classes and Associations tab.

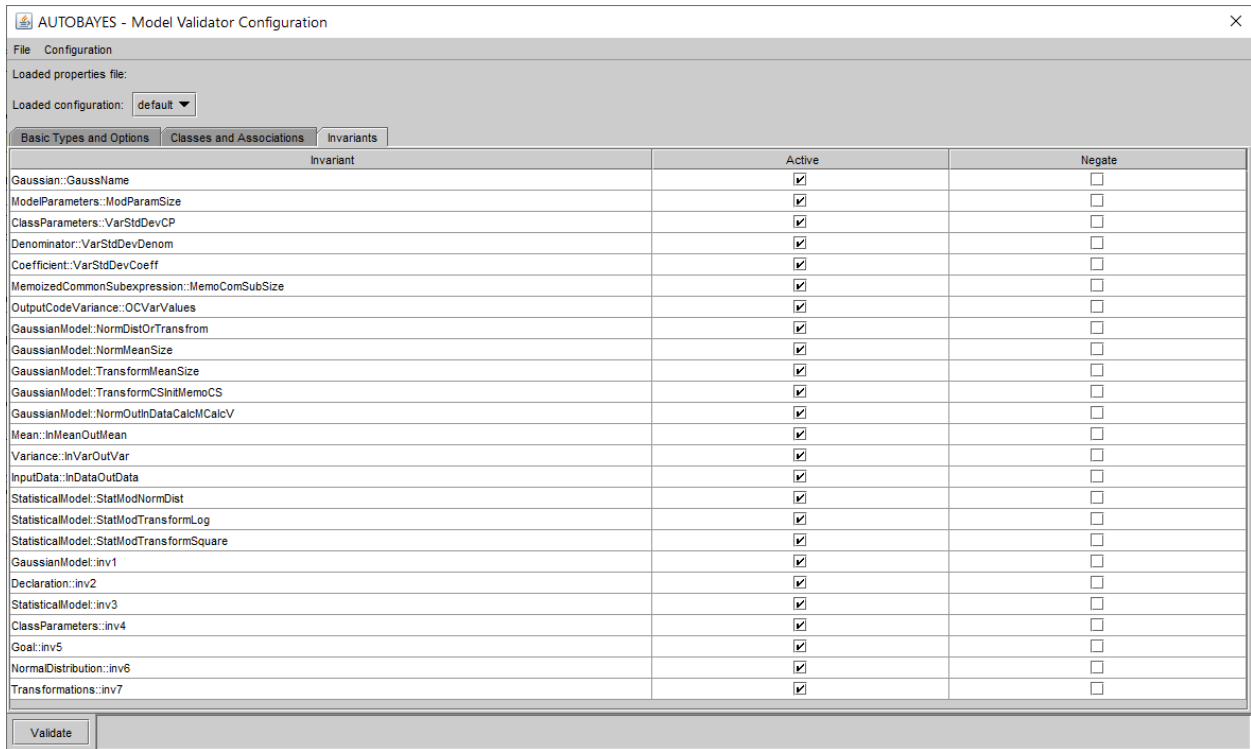


Figure A.11. The USE Model Validator Configuration - Invariants tab.

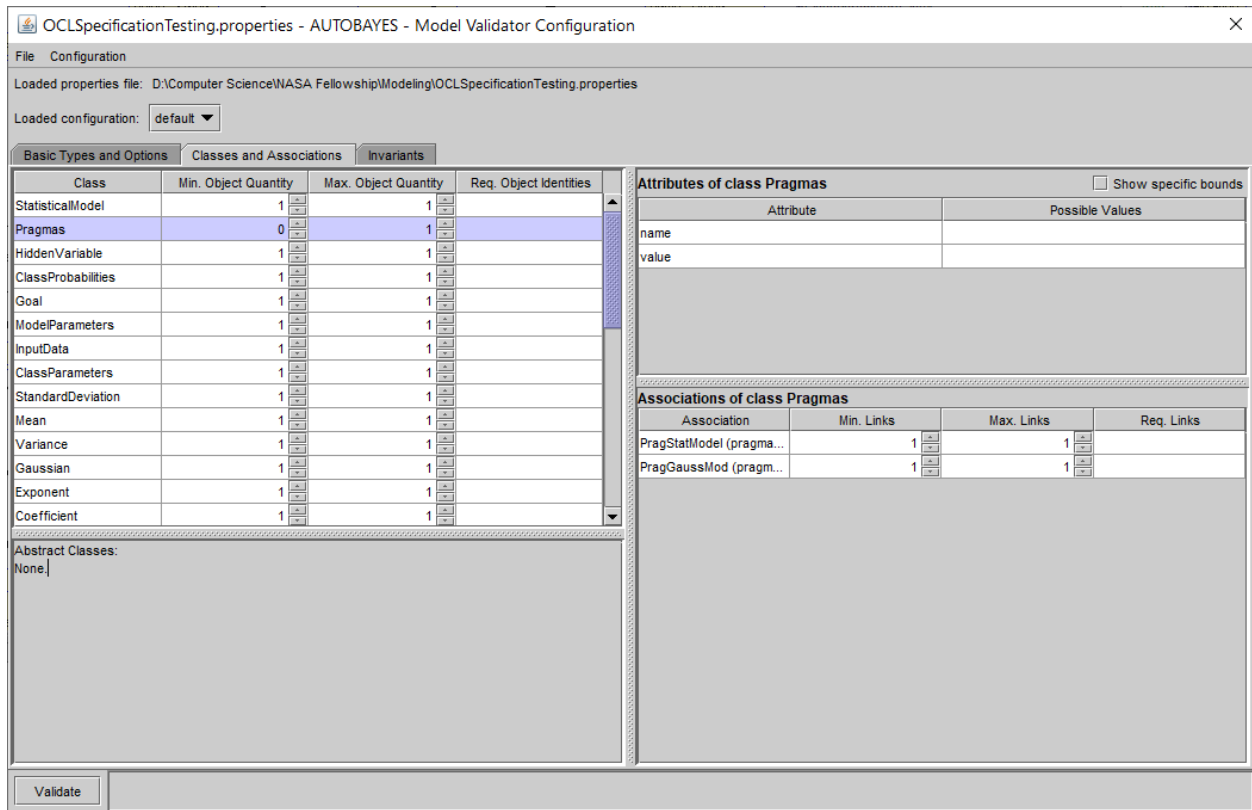


Figure A.12. The USE Model Validator Configuration - Classes and Associations tab with fix for the Pragma Class error.

OCLSpecificationTesting.properties - AUTOBAYES - Model Validator Configuration

File Configuration

Loaded properties file: D:\Computer Science\WASA Fellowship\Modeling\OCLSpecificationTesting.properties

Loaded configuration: default

Basic Types and Options Classes and Associations Invariants

Class	Min. Object Quantity	Max. Object Quantity	Req. Object Identities
StatisticalModel	1	1	
Pragmas	0	1	
HiddenVariable	0	1	
ClassProbabilities	0	1	
Goal	1	1	
ModelParameters	0	1	
InputData	0	1	
ClassParameters	0	1	
StandardDeviation	0	1	
Mean	1	1	
Variance	0	1	
Gaussian	1	1	
Exponent	1	1	
Coefficient	1	1	
Numerator	1	1	
Denominator	1	1	
GaussianModel	1	1	
Retval	1	1	
Declaration	0	1	
InputDeclaration	1	1	
ConstantDeclaration	1	1	
OutputDeclaration	1	1	
LocaDeclaration	1	1	
Initialization	0	1	
Matrix	1	1	
OutputCodeMean	1	2	
OutputCodeVariance	0	1	
OutputCodeInputData	1	2	
MemoizedCommonSubexpr...	1	2	
NormalDistribution	0	1	
CalculateVarianceLoop	0	1	
CalculateMeanLoop	0	1	
Transformations	0	1	
CommonSubexpressionInt...	0	1	
CheckDivideByZero	1	1	

Abstract Classes:
None.

Validate

Attributes of class InputData Show specific bounds

Attribute	Possible Values
col_size	
name	
row_size	
values	

Associations of class InputData

Association	Min. Links	Max. Links	Req. Links
InDataGoal (inputData.In...	1	1	
InDataNum (inputData.In...	1	1	
InDataMean (inputData:L...	1	1	
InInputDataGaussMod (i...	1	1	
InInputDataOutputData...	1	1	
StatModelInData (statisti...	1	1	
ModParainData (modeIP...	1	1	

Figure A.13. The USE Model Validator Configuration - Classes and Associations tab with fix for each of the Classes that can have a multiplicity of 0.

Basic Types and Options				Classes and Associations				Invariants			
Class	Min. Object Quantity		Max. Object Quantity		Req. Object Identities						
StatisticalModel	1		1								
Pragmas	0		1								
HiddenVariable	0		1								
ClassProbabilities	0		1								
Goal	0		1								
ModelParameters	0		1								
InputData	0		1								
ClassParameters	0		1								
StandardDeviation	0		1								
Mean	1		1								
Variance	0		1								

Figure A.14. The USE Model Validator Configuration - Classes and Associations tab with fix for the Goal Class error.

Basic Types and Options				Classes and Associations				Invariants			
Class	Min. Object Quantity		Max. Object Quantity		Req. Object Identities						
StatisticalModel	1		1								
Pragmas	0		1								
HiddenVariable	0		1								
ClassProbabilities	0		1								
Goal	0		1								
ModelParameters	0		1								
InputData	0		1								
ClassParameters	0		1								
StandardDeviation	0		1								
Mean	0		1								
Variance	0		1								
Gaussian	1		1								

Figure A.15. The USE Model Validator Configuration - Classes and Associations tab with fix for the Mean Class error.

Basic Types and Options		Classes and Associations		Invariants
Class	Min. Object Quantity	Max. Object Quantity	Req. Object Identities	
Mean	0	1		
Variance	0	1		
Gaussian	0	1		
Exponent	1	1		

Figure A.16. The USE Model Validator Configuration - Classes and Associations tab with fix for the Gaussian Class error.

Basic Types and Options		Classes and Associations		Invariants
Class	Min. Object Quantity	Max. Object Quantity	Req. Object Identities	
Numerator	1	1		
Denominator	1	1		
GaussianModel	0	1		
Retval	1	1		
Declaration	0	1		

Figure A.17. The USE Model Validator Configuration - Classes and Associations tab with fix for the GaussianModel Class error.

REFERENCES

- [1] J. Schumann, H. Jafari, T. Pressburger, E. Denney, W. Buntine, and B. Fischer, “AutoBayes Program Synthesis System Users Manual,” NASA Aeronautics and Space Administration, NASA/TM—2008–215366, Tech. Rep., 2008.
- [2] A. Church, “Applications of recursive arithmetic to the problem of circuit synthesis,” *Summaries of the Summer Institute of Symbolic Logic*, vol. 1, pp. 3–50, 1957.
- [3] J. R. Buchi and L. H. Landweber, “Solving Sequential Conditions by Finite-State Strategies,” *Transactions of the American Mathematical Society*, vol. 138, p. 295, apr 1969. [Online]. Available: <https://www.jstor.org/stable/1994916?origin=crossref>
- [4] Z. Manna and R. Waldinger, “A Deductive Approach to Program Synthesis,” *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 90–121, jan 1980. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=357084.357090>
- [5] A. Solar-Lezama, “Program synthesis by sketching,” Ph.D. dissertation, University of California, Berkeley, 2008.
- [6] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, “Syntax-guided synthesis,” in *2013 Formal Methods in Computer-Aided Design*. IEEE, oct 2013, pp. 1–8. [Online]. Available: <http://ieeexplore.ieee.org/document/6679385/>
- [7] C. Volkstorf, “Program Synthesis from Axiomatic Proof of Correctness,” jan 2015. [Online]. Available: <http://arxiv.org/abs/1501.01363>
- [8] S. Gulwani, “Dimensions in program synthesis,” in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '10*. New York, New York, USA: ACM Press, 2010, pp. 13–24. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1836089.1836091>
- [9] X. Wang, I. Dillig, and R. Singh, “Program synthesis using abstraction refinement,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, dec 2017. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3177123.3158151>
- [10] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, “RobustFill: Neural Program Learning under Noisy I/O,” mar 2017. [Online]. Available: <http://arxiv.org/abs/1703.07469>
- [11] S. Gulwani, O. Polozov, and R. Singh, “Program Synthesis,” *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: <http://www.nowpublishers.com/article/Details/PGL-010>

- [12] R. Simmons-Edler, A. Miltner, and S. Seung, “Program Synthesis Through Reinforcement Learning Guided Tree Search,” jun 2018. [Online]. Available: <http://arxiv.org/abs/1806.02932>
- [13] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, “The Gen Voca Model of Software-System Generators,” *IEEE Software*, 1994.
- [14] G. Rosu and J. Whittle, “Towards certifying domain-specific properties of synthesized code,” in *Proc. VCL’02: Third Int’l Workshop on Verification and Computational Logic*, Pittsburgh, PA, 2002, pp. 46–56.
- [15] J. Schumann and P. Robinson, “[] or Success is Not Enough: Current Technology and Future Directions in Proof Presentation,” in *Workshop IJCAR 2001: Int’l Joint Conf. on Automated Reasoning*, Siena, Italy, 2001, pp. 702–710.
- [16] G. Rosu and J. Whittle, “Towards certifying domain-specific properties of synthesized code,” in *Proceedings 17th IEEE International Conference on Automated Software Engineering*,. IEEE Comput. Soc, 2002, pp. 289–294. [Online]. Available: <http://ieeexplore.ieee.org/document/1115032/>
- [17] J. Whittle and J. Schumann, “Automating the implementation of Kalman filter algorithms,” *ACM Transactions on Mathematical Software*, 2004.
- [18] J. Whittle, J. Van Baalen, J. Schumann, P. Robinson, T. Pressburger, J. Penix, P. Oh, M. Lowry, and G. Brat, “Amphion/NAV: deductive synthesis of state estimation software,” 2005.
- [19] W. Buntine, B. Fischer, and T. Pressburger, “Towards automated synthesis of data mining programs,” 1999.
- [20] B. Fischer and J. Schumann, “AutoBayes: A System for the Synthesis of Data Analysis Programs,” in *Proc. NIPS 2000: Workshop on Software Support for Bayesian Analysis Systems*, Breckenridge, CO, 2000.
- [21] B. Fischer, T. Pressburger, G. Rosu, and J. Schumann, “The AutoBayes Program Synthesis System - System Description,” in *Proc. CALCULEMUS 2001: 9th Symp. on the Integration of Symbolic Computation and Mechanized Reasoning*, Siena, Italy, 2001, pp. 182–187.
- [22] B. Fisher and J. Schumann, “Automated Synthesis of Statistical Data Analysis Programs,” in *Proc. SDP’02: Workshop Science Data Processing*, Greenbelt, MD, 2002.
- [23] B. Fischer and J. Schumann, “AutoBayes: A system for generating data analysis programs from statistical models,” 2003.
- [24] K. A. Huyser, “Discovering Planetary Nebula Geometries: Explorations with a Hierarchy of Models,” 2004.

- [25] K. Gundy-Burlet, J. Schumann, T. Menzies, and T. Barrett, “Parametric Analysis of Antares Re-Entry Guidance Algorithms using Advanced Test Generation and Data Analysis,” in *Proc. i-SAIRAS’08: 9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, Universal City, California, 2008.
- [26] E. S. Grant, “Defining Domain-Specific Object-Oriented Modeling Languages as UML Profiles,” PhD thesis, Colorado State University, Ft. Collins, Colorado, USA, 2002.
- [27] E. S. Grant, J. Whittle, and R. Chennamaneni, “Checking Program Synthesizer Input/Output,” in *OOPSLA2003: 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, 2003, pp. 395–399.
- [28] M. Whalen, J. Schumann, and B. Fischer, “Synthesizing certified code,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2002.
- [29] J. Schumann, “Automatic Synthesis of Safety-Related Software -Short Paper-,” *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 2, pp. 105–108, 2002.
- [30] M. Whalen, J. Schumann, and B. Fischer, “AutoBayes/CC-Combining Program Synthesis with Automatic Code Certification -System Description-,” in *Proc. CADE-18: Conf. on Automated Deduction*. Copenhagen, Denmark: Springer LNCS, vol. 2392, 2002, pp. 290–294.
- [31] J. Schumann, B. Fischer, M. Whalen, and J. Whittle, “Certification support for automatically generated programs,” in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences, HICSS 2003*, 2003.
- [32] G. Roşu, R. P. Venkatesan, J. Whittle, and L. Leuştean, “Certifying optimality of state estimation programs,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2003.
- [33] E. Denney and B. Fischer, “Correctness of source-level safety policies,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2003.
- [34] J. Richardson and J. Green, “Traceability Through Automatic Program Generation,” *Proceedings of Second International Workshop on Traceability in Emerging Forms of Software Engineering TEFSE 2003*, 2003.
- [35] E. Denney, B. Fischer, and J. Schumann, “Adding assurance to automatically generated code,” in *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, 2004.
- [36] E. Denney, B. Fischer, and J. Schumann, “Using automated theorem provers to certify auto-generated aerospace software,” in *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, 2004.

- [37] E. Denney and R. P. Venkatesan, “A generic software safety document generator,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2004.
- [38] E. Denney, B. Fischer, J. Schumann, and J. Richardson, “Automatic certification of kalman filters for reliable code generation,” in *IEEE Aerospace Conference Proceedings*, 2005.
- [39] J. Richardson, J. Schumann, B. Fischer, and E. Denney, “Rapid exploration of the design space during automatic generation of kalman filter code,” in *IEEE Aerospace Conference Proceedings*, 2005.
- [40] E. Denney and B. Fischer, “A Program Certification Assistant Based on Fully Automated Theorem Provers,” in *Proc. UITP’05: International Workshop on User Interfaces for Theorem Provers*, Edinburgh, Scotland, 2005.
- [41] G. Sutcliffe, E. Denney, and B. Fischer, “Practical Proof Checking for Program Certification,” in *Proc. ESCAR’05: Empirically Successful Classical Automated Reasoning*, Tallin, Estonia, 2005.
- [42] E. Denney and B. Fischer, “Formal Safety Certification of Aerospace Software,” in *Proc. AIAA’05: Infotech at Aerospace*, Arlington, VA, 2005.
- [43] E. Denney and B. Fischer, “Certifiable Program Generation,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2005, pp. 17–28. [Online]. Available: http://link.springer.com/10.1007/11561347_3
- [44] E. Denney and B. Fischer, “Software Certification and Software Certificate Management Systems,” in *Proc. SoftCeMent’05: 2005 ASE Workshop on Software Certificate Management*, Long Beach, CA, 2005, pp. 1–5.
- [45] E. Denney, B. Fischer, D. Hutter, and M. Jones, “2005 ASE Workshop on Software Certificate Management,” in *Proc. SoftCeMent’ 05: 2005 ASE Workshop on Software Certificate Management*, Long Beach, CA, 2005.
- [46] E. Denney and B. Fischer, “Formal Approaches to Ensuring the Safety of Space Software,” in *Proc. Brazilian Sym on Formal M*, Porto Alegre, Brazil, 2005.
- [47] E. Denney and B. Fischer, “Explaining Verification Conditions,” in *Proc. Formal Methods 2006*, Hamilton, Ontario, Canada, 2006.
- [48] E. Denney, B. Fischer, and J. Schumann, “An empirical evaluation of automated theorem provers in software certification,” in *International Journal on Artificial Intelligence Tools*, 2006.
- [49] E. Denney and B. Fischer, “Annotation inference for safety certification of automatically generated code (extended abstract),” in *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*, 2006.

- [50] E. Denney and B. Fischer, "A generic annotation inference algorithm for the safety certification of automatically generated code," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE'06*, 2006.
- [51] E. Denney and B. Fischer, "Extending source code generators for evidence-based software certification," in *Proceedings - ISoLA 2006: 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2006.
- [52] E. Denney, B. Fischer, and C. Pasareanu, "A High-Level Certification Language for Automatically Generated Code," in *Proc. Generative Programming and Component Engineering (GPCE'07)*, Salzburg, Austria, 2007.
- [53] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," in *IEEE Aerospace Conference Proceedings*, 2008.
- [54] M. Lowry, "Intelligent software engineering tools for NASA's crew exploration vehicle," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.
- [55] E. Denney and B. Fischer, "Explaining Verification Conditions," in *Proc. AMAST '08: 12th International Conference on Algebraic Methodology and Software Technology*, Urbana, Illinois, 2008.
- [56] N. Basir, E. Denney, and B. Fischer, "Constructing a safety case for automatically generated code from formal program verification information," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.
- [57] E. Denney and B. Fischer, "Generating customized verifiers for automatically generated code," in *GPCE'08: Proceedings of the ACM SIGPLAN 7th International Conference on Generative Programming and Component Engineering*, 2008.
- [58] N. Basir, E. Denney, and B. Fischer, "Deriving safety cases from automatically constructed proofs," in *IET Conference Publications*, 2009.
- [59] N. Basir, E. Denney, and B. Fischer, "Deriving Safety Cases for the Formal Safety Certification of Automatically Generated Code," *Electronic Notes in Theoretical Computer Science*, 2009.
- [60] N. G. Leveson, "Safety as a system property," *Communications of the ACM*, vol. 38, no. 11, p. 146, nov 1995. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=219717.219816>
- [61] L. E. G. Martins and T. Gorschek, "Requirements engineering for safety-critical systems: A systematic literature review," 2016.
- [62] S. Ian, *Software Engineering*, 10th ed. Pearson Education, Inc., 2016.
- [63] L. Hall, "2015 NASA Technology Roadmaps," Tech. Rep., 2015.

- [64] D. Terrier, “2020 NASA Technology Taxonomy - TX 11: Software, Modeling, Simulation, Information Processing,” Tech. Rep., 2020. [Online]. Available: <https://www.nasa.gov/offices/oct/taxonomy/index.html>
- [65] D. Terrier, F. Chandler, J.-F. Barthelemy, R. Clayton, P. Hughes, H. Partridge, L. Barbier, T. Chytka, P. Desai, A. Eckel, H. Grant, M. Seablom, N. Mendonca, C. Moore, M. Weyland, T. Spagnuolo, J. Spragu, E. Gresham, C. Mullins, and T. Doom, “NASA Strategic Technology Investment Plan,” Tech. Rep., 2017.
- [66] NASA Contributors, “NASA 2018 Strategic Plan,” Tech. Rep., 2018.
- [67] O. M. Group, “OMG Unified Modeling Language TM (OMG UML), Version 2.5,” *InformatikSpektrum*, 2017.
- [68] M. Gogolla, F. Büttner, and M. Richters, “USE: A UML-based specification environment for validating UML and OCL,” *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, dec 2007. [Online]. Available: http://useocl.sourceforge.net/w/index.php/Main_Page <https://linkinghub.elsevier.com/retrieve/pii/S0167642307001608>
- [69] S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*. Upper Saddle River, NJ: Prentice Hall, Inc., 2010.
- [70] G. C. Necula, “Proof-carrying code,” in *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 1997.
- [71] H. Wasserman and M. Blum, “Software reliability via run-time result-checking,” *Journal of the ACM*, 1997.