January 2020

Automation And Visualization Of Program Correctness For Automatically Generating Code

Jason Michael Hicks

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

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
10A2B20E61AE418...
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
2E0AF880C733409...
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

Jason M. Hicks
9/10/2020

Contents

List of Figures

List of Tables

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 successful 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 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 MATLAB^{TM} 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 the 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 import 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 distribution 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 MATLAB^{TM} and Octave environments [1, 19–25].

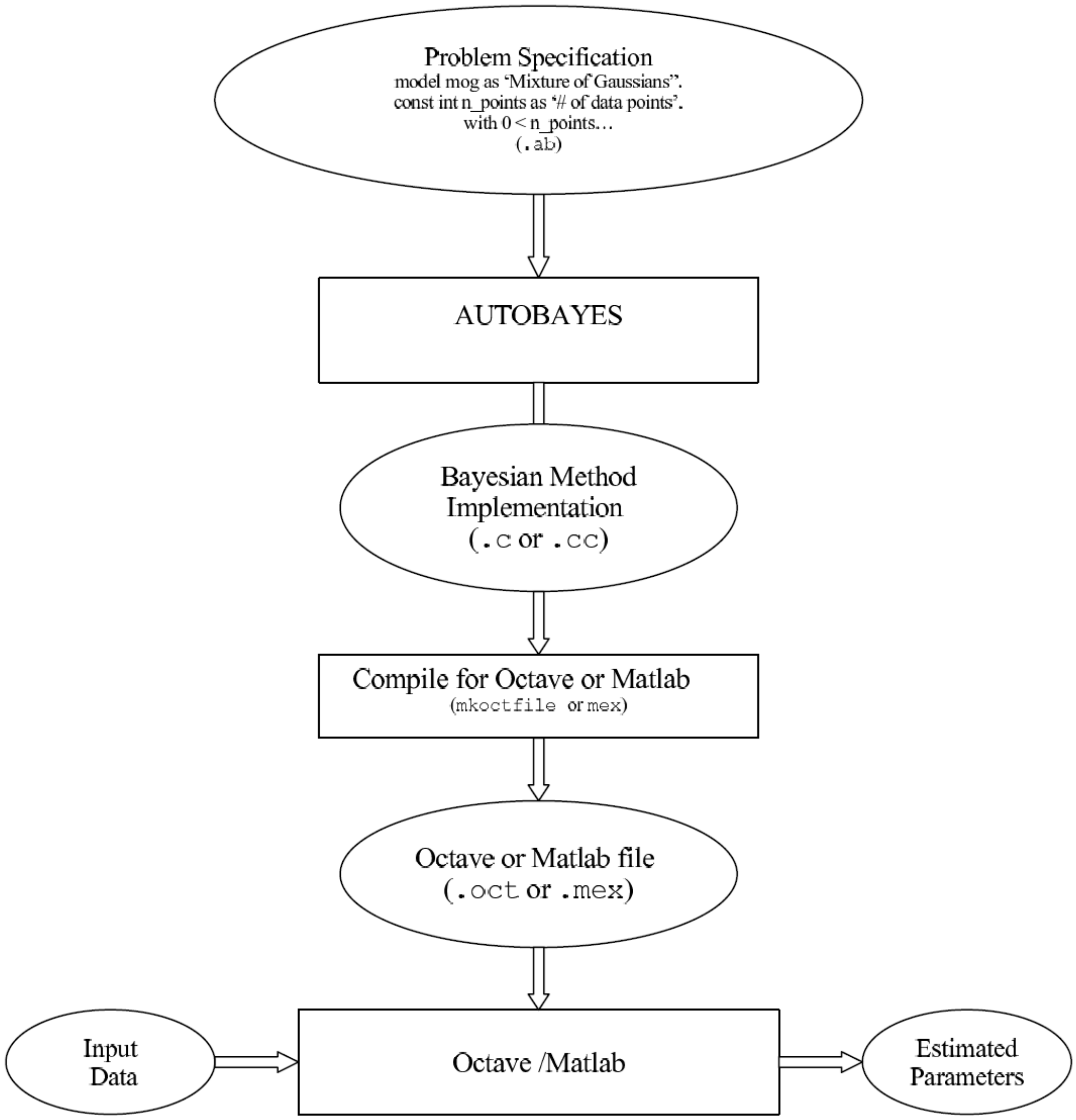

Figure 2.1. AUTOBAYES system architecture [1].

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	$x \sim bernoulli(p)$	Y	
Beta	$x \sim beta(\alpha, \beta)$	N	1
Binomial	$x \sim binomial(n, p)$	Y	2
Cauchy	$x \sim cauchy(x, y)$	N	1
Exponential	$x \sim exp(\lambda)$	Y	
Gamma	$x \sim gamma(k, \theta)$	Y	k known
Gamma	$x \sim gamma(k, \theta)$	N	1
Gauss	$x \sim gauss(\mu, \sigma^2)$	Y	
Poisson	$x \sim poisson(\lambda)$	Y	
vonMises	$x \sim vonmises(\mu, k)$	Y	3
Weibull	$x \sim weibull(\alpha, \beta)$	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 is is normally distributed.

The 1D Gaussian equation is

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

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

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

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

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

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

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

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 orientation 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.

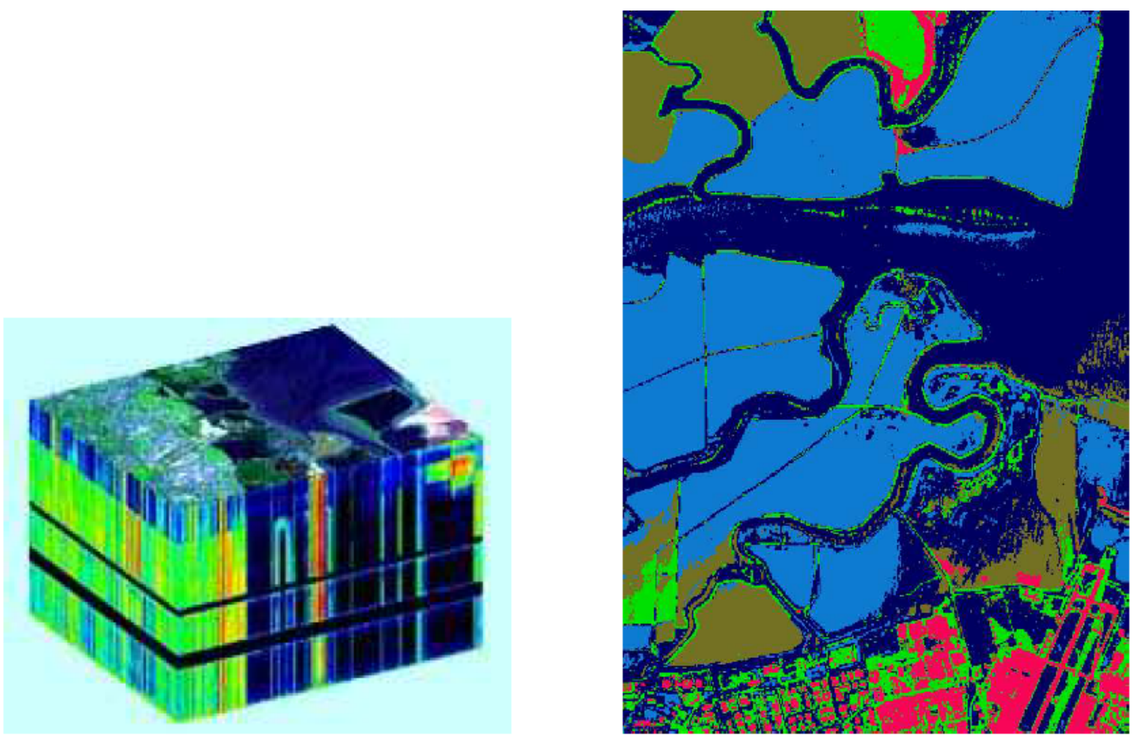

Figure 2.2. Left image: Hyperspectral image cube (MODIS). Right image: Clustering result for hyperspectral data and 5 classes as produced by AUTOBAYES [1].

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 publish 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 algorithms 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.

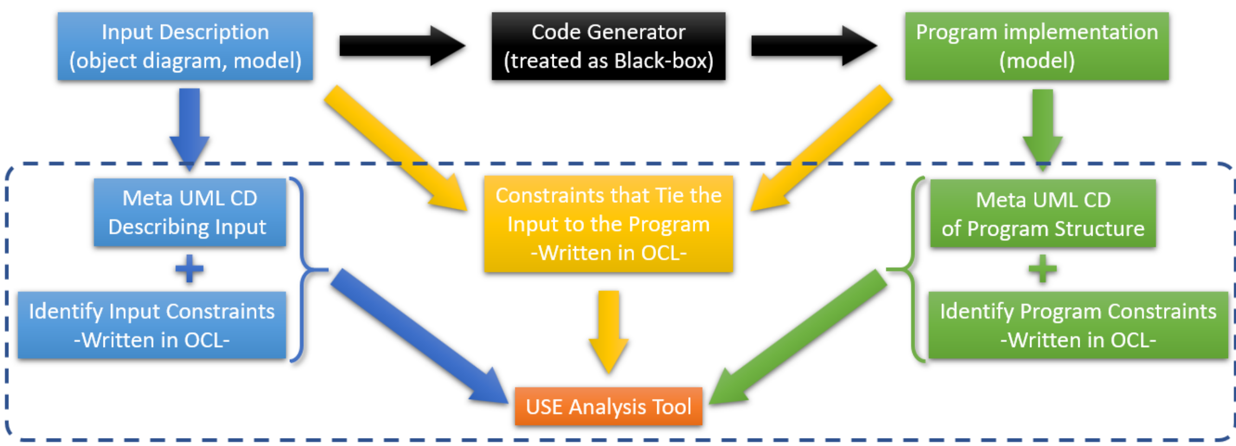

Figure 3.1. Diagrammatic representation of approach to demonstrate program correctness.

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.

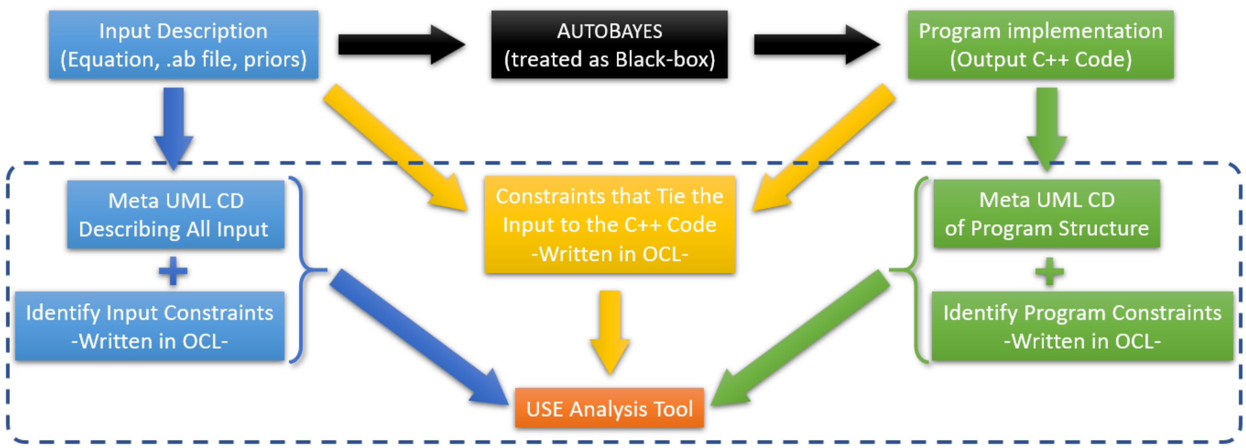

Figure 4.1. Diagrammatic representation of approach to demonstrate program correctness applied to AUTOBAYES.

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 gerneralization 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.

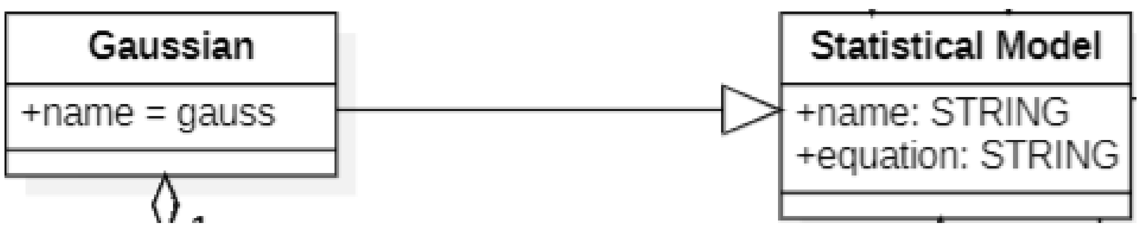

Figure 4.2. Examples of classes representing potential input into AUTOBAYES.

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.

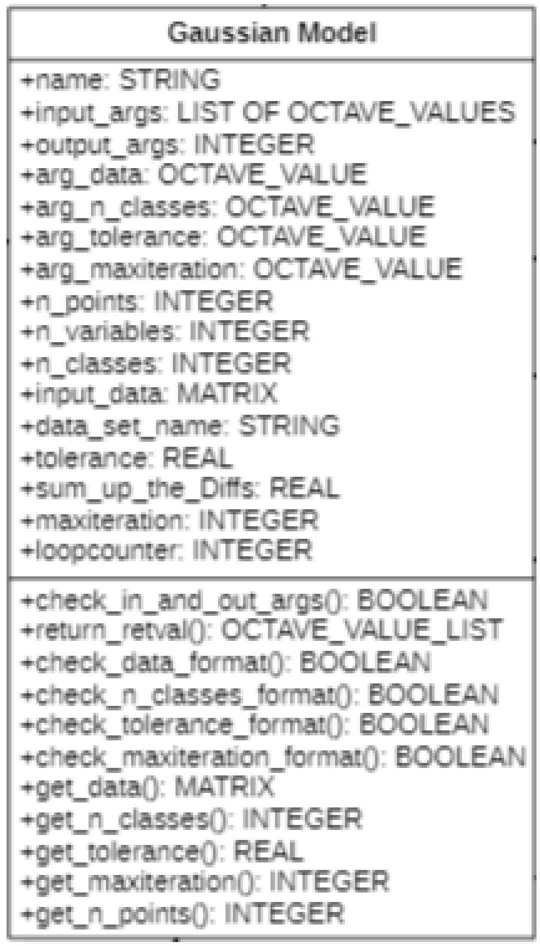

Figure 4.3. The Gaussian Model Class of the AUTOBAYES output code.

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 Dijksta'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.

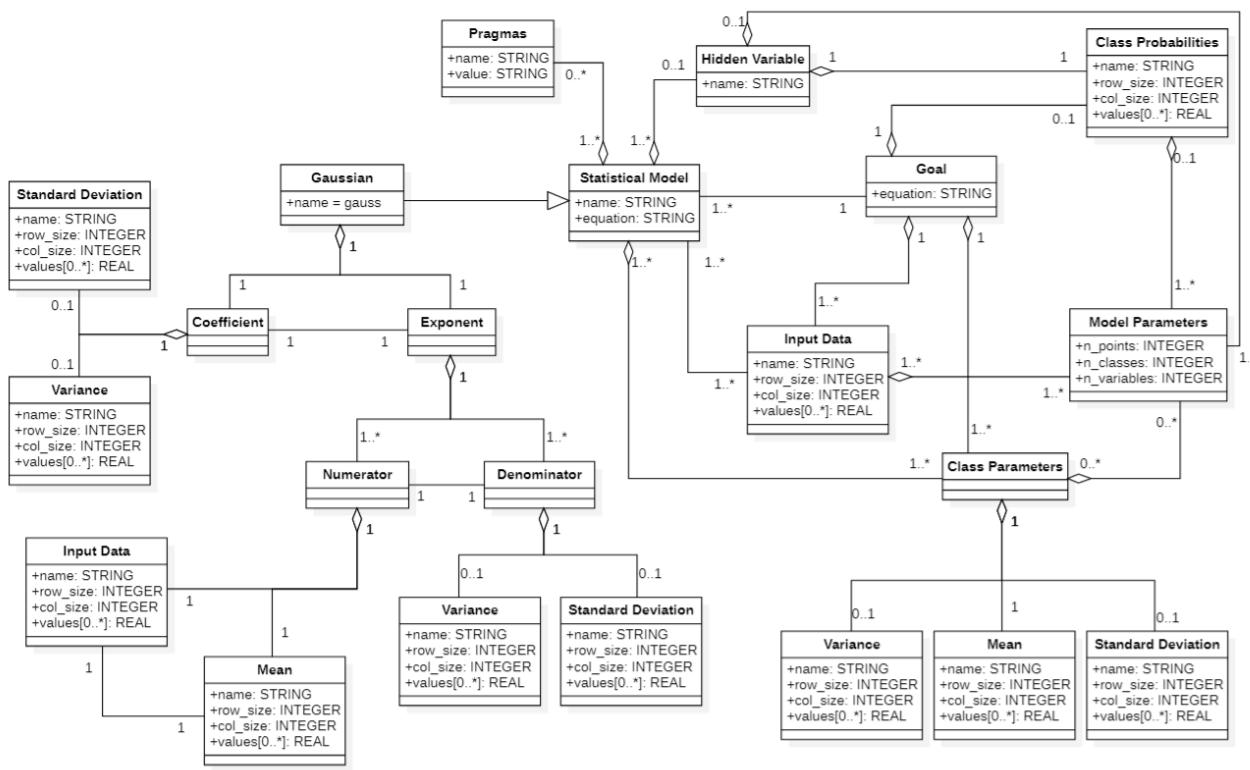

Figure 5.1. Input CD for AUTOBAYES when a Normal distribution of data is assumed and thus the Gaussian equation is used.

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 into smaller "sub-class diagrams or sub-CDs and describe various aspects of them over the next few pages.

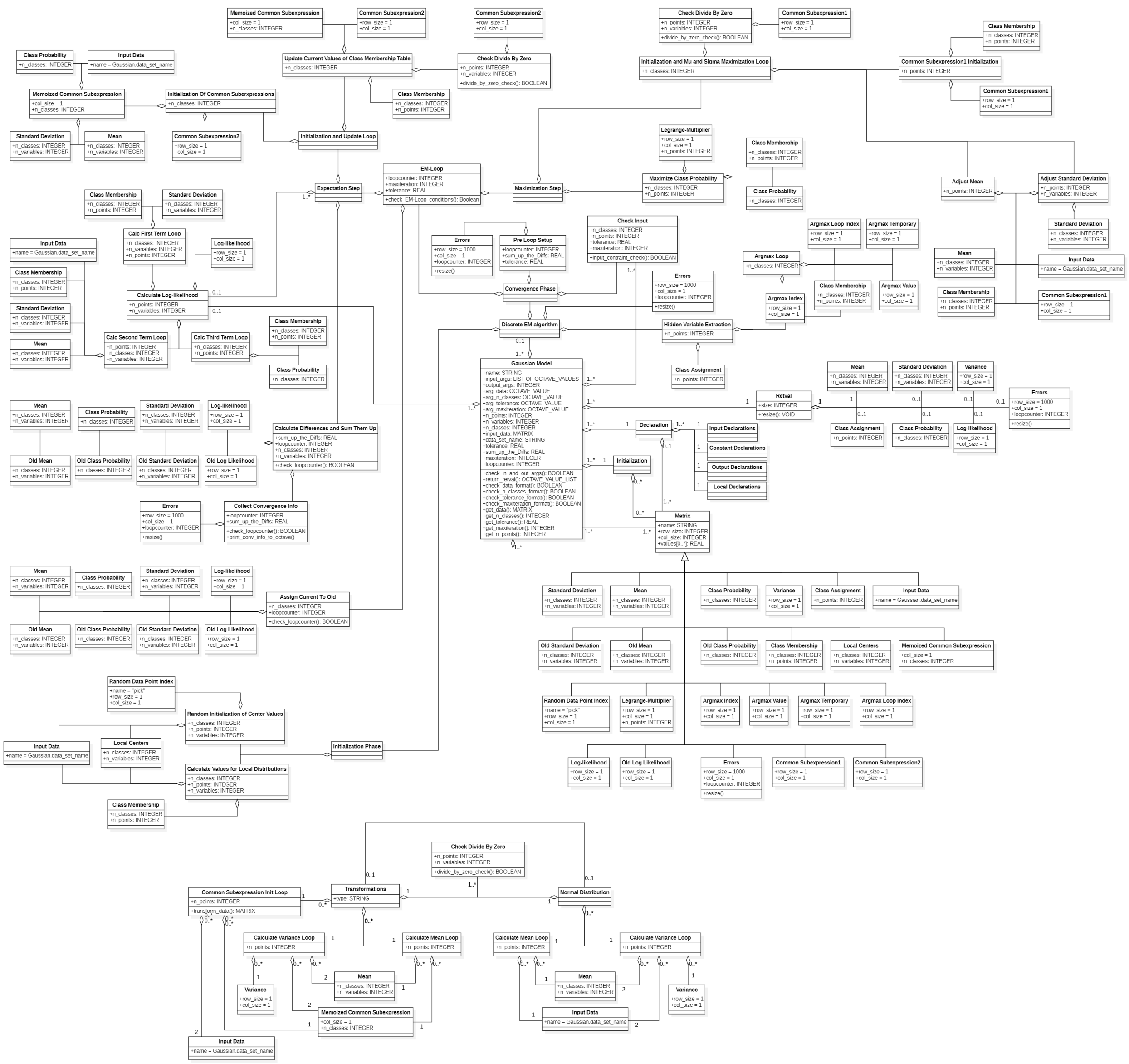

Figure 5.2. Output CD.

A sub-CD for the section of the output CD that gives all of the possible Matrices that can be produced in the output code from AUTOBAYES is given below in Figure 5.3. Each of the various classes listed here as a generalization of a the Matrix class represent important variables (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.

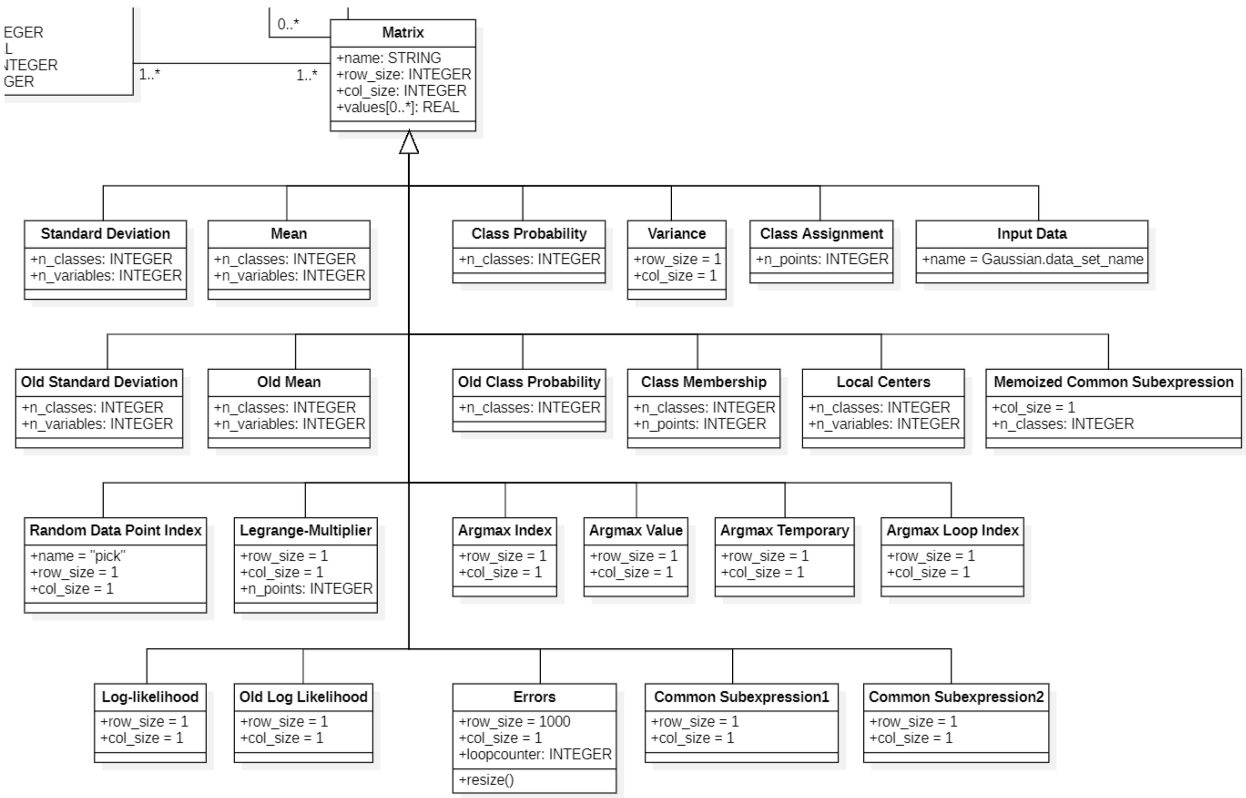

Figure 5.3. Output CD - Matrices.

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.

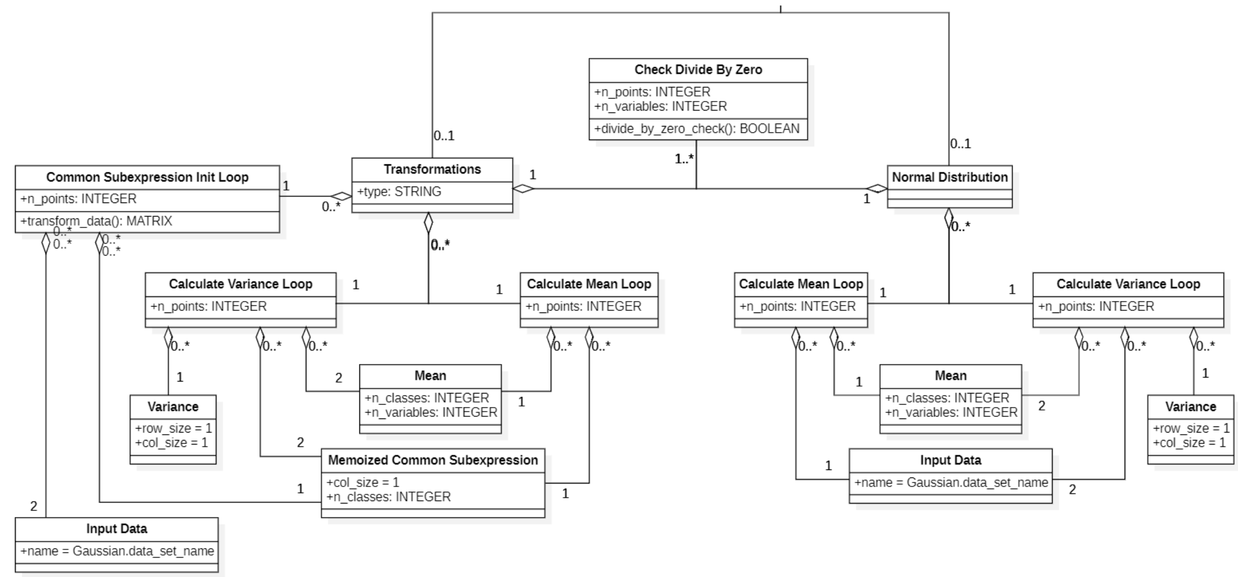

Figure 5.4. Output CD - Normal Distribution and Transformations.

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.

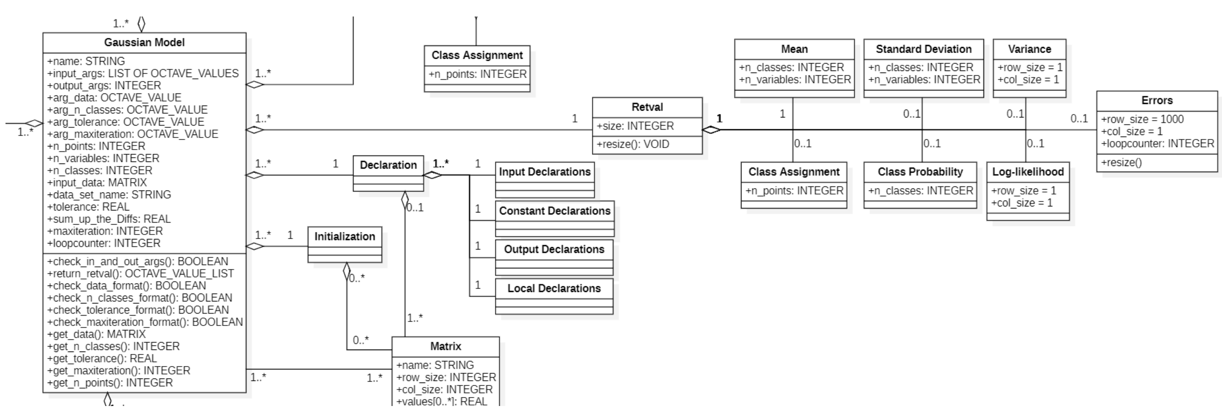

Figure 5.5. Output CD - Gaussian and Retval.

Next, the Discrete EM-algorithm Class is focused upon. This appears in the code when a mixture of Gaussians or multivarient Gaussians are used in the AUTOBAYES input files (e.g., Figures A.4, A.5, and A.6). When present in the code, this is made up of an initialization phase, a hidden variable extraction, and a convergence phase.

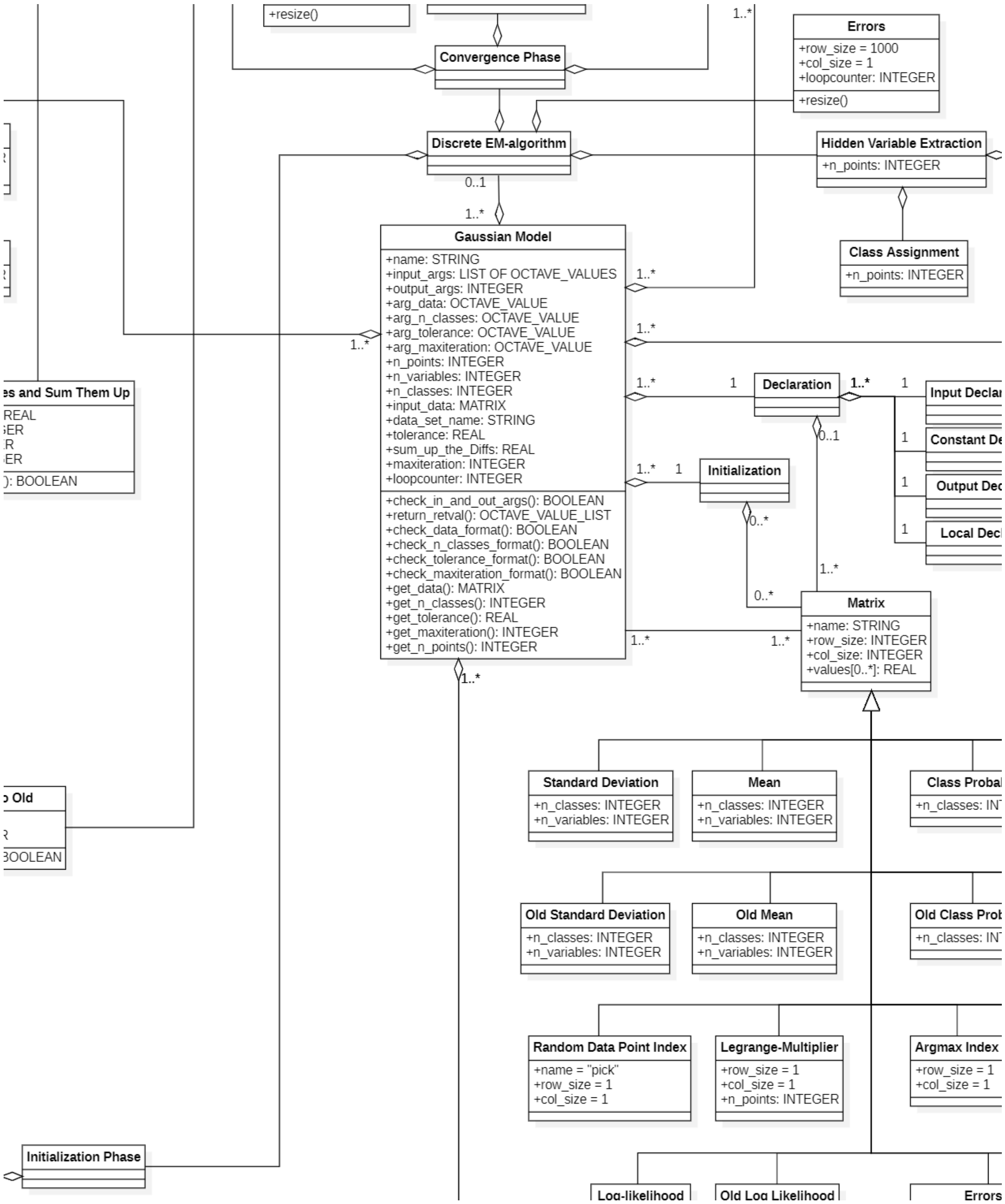

Figure 5.6. Output CD - Discrete EM.

First, the Initialization Phase Class of the Discrete EM-algorithm Class is given below in

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

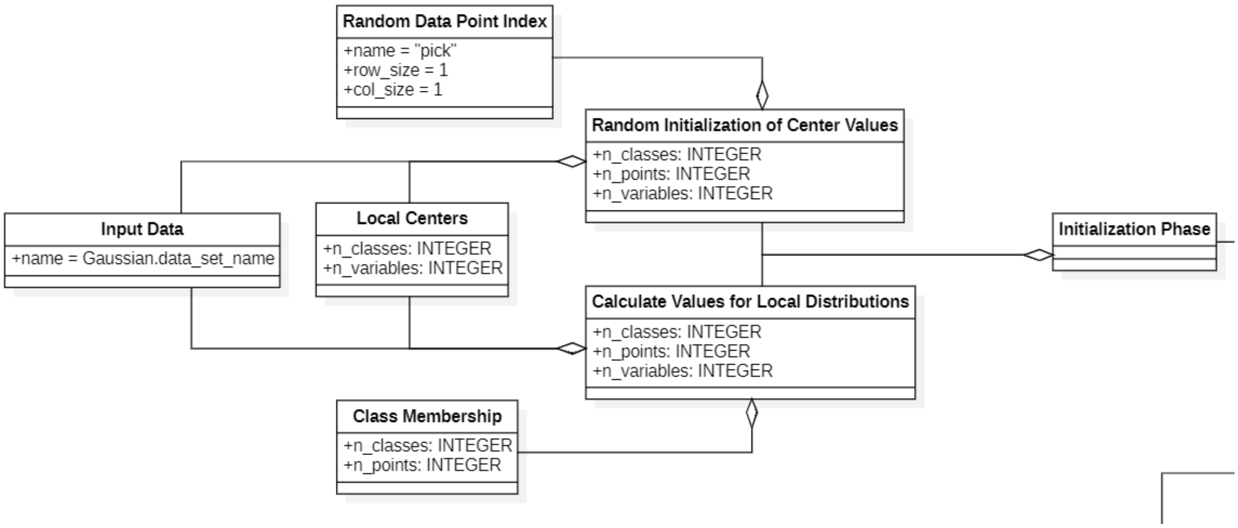

Figure 5.7. Output CD - Initialization Phase.

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

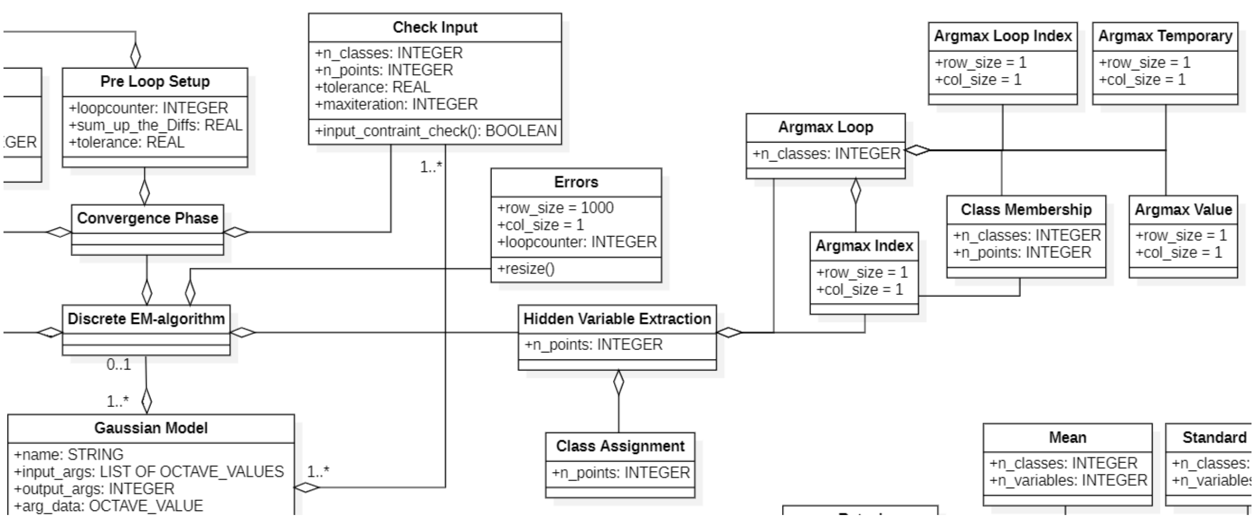

Figure 5.8. Output CD - Hidden Variable Extraction.

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.

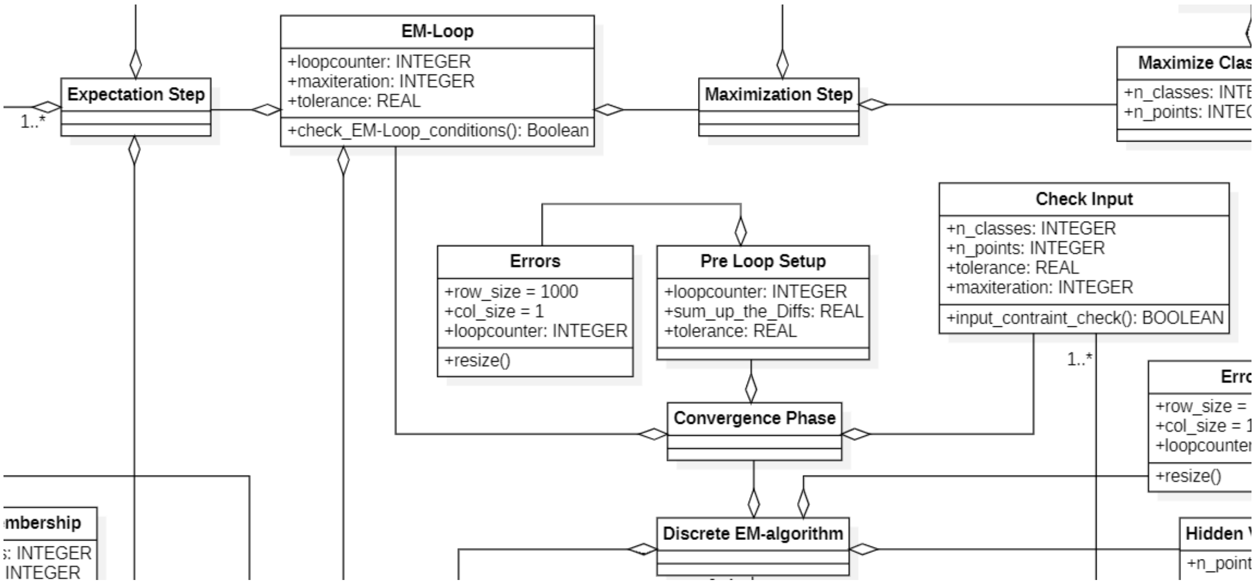

Figure 5.9. Output CD - Convergence Phase.

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.

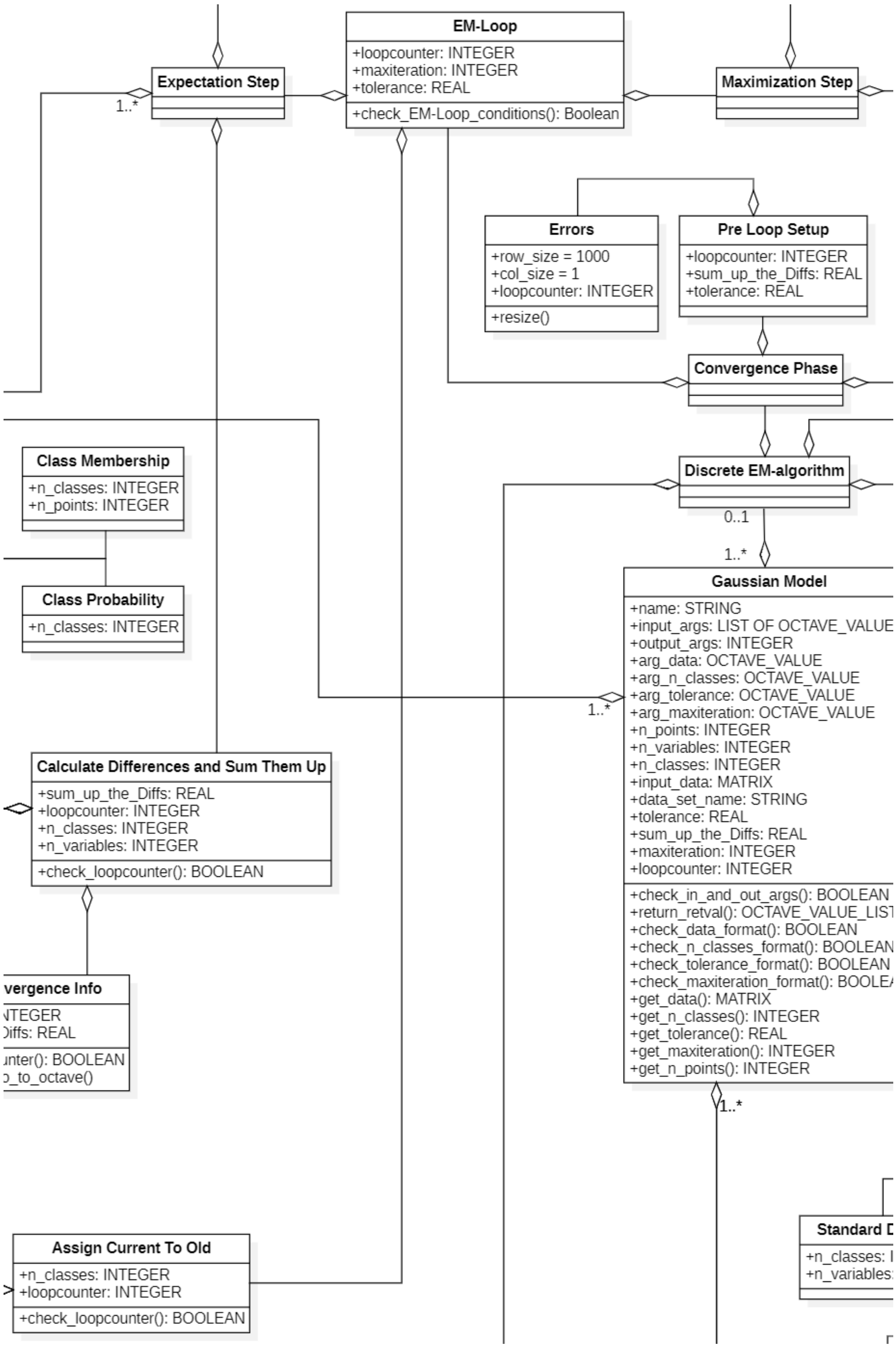

Figure 5.10. Output CD - EM Loop.

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.

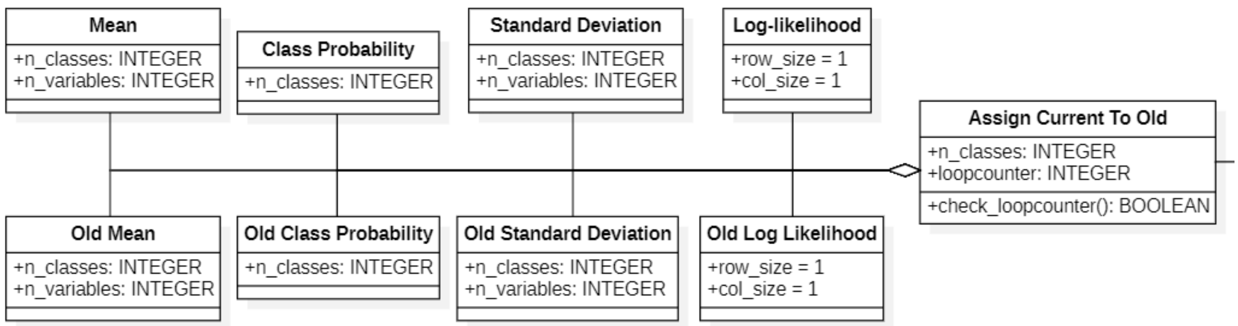

Figure 5.11. Output CD - Current To Old.

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.

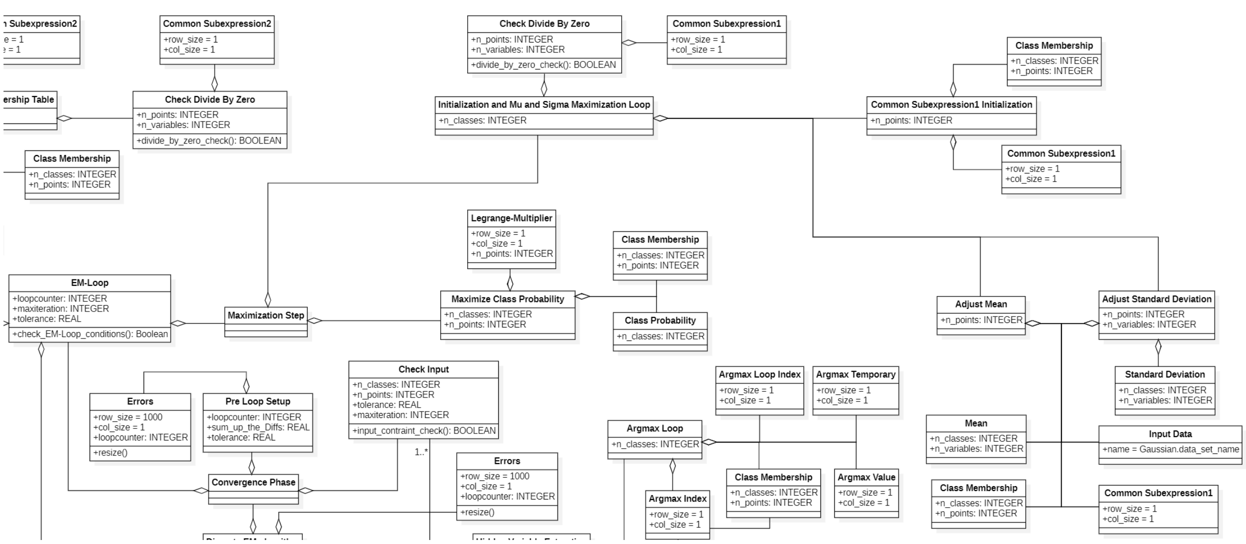

Figure 5.12. Output CD - Maximization Step.

Lastly, the Expectation Step class is focused upon below in Figure 5.13. In the code, the Expectation Step is made up of an initialization and update loop, a section in which the differences between the current and old values of values of interest are calculated, and if the log-likelihood pragma was specified, the log-likelihood will be calculated.

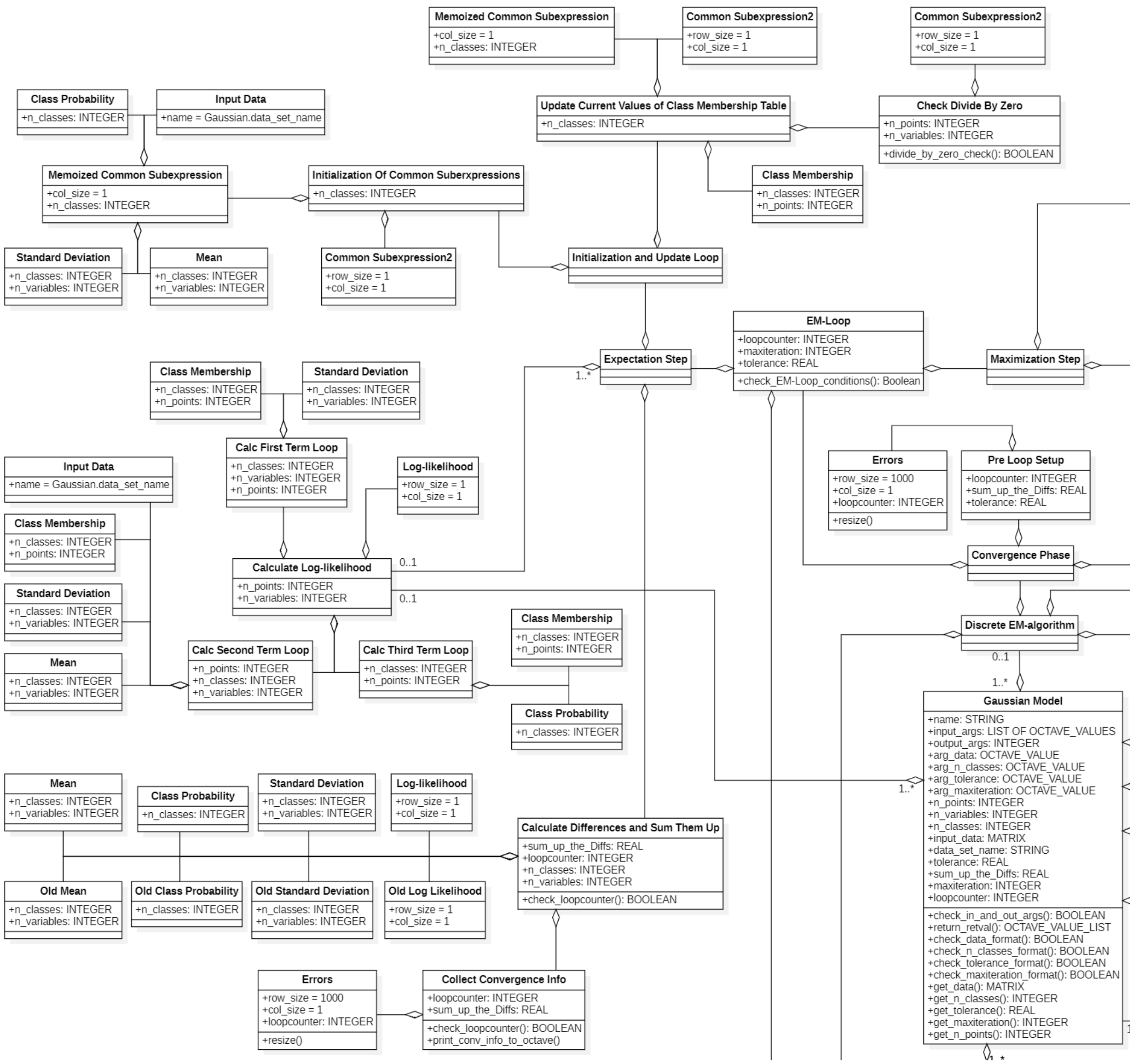

Figure 5.13. Output CD - Expectation Step.

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 bellow 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 vise versa.
2	IF Variance is used in Denominator, StandardDeviation is not used and vise versa.
3	IF Variance is used in Coefficient, StandardDeviation is not used and vise 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::NormDistOrTransfrom	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 combining 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.

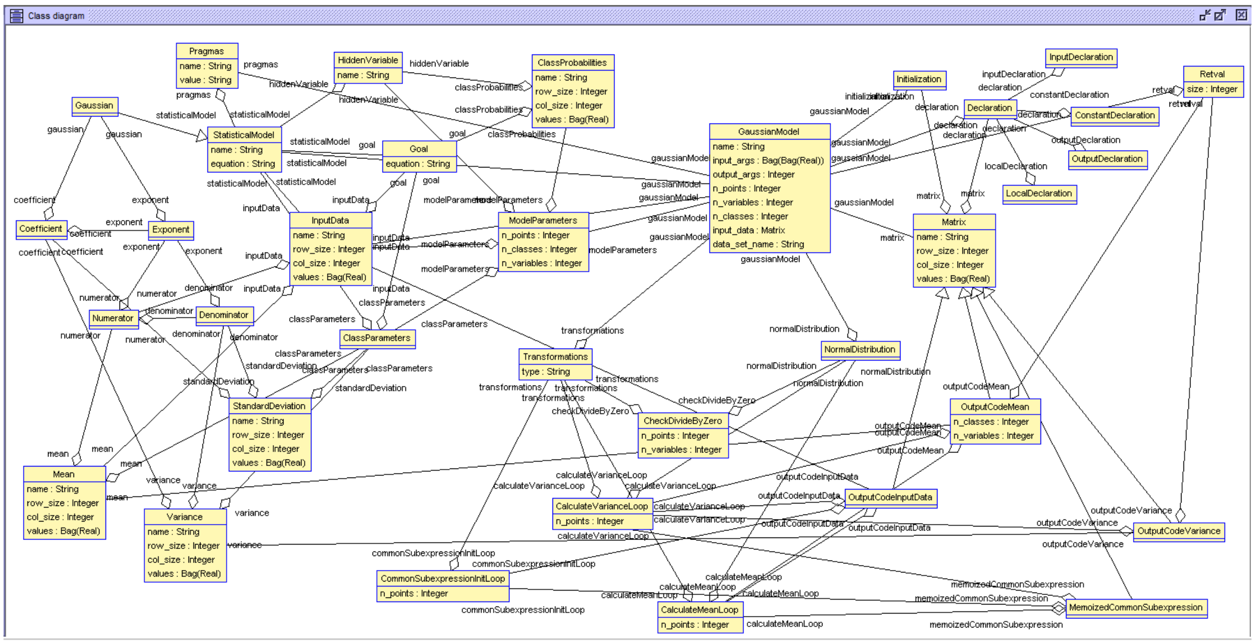

Figure 5.15. The USE class diagram.

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: TRIVIALLY_UNSATISFIABLE
INFO: Translation time (Kodkod to SAT): 162 ms; Solving time: 0 ms
INFO: Unsatisfiable proof:
< node: (all c: one Pragmas | one (c . Pragmas_name)), literal: 5, env: {}>
< node: !(Undefined in (univ . Pragmas_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: TRIVIALLY_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: TRIVIALLY_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: TRIVIALLY_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: TRIVIALLY_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: TRIVIALLY_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 is 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 and 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 then 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 came 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".

```
model normal as 'Normal Distributed Data'.

const nat n as 'NUMBER OF DATA POINTS'.

double mu as 'UNKNOWN MEAN'.
double sigma_sq as 'UNKNOWN VARIANCE'.
        where 0 < sigma_sq.

data double x(0..n-1) as 'GIVEN DATA POINTS'.

x(_) ~ gauss(mu, sqrt(sigma_sq)).

max pr( x | {mu, sigma_sq} ) for {mu, sigma_sq}.
```

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

```
model square_normal as 'SQUARE-NORMAL MODEL'.

const nat n as 'NUMBER OF DATA POINTS'.

double mu as 'UNKNOWN MEAN'.
double sigma_sq as 'UNKNOWN VARIANCE'.
        where 0 < sigma_sq.

data double x(0..n-1) as 'CURRENT DATA POINTS (KNOWN)'.
x(_)**2 ~ gauss(mu, sqrt(sigma_sq)).

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.

```
model log_normal as 'lOG-NORMAL MODEL'.

const nat n as 'NUMBER OF DATA POINTS'.

double mu as 'UNKNOWN MEAN'.
double sigma_sq as 'UNKNOWN VARIANCE'.
        where 0 < sigma_sq.

data double x(0..n-1) as 'CURRENT DATA POINTS (KNOWN)'.
log(x(_)) ~ gauss(mu, sqrt(sigma_sq)).

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.

```
model mog as 'MIXTURE OF GAUSSIANS'.

        % MODEL PARAMETERS
const nat n_points as 'NUMBER OF DATA POINTS'.
        where 0 < n_points.
const nat n_classes as 'NUMBER OF CLASSES'.
        where 0 < n_classes.
        where n_classes << n_points.

        % CLASS PROBABILITIES
double phi(0..n_classes-1) as 'CLASS PROBABILITY VECTOR.'.
        where 0 = sum(I := 0 .. n_classes-1, phi(I))-1.

        % CLASS PARAMETERS
double mu(0..n_classes-1) as 'COLUMN VECTOR OF MEANS'.
double sigma(0..n_classes-1) as 'COLUMN VECTOR OF STD DEVS'.
        where 0 < sigma(_).

        % HIDDEN VARIABLE
output nat c(0..n_points-1) as 'CLASS ASSIGNMENT VECTOR'.
c(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).

        % DATA
data double x(0..n_points-1).
x(I) ~ gauss(mu(c(I)), sigma(c(I))).

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.

```
model mult_cluster as 'SIMPLE MULTIVARIATE CLUSTERING MODEL'.

        % MODEL PARAMETERS
const nat n_variables as 'NUMBER OF VARIABLES'.
const nat n_points as 'NUMBER OF DATA POINTS'.
const nat n_classes as 'NUMBER OF CLASSES'.
        where 0 < n_classes.
        where n_classes << n_points.

        % CLASS PROBABILITIES
double phi(0..n_classes-1) as 'CLASS PROBABILITY VECTOR.'.
        where 0 = sum(I := 0 .. n_classes-1, phi(I))-1.

        % CLASS PARAMETERS
double mu(0..n_variables-1, 0..n_classes-1) as 'COLUMN VECTOR OF MEANS'.
double sigma(0..n_variables-1, 0..n_classes-1) as 'COLUMN VECTOR OF STD DEVS'.
        where 0 < sigma(_,_).

        % HIDDEN VARIABLE
output nat class_assignment(0..n_points-1) as 'HIDDEN VARIABLE'.
class_assignment(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).

        % DATA
data double sim_data(0..n_variables-1, 0..n_points-1).
sim_data(C,I) ~ gauss(mu(C,class_assignment(I)), sigma(C,class_assignment(I))).

        % GOAL
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.

```
model iris as
        'SIMPLE MULTIVARIATE CLUSTERING MODEL FOR CLASSICAL IRIS FLOWER EXAMPLE'.

const nat n_variables as 'NUMBER OF FEATURES'.
const nat n_points as 'NUMBER OF DATA POINTS'.
const nat n_classes as 'NUMBER OF CLASSES'.
        where 0 < n_classes.
        where n_classes << n_points.

double phi(0..n_classes-1) as 'CLASS PROBABILITY VECTOR.'.
        where sum(I := 0 .. n_classes-1, phi(I)) = 1.

double mu(0..n_variables-1, 0..n_classes-1) as 'MATRIX OF MEANS'.
double sigma(0..n_variables-1, 0..n_classes-1) as 'MATRIX OF STD DEVS'.
        where 0 < sigma(_,_).

output nat class_assignment(0..n_points-1) as 'CLASS OF EACH POINT'.
class_assignment(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).

data double iris_data(0..n_variables-1, 0..n_points-1).
iris_data(C,I) ~ gauss(mu(C, class_assignment(I)), sigma(C, class_assignment(I))).

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 MATLAB^{TM} 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.

```
//------------------------------------------------------------------------------
// Code file generated by AutoBayes V0.9.9
// AutoBayes(c) 2008-2011 United States Government as represented by
// the Administrator of NASA. AutoBayes is distributed under the NASA
// Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
// details.
// Problem:   Normal Distributed Data
// Source:    normal.ab
// Command:
//
//    PROLOG_VAR
//

//              -designdoc
//              normal.ab
// Generated: Fri Jun 26 12:08:31 2020
//------------------------------------------------------------------------------

#include "autobayes.h"



//------------------------------------------------------------------------------
// Octave Function: normal
//------------------------------------------------------------------------------

DEFUN_DLD(normal,input_args,output_args,
          "usage: [double mu,double sigma_sq] = normal(vector x)\n\n"
         )
{
  octave_value_list retval;
  if (input_args.length () != 1 || output_args != 2 ){
    octave_stdout << "usage: [double mu,double sigma_sq] = normal(vector x)\n\
   n";
    return retval;
  }

  //-- Input declarations ----------------------------------------------------

    // GIVEN DATA POINTS
  octave_value arg_x = input_args(0);
  if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
    gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
    return retval;
  }
  ColumnVector x = (ColumnVector)(arg_x.vector_value());

  //-- Constant declarations -------------------------------------------------

    // NUMBER OF DATA POINTS
  int n = arg_x.rows();

  //-- Output declarations ---------------------------------------------------
```

```

  // UNKNOWN MEAN
double mu;
  // UNKNOWN VARIANCE
double sigma_sq;

//-- Local declarations ----------------------------------------------------

// Summation accumulator
//   sum([pv1 := 0 .. -1 + n], x(pv1))
double pv3;

int pv1;

// Summation accumulator
//   sum([pv2 := 0 .. -1 + n], (-1 * mu + x(pv2)) ** 2)
double pv5;

int pv2;


// The conditional probability pr(x | {mu,sigma_sq}) is under the
// dependencies given in the model equivalent to
//
//   prod([pv0 := 0 .. -1 + n], pr(x(pv0) | {mu,sigma_sq}))
//
// The probability occuring here is atomic and can thus be replaced by the
// respective probability density function given in the model. This yields
// the log-likelihood function
//
//   log(prod([pv0 := 0 .. -1 + n],
//            exp(-1 / 2 * (x(pv0) - mu) ** 2 / (sigma_sq ** (1 / 2)) ** 2)
  *
//             (1 / (sqrt(2 * pi) * sigma_sq ** (1 / 2)))))
//
// which can be simplified to
//
//   -1 / 2 * n * log(2) + -1 / 2 * n * log(pi) + -1 / 2 * n * log(sigma_sq)
   +
//    -1 / 2 * sigma_sq ** -1 *
//     sum([pv0 := 0 .. -1 + n], (-1 * mu + x(pv0)) ** 2)
//
// This function is then optimized w.r.t. the goal variables mu and sigma_sq
  .
//
// The summands
//
//   -1 / 2 * n * log(2)
//   -1 / 2 * n * log(pi)
//
// are constant with respect to the goal variables mu and sigma_sq and can
// thus be ignored for maximization.
//
// The factor
```

```
//
//   1 / 2
//
// is non-negative and constant with respect to the goal variables mu and
// sigma_sq and can thus be ignored for maximization.
//
// The function
//
//   -1 * n * log(sigma_sq) +
//    -1 * sigma_sq ** -1 * sum([pv0 := 0 .. -1 + n], (-1 * mu + x(pv0)) ** 2)
//
// is then symbolically maximized w.r.t. the goal variables mu and sigma_sq.
// The partial differentials
//
//   df / d_mu ==
//    -2 * mu * n * sigma_sq ** -1 +
//     2 * sigma_sq ** -1 * sum([pv0 := 0 .. -1 + n], x(pv0))
//   df / d_sigma_sq ==
//    -1 * n * sigma_sq ** -1 +
//     sigma_sq ** -2 * sum([pv0 := 0 .. -1 + n], (-1 * mu + x(pv0)) ** 2)
//
// are set to zero; these equations yield the solutions
//
//   mu ==
//    cond(0 == n, fail(division_by_zero),
//         n ** -1 * sum([pv1 := 0 .. -1 + n], x(pv1)))
//   sigma_sq ==
//    cond(0 == n, fail(division_by_zero),
//         n ** -1 * sum([pv2 := 0 .. -1 + n], (-1 * mu + x(pv2)) ** 2))
//
if ( 0 == n )
  { ab_error( division_by_zero ); }
else
  {
    pv3 = 0.0;
    for( pv1 = 0;pv1 <= n - 1;pv1++ )
      pv3 += x(pv1);
    mu = pv3 / (double)(n);
  }
if ( 0 == n )
  { ab_error( division_by_zero ); }
else
  {
    pv5 = 0.0;
    for( pv2 = 0;pv2 <= n - 1;pv2++ )
      pv5 += (x(pv2) - mu) * (x(pv2) - mu);
    sigma_sq = pv5 / (double)(n);
  }

retval.resize(2);
retval(0) = mu;
retval(1) = sigma_sq;
```

```
  return retval;
}
//-- End of code
    ---------------------------------------------------------------
```

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

```

//----------------------------------------------------------------------------
// Code file generated by AutoBayes V0.9.9
// AutoBayes(c) 2008-2011 United States Government as represented by
// the Administrator of NASA. AutoBayes is distributed under the NASA
// Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
// details.
// Problem:   SQUARE-NORMAL MODEL
// Source:    square_normal.ab
// Command:
//
//   PROLOG_VAR
//

//             -designdoc
//             square_normal.ab
// Generated: Fri Jun 26 13:43:50 2020
//----------------------------------------------------------------------------

#include "autobayes.h"



//----------------------------------------------------------------------------
// Octave Function: square_normal
//----------------------------------------------------------------------------

DEFUN_DLD(square_normal,input_args,output_args,
          "usage: [double mu,double sigma_sq] = square_normal(vector x)\n\n"
         )
{
  octave_value_list retval;
  if (input_args.length () != 1 || output_args != 2 ){
    octave_stdout << "usage: [double mu,double sigma_sq] = square_normal(
   vector x)\n\n";
    return retval;
  }

  //-- Input declarations ----------------------------------------------------

    // CURRENT DATA POINTS (KNOWN)
  octave_value arg_x = input_args(0);
  if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
    gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
    return retval;
  }
```

```
ColumnVector x = (ColumnVector)(arg_x.vector_value());

//-- Constant declarations ---------------------------------------------

  // NUMBER OF DATA POINTS
int n = arg_x.rows();

//-- Output declarations -----------------------------------------------

  // UNKNOWN MEAN
double mu;
  // UNKNOWN VARIANCE
double sigma_sq;

//-- Local declarations ------------------------------------------------

// Memoized common subexpression
//   x(pv2) ** 2
ColumnVector pv4(n);

// Summation accumulator
//   sum([pv2 := 0 .. -1 + n], pv4(pv2))
double pv7;

// Loop variable
int pv2;

// Summation accumulator
//   sum([pv3 := 0 .. -1 + n], (-mu + pv4(pv3)) ** 2)
double pv8;

int pv3;


// The conditional probability pr(x | {mu,sigma_sq}) is under the
// dependencies given in the model equivalent to
//
//   prod([pv1 := 0 .. -1 + n], pr(x(pv1) | {mu,sigma_sq}))
//
// The probability occuring here is atomic and can thus be replaced by the
// respective probability density function given in the model. This yields
// the log-likelihood function
//
//   log(prod([pv1 := 0 .. -1 + n],
//            abs(deriv(x(pv1) ** 2, x(pv1))) *
//             exp(-1 / 2 * (x(pv1) ** 2 - mu) ** 2 /
//                  (sigma_sq ** (1 / 2)) ** 2) *
//             (1 / (sqrt(2 * pi) * sigma_sq ** (1 / 2)))))
//
// which can be simplified to
//
//   -1 / 2 * n * log(2) + -1 / 2 * n * log(pi) + -1 / 2 * n * log(sigma_sq)
  +
//    -1 / 2 * sigma_sq ** -1 *
```

```
//     sum([pv1 := 0 .. -1 + n], (-1 * mu + x(pv1) ** 2) ** 2) +
//    sum([pv1 := 0 .. -1 + n], log(abs(2 * x(pv1))))
//
// This function is then optimized w.r.t. the goal variables mu and sigma_sq
  .
//
// The summands
//
//   -1 / 2 * n * log(2)
//   -1 / 2 * n * log(pi)
//   sum([pv1 := 0 .. -1 + n], log(abs(2 * x(pv1))))
//
// are constant with respect to the goal variables mu and sigma_sq and can
// thus be ignored for maximization.
//
// The factor
//
//   1 / 2
//
// is non-negative and constant with respect to the goal variables mu and
// sigma_sq and can thus be ignored for maximization.
//
// The function
//
//   -1 * n * log(sigma_sq) +
//    -1 * sigma_sq ** -1 *
//     sum([pv1 := 0 .. -1 + n], (-1 * mu + x(pv1) ** 2) ** 2)
//
// is then symbolically maximized w.r.t. the goal variables mu and sigma_sq.
// The partial differentials
//
//   df / d_mu ==
//    -2 * mu * n * sigma_sq ** -1 +
//     2 * sigma_sq ** -1 * sum([pv1 := 0 .. -1 + n], x(pv1) ** 2)
//   df / d_sigma_sq ==
//    -1 * n * sigma_sq ** -1 +
//     sigma_sq ** -2 *
//      sum([pv1 := 0 .. -1 + n], (-1 * mu + x(pv1) ** 2) ** 2)
//
// are set to zero; these equations yield the solutions
//
//   mu ==
//    cond(0 == n, fail(division_by_zero),
//         n ** -1 * sum([pv2 := 0 .. -1 + n], x(pv2) ** 2))
//   sigma_sq ==
//    cond(0 == n, fail(division_by_zero),
//         n ** -1 * sum([pv3 := 0 .. -1 + n], (-1 * mu + x(pv3) ** 2) ** 2)
  )
//
//
// Initialization of common subexpression
for( pv2 = 0;pv2 <= n - 1;pv2++ )
  pv4(pv2) = x(pv2) * x(pv2);

```

```
  if ( 0 == n )
    { ab_error( division_by_zero ); }
  else
    {
      pv7 = 0.0;
      for( pv2 = 0;pv2 <= n - 1;pv2++ )
        pv7 += pv4(pv2);
      mu = pv7 * ((double)(1) / (double)(n));
    }
  if ( 0 == n )
    { ab_error( division_by_zero ); }
  else
    {
      pv8 = 0.0;
      for( pv3 = 0;pv3 <= n - 1;pv3++ )
        pv8 += (pv4(pv3) - mu) * (pv4(pv3) - mu);
      sigma_sq = pv8 * ((double)(1) / (double)(n));
    }

  retval.resize(2);
  retval(0) = mu;
  retval(1) = sigma_sq;

  return retval;
}
//-- End of code
    ---------------------------------------------------------------
```

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

```

//----------------------------------------------------------------------------
// Code file generated by AutoBayes V0.9.9
// AutoBayes(c) 2008-2011 United States Government as represented by
// the Administrator of NASA. AutoBayes is distributed under the NASA
// Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
// details.
// Problem:   lOG-NORMAL MODEL
// Source:    log_normal.ab
// Command:
//
//   PROLOG_VAR
//

//             -designdoc
//             log_normal.ab
// Generated: Fri Jun 26 13:38:58 2020
//----------------------------------------------------------------------------

#include "autobayes.h"



```

```
//----------------------------------------------------------------------
// Octave Function: log_normal
//----------------------------------------------------------------------

DEFUN_DLD(log_normal,input_args,output_args,
          "usage: [double mu,double sigma_sq] = log_normal(vector x)\n\n"
         )
{
  octave_value_list retval;
  if (input_args.length () != 1 || output_args != 2 ){
    octave_stdout << "usage: [double mu,double sigma_sq] = log_normal(vector x
    )\n\n";
    return retval;
  }

  //-- Input declarations ----------------------------------------------

    // CURRENT DATA POINTS (KNOWN)
  octave_value arg_x = input_args(0);
  if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
    gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
    return retval;
  }
  ColumnVector x = (ColumnVector)(arg_x.vector_value());

  //-- Constant declarations -------------------------------------------

    // NUMBER OF DATA POINTS
  int n = arg_x.rows();

  //-- Output declarations ---------------------------------------------

    // UNKNOWN MEAN
  double mu;
    // UNKNOWN VARIANCE
  double sigma_sq;

  //-- Local declarations ----------------------------------------------

  // Memoized common subexpression
  //   log(x(pv2))
  ColumnVector pv4(n);

  // Summation accumulator
  //   sum([pv2 := 0 .. -1 + n], pv4(pv2))
  double pv7;

  // Loop variable
  int pv2;

  // Summation accumulator
  //   sum([pv3 := 0 .. -1 + n], (-mu + pv4(pv3)) ** 2)
  double pv8;

```

```
int pv3;


// The conditional probability pr(x | {mu,sigma_sq}) is under the
// dependencies given in the model equivalent to
//
//   prod([pv1 := 0 .. -1 + n], pr(x(pv1) | {mu,sigma_sq}))
//
// The probability occuring here is atomic and can thus be replaced by the
// respective probability density function given in the model. This yields
// the log-likelihood function
//
//   log(prod([pv1 := 0 .. -1 + n],
//              abs(deriv(log(x(pv1)), x(pv1))) *
//               exp(-1 / 2 * (log(x(pv1)) - mu) ** 2 /
//                    (sigma_sq ** (1 / 2)) ** 2) *
//               (1 / (sqrt(2 * pi) * sigma_sq ** (1 / 2)))))
//
// which can be simplified to
//
//   -1 / 2 * n * log(2) + -1 / 2 * n * log(pi) + -1 / 2 * n * log(sigma_sq) +
//    -1 / 2 * sigma_sq ** -1 *
//     sum([pv1 := 0 .. -1 + n], (-1 * mu + log(x(pv1))) ** 2) +
//    sum([pv1 := 0 .. -1 + n], log(abs(x(pv1) ** -1)))
//
// This function is then optimized w.r.t. the goal variables mu and sigma_sq.
//
// The summands
//
//   -1 / 2 * n * log(2)
//   -1 / 2 * n * log(pi)
//   sum([pv1 := 0 .. -1 + n], log(abs(x(pv1) ** -1)))
//
// are constant with respect to the goal variables mu and sigma_sq and can
// thus be ignored for maximization.
//
// The factor
//
//   1 / 2
//
// is non-negative and constant with respect to the goal variables mu and
// sigma_sq and can thus be ignored for maximization.
//
// The function
//
//   -1 * n * log(sigma_sq) +
//    -1 * sigma_sq ** -1 *
//     sum([pv1 := 0 .. -1 + n], (-1 * mu + log(x(pv1))) ** 2)
//
// is then symbolically maximized w.r.t. the goal variables mu and sigma_sq.
// The partial differentials
//
```

```
  //   df / d_mu ==
  //    -2 * mu * n * sigma_sq ** -1 +
  //     2 * sigma_sq ** -1 * sum([pv1 := 0 .. -1 + n], log(x(pv1)))
  //   df / d_sigma_sq ==
  //    -1 * n * sigma_sq ** -1 +
  //     sigma_sq ** -2 *
  //      sum([pv1 := 0 .. -1 + n], (-1 * mu + log(x(pv1))) ** 2)
  //
  // are set to zero; these equations yield the solutions
  //
  //   mu ==
  //    cond(0 == n, fail(division_by_zero),
  //         n ** -1 * sum([pv2 := 0 .. -1 + n], log(x(pv2))))
  //   sigma_sq ==
  //    cond(0 == n, fail(division_by_zero),
  //         n ** -1 * sum([pv3 := 0 .. -1 + n], (-1 * mu + log(x(pv3))) ** 2)
   )
  //
  //
  // Initialization of common subexpression
  for( pv2 = 0;pv2 <= n - 1;pv2++ )
    pv4(pv2) = safelog(x(pv2));

  if ( 0 == n )
    { ab_error( division_by_zero ); }
  else
    {
      pv7 = 0.0;
      for( pv2 = 0;pv2 <= n - 1;pv2++ )
        pv7 += pv4(pv2);
      mu = pv7 * ((double)(1) / (double)(n));
    }
  if ( 0 == n )
    { ab_error( division_by_zero ); }
  else
    {
      pv8 = 0.0;
      for( pv3 = 0;pv3 <= n - 1;pv3++ )
        pv8 += (pv4(pv3) - mu) * (pv4(pv3) - mu);
      sigma_sq = pv8 * ((double)(1) / (double)(n));
    }

  retval.resize(2);
  retval(0) = mu;
  retval(1) = sigma_sq;

  return retval;
}
//-- End of code
    --------------------------------------------------------------
```

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

```

//------------------------------------------------------------------------------
// Code file generated by AutoBayes V0.9.9
// AutoBayes(c) 2008-2011 United States Government as represented by
// the Administrator of NASA. AutoBayes is distributed under the NASA
// Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
// details.
// Problem:   MIXTURE OF GAUSSIANS
// Source:    mog.ab
// Command:
//
//   PROLOG_VAR
//

//             -designdoc
//             mog.ab
// Generated: Fri Jun 26 14:21:50 2020
//------------------------------------------------------------------------------

#include "autobayes.h"



//------------------------------------------------------------------------------
// Octave Function: mog
//------------------------------------------------------------------------------

DEFUN_DLD(mog,input_args,output_args,
          "usage: [vector c,vector mu,vector phi,vector sigma] = mog(int
    n_classes,vector x,double tolerance,int maxiteration)\n\n"
         )
{
  octave_value_list retval;
  if (input_args.length () != 4 || output_args != 4 ){
    octave_stdout << "usage: [vector c,vector mu,vector phi,vector sigma] =
    mog(int n_classes,vector x,double tolerance,int maxiteration)\n\n";
    return retval;
  }

  //-- Input declarations ----------------------------------------------------

    // NUMBER OF CLASSES
  octave_value arg_n_classes = input_args(0);
  if (!arg_n_classes.is_real_scalar()){
    gripe_wrong_type_arg("n_classes", (const std::string &)"int expected");
    return retval;
  }
  int n_classes = (int)(arg_n_classes.int_value());

  octave_value arg_x = input_args(1);
  if (!arg_x.is_real_matrix() || arg_x.columns() != 1){
    gripe_wrong_type_arg("x", (const std::string &)"ColumnVector expected");
    return retval;
  }
```

```
  ColumnVector x = (ColumnVector)(arg_x.vector_value());

    // Iteration tolerance for convergence loop
  octave_value arg_tolerance = input_args(2);
  if (!arg_tolerance.is_real_scalar()){
    gripe_wrong_type_arg("tolerance", (const std::string &)"double expected");
    return retval;
  }
  double tolerance = (double)(arg_tolerance.double_value());

    // maximal number of iterations
  octave_value arg_maxiteration = input_args(3);
  if (!arg_maxiteration.is_real_scalar()){
    gripe_wrong_type_arg("maxiteration", (const std::string &)"int expected");
    return retval;
  }
  int maxiteration = (int)(arg_maxiteration.int_value());

  //-- Constant declarations -----------------------------------------------

    // NUMBER OF DATA POINTS
  int n_points = arg_x.rows();

  //-- Output declarations -------------------------------------------------

    // CLASS ASSIGNMENT VECTOR
  ColumnVector c(n_points);

    // COLUMN VECTOR OF MEANS
  ColumnVector mu(n_classes);

    // CLASS PROBABILITY VECTOR.
  ColumnVector phi(n_classes);

    // COLUMN VECTOR OF STD DEVS
  ColumnVector sigma(n_classes);


  //-- Local declarations --------------------------------------------------

  // Label: label0
  // class membership table used in Discrete EM-algorithm
  Matrix q(n_points, n_classes);

  // local centers used for center-based initialization
  Matrix center(n_classes, 1);

  // Random index of data point
  int pick;

  // Loop variable
  int pv53;

  // Loop variable
```

```
int pv51;

// Lagrange-multiplier
double l;

// Loop variable
int pv32;

// Loop variable
int pv11;

// Loop variable
int pv22;

// Common subexpression
//   sum([pv37 := 0 .. -1 + n_points], q(pv37, pv32))
double pv40;

// Memoized common subexpression
//   exp(-1 / 2 * (x(pv11) - mu(pv42)) ** 2 / sigma(pv42) ** 2) * phi(pv42)
 *
//    (1 / (sigma(pv42) * sqrt(2 * pi)))
ColumnVector pv44(n_classes);

// Common subexpression
//   sum([pv41 := 0 .. -1 + n_classes], pv44(pv41))
double pv46;

// Loop variable
int pv42;

// Loop variable
int pv61;

// Summation accumulator
//   sum([pv54 := 0 .. -1 + n_classes],
//       sqrt((center(pv54, 0) - x(pv11)) ** 2))
double pv66;

int pv54;

ColumnVector muold(n_classes);

ColumnVector phiold(n_classes);

ColumnVector sigmaold(n_classes);

int pv56;

int pv57;

int pv58;

// convergence loop counter
```

```
int loopcounter;

// sum up the Diffs
double pv67;

// Summation accumulator
//   sum([pv24 := 0 .. -1 + n_points], q(pv24, pv22))
double pv73;

int pv24;

// Summation accumulator
//   sum([pv37 := 0 .. -1 + n_points], q(pv37, pv32))
double pv74;

int pv37;

// Summation accumulator
//   sum([pv36 := 0 .. -1 + n_points], x(pv36) * q(pv36, pv32))
double pv75;

int pv36;

// Summation accumulator
//   sum([pv38 := 0 .. -1 + n_points],
//       (-mu(pv32) + x(pv38)) ** 2 * q(pv38, pv32))
double pv76;

int pv38;

// Summation accumulator
//   sum([pv41 := 0 .. -1 + n_classes], pv44(pv41))
double pv81;

int pv41;

int pv69;

// Summation accumulator
//   sum([pv69 := 0 .. -1 + n_classes],
//       abs(phi(pv69) - phiold(pv69)) / (abs(phi(pv69)) + abs(phiold(pv69))
 ))
double pv83;

int pv68;

// Summation accumulator
//   sum([pv68 := 0 .. -1 + n_classes],
//       abs(mu(pv68) - muold(pv68)) / (abs(mu(pv68)) + abs(muold(pv68))))
double pv82;

// Summation accumulator
//   sum([pv70 := 0 .. -1 + n_classes],
//       abs(sigma(pv70) - sigmaold(pv70)) /
```

```
//        (abs(sigma(pv70)) + abs(sigmaold(pv70))))
double pv84;

int pv70;

// Argmax index
int pv85;

// Argmax value
double pv86;

// Argmax temporary
double pv87;

// Argmax loop index
int pv64;

// Check constraints on inputs
ab_assert( 0 < n_classes );
ab_assert( 10 * n_classes < n_points );
ab_assert( 0 < n_points );

// Label: label1
// Label: label2
// Label: label4
// Discrete EM-algorithm
//
// The model describes a discrete latent (or hidden) variable problem with
// the latent variable c and the data variable x. The problem to optimize
// the conditional probability pr(x | {mu,phi,sigma}) w.r.t. the variables
// mu, phi, and sigma can thus be solved by an application of the (discrete)
// EM-algorithm.
// The algorithm maintains as central data structure a class membership
// table q (see "label0") such that q(pv11,pv47) is the probability that
// data point pv11 belongs to class pv47, i.e.,
//
//   q(pv11, pv47) == pr([c(pv11) == pv47])
//
// The algorithm consists of an initialization phase for q (see "label2"),
// followed by a convergence phase (see "label5"), followed by the
// extraction of the hidden variable c (see "label6").
//
// Initialization
//
// The initialization is center-based, i.e., for each class (i.e., value of
// the hidden variable c) a center value center is chosen first
// (see "label4"). Then, the values for the local distribution are
// calculated as distances between the data points and these center values
// (see "label7").
//
// Random initialization of the centers center with data points;
// note that a data point can be picked as center more than once.
for( pv51 = 0;pv51 <= n_classes - 1;pv51++ )
  {
```

```
      pick = uniform_int_rnd(n_points - 1);
      center(pv51, 0) = x(pick);
    }
  // Label: label7
  for( pv11 = 0;pv11 <= n_points - 1;pv11++ )
    for( pv53 = 0;pv53 <= n_classes - 1;pv53++ )
      {
        pv66 = 0.0;
        for( pv54 = 0;pv54 <= n_classes - 1;pv54++ )
          pv66 += sqrt((center(pv54, 0) - x(pv11)) *
                       (center(pv54, 0) - x(pv11)));
        q(pv11, pv53) = sqrt((center(pv53, 0) - x(pv11)) *
                             (center(pv53, 0) - x(pv11))) / pv66;
      }

  // Label: label5
  // EM-loop
  //
  // The EM-loop iterates two steps, expectation (or E-Step) (see "label8"),
  // and maximization (or M-Step) (see "label9"); however, due to the form of
  // the initialization used here, the are ordered the other way around. The
  // loop runs until convergence in the values of the variables mu, phi, and
  // sigma is achieved.
  //
  // Tolerance value must be positive
  ab_assert( tolerance > 0 );
  // max nr of iterations must be positive
  ab_assert( maxiteration > 0 );
  loopcounter = 0;
  // repeat at least once
  pv67 = tolerance;
  while( ((loopcounter < maxiteration) && (pv67 >= tolerance)) )
    {
      loopcounter = 1 + loopcounter;
      if ( loopcounter > 1 )
        {
          // assign current values to old values
          for( pv56 = 0;pv56 <= n_classes - 1;pv56++ )
            muold(pv56) = mu(pv56);
          // assign current values to old values
          for( pv57 = 0;pv57 <= n_classes - 1;pv57++ )
            phiold(pv57) = phi(pv57);
          // assign current values to old values
          for( pv58 = 0;pv58 <= n_classes - 1;pv58++ )
            sigmaold(pv58) = sigma(pv58);
        }
      else
        ;

      // Label: label9
      // Label: label3
      // M-Step
      //
      // Decomposition I
```

```
      //
      // The problem to optimize the conditional probability pr({c,x} |
      // {mu,phi,sigma}) w.r.t. the variables mu, phi, and sigma can under the
      // given dependencies by Bayes rule be decomposed into two independent
      // subproblems:
      //
      //   max pr(c | phi) for phi
      //   max pr(x | {c,mu,sigma}) for {mu,sigma}
      //
      //
      // The conditional probability pr(c | phi) is under the dependencies
      // given in the model equivalent to
      //
      //   prod([pv15 := 0 .. -1 + n_points], pr(c(pv15) | phi))
      //
      // The probability occuring here is atomic and can thus be replaced by
      // the respective probability density function given in the model.
      // Summing out the expected variable c(pv11) yields the log-likelihood
      // function
      //
      //   sum_domain([pv11 := 0 .. -1 + n_points],
      //              [pv16 := 0 .. -1 + n_classes], [c(pv11)], q(pv11, pv16),
      //              log(prod([pv15 := 0 .. -1 + n_points], phi(c(pv15)))))
      //
      // which can be simplified to
      //
      //   sum([pv16 := 0 .. -1 + n_classes],
      //       log(phi(pv16)) *
      //        sum([pv15 := 0 .. -1 + n_points], q(pv15, pv16)))
      //
      // This function is then optimized w.r.t. the goal variable phi.
      //
      // The expression
      //
      //   sum([pv16 := 0 .. -1 + n_classes],
      //       log(phi(pv16)) *
      //        sum([pv15 := 0 .. -1 + n_points], q(pv15, pv16)))
      //
      // is maximized w.r.t. the variable phi under the constraint
      //
      //   0 == -1 + sum([pv21 := 0 .. -1 + n_classes], phi(pv21))
      //
      // using the Lagrange-multiplier l.
      l = (double)(n_points);
      for( pv22 = 0;pv22 <= n_classes - 1;pv22++ )
        // The summand
        //
        //   l
        //
        // is constant with respect to the goal variable phi(pv22) and can
        // thus be ignored for maximization.
        //
        // The function
        //
```

```
        //   -1 * l * sum([pv21 := 0 .. -1 + n_classes], phi(pv21)) +
        //    sum([pv16 := 0 .. -1 + n_classes],
        //        log(phi(pv16)) *
        //         sum([pv15 := 0 .. -1 + n_points], q(pv15, pv16)))
        //
        // is then symbolically maximized w.r.t. the goal variable phi(pv22).
        // The differential
        //
        //   -1 * l +
        //    phi(pv22) ** -1 * sum([pv15 := 0 .. -1 + n_points], q(pv15, pv22
    ))
        //
        // is set to zero; this equation yields the solution
        //
        //   l ** -1 * sum([pv24 := 0 .. -1 + n_points], q(pv24, pv22))
        //
        {
          pv73 = 0.0;
          for( pv24 = 0;pv24 <= n_points - 1;pv24++ )
            pv73 += q(pv24, pv22);
          phi(pv22) = pv73 / l;
        }

      // The conditional probability pr(x | {c,mu,sigma}) is under the
      // dependencies given in the model equivalent to
      //
      //   prod([pv28 := 0 .. -1 + n_points], pr(x(pv28) | {c(pv28),mu,sigma})
    )
      //
      // The probability occuring here is atomic and can thus be replaced by
      // the respective probability density function given in the model.
      // Summing out the expected variable c(pv11) yields the log-likelihood
      // function
      //
      //   sum_domain([pv11 := 0 .. -1 + n_points],
      //              [pv29 := 0 .. -1 + n_classes], [c(pv11)], q(pv11, pv29),
      //              log(prod([pv28 := 0 .. -1 + n_points],
      //                         exp(-1 / 2 * (x(pv28) - mu(c(pv28))) ** 2 /
      //                              sigma(c(pv28)) ** 2) *
      //                          (1 / (sigma(c(pv28)) * sqrt(2 * pi))))))
      //
      // which can be simplified to
      //
      //   -1 *
      //    sum([pv29 := 0 .. -1 + n_classes],
      //        log(sigma(pv29)) *
      //         sum([pv28 := 0 .. -1 + n_points], q(pv28, pv29))) +
      //    -1 / 2 * n_points * log(2) + -1 / 2 * n_points * log(pi) +
      //    -1 / 2 *
      //     sum([pv29 := 0 .. -1 + n_classes],
      //         sigma(pv29) ** -2 *
      //          sum([pv28 := 0 .. -1 + n_points],
      //              (-1 * mu(pv29) + x(pv28)) ** 2 * q(pv28, pv29)))
      //
```

```
      // This function is then optimized w.r.t. the goal variables mu and
      // sigma.
      //
      // The summands
      //
      //   -1 / 2 * n_points * log(2)
      //   -1 / 2 * n_points * log(pi)
      //
      // are constant with respect to the goal variables mu and sigma and can
      // thus be ignored for maximization.
      //
      // Index decomposition
      //
      // The function
      //
      //   -1 *
      //    sum([pv29 := 0 .. -1 + n_classes],
      //        log(sigma(pv29)) *
      //         sum([pv28 := 0 .. -1 + n_points], q(pv28, pv29))) +
      //    -1 / 2 *
      //     sum([pv29 := 0 .. -1 + n_classes],
      //         sigma(pv29) ** -2 *
      //          sum([pv28 := 0 .. -1 + n_points],
      //              (-1 * mu(pv29) + x(pv28)) ** 2 * q(pv28, pv29)))
      //
      // can be optimized w.r.t. the variables mu(pv32) and sigma(pv32)
      // element by element (i.e., along the index variable pv32) because
      // there are no dependencies along that dimension.
      for( pv32 = 0;pv32 <= n_classes - 1;pv32++ )
        // The factor
        //
        //   n_classes
        //
        // is non-negative and constant with respect to the goal variables
        // mu(pv32) and sigma(pv32) and can thus be ignored for maximization.
        //
        // The function
        //
        //   -1 * log(sigma(pv32)) *
        //    sum([pv28 := 0 .. -1 + n_points], q(pv28, pv32)) +
        //    -1 / 2 * sigma(pv32) ** -2 *
        //     sum([pv28 := 0 .. -1 + n_points],
        //         (-1 * mu(pv32) + x(pv28)) ** 2 * q(pv28, pv32))
        //
        // is then symbolically maximized w.r.t. the goal variables mu(pv32)
        // and sigma(pv32). The partial differentials
        //
        //   df / d_mu(pvar(32)) ==
        //    -1 * mu(pv32) * sigma(pv32) ** -2 *
        //     sum([pv28 := 0 .. -1 + n_points], q(pv28, pv32)) +
        //     sigma(pv32) ** -2 *
        //      sum([pv28 := 0 .. -1 + n_points], x(pv28) * q(pv28, pv32))
        //   df / d_sigma(pvar(32)) ==
        //    -1 * sigma(pv32) ** -1 *
```

```
        //      sum([pv28 := 0 .. -1 + n_points], q(pv28, pv32)) +
        //      sigma(pv32) ** -3 *
        //       sum([pv28 := 0 .. -1 + n_points],
        //           (-1 * mu(pv32) + x(pv28)) ** 2 * q(pv28, pv32))
        //
        // are set to zero; these equations yield the solutions
        //
        //    mu(pv32) ==
        //     cond(0 == sum([pv34 := 0 .. -1 + n_points], q(pv34, pv32)),
        //          fail(division_by_zero),
        //          sum([pv35 := 0 .. -1 + n_points], q(pv35, pv32)) ** -1 *
        //           sum([pv36 := 0 .. -1 + n_points], x(pv36) * q(pv36, pv32))
)
        //    sigma(pv32) ==
        //     cond(0 == sum([pv37 := 0 .. -1 + n_points], q(pv37, pv32)),
        //          fail(division_by_zero),
        //          sum([pv38 := 0 .. -1 + n_points],
        //              (-1 * mu(pv32) + x(pv38)) ** 2 * q(pv38, pv32)) **
        //           (1 / 2) *
        //           sum([pv39 := 0 .. -1 + n_points], q(pv39, pv32)) **
        //            (-1 / 2))
        //
        {
          // Initialization of common subexpression
          pv74 = 0.0;
          for( pv37 = 0;pv37 <= n_points - 1;pv37++ )
            pv74 += q(pv37, pv32);
          pv40 = pv74;

          if ( 0 == pv40 )
            { ab_error( division_by_zero ); }
          else
            {
              pv75 = 0.0;
              for( pv36 = 0;pv36 <= n_points - 1;pv36++ )
                pv75 += x(pv36) * q(pv36, pv32);
              mu(pv32) = pv75 * ((double)(1) / pv40);
            }
          if ( 0 == pv40 )
            { ab_error( division_by_zero ); }
          else
            {
              pv76 = 0.0;
              for( pv38 = 0;pv38 <= n_points - 1;pv38++ )
                pv76 += (x(pv38) - mu(pv32)) * (x(pv38) - mu(pv32)) *
                         q(pv38, pv32);
              sigma(pv32) = sqrt(pv76) * ((double)(1) / sqrt(pv40));
            }
        }
      // Label: label8
      // E-Step
      // Update the current values of the class membership table q.
      for( pv11 = 0;pv11 <= n_points - 1;pv11++ )
        {
```

```
          // Initialization of common subexpression
          for( pv42 = 0;pv42 <= n_classes - 1;pv42++ )
            pv44(pv42) = exp(-0.5 * (x(pv11) - mu(pv42)) *
                               (x(pv11) - mu(pv42)) /
                               (sigma(pv42) * sigma(pv42))) * phi(pv42) *
                           ((double)(1) /
                            (sigma(pv42) * sqrt(M_PI * (double)(2))));

          pv81 = 0.0;
          for( pv41 = 0;pv41 <= n_classes - 1;pv41++ )
            pv81 += pv44(pv41);
          pv46 = pv81;
          for( pv61 = 0;pv61 <= n_classes - 1;pv61++ )
            // The denominator pv46 can become zero due to round-off errors.
            // In that case, each class is considered to be equally likely.
            if ( pv46 == 0.0 )
              q(pv11, pv61) = (double)(1) / (double)(n_classes);
            else
              q(pv11, pv61) = pv44(pv61) / pv46;
        }
      if ( loopcounter > 1 )
        {
          pv82 = 0.0;
          for( pv68 = 0;pv68 <= n_classes - 1;pv68++ )
            pv82 += abs(mu(pv68) - muold(pv68)) /
                     (abs(mu(pv68)) + abs(muold(pv68)));

          pv83 = 0.0;
          for( pv69 = 0;pv69 <= n_classes - 1;pv69++ )
            pv83 += abs(phi(pv69) - phiold(pv69)) /
                     (abs(phi(pv69)) + abs(phiold(pv69)));

          pv84 = 0.0;
          for( pv70 = 0;pv70 <= n_classes - 1;pv70++ )
            pv84 += abs(sigma(pv70) - sigmaold(pv70)) /
                     (abs(sigma(pv70)) + abs(sigmaold(pv70)));
          pv67 = pv82 + pv83 + pv84;
        }
      else
        ;
    }
  // Label: label6
  // Extract the most likely values of the hidden variable c(pv11) from the
  // class membership table q.
  for( pv11 = 0;pv11 <= n_points - 1;pv11++ )
    {
      // Determine the position of the maximum with in the range
      //   0
      // ...
      //   -1 + n_classes
      // by iterating over this range and calculating the value at each point
      // (argmax).
      //
      // Argmax loop
```

```
      for( pv64 = 0;pv64 <= n_classes - 1;pv64++ )
        {
          pv87 = q(pv11, pv64);
          if ( ((pv64 == 0) || (pv87 > pv86)) )
            // Save new maximum
            {
              pv86 = pv87;
              pv85 = pv64;
            }
          else
            ;
        }
      c(pv11) = pv85;
    }

  retval.resize(4);
  retval(0) = c;
  retval(1) = mu;
  retval(2) = phi;
  retval(3) = sigma;

  return retval;
}
//-- End of code
    ----------------------------------------------------------------
```

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

```

//-----------------------------------------------------------------------------
// Code file generated by AutoBayes V0.9.9
// AutoBayes(c) 2008-2011 United States Government as represented by
// the Administrator of NASA. AutoBayes is distributed under the NASA
// Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
// details.
// Problem:   SIMPLE MULTIVARIATE CLUSTERING MODEL
// Source:    mult_cluster.ab
// Command:
//
//    PROLOG_VAR
//

//             -designdoc
//             -instrument
//             -pragma em_log_likelihood_convergence=true
//             mult_cluster.ab
// Generated: Fri Jun 26 15:51:13 2020
//-----------------------------------------------------------------------------

#include "autobayes.h"



```

```
//---------------------------------------------------------------------------
// Octave Function: mult_cluster
//---------------------------------------------------------------------------

DEFUN_DLD(mult_cluster,input_args,output_args,
          "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"
         )
{
  octave_value_list retval;
  if (input_args.length () != 4 || output_args != 6 ){
    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";
    return retval;
  }

  //-- Input declarations ---------------------------------------------------

    // NUMBER OF CLASSES
  octave_value arg_n_classes = input_args(0);
  if (!arg_n_classes.is_real_scalar()){
    gripe_wrong_type_arg("n_classes", (const std::string &)"int expected");
    return retval;
  }
  int n_classes = (int)(arg_n_classes.int_value());

  octave_value arg_sim_data = input_args(1);
  if (!arg_sim_data.is_real_matrix()){
    gripe_wrong_type_arg("sim_data", (const std::string &)"Matrix expected");
    return retval;
  }
  Matrix sim_data = (Matrix)(arg_sim_data.matrix_value());

    // Iteration tolerance for convergence loop
  octave_value arg_tolerance = input_args(2);
  if (!arg_tolerance.is_real_scalar()){
    gripe_wrong_type_arg("tolerance", (const std::string &)"double expected");
    return retval;
  }
  double tolerance = (double)(arg_tolerance.double_value());

    // maximal number of iterations
  octave_value arg_maxiteration = input_args(3);
  if (!arg_maxiteration.is_real_scalar()){
    gripe_wrong_type_arg("maxiteration", (const std::string &)"int expected");
    return retval;
  }
  int maxiteration = (int)(arg_maxiteration.int_value());

  //-- Constant declarations ------------------------------------------------

    // NUMBER OF DATA POINTS
```

```
int n_points = arg_sim_data.columns();

  // NUMBER OF VARIABLES
int n_variables = arg_sim_data.rows();

//-- Output declarations ----------------------------------------------------

  // HIDDEN VARIABLE
ColumnVector class_assignment(n_points);

  // COLUMN VECTOR OF MEANS
Matrix mu(n_variables, n_classes);

  // CLASS PROBABILITY VECTOR.
ColumnVector phi(n_classes);

  // COLUMN VECTOR OF STD DEVS
Matrix sigma(n_variables, n_classes);

  // log likelihood
double loglikelihood;
  // intrumentation: assembly of convergence data
ColumnVector errors(1000);


//-- Local declarations -----------------------------------------------------

// Label: label0
// class membership table used in Discrete EM-algorithm
Matrix q(n_points, n_classes);

// local centers used for center-based initialization
Matrix center(n_classes, n_variables);

// Random index of data point
int pick;

// Loop variable
int pv74;

// Loop variable
int pv71;

// Loop variable
int pv72;

// Lagrange-multiplier
double l;

// Loop variable
int pv43;

// Loop variable
int pv44;
```

```
// Loop variable
int pv14;

// Loop variable
int pv25;

// Common subexpression
//   sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))
double pv53;

// Memoized common subexpression
//   phi(pv57) *
//    prod([pv54 := 0 .. -1 + n_variables],
//         exp(-1 / 2 * (sim_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
//              sigma(pv54, pv57) ** 2) *
//          (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
ColumnVector pv59(n_classes);

// Common subexpression
//   sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
double pv61;

// Loop variable
int pv57;

// Loop variable
int pv81;

int pv76;

// Summation accumulator
//   sum([pv76 := 0 .. -1 + n_classes], sqrt(pv87))
double pv88;

// Summation accumulator
//   sum([pv75 := 0 .. -1 + n_variables],
//       (center(pv74, pv75) - sim_data(pv75, pv14)) ** 2)
double pv86;

int pv75;

// Summation accumulator
//   sum([pv77 := 0 .. -1 + n_variables],
//       (center(pv76, pv77) - sim_data(pv77, pv14)) ** 2)
double pv87;

int pv77;

double pv89;

// convergence loop counter
int loopcounter;
```

```
// sum up the Diffs
double pv90;

// Summation accumulator
//   sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
double pv93;

int pv27;

// Summation accumulator
//   sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))
double pv94;

int pv50;

// Summation accumulator
//   sum([pv49 := 0 .. -1 + n_points], q(pv49, pv44) * sim_data(pv43, pv49))
double pv95;

int pv49;

// Summation accumulator
//   sum([pv51 := 0 .. -1 + n_points],
//       (-mu(pv43, pv44) + sim_data(pv43, pv51)) ** 2 * q(pv51, pv44))
double pv96;

int pv51;

// Product accumulator
//   sum([pv54 := 0 .. -1 + n_variables],
//       exp(-1 / 2 * (sim_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
//             sigma(pv54, pv57) ** 2) *
//        (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
double pv102;

int pv54;

// Summation accumulator
//   sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
double pv103;

int pv56;

// Summation accumulator
//   sum([pv19 := 0 .. -1 + n_classes], log(phi(pv19)) * pv109)
double pv110;

// Summation accumulator
//   sum([pv35 := 0 .. -1 + n_classes], pv104 * pv105)
double pv106;

// Summation accumulator
//   sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
//       pv107 * q(pv34, pv35))
```

```
double pv108;

// Summation accumulator
//   sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35)))
double pv104;

// Summation accumulator
//   sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))
double pv105;

// Summation accumulator
//   sum([pv33 := 0 .. -1 + n_variables],
//       (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
//        sigma(pv33, pv35) ** -2)
double pv107;

// Summation accumulator
//   sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19))
double pv109;

// Argmax index
int pv111;

// Argmax value
double pv112;

// Argmax temporary
double pv113;

// Argmax loop index
int pv84;

int pv19;

// Summation accumulator
//   sum([pv19 := 0 .. -1 + n_classes], log(phi(pv19)) * pv119)
double pv120;

// Summation accumulator
//   sum([pv35 := 0 .. -1 + n_classes], pv114 * pv115)
double pv116;

// Summation accumulator
//   sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
//       pv117 * q(pv34, pv35))
double pv118;

int pv35;

// Summation accumulator
//   sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35)))
double pv114;

// Summation accumulator
```

```
//   sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))
double pv115;

int pv34;

// Summation accumulator
//   sum([pv33 := 0 .. -1 + n_variables],
//       (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
//        sigma(pv33, pv35) ** -2)
double pv117;

int pv33;

// Summation accumulator
//   sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19))
double pv119;

int pv18;

// Check constraints on inputs
ab_assert( 0 < n_classes );
ab_assert( 10 * n_classes < n_points );

// Label: label0
// Label: label0
// Label: label0
// Discrete EM-algorithm
//
// The model describes a discrete latent (or hidden) variable problem with
// the latent variable class_assignment and the data variable sim_data. The
// problem to optimize the conditional probability pr(sim_data |
// {mu,phi,sigma}) w.r.t. the variables mu, phi, and sigma can thus be
// solved by an application of the (discrete) EM-algorithm.
// The algorithm maintains as central data structure a class membership
// table q (see "label0") such that q(pv14,pv67) is the probability that
// data point pv14 belongs to class pv67, i.e.,
//
//   q(pv14, pv67) == pr([class_assignment(pv14) == pv67])
//
// The algorithm consists of an initialization phase for q (see "label0"),
// followed by a convergence phase (see "label0"), followed by the
// extraction of the hidden variable class_assignment (see "label0").
//
// Initialization
//
// The initialization is center-based, i.e., for each class (i.e., value of
// the hidden variable class_assignment) a center value center is chosen
// first (see "label0"). Then, the values for the local distribution are
// calculated as distances between the data points and these center values
// (see "label0").
//
// Random initialization of the centers center with data points;
// note that a data point can be picked as center more than once.
for( pv71 = 0;pv71 <= n_classes - 1;pv71++ )
```

```
    {
      pick = uniform_int_rnd(n_points - 1);
      for( pv72 = 0;pv72 <= n_variables - 1;pv72++ )
        center(pv71, pv72) = sim_data(pv72, pick);
    }
  // Label: label0
  for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
    for( pv74 = 0;pv74 <= n_classes - 1;pv74++ )
      {
        pv86 = 0.0;
        for( pv75 = 0;pv75 <= n_variables - 1;pv75++ )
          pv86 += (center(pv74, pv75) - sim_data(pv75, pv14)) *
                   (center(pv74, pv75) - sim_data(pv75, pv14));

        pv88 = 0.0;
        for( pv76 = 0;pv76 <= n_classes - 1;pv76++ )
          {
            pv87 = 0.0;
            for( pv77 = 0;pv77 <= n_variables - 1;pv77++ )
              pv87 += (center(pv76, pv77) - sim_data(pv77, pv14)) *
                       (center(pv76, pv77) - sim_data(pv77, pv14));
            pv88 += sqrt(pv87);
          }
        q(pv14, pv74) = sqrt(pv86) / pv88;
      }
  // resize vector to maximal size
  errors.resize(1000);
  // initialize convergence output
  for( loopcounter = 0;loopcounter <= 999;loopcounter++ )
    errors(loopcounter) = 0;
  // Tolerance value must be positive
  ab_assert( tolerance > 0 );
  // max nr of iterations must be positive
  ab_assert( maxiteration > 0 );
  loopcounter = 0;
  // repeat at least once
  pv90 = tolerance;
  while( ((loopcounter < maxiteration) && (pv90 >= tolerance)) )
    {
      loopcounter = 1 + loopcounter;
      if ( loopcounter > 1 )
        // assign current values to old values
        pv89 = loglikelihood;
      else
        ;

      // Label: label0
      // Label: label0
      // M-Step
      //
      // Decomposition I
      //
      // The problem to optimize the conditional probability
      // pr({class_assignment,sim_data} | {mu,phi,sigma}) w.r.t. the variables
```

```
      // mu, phi, and sigma can under the given dependencies by Bayes rule be
      // decomposed into two independent subproblems:
      //
      //   max pr(class_assignment | phi) for phi
      //   max pr(sim_data | {class_assignment,mu,sigma}) for {mu,sigma}
      //
      //
      // The conditional probability pr(class_assignment | phi) is under the
      // dependencies given in the model equivalent to
      //
      //   prod([pv18 := 0 .. -1 + n_points], pr(class_assignment(pv18) | phi))
      //
      // The probability occuring here is atomic and can thus be replaced by
      // the respective probability density function given in the model.
      // Summing out the expected variable class_assignment(pv14) yields the
      // log-likelihood function
      //
      //   sum_domain([pv14 := 0 .. -1 + n_points],
      //              [pv19 := 0 .. -1 + n_classes], [class_assignment(pv14)],
      //              q(pv14, pv19),
      //              log(prod([pv18 := 0 .. -1 + n_points],
      //                       phi(class_assignment(pv18)))))
      //
      // which can be simplified to
      //
      //   sum([pv19 := 0 .. -1 + n_classes],
      //       log(phi(pv19)) *
      //        sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
      //
      // This function is then optimized w.r.t. the goal variable phi.
      //
      // The expression
      //
      //   sum([pv19 := 0 .. -1 + n_classes],
      //       log(phi(pv19)) *
      //        sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
      //
      // is maximized w.r.t. the variable phi under the constraint
      //
      //   0 == -1 + sum([pv24 := 0 .. -1 + n_classes], phi(pv24))
      //
      // using the Lagrange-multiplier l.
      l = (double)(n_points);
      for( pv25 = 0;pv25 <= n_classes - 1;pv25++ )
        // The summand
        //
        //   l
        //
        // is constant with respect to the goal variable phi(pv25) and can
        // thus be ignored for maximization.
        //
        // The function
        //
```

```
      //   -1 * l * sum([pv24 := 0 .. -1 + n_classes], phi(pv24)) +
      //    sum([pv19 := 0 .. -1 + n_classes],
      //        log(phi(pv19)) *
      //         sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
      //
      // is then symbolically maximized w.r.t. the goal variable phi(pv25).
      // The differential
      //
      //   -1 * l +
      //    phi(pv25) ** -1 * sum([pv18 := 0 .. -1 + n_points], q(pv18, pv25))
      //
      // is set to zero; this equation yields the solution
      //
      //   l ** -1 * sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
      //
      {
        pv93 = 0.0;
        for( pv27 = 0;pv27 <= n_points - 1;pv27++ )
          pv93 += q(pv27, pv25);
        phi(pv25) = pv93 / l;
      }

    // The conditional probability pr(sim_data |
    // {class_assignment,mu,sigma}) is under the dependencies given in the
    // model equivalent to
    //
    //   prod([pv33 := 0 .. -1 + n_variables, pv34 := 0 .. -1 + n_points],
    //        pr(sim_data(pv33,pv34) | {class_assignment(pv34),mu(pv33,*), sigma(pv33,*)}))
    //
    // The probability occuring here is atomic and can thus be replaced by
    // the respective probability density function given in the model.
    // Summing out the expected variable class_assignment(pv14) yields the
    // log-likelihood function
    //
    //   sum_domain([pv14 := 0 .. -1 + n_points],
    //              [pv35 := 0 .. -1 + n_classes], [class_assignment(pv14)],
    //              q(pv14, pv35),
    //              log(prod([pv33 := 0 .. -1 + n_variables,
    //                        pv34 := 0 .. -1 + n_points],
    //                       exp(-1 / 2 *
    //                            (sim_data(pv33, pv34) -
    //                             mu(pv33, class_assignment(pv34))) ** 2 /
    //                            sigma(pv33, class_assignment(pv34)) ** 2) *
    //                        (1 /
    //                         (sqrt(2 * pi) *
    //                          sigma(pv33, class_assignment(pv34))))))
    //
    // which can be simplified to
    //
    //   -1 *
    //    sum([pv35 := 0 .. -1 + n_classes],
```

```
    //         sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35))) *
    //          sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
    //    -1 / 2 * n_points * n_variables * log(2) +
    //    -1 / 2 * n_points * n_variables * log(pi) +
    //    -1 / 2 *
    //     sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
    //         q(pv34, pv35) *
    //          sum([pv33 := 0 .. -1 + n_variables],
    //              (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
    //               sigma(pv33, pv35) ** -2))
    //
    // This function is then optimized w.r.t. the goal variables mu and
    // sigma.
    //
    // The summands
    //
    //   -1 / 2 * n_points * n_variables * log(2)
    //   -1 / 2 * n_points * n_variables * log(pi)
    //
    // are constant with respect to the goal variables mu and sigma and can
    // thus be ignored for maximization.
    //
    // Index decomposition
    //
    // The function
    //
    //   -1 *
    //    sum([pv35 := 0 .. -1 + n_classes],
    //        sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35))) *
    //         sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
    //    -1 / 2 *
    //     sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
    //         q(pv34, pv35) *
    //          sum([pv33 := 0 .. -1 + n_variables],
    //              (-1 * mu(pv33, pv35) + sim_data(pv33, pv34)) ** 2 *
    //               sigma(pv33, pv35) ** -2))
    //
    // can be optimized w.r.t. the variables mu(pv43,pv44) and
    // sigma(pv43,pv44) element by element (i.e., along the index variables
    // pv43 and pv44) because there are no dependencies along thats
    // dimensions.
    for( pv43 = 0;pv43 <= n_variables - 1;pv43++ )
      for( pv44 = 0;pv44 <= n_classes - 1;pv44++ )
        // The factor
        //
        //   n_classes
        //
        // is non-negative and constant with respect to the goal variables
        // mu(pv43,pv44) and sigma(pv43,pv44) and can thus be ignored for
        // maximization.
        //
        // The function
        //
        //   -1 * n_variables * log(sigma(pv43, pv44)) *
```

```
          //    sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
          //    -1 / 2 * n_variables * sigma(pv43, pv44) ** -2 *
          //     sum([pv34 := 0 .. -1 + n_points],
          //         (-1 * mu(pv43, pv44) + sim_data(pv43, pv34)) ** 2 *
          //          q(pv34, pv44))
          //
          // is then symbolically maximized w.r.t. the goal variables
          // mu(pv43,pv44) and sigma(pv43,pv44). The partial differentials
          //
          //   df / d_mu(pvar(43),pvar(44)) ==
          //    -1 * n_variables * sigma(pv43, pv44) ** -2 * mu(pv43, pv44) *
          //     sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
          //     n_variables * sigma(pv43, pv44) ** -2 *
          //      sum([pv34 := 0 .. -1 + n_points],
          //          q(pv34, pv44) * sim_data(pv43, pv34))
          //   df / d_sigma(pvar(43),pvar(44)) ==
          //    -1 * n_variables * sigma(pv43, pv44) ** -1 *
          //     sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
          //     n_variables * sigma(pv43, pv44) ** -3 *
          //      sum([pv34 := 0 .. -1 + n_points],
          //          (-1 * mu(pv43, pv44) + sim_data(pv43, pv34)) ** 2 *
          //           q(pv34, pv44))
          //
          // are set to zero; these equations yield the solutions
          //
          //   mu(pv43, pv44) ==
          //    cond(0 == n_variables or
          //          0 == sum([pv47 := 0 .. -1 + n_points], q(pv47, pv44)),
          //         fail(division_by_zero),
          //         sum([pv48 := 0 .. -1 + n_points], q(pv48, pv44)) ** -1 *
          //          sum([pv49 := 0 .. -1 + n_points],
          //              q(pv49, pv44) * sim_data(pv43, pv49)))
          //   sigma(pv43, pv44) ==
          //    cond(0 == n_variables or
          //          0 == sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44)),
          //         fail(division_by_zero),
          //         abs(n_variables) * n_variables ** -1 *
          //          sum([pv51 := 0 .. -1 + n_points],
          //              (-1 * mu(pv43, pv44) + sim_data(pv43, pv51)) ** 2 *
          //               q(pv51, pv44)) ** (1 / 2) *
          //          sum([pv52 := 0 .. -1 + n_points], q(pv52, pv44)) **
          //           (-1 / 2))
          //
          {
            // Initialization of common subexpression
            pv94 = 0.0;
            for( pv50 = 0;pv50 <= n_points - 1;pv50++ )
              pv94 += q(pv50, pv44);
            pv53 = pv94;

            if ( ((0 == n_variables) || (0 == pv53)) )
              { ab_error( division_by_zero ); }
            else
              {
```

```
                pv95 = 0.0;
                for( pv49 = 0;pv49 <= n_points - 1;pv49++ )
                  pv95 += q(pv49, pv44) * sim_data(pv43, pv49);
                mu(pv43, pv44) = pv95 * ((double)(1) / pv53);
              }
            if ( ((0 == n_variables) || (0 == pv53)) )
              { ab_error( division_by_zero ); }
            else
              {
                pv96 = 0.0;
                for( pv51 = 0;pv51 <= n_points - 1;pv51++ )
                  pv96 += (sim_data(pv43, pv51) - mu(pv43, pv44)) *
                           (sim_data(pv43, pv51) - mu(pv43, pv44)) *
                           q(pv51, pv44);
                sigma(pv43, pv44) = abs(n_variables) * sqrt(pv96) *
                                      ((double)(1) / (double)(n_variables)) *
                                      ((double)(1) / sqrt(pv53));
              }
          }
      // Label: label0
      // E-Step
      // Update the current values of the class membership table q.
      for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
        {
          // Initialization of common subexpression
          for( pv57 = 0;pv57 <= n_classes - 1;pv57++ )
            {
              pv102 = 1.0;
              for( pv54 = 0;pv54 <= n_variables - 1;pv54++ )
                pv102 *= exp(-0.5 * (sim_data(pv54, pv14) - mu(pv54, pv57)) *
                                (sim_data(pv54, pv14) - mu(pv54, pv57)) /
                                (sigma(pv54, pv57) * sigma(pv54, pv57))) *
                            ((double)(1) /
                             (sqrt(M_PI * (double)(2)) * sigma(pv54, pv57)));
              pv59(pv57) = phi(pv57) * pv102;
            }

          pv103 = 0.0;
          for( pv56 = 0;pv56 <= n_classes - 1;pv56++ )
            pv103 += pv59(pv56);
          pv61 = pv103;
          for( pv81 = 0;pv81 <= n_classes - 1;pv81++ )
            // The denominator pv61 can become zero due to round-off errors.
            // In that case, each class is considered to be equally likely.
            if ( pv61 == 0.0 )
              q(pv14, pv81) = (double)(1) / (double)(n_classes);
            else
              q(pv14, pv81) = pv59(pv81) / pv61;
        }

      // Calculate the Log-likelihood as a convergence measure
      pv106 = 0.0;
      for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
        {
```

```
          pv104 = 0.0;
          for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
            pv104 += safelog(sigma(pv33, pv35));

          pv105 = 0.0;
          for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
            pv105 += q(pv34, pv35);
          pv106 += pv104 * pv105;
        }

      pv108 = 0.0;
      for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
        for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
          {
            pv107 = 0.0;
            for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
              pv107 += (sim_data(pv33, pv34) - mu(pv33, pv35)) *
                        (sim_data(pv33, pv34) - mu(pv33, pv35)) /
                        (sigma(pv33, pv35) * sigma(pv33, pv35));
            pv108 += pv107 * q(pv34, pv35);
          }

      pv110 = 0.0;
      for( pv19 = 0;pv19 <= n_classes - 1;pv19++ )
        {
          pv109 = 0.0;
          for( pv18 = 0;pv18 <= n_points - 1;pv18++ )
            pv109 += q(pv18, pv19);
          pv110 += pv109 * safelog(phi(pv19));
        }
      loglikelihood = -0.5 *
                        (n_points * n_variables *
                          (safelog(2) + safelog(M_PI)) + pv108) + pv110 -
                        pv106;
      if ( loopcounter > 1 )
        {
          pv90 = abs(loglikelihood - pv89) / (abs(loglikelihood) + abs(pv89));

          if ( loopcounter <= 1000 )
            // collect convergence info
            errors(loopcounter - 2) = pv90;
          else
            ;
          octave_stdout << " pvar(90) = " << pv90 << endl;
        }
      else
        ;
    }
  errors.resize(loopcounter);
  // Label: label0
  // Extract the most likely values of the hidden variable
  // class_assignment(pv14) from the class membership table q.
  for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
    {
```

```
      // Determine the position of the maximum with in the range
      //   0
      // ...
      //   -1 + n_classes
      // by iterating over this range and calculating the value at each point
      // (argmax).
      //
      // Argmax loop
      for( pv84 = 0;pv84 <= n_classes - 1;pv84++ )
        {
          pv113 = q(pv14, pv84);
          if ( ((pv84 == 0) || (pv113 > pv112)) )
            // Save new maximum
            {
              pv112 = pv113;
              pv111 = pv84;
            }
          else
            ;
        }
      class_assignment(pv14) = pv111;
    }

  // Calculation of Log-likelihood
  pv116 = 0.0;
  for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
    {
      pv114 = 0.0;
      for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
        pv114 += safelog(sigma(pv33, pv35));

      pv115 = 0.0;
      for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
        pv115 += q(pv34, pv35);
      pv116 += pv114 * pv115;
    }

  pv118 = 0.0;
  for( pv34 = 0;pv34 <= n_points - 1;pv34++ )
    for( pv35 = 0;pv35 <= n_classes - 1;pv35++ )
      {
        pv117 = 0.0;
        for( pv33 = 0;pv33 <= n_variables - 1;pv33++ )
          pv117 += (sim_data(pv33, pv34) - mu(pv33, pv35)) *
                     (sim_data(pv33, pv34) - mu(pv33, pv35)) /
                     (sigma(pv33, pv35) * sigma(pv33, pv35));
        pv118 += pv117 * q(pv34, pv35);
      }

  pv120 = 0.0;
  for( pv19 = 0;pv19 <= n_classes - 1;pv19++ )
    {
      pv119 = 0.0;
      for( pv18 = 0;pv18 <= n_points - 1;pv18++ )
```

```
        pv119 += q(pv18, pv19);
      pv120 += pv119 * safelog(phi(pv19));
    }
  loglikelihood = -0.5 *
                    (n_points * n_variables * (safelog(2) + safelog(M_PI)) +
                     pv118) + pv120 - pv116;

  retval.resize(6);
  retval(0) = class_assignment;
  retval(1) = mu;
  retval(2) = phi;
  retval(3) = sigma;
  retval(4) = loglikelihood;
  retval(5) = errors;

  return retval;
}
//-- End of code
    ---------------------------------------------------------------
```

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

```

//----------------------------------------------------------------------------
// Code file generated by AutoBayes V0.9.9
// AutoBayes(c) 2008-2011 United States Government as represented by
// the Administrator of NASA. AutoBayes is distributed under the NASA
// Open Source Agreement (NOSA), version 1.3. See AutoBayes license for
// details.
// Problem:   SIMPLE MULTIVARIATE CLUSTERING MODEL FOR CLASSICAL IRIS FLOWER
// EXAMPLE
// Source:    iris.ab
// Command:
//
//   PROLOG_VAR
//

//              -instrument
//              iris.ab
// Generated: Thu Jun  4 11:08:35 2020
//----------------------------------------------------------------------------

#include "autobayes.h"




//----------------------------------------------------------------------------
// Octave Function: iris
//----------------------------------------------------------------------------

DEFUN_DLD(iris,input_args,output_args,
          "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"
         )
{
  octave_value_list retval;
  if (input_args.length () != 4 || output_args != 5 ){
    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";
    return retval;
  }

  //-- Input declarations -------------------------------------------------------

  octave_value arg_iris_data = input_args(0);
  if (!arg_iris_data.is_real_matrix()){
    gripe_wrong_type_arg("iris_data", (const std::string &)"Matrix expected");
    return retval;
  }
  Matrix iris_data = (Matrix)(arg_iris_data.matrix_value());

    // NUMBER OF CLASSES
  octave_value arg_n_classes = input_args(1);
  if (!arg_n_classes.is_real_scalar()){
    gripe_wrong_type_arg("n_classes", (const std::string &)"int expected");
    return retval;
  }
  int n_classes = (int)(arg_n_classes.int_value());

    // Iteration tolerance for convergence loop
  octave_value arg_tolerance = input_args(2);
  if (!arg_tolerance.is_real_scalar()){
    gripe_wrong_type_arg("tolerance", (const std::string &)"double expected");
    return retval;
  }
  double tolerance = (double)(arg_tolerance.double_value());

    // maximal number of iterations
  octave_value arg_maxiteration = input_args(3);
  if (!arg_maxiteration.is_real_scalar()){
    gripe_wrong_type_arg("maxiteration", (const std::string &)"int expected");
    return retval;
  }
  int maxiteration = (int)(arg_maxiteration.int_value());

  //-- Constant declarations ----------------------------------------------------

    // NUMBER OF DATA POINTS
  int n_points = arg_iris_data.columns();

    // NUMBER OF FEATURES
  int n_variables = arg_iris_data.rows();

  //-- Output declarations ------------------------------------------------------

```

```
  // CLASS OF EACH POINT
ColumnVector class_assignment(n_points);

  // MATRIX OF MEANS
Matrix mu(n_variables, n_classes);

  // CLASS PROBABILITY VECTOR.
ColumnVector phi(n_classes);

  // MATRIX OF STD DEVS
Matrix sigma(n_variables, n_classes);

  // intrumentation: assembly of convergence data
ColumnVector errors(1000);


//-- Local declarations ----------------------------------------------------

// Label: label0
// class membership table used in Discrete EM-algorithm
Matrix q(n_points, n_classes);

// local centers used for center-based initialization
Matrix center(n_classes, n_variables);

// Random index of data point
int pick;

// Loop variable
int pv69;

// Loop variable
int pv66;

// Loop variable
int pv67;

// Lagrange-multiplier
double l;

// Loop variable
int pv43;

// Loop variable
int pv44;

// Loop variable
int pv14;

// Loop variable
int pv25;

// Common subexpression
//   sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))
```

```
double pv53;

// Memoized common subexpression
//   phi(pv57) *
//    prod([pv54 := 0 .. -1 + n_variables],
//         exp(-1 / 2 * (iris_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
//              sigma(pv54, pv57) ** 2) *
//          (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
ColumnVector pv59(n_classes);

// Common subexpression
//   sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
double pv61;

// Loop variable
int pv57;

// Loop variable
int pv81;

int pv71;

// Summation accumulator
//   sum([pv71 := 0 .. -1 + n_classes], sqrt(pv87))
double pv88;

// Summation accumulator
//   sum([pv70 := 0 .. -1 + n_variables],
//       (center(pv69, pv70) - iris_data(pv70, pv14)) ** 2)
double pv86;

int pv70;

// Summation accumulator
//   sum([pv72 := 0 .. -1 + n_variables],
//       (center(pv71, pv72) - iris_data(pv72, pv14)) ** 2)
double pv87;

int pv72;

Matrix muold(n_variables, n_classes);

ColumnVector phiold(n_classes);

Matrix sigmaold(n_variables, n_classes);

int pv74;

int pv75;

int pv76;

int pv77;

```

```
int pv78;

// convergence loop counter
int loopcounter;

// sum up the Diffs
double pv89;

// Summation accumulator
//   sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
double pv97;

int pv27;

// Summation accumulator
//   sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44))
double pv98;

int pv50;

// Summation accumulator
//   sum([pv49 := 0 .. -1 + n_points], iris_data(pv43, pv49) * q(pv49, pv44)
  )
double pv99;

int pv49;

// Summation accumulator
//   sum([pv51 := 0 .. -1 + n_points],
//       (-mu(pv43, pv44) + iris_data(pv43, pv51)) ** 2 * q(pv51, pv44))
double pv100;

int pv51;

// Product accumulator
//   sum([pv54 := 0 .. -1 + n_variables],
//       exp(-1 / 2 * (iris_data(pv54, pv14) - mu(pv54, pv57)) ** 2 /
//              sigma(pv54, pv57) ** 2) *
//        (1 / (sqrt(2 * pi) * sigma(pv54, pv57))))
double pv106;

int pv54;

// Summation accumulator
//   sum([pv56 := 0 .. -1 + n_classes], pv59(pv56))
double pv107;

int pv56;

int pv92;

// Summation accumulator
//   sum([pv92 := 0 .. -1 + n_classes],
//       abs(phi(pv92) - phiold(pv92)) / (abs(phi(pv92)) + abs(phiold(pv92))
```

```
  ))
double pv109;

int pv91;

int pv90;

// Summation accumulator
//   sum([pv90 := 0 .. -1 + n_variables, pv91 := 0 .. -1 + n_classes],
//       abs(mu(pv90, pv91) - muold(pv90, pv91)) /
//        (abs(mu(pv90, pv91)) + abs(muold(pv90, pv91))))
double pv108;

// Summation accumulator
//   sum([pv93 := 0 .. -1 + n_variables, pv94 := 0 .. -1 + n_classes],
//       abs(sigma(pv93, pv94) - sigmaold(pv93, pv94)) /
//        (abs(sigma(pv93, pv94)) + abs(sigmaold(pv93, pv94))))
double pv110;

int pv93;

int pv94;

// Argmax index
int pv111;

// Argmax value
double pv112;

// Argmax temporary
double pv113;

// Argmax loop index
int pv84;

// Check constraints on inputs
ab_assert( 0 < n_classes );
ab_assert( 10 * n_classes < n_points );

// Label: label1
// Label: label2
// Label: label4
// Discrete EM-algorithm
//
// The model describes a discrete latent (or hidden) variable problem with
// the latent variable class_assignment and the data variable iris_data. The
// problem to optimize the conditional probability pr(iris_data |
// {mu,phi,sigma}) w.r.t. the variables mu, phi, and sigma can thus be
// solved by an application of the (discrete) EM-algorithm.
// The algorithm maintains as central data structure a class membership
// table q (see "label0") such that q(pv14,pv62) is the probability that
// data point pv14 belongs to class pv62, i.e.,
//
//   q(pv14, pv62) == pr([class_assignment(pv14) == pv62])
```

```
//
// The algorithm consists of an initialization phase for q (see "label2"),
// followed by a convergence phase (see "label5"), followed by the
// extraction of the hidden variable class_assignment (see "label6").
//
// Initialization
//
// The initialization is center-based, i.e., for each class (i.e., value of
// the hidden variable class_assignment) a center value center is chosen
// first (see "label4"). Then, the values for the local distribution are
// calculated as distances between the data points and these center values
// (see "label7").
//
// Random initialization of the centers center with data points;
// note that a data point can be picked as center more than once.
for( pv66 = 0;pv66 <= n_classes - 1;pv66++ )
  {
    pick = uniform_int_rnd(n_points - 1);
    for( pv67 = 0;pv67 <= n_variables - 1;pv67++ )
      center(pv66, pv67) = iris_data(pv67, pick);
  }
// Label: label7
for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
  for( pv69 = 0;pv69 <= n_classes - 1;pv69++ )
    {
      pv86 = 0.0;
      for( pv70 = 0;pv70 <= n_variables - 1;pv70++ )
        pv86 += (center(pv69, pv70) - iris_data(pv70, pv14)) *
                 (center(pv69, pv70) - iris_data(pv70, pv14));

      pv88 = 0.0;
      for( pv71 = 0;pv71 <= n_classes - 1;pv71++ )
        {
          pv87 = 0.0;
          for( pv72 = 0;pv72 <= n_variables - 1;pv72++ )
            pv87 += (center(pv71, pv72) - iris_data(pv72, pv14)) *
                     (center(pv71, pv72) - iris_data(pv72, pv14));
          pv88 += sqrt(pv87);
        }
      q(pv14, pv69) = sqrt(pv86) / pv88;
    }
// resize vector to maximal size
errors.resize(1000);
// initialize convergence output
for( loopcounter = 0;loopcounter <= 999;loopcounter++ )
  errors(loopcounter) = 0;
// Tolerance value must be positive
ab_assert( tolerance > 0 );
// max nr of iterations must be positive
ab_assert( maxiteration > 0 );
loopcounter = 0;
// repeat at least once
pv89 = tolerance;
while( ((loopcounter < maxiteration) && (pv89 >= tolerance)) )
```

```
  {
    loopcounter = 1 + loopcounter;
    if ( loopcounter > 1 )
      {
        // assign current values to old values
        for( pv74 = 0;pv74 <= n_variables - 1;pv74++ )
          for( pv75 = 0;pv75 <= n_classes - 1;pv75++ )
            muold(pv74, pv75) = mu(pv74, pv75);
        // assign current values to old values
        for( pv76 = 0;pv76 <= n_classes - 1;pv76++ )
          phiold(pv76) = phi(pv76);
        // assign current values to old values
        for( pv77 = 0;pv77 <= n_variables - 1;pv77++ )
          for( pv78 = 0;pv78 <= n_classes - 1;pv78++ )
            sigmaold(pv77, pv78) = sigma(pv77, pv78);
      }
    else
      ;

    // Label: label8
    // Label: label3
    // M-Step
    //
    // Decomposition I
    //
    // The problem to optimize the conditional probability
    // pr({class_assignment,iris_data} | {mu,phi,sigma}) w.r.t. the
    // variables mu, phi, and sigma can under the given dependencies by
    // Bayes rule be decomposed into two independent subproblems:
    //
    //   max pr(class_assignment | phi) for phi
    //   max pr(iris_data | {class_assignment,mu,sigma}) for {mu,sigma}
    //
    //
    // The conditional probability pr(class_assignment | phi) is under the
    // dependencies given in the model equivalent to
    //
    //   prod([pv18 := 0 .. -1 + n_points], pr(class_assignment(pv18) | phi)
)
    //
    // The probability occuring here is atomic and can thus be replaced by
    // the respective probability density function given in the model.
    // Summing out the expected variable class_assignment(pv14) yields the
    // log-likelihood function
    //
    //   sum_domain([pv14 := 0 .. -1 + n_points],
    //              [pv19 := 0 .. -1 + n_classes], [class_assignment(pv14)],
    //              q(pv14, pv19),
    //              log(prod([pv18 := 0 .. -1 + n_points],
    //                       phi(class_assignment(pv18)))))
    //
    // which can be simplified to
    //
    //   sum([pv19 := 0 .. -1 + n_classes],
```

```
    //        log(phi(pv19)) *
    //         sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
    //
    // This function is then optimized w.r.t. the goal variable phi.
    //
    // The expression
    //
    //    sum([pv19 := 0 .. -1 + n_classes],
    //        log(phi(pv19)) *
    //         sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
    //
    // is maximized w.r.t. the variable phi under the constraint
    //
    //    0 == 1 + -1 * sum([pv24 := 0 .. -1 + n_classes], phi(pv24))
    //
    // using the Lagrange-multiplier l.
    l = (double)(-n_points);
    for( pv25 = 0;pv25 <= n_classes - 1;pv25++ )
      // The summand
      //
      //    -1 * l
      //
      // is constant with respect to the goal variable phi(pv25) and can
      // thus be ignored for maximization.
      //
      // The function
      //
      //    l * sum([pv24 := 0 .. -1 + n_classes], phi(pv24)) +
      //     sum([pv19 := 0 .. -1 + n_classes],
      //         log(phi(pv19)) *
      //          sum([pv18 := 0 .. -1 + n_points], q(pv18, pv19)))
      //
      // is then symbolically maximized w.r.t. the goal variable phi(pv25).
      // The differential
      //
      //    l +
      //     phi(pv25) ** -1 * sum([pv18 := 0 .. -1 + n_points], q(pv18, pv25
))
      //
      // is set to zero; this equation yields the solution
      //
      //    -1 * l ** -1 * sum([pv27 := 0 .. -1 + n_points], q(pv27, pv25))
      //
      {
        pv97 = 0.0;
        for( pv27 = 0;pv27 <= n_points - 1;pv27++ )
          pv97 += q(pv27, pv25);
        phi(pv25) = -pv97 / l;
      }

    // The conditional probability pr(iris_data |
    // {class_assignment,mu,sigma}) is under the dependencies given in the
    // model equivalent to
    //
```

```
    //   prod([pv33 := 0 .. -1 + n_variables, pv34 := 0 .. -1 + n_points],
    //        pr(iris_data(pv33,pv34) | {class_assignment(pv34),mu(pv33,*),sigma(pv33,*)}))
    //
    // The probability occuring here is atomic and can thus be replaced by
    // the respective probability density function given in the model.
    // Summing out the expected variable class_assignment(pv14) yields the
    // log-likelihood function
    //
    //   sum_domain([pv14 := 0 .. -1 + n_points],
    //              [pv35 := 0 .. -1 + n_classes], [class_assignment(pv14)],
    //              q(pv14, pv35),
    //              log(prod([pv33 := 0 .. -1 + n_variables,
    //                        pv34 := 0 .. -1 + n_points],
    //                       exp(-1 / 2 *
    //                            (iris_data(pv33, pv34) -
    //                             mu(pv33, class_assignment(pv34))) ** 2 /
    //                            sigma(pv33, class_assignment(pv34)) ** 2) *
    //                        (1 /
    //                         (sqrt(2 * pi) *
    //                          sigma(pv33, class_assignment(pv34))))))
    //
    // which can be simplified to
    //
    //   -1 *
    //    sum([pv35 := 0 .. -1 + n_classes],
    //        sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35))) *
    //         sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
    //    -1 / 2 * n_points * n_variables * log(2) +
    //    -1 / 2 * n_points * n_variables * log(pi) +
    //    -1 / 2 *
    //     sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
    //         q(pv34, pv35) *
    //          sum([pv33 := 0 .. -1 + n_variables],
    //              (-1 * mu(pv33, pv35) + iris_data(pv33, pv34)) ** 2 *
    //               sigma(pv33, pv35) ** -2))
    //
    // This function is then optimized w.r.t. the goal variables mu and
    // sigma.
    //
    // The summands
    //
    //   -1 / 2 * n_points * n_variables * log(2)
    //   -1 / 2 * n_points * n_variables * log(pi)
    //
    // are constant with respect to the goal variables mu and sigma and can
    // thus be ignored for maximization.
    //
    // Index decomposition
    //
    // The function
    //
    //   -1 *
```

```
      //    sum([pv35 := 0 .. -1 + n_classes],
      //        sum([pv33 := 0 .. -1 + n_variables], log(sigma(pv33, pv35))) *
      //         sum([pv34 := 0 .. -1 + n_points], q(pv34, pv35))) +
      //    -1 / 2 *
      //     sum([pv34 := 0 .. -1 + n_points, pv35 := 0 .. -1 + n_classes],
      //         q(pv34, pv35) *
      //          sum([pv33 := 0 .. -1 + n_variables],
      //              (-1 * mu(pv33, pv35) + iris_data(pv33, pv34)) ** 2 *
      //               sigma(pv33, pv35) ** -2))
      //
      // can be optimized w.r.t. the variables mu(pv43,pv44) and
      // sigma(pv43,pv44) element by element (i.e., along the index variables
      // pv43 and pv44) because there are no dependencies along thats
      // dimensions.
      for( pv43 = 0;pv43 <= n_variables - 1;pv43++ )
        for( pv44 = 0;pv44 <= n_classes - 1;pv44++ )
          // The factor
          //
          //   n_classes
          //
          // is non-negative and constant with respect to the goal variables
          // mu(pv43,pv44) and sigma(pv43,pv44) and can thus be ignored for
          // maximization.
          //
          // The function
          //
          //   -1 * n_variables * log(sigma(pv43, pv44)) *
          //    sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
          //    -1 / 2 * n_variables * sigma(pv43, pv44) ** -2 *
          //     sum([pv34 := 0 .. -1 + n_points],
          //         (-1 * mu(pv43, pv44) + iris_data(pv43, pv34)) ** 2 *
          //          q(pv34, pv44))
          //
          // is then symbolically maximized w.r.t. the goal variables
          // mu(pv43,pv44) and sigma(pv43,pv44). The partial differentials
          //
          //   df / d_mu(pvar(43),pvar(44)) ==
          //    -1 * n_variables * sigma(pv43, pv44) ** -2 * mu(pv43, pv44) *
          //     sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
          //     n_variables * sigma(pv43, pv44) ** -2 *
          //      sum([pv34 := 0 .. -1 + n_points],
          //          iris_data(pv43, pv34) * q(pv34, pv44))
          //   df / d_sigma(pvar(43),pvar(44)) ==
          //    -1 * n_variables * sigma(pv43, pv44) ** -1 *
          //     sum([pv34 := 0 .. -1 + n_points], q(pv34, pv44)) +
          //     n_variables * sigma(pv43, pv44) ** -3 *
          //      sum([pv34 := 0 .. -1 + n_points],
          //          (-1 * mu(pv43, pv44) + iris_data(pv43, pv34)) ** 2 *
          //           q(pv34, pv44))
          //
          // are set to zero; these equations yield the solutions
          //
          //   mu(pv43, pv44) ==
          //    cond(0 == n_variables or
```

```
          //          0 == sum([pv47 := 0 .. -1 + n_points], q(pv47, pv44)),
          //         fail(division_by_zero),
          //         sum([pv48 := 0 .. -1 + n_points], q(pv48, pv44)) ** -1 *
          //          sum([pv49 := 0 .. -1 + n_points],
          //              iris_data(pv43, pv49) * q(pv49, pv44)))
          //   sigma(pv43, pv44) ==
          //    cond(0 == n_variables or
          //          0 == sum([pv50 := 0 .. -1 + n_points], q(pv50, pv44)),
          //         fail(division_by_zero),
          //         abs(n_variables) * n_variables ** -1 *
          //          sum([pv51 := 0 .. -1 + n_points],
          //              (-1 * mu(pv43, pv44) + iris_data(pv43, pv51)) ** 2 *
          //               q(pv51, pv44)) ** (1 / 2) *
          //          sum([pv52 := 0 .. -1 + n_points], q(pv52, pv44)) **
          //           (-1 / 2))
          //
          {
            // Initialization of common subexpression
            pv98 = 0.0;
            for( pv50 = 0;pv50 <= n_points - 1;pv50++ )
              pv98 += q(pv50, pv44);
            pv53 = pv98;

            if ( ((0 == n_variables) || (0 == pv53)) )
              { ab_error( division_by_zero ); }
            else
              {
                pv99 = 0.0;
                for( pv49 = 0;pv49 <= n_points - 1;pv49++ )
                  pv99 += iris_data(pv43, pv49) * q(pv49, pv44);
                mu(pv43, pv44) = pv99 * ((double)(1) / pv53);
              }
            if ( ((0 == n_variables) || (0 == pv53)) )
              { ab_error( division_by_zero ); }
            else
              {
                pv100 = 0.0;
                for( pv51 = 0;pv51 <= n_points - 1;pv51++ )
                  pv100 += (iris_data(pv43, pv51) - mu(pv43, pv44)) *
                            (iris_data(pv43, pv51) - mu(pv43, pv44)) *
                            q(pv51, pv44);
                sigma(pv43, pv44) = abs(n_variables) * sqrt(pv100) *
                                      ((double)(1) / (double)(n_variables)) *
                                      ((double)(1) / sqrt(pv53));
              }
          }
      // Label: label9
      // E-Step
      // Update the current values of the class membership table q.
      for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
        {
          // Initialization of common subexpression
          for( pv57 = 0;pv57 <= n_classes - 1;pv57++ )
            {
```

```
              pv106 = 1.0;
              for( pv54 = 0;pv54 <= n_variables - 1;pv54++ )
                pv106 *= exp(-0.5 * (iris_data(pv54, pv14) - mu(pv54, pv57)) *
                               (iris_data(pv54, pv14) - mu(pv54, pv57)) /
                               (sigma(pv54, pv57) * sigma(pv54, pv57))) *
                           ((double)(1) /
                            (sqrt(M_PI * (double)(2)) * sigma(pv54, pv57)));
              pv59(pv57) = phi(pv57) * pv106;
            }

          pv107 = 0.0;
          for( pv56 = 0;pv56 <= n_classes - 1;pv56++ )
            pv107 += pv59(pv56);
          pv61 = pv107;
          for( pv81 = 0;pv81 <= n_classes - 1;pv81++ )
            // The denominator pv61 can become zero due to round-off errors.
            // In that case, each class is considered to be equally likely.
            if ( pv61 == 0.0 )
              q(pv14, pv81) = (double)(1) / (double)(n_classes);
            else
              q(pv14, pv81) = pv59(pv81) / pv61;
        }
      if ( loopcounter > 1 )
        {
          pv108 = 0.0;
          for( pv90 = 0;pv90 <= n_variables - 1;pv90++ )
            for( pv91 = 0;pv91 <= n_classes - 1;pv91++ )
              pv108 += abs(mu(pv90, pv91) - muold(pv90, pv91)) /
                         (abs(mu(pv90, pv91)) + abs(muold(pv90, pv91)));

          pv109 = 0.0;
          for( pv92 = 0;pv92 <= n_classes - 1;pv92++ )
            pv109 += abs(phi(pv92) - phiold(pv92)) /
                       (abs(phi(pv92)) + abs(phiold(pv92)));

          pv110 = 0.0;
          for( pv93 = 0;pv93 <= n_variables - 1;pv93++ )
            for( pv94 = 0;pv94 <= n_classes - 1;pv94++ )
              pv110 += abs(sigma(pv93, pv94) - sigmaold(pv93, pv94)) /
                         (abs(sigma(pv93, pv94)) + abs(sigmaold(pv93, pv94)));
          pv89 = pv108 + pv109 + pv110;

          if ( loopcounter <= 1000 )
            // collect convergence info
            errors(loopcounter - 2) = pv89;
          else
            ;
          octave_stdout << " pvar(89) = " << pv89 << endl;
        }
      else
        ;
    }
  errors.resize(loopcounter);
  // Label: label6
```

```
  // Extract the most likely values of the hidden variable
  // class_assignment(pv14) from the class membership table q.
  for( pv14 = 0;pv14 <= n_points - 1;pv14++ )
    {
      // Determine the position of the maximum with in the range
      //   0
      // ...
      //   -1 + n_classes
      // by iterating over this range and calculating the value at each point
      // (argmax).
      //
      // Argmax loop
      for( pv84 = 0;pv84 <= n_classes - 1;pv84++ )
        {
          pv113 = q(pv14, pv84);
          if ( ((pv84 == 0) || (pv113 > pv112)) )
            // Save new maximum
            {
              pv112 = pv113;
              pv111 = pv84;
            }
          else
            ;
        }
      class_assignment(pv14) = pv111;
    }

  retval.resize(5);
  retval(0) = class_assignment;
  retval(1) = mu;
  retval(2) = phi;
  retval(3) = sigma;
  retval(4) = errors;

  return retval;
}
//-- 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 (Varianace.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 vise versa.
7	IF Variance is used in Denominator, StandardDeviation is not used and vise versa.
8	IF Variance is used in Coefficient, StandardDeviation is not used and vise 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 CalcuateMean 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

```
-- $ProjectHeader: use 5.2.0 Thurs, 15 October 2020-CSci Master Thesis-Jason
   Hicks $

model AUTOBAYES

----------------------------------------------
-- Classes from input CD ----------------------
----------------------------------------------

class StatisticalModel
attributes
  name : String
  equation : String
end

class Pragmas
attributes
  name : String
  value : String
end

class HiddenVariable
attributes
  name : String
end

class ClassProbabilities
attributes
  name : String
  row_size : Integer
  col_size : Integer
  values : Bag(Real)
end

class Goal
attributes
  equation : String
end

class ModelParameters
attributes
  n_points : Integer
  n_classes : Integer
  n_variables : Integer
end

class InputData
attributes
```

```
  name : String
  row_size : Integer
  col_size : Integer
  values : Bag(Real)
end

class ClassParameters
end

class StandardDeviation
attributes
  name : String
  row_size : Integer
  col_size : Integer
  values : Bag(Real)
end

class Mean
attributes
  name : String
  row_size : Integer
  col_size : Integer
  values : Bag(Real)
end

class Variance
attributes
  name : String
  row_size : Integer
  col_size : Integer
  values : Bag(Real)
end

class Gaussian < StatisticalModel
end

class Exponent
end

class Coefficient
end

class Numerator
end

class Denominator
end


----------------------------------------------
-- Classes from subset of output CD -----------
----------------------------------------------

class GaussianModel
```

```
attributes
  name : String
  input_args : Bag(Bag(Real))
  output_args : Integer
  n_points : Integer
  n_variables : Integer
  n_classes : Integer
  input_data : Matrix
  data_set_name : String
operations
  check_in_and_out_args(input_args : Bag(Bag(Real)), output_args : Integer) :
   Boolean
  check_data_format(input_data : Matrix) : Boolean
  get_data() : Matrix
  get_n_points() : Integer
end

class Retval
attributes
  size : Integer
operations
  resize()
end

class Declaration
end

class InputDeclaration
end

class ConstantDeclaration
end

class OutputDeclaration
end

class LocalDeclaration
end

class Initialization
end

class Matrix
attributes
  name : String
  row_size : Integer
  col_size : Integer
  values : Bag(Real)
end

class OutputCodeMean < Matrix
attributes
  n_classes : Integer
  n_variables : Integer
```

```
end

class OutputCodeVariance < Matrix
end

class OutputCodeInputData < Matrix
end

class MemoizedCommonSubexpression < Matrix
end

class NormalDistribution
end

class CalculateVarianceLoop
attributes
  n_points : Integer
end

class CalculateMeanLoop
attributes
  n_points : Integer
end

class Transformations
attributes
  type : String
end

class CommonSubexpressionInitLoop
attributes
  n_points : Integer
operations
  transform_data(OutputCodeInputData : Matrix) : Matrix
end

class CheckDivideByZero
attributes
  n_points : Integer
  n_variables : Integer
operations
  divide_by_zero_check(IntToCheck : Integer) : Boolean
end

-------------------------------------------------------
-- Associations within input CD -----------------------
-------------------------------------------------------

association StatModelGoal between
  StatisticalModel[1..*]
  Goal[1]
end

association StatModelInData between
```

```
  StatisticalModel[1..*]
  InputData[1..*]
end

aggregation PragStatModel between
  Pragmas[0..*]
  StatisticalModel[1..*]
end

aggregation HidVarStatModel between
  HiddenVariable[0..1]
  StatisticalModel[1..*]
end

aggregation ClassParaStatModel between
  ClassParameters[1..*]
  StatisticalModel[1..*]
end

aggregation ModParaHidVar between
  ModelParameters[1..*]
  HiddenVariable[0..1]
end

aggregation ClassProbHidVar between
  ClassProbabilities[1]
  HiddenVariable[0..1]
end

aggregation ClassProbGoal between
  ClassProbabilities[0..1]
  Goal[1]
end

aggregation ModParaClassProb between
  ModelParameters[1..*]
  ClassProbabilities[0..1]
end

aggregation ModParaInData between
  ModelParameters[1..*]
  InputData[1..*]
end

aggregation ModParaClassPara between
  ModelParameters[0..*]
  ClassParameters[0..*]
end

aggregation ClassParaGoal between
  ClassParameters[1..*]
  Goal[1]
end
```

```
aggregation VarClassPara between
  Variance[0..1]
  ClassParameters[1]
end

aggregation MeanClassPara between
  Mean[1]
  ClassParameters[1]
end

aggregation StandDevClassPara between
  StandardDeviation[0..1]
  ClassParameters[1]
end

aggregation InDataGoal between
  InputData[1..*]
  Goal[1]
end

---- From breaking down the Guassian equ ----
aggregation CoeffGauss between
  Coefficient[1]
  Gaussian[1]
end

aggregation ExpGauss between
  Exponent[1]
  Gaussian[1]
end

aggregation CoeffExp between
  Coefficient[1]
  Exponent[1]
end

aggregation StandDevCoeff between
  StandardDeviation[0..1]
  Coefficient[1]
end

aggregation VarCoeff between
  Variance[0..1]
  Coefficient[1]
end

aggregation NumExp between
  Numerator[1..*]
  Exponent[1]
end

aggregation DenomfExp between
  Denominator[1..*]
  Exponent[1]
```

```
end

aggregation NumDenom between
  Numerator[1]
  Denominator[1]
end

aggregation InDataNum between
  InputData[1]
  Numerator[1]
end

aggregation MeanNum between
  Mean[1]
  Numerator[1]
end

aggregation InDataMean between
  InputData[1]
  Mean[1]
end

aggregation VarDenom between
  Variance[0..1]
  Denominator[1]
end

aggregation StandDevDenom between
  StandardDeviation[0..1]
  Denominator[1]
end


-------------------------------------------------------
-- Associations within output CD ----------------------
-------------------------------------------------------

association GaussModMatrix between
  GaussianModel[1..*]
  Matrix[1..*]
end

aggregation MatrixInit between
  Matrix[1..*]
  Initialization[0..*]
end

aggregation InitGaussMod between
  Initialization[1]
  GaussianModel[1..*]
end

--------------- Declarations ---------------
aggregation DeclarGaussMod between
```

```
  Declaration[1]
  GaussianModel[1..*]
end

aggregation InDeclar between
  InputDeclaration[1]
  Declaration[1..*]
end

aggregation ConstDeclar between
  ConstantDeclaration[1]
  Declaration[1..*]
end

aggregation OutDeclar between
  OutputDeclaration[1]
  Declaration[1..*]
end

aggregation LocalDeclar between
  LocalDeclaration[1]
  Declaration[1..*]
end

aggregation MatrixDeclar between
  Matrix[1..*]
  Declaration[0..1]
end

------------------ Retval ------------------
aggregation RetvalGaussMod between
  Retval[1]
  GaussianModel[1..*]
end

aggregation OutMeanRetval between
  OutputCodeMean[1]
  Retval[1]
end

aggregation OutVarRetval between
  OutputCodeVariance[0..1]
  Retval[1]
end

------------------ Normal ------------------
aggregation NormDistGaussMod between
  NormalDistribution[0..1]
  GaussianModel[1..*]
end

aggregation ChDivNormDist between
  CheckDivideByZero[1..*]
  NormalDistribution[1]
```

```
end

aggregation CalcMeanNormDist between
  CalculateMeanLoop[1]
  NormalDistribution[0..*]
end

aggregation CalcVarNormDist between
  CalculateVarianceLoop[1]
  NormalDistribution[0..*]
end

aggregation OutMeanCalcMean between
  OutputCodeMean[1]
  CalculateMeanLoop[0..*]
end

aggregation OutInDataCalcMean between
  OutputCodeInputData[1]
  CalculateMeanLoop[0..*]
end

aggregation OutMeanCalcVar between
  OutputCodeMean[2]
  CalculateVarianceLoop[0..*]
end

aggregation OutInDataCalcVar between
  OutputCodeInputData[2]
  CalculateVarianceLoop[0..*]
end

aggregation OutVarCalcVar between
  OutputCodeVariance[1]
  CalculateVarianceLoop[0..*]
end

---------------- Transfrom -----------------
aggregation TransformGaussMod between
  Transformations[0..1]
  GaussianModel[1..*]
end

aggregation ChDivTransform between
  CheckDivideByZero[1..*]
  Transformations[1]
end

aggregation ComSubInitTransform between
  CommonSubexpressionInitLoop[1]
  Transformations[0..*]
end

aggregation CalcMeanTransform between
```

```
  CalculateMeanLoop[1]
  Transformations[0..*]
end

aggregation CalcVarTransform between
  CalculateVarianceLoop[1]
  Transformations[0..*]
end

aggregation MemoComSubCalcMean between
  MemoizedCommonSubexpression[1]
  CalculateMeanLoop[0..*]
end

aggregation MemoComSubCalcVar between
  MemoizedCommonSubexpression[2]
  CalculateVarianceLoop[0..*]
end

aggregation MemoComSubComSubInit between
  MemoizedCommonSubexpression[1]
  CommonSubexpressionInitLoop[0..*]
end

aggregation OutInDataComSubInit between
  OutputCodeInputData[2]
  CommonSubexpressionInitLoop[0..*]
end

-------------------------------------------------------
-- Associations input CD and output CD ----------------
-------------------------------------------------------

association InInputDataGaussMod between
  InputData[0..*]
  GaussianModel[1]
end

association ModParaGaussMod between
  ModelParameters[1..*]
  GaussianModel[1]
end

association StatModGaussMod between
  StatisticalModel[1]
  GaussianModel[1]
end

association InMeanOutMean between
  Mean[1]
  OutputCodeMean[1]
end

association InVarOutVar between
```

```
  Variance[1]
  OutputCodeVariance[1]
end

association InInputDataOutInputData between
  InputData[1]
  OutputCodeInputData[1]
end

association PragGaussMod between
  Pragmas[0..*]
  GaussianModel[1]
end


-------------------------------------------------------
-- OCL constraints ------------------------------------
-------------------------------------------------------

constraints

---- Constraints for input ----

context Gaussian
  inv GaussName:
    self.name = 'gauss'

context Variance
  inv VarSize:
  self.name.size() > 0
    and self.row_size = 1
  and self.col_size = 1

context Mean
  inv MeanSize:
    self.name.size() > 0
    and self.row_size >= 1
  and self.col_size >= 1

context ModelParameters
  inv ModParamSize:
    self.n_classes >= 1
  and self.n_variables >= 1
  and self.n_points >= 1

context InputData
  inv InputDataName:
    self.name.size() > 0

context ClassParameters
  inv VarStdDevCP:
    self.variance->size() = 1 implies self.standardDeviation->size() = 0
  and self.standardDeviation->size() = 1 implies self.variance->size() = 0

```

```
context Denominator
  inv VarStdDevDenom:
    self.variance->size() = 1 implies self.standardDeviation->size() = 0
  and self.standardDeviation->size() = 1 implies self.variance->size() = 0

context Coefficient
  inv VarStdDevCoeff:
    self.variance->size() = 1 implies self.standardDeviation->size() = 0
  and self.standardDeviation->size() = 1 implies self.variance->size() = 0

---- Constraints for output ----

context MemoizedCommonSubexpression
  inv MemoComSubSize:
    self.col_size = 1

context OutputCodeVariance
  inv OCVarSize:
  self.name.size() > 0
    and self.row_size = 1
  and self.col_size = 1

context OutputCodeVariance
  inv OCVarValues:
  self.values->forAll(v | v > 0)

context OutputCodeMean
  inv OCMeanSize:
    self.name.size() > 0
    and self.row_size >= 1
  and self.col_size >= 1

context GaussianModel
  inv NormDistOrTransfrom:
    self.normalDistribution->size() = 1 implies self.transformations->size() =
     0
  and self.transformations->size() = 1 implies self.normalDistribution->size()
     = 0

context GaussianModel
  inv NormMeanSize:
    self.normalDistribution->size() = 1
    implies self.normalDistribution.calculateMeanLoop.outputCodeMean.row_size
    = 1
    and self.normalDistribution.calculateMeanLoop.outputCodeMean.col_size = 1

context GaussianModel
  inv TransformMeanSize:
    self.transformations->size() = 1
    implies self.transformations.calculateMeanLoop.outputCodeMean.row_size = 1
    and self.transformations.calculateMeanLoop.outputCodeMean.col_size = 1

context GaussianModel
  inv TransformCSInitMemoCS:
```

```
    self.transformations->size() = 1
  implies self.transformations.commonSubexpressionInitLoop->size() = 1
  and self.transformations.commonSubexpressionInitLoop.memoizedCommonSubexpression->size() = 1

context GaussianModel
  inv NormOutInDataCalcMCalcV:
    self.normalDistribution->size() = 1
  implies self.normalDistribution.calculateMeanLoop.outputCodeInputData->size() = 1
  and self.normalDistribution.calculateVarianceLoop.outputCodeInputData->size() = 2


---- Constraints connecting input and output ----

context Mean
  inv InMeanOutMean:
    self.name = self.outputCodeMean.name
  and self.row_size = self.outputCodeMean.row_size
  and self.col_size = self.outputCodeMean.col_size

context Variance
  inv InVarOutVar:
    self.name = self.outputCodeVariance.name
  and self.row_size = self.outputCodeVariance.row_size
  and self.col_size = self.outputCodeVariance.col_size

context InputData
  inv InDataOutData:
    self.name = self.outputCodeInputData.name
  and self.row_size = self.outputCodeInputData.row_size
  and self.col_size = self.outputCodeInputData.col_size
    and self.values = self.outputCodeInputData.values

context StatisticalModel
   inv StatModNormDist:
     self.equation = 'x(_) ~ gauss(mu, sqrt(sigma_sq)).' implies (self.gaussianModel.normalDistribution->size() = 1
   and self.gaussianModel.transformations->size() = 0
   and self.classParameters.variance->size() = 1)

context StatisticalModel
   inv StatModTransformLog:
     self.equation = 'log(x(_)) ~ gauss(mu, sqrt(sigma_sq)).' implies (self.gaussianModel.transformations->size() = 1
   and self.gaussianModel.transformations.type = 'log'
   and self.gaussianModel.normalDistribution->size() = 0
   and self.classParameters.variance->size() = 1)

context StatisticalModel
   inv StatModTransformSquare:
     self.equation = 'x(_)**2 ~ gauss(mu, sqrt(sigma_sq)).' implies (self.gaussianModel.transformations->size() = 1
```

```
    and self.gaussianModel.transformations.type = 'square'
    and self.gaussianModel.normalDistribution->size() = 1
    and self.classParameters.variance->size() = 1)

---- The constraints from the multiplicities of the CDs ----

context GaussianModel inv:
  self.declaration->size() = 1
  and self.initialization->size() = 1
  and self.retval->size() = 1
  and self.matrix->size() >= 1
  and (self.normalDistribution->size() = 0 or self.normalDistribution->size()
    = 1)
  and (self.transformations->size() = 0 or self.transformations->size() = 1)

context Declaration inv:
  self.inputDeclaration->size() = 1
  and self.constantDeclaration->size() = 1
  and self.outputDeclaration->size() = 1
  and self.localDeclaration->size() = 1
  and self.matrix->size() >= 1

context StatisticalModel inv:
  self.pragmas->size() >= 0
  and (self.hiddenVariable->size() = 0 or self.hiddenVariable->size() = 1)
  and self.goal->size() = 1
  and self.inputData->size() >= 1
  and self.classParameters->size() >= 1

context ClassParameters inv:
  self.statisticalModel->size() >= 1
  and ( (self.variance->size() = 0 and self.standardDeviation->size() = 1)
  or (self.variance->size() = 1 and self.standardDeviation->size() = 0) )
  and self.mean->size() = 1
  and self.modelParameters->size() >= 0
  and self.goal->size() = 1

context Goal inv:
  self.statisticalModel->size() >= 1
  and (self.classProbabilities->size() = 0 or self.classProbabilities->size()
    = 1)
  and self.classParameters->size() >= 1
  and self.inputData->size() >= 1

context NormalDistribution inv:
  self.gaussianModel->size() >= 1
  and self.calculateMeanLoop->size() = 1
  and self.calculateVarianceLoop->size() = 1
  and self.checkDivideByZero->size() >= 1

context Transformations inv:
  self.gaussianModel->size() >= 1
  and self.calculateMeanLoop->size() = 1
  and self.calculateVarianceLoop->size() = 1
```

```
and self.checkDivideByZero->size() >= 1
and self.commonSubexpressionInitLoop->size() = 1
```

Listing A.7. The input file for the analysis with the USE tool.

A.5 User Interface Screens and Output From USE

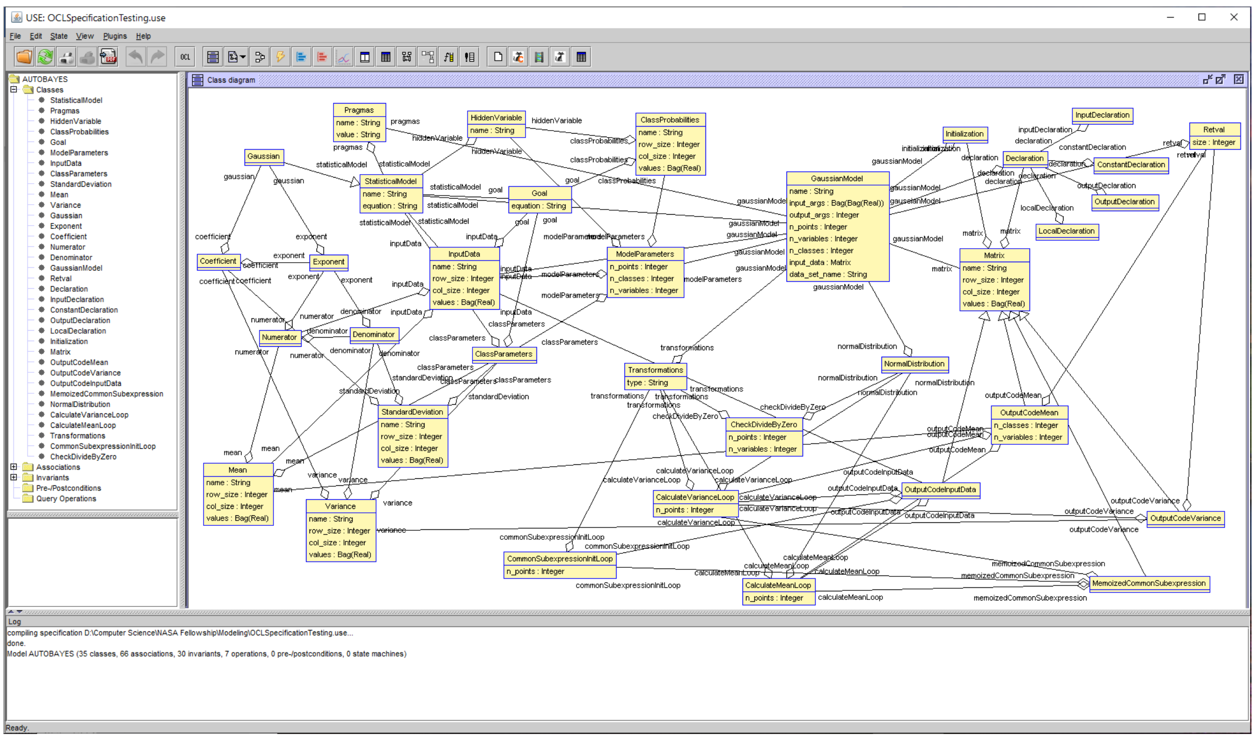

Figure A.7. The USE tool user interface after loading my input file.

```
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.

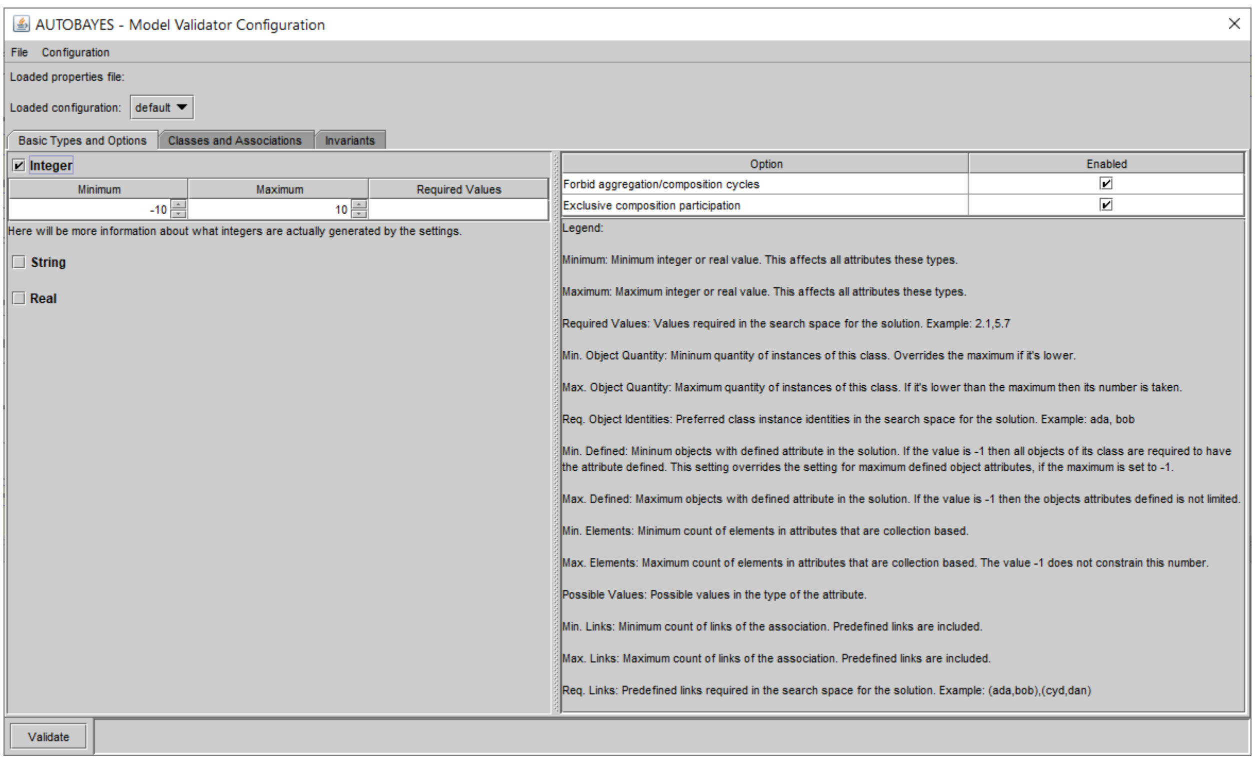

Figure A.9. The USE Model Validator Configuration - Basic Types and Options tab.

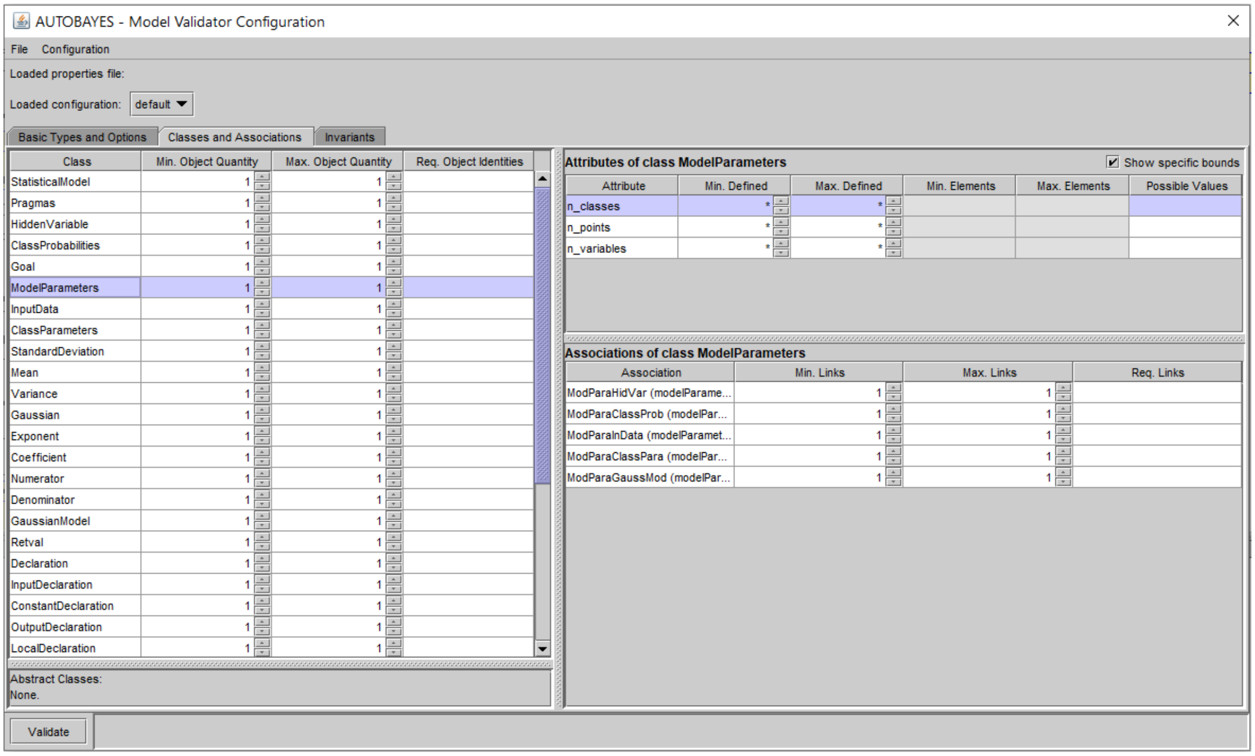

Figure A.10. The USE Model Validator Configuration - Classes and Associations tab.

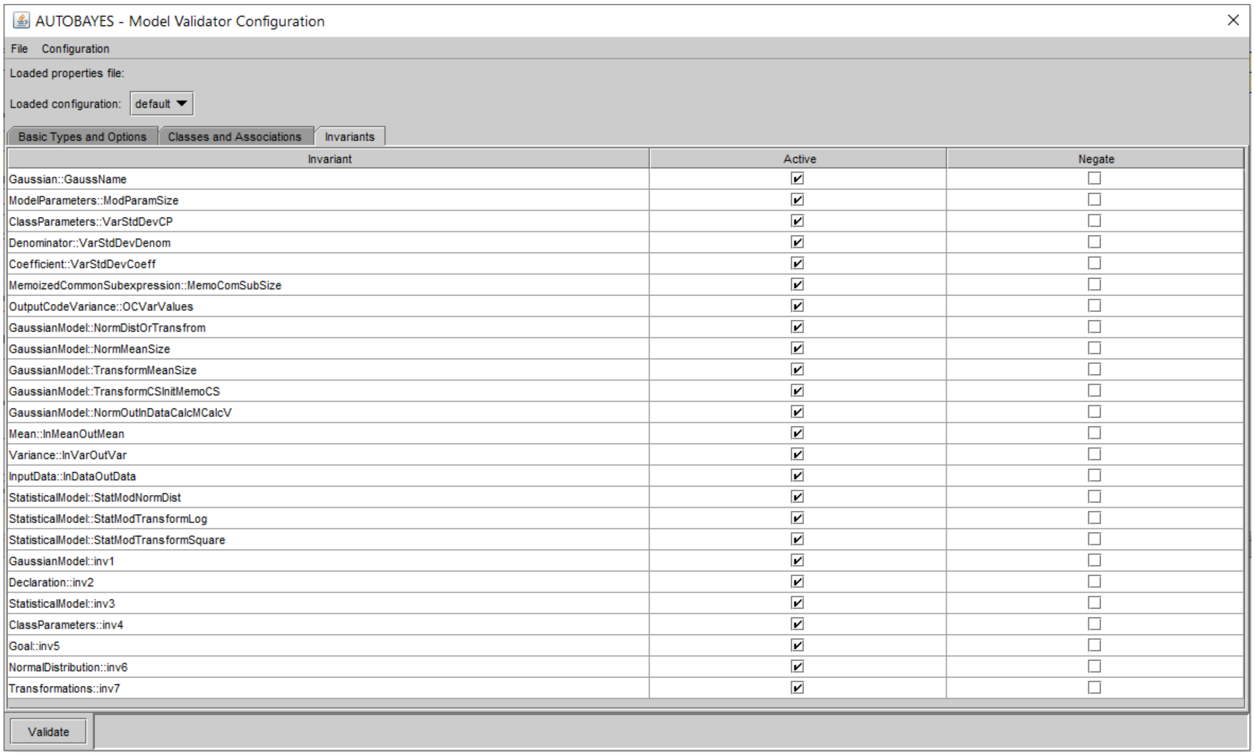

Figure A.11. The USE Model Validator Configuration - Invariants tab.

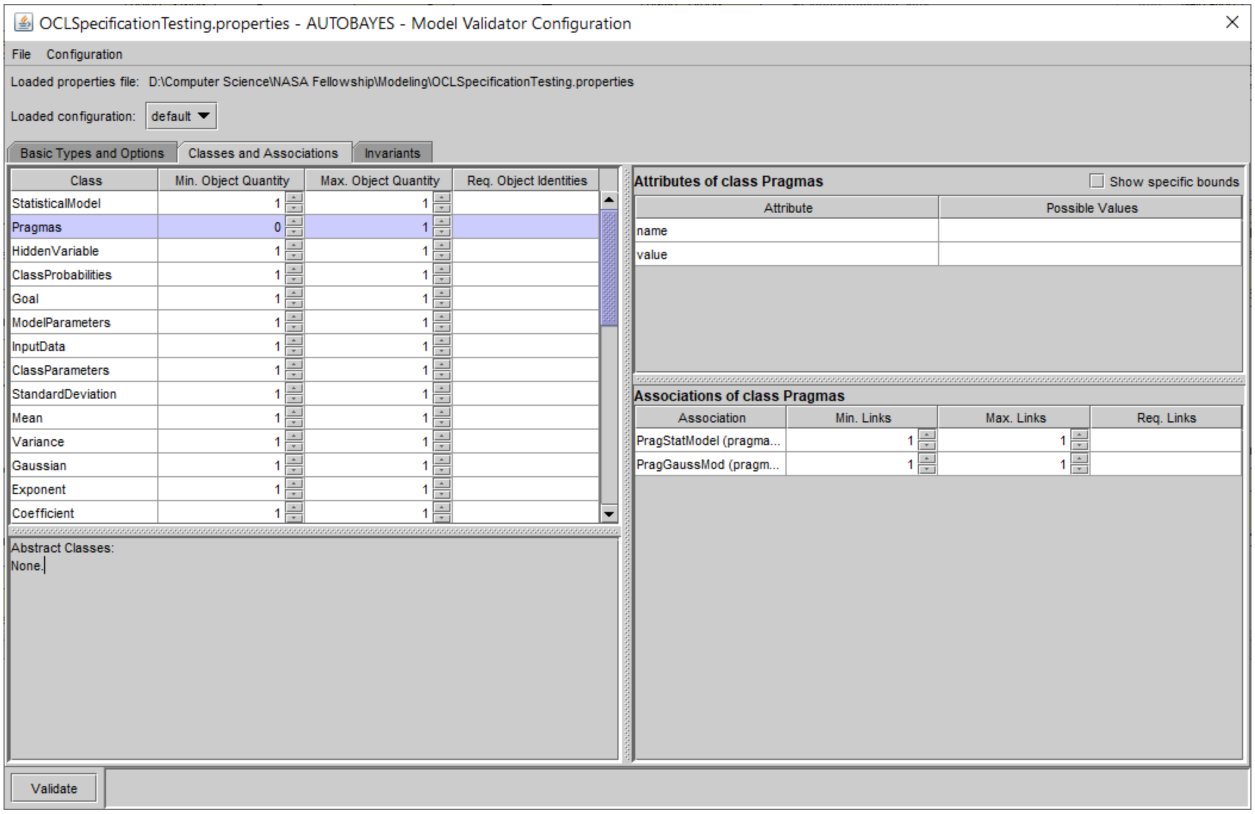

Figure A.12. The USE Model Validator Configuration - Classes and Associations tab with fix for the Pragma Class error.

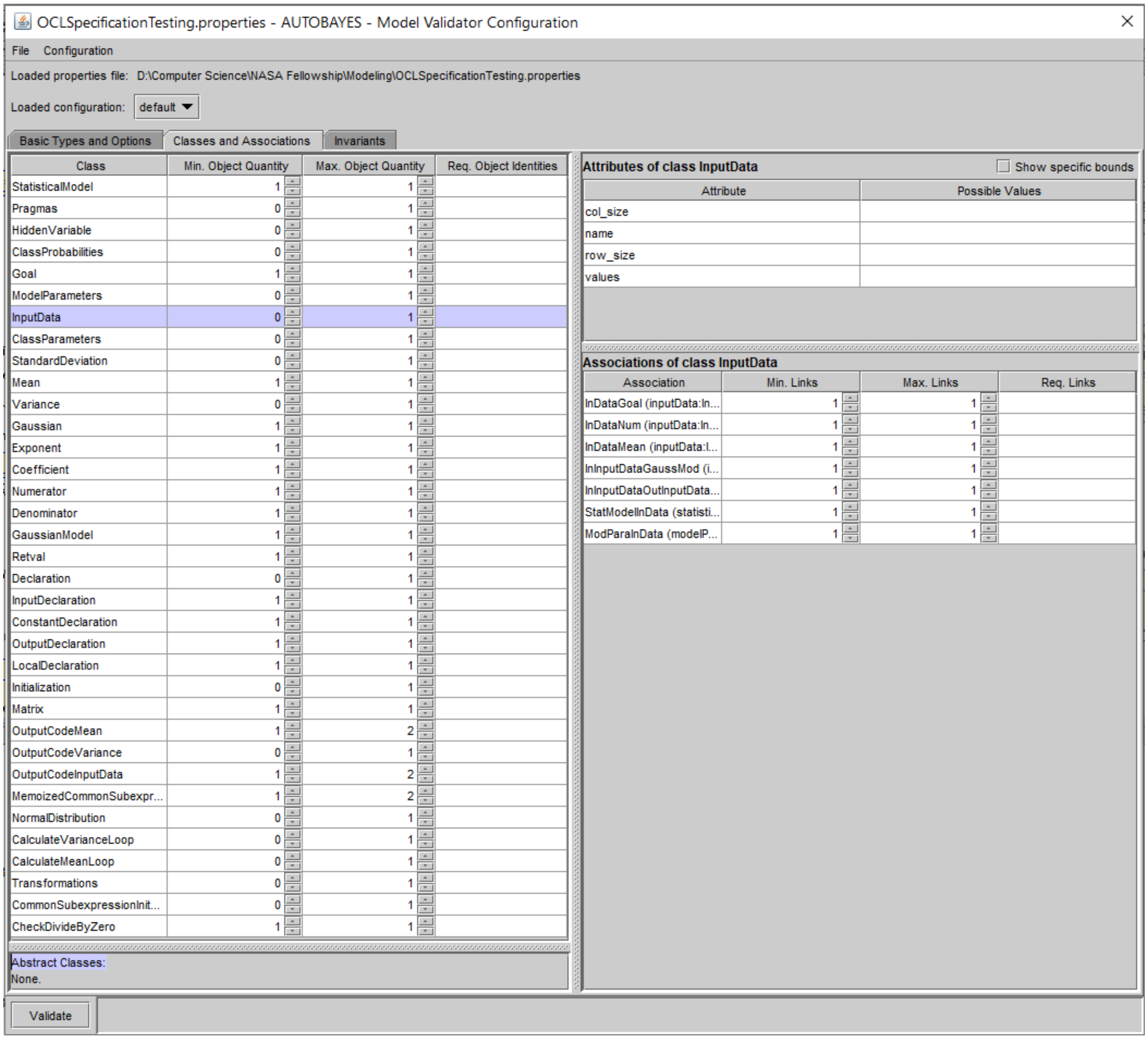

Figure A.13. The USE Model Validator Configuration - Classes and Associations tab with fix for each of the Classes that can have a multiplicity of 0.

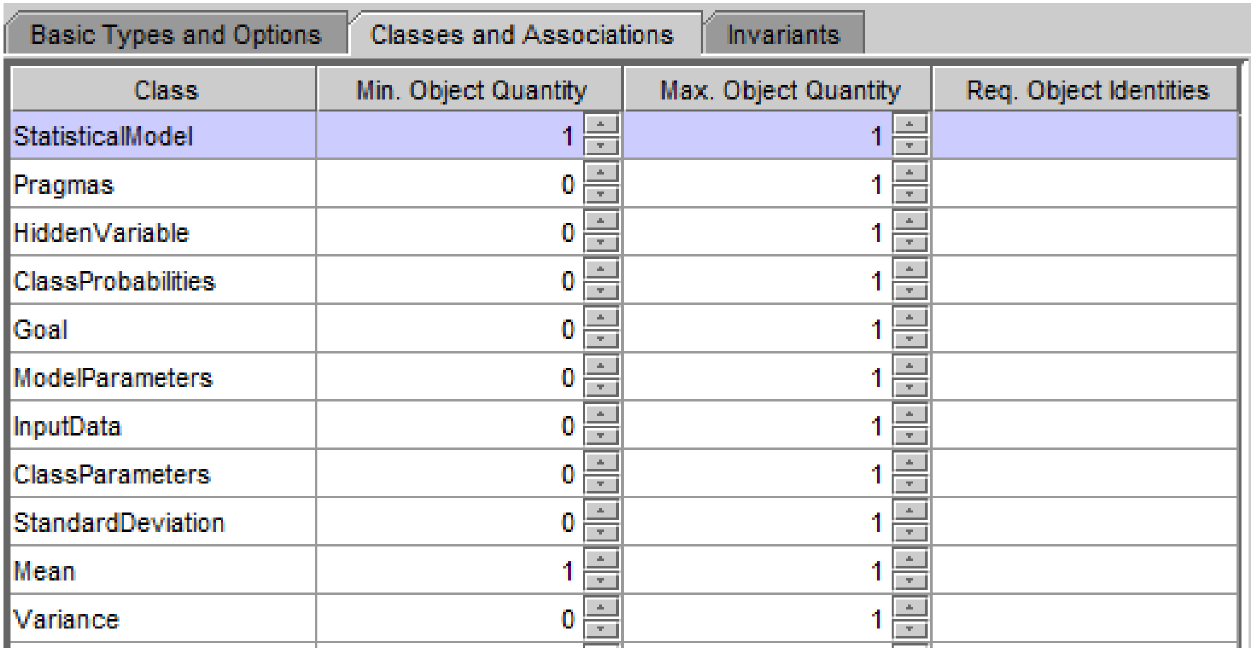

Figure A.14. The USE Model Validator Configuration - Classes and Associations tab with fix for the Goal Class error.

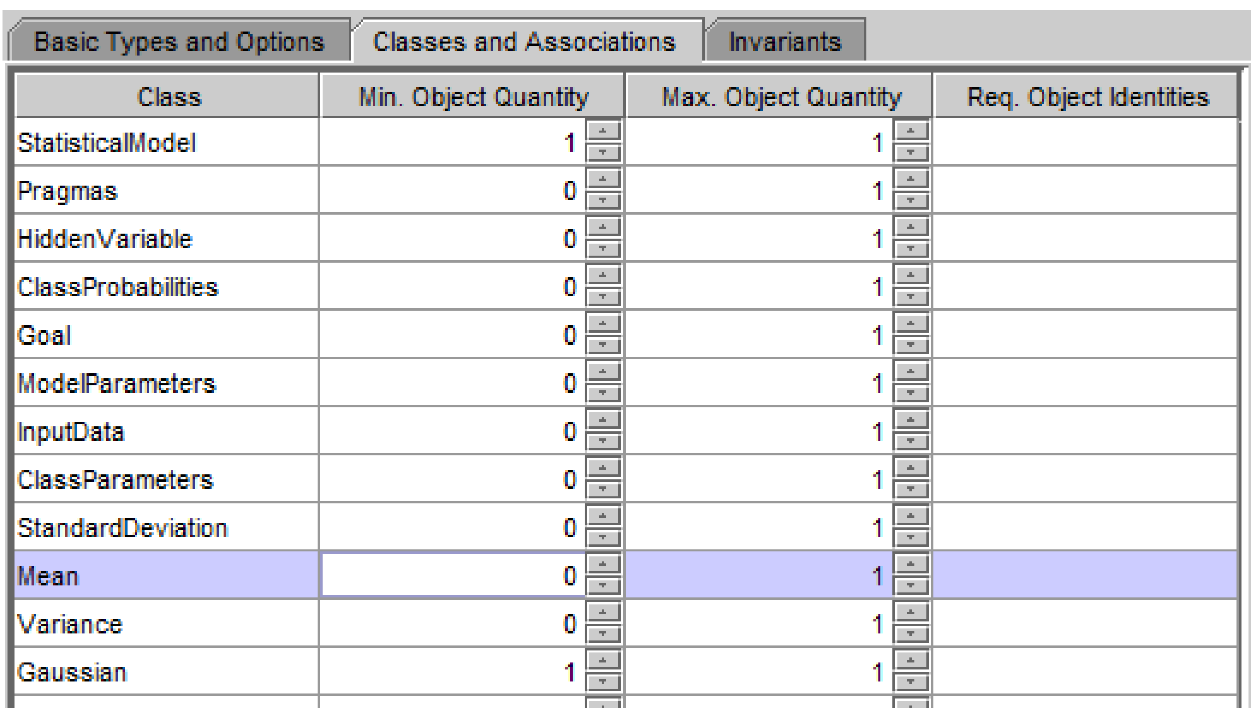

Figure A.15. The USE Model Validator Configuration - Classes and Associations tab with fix for the Mean Class error.

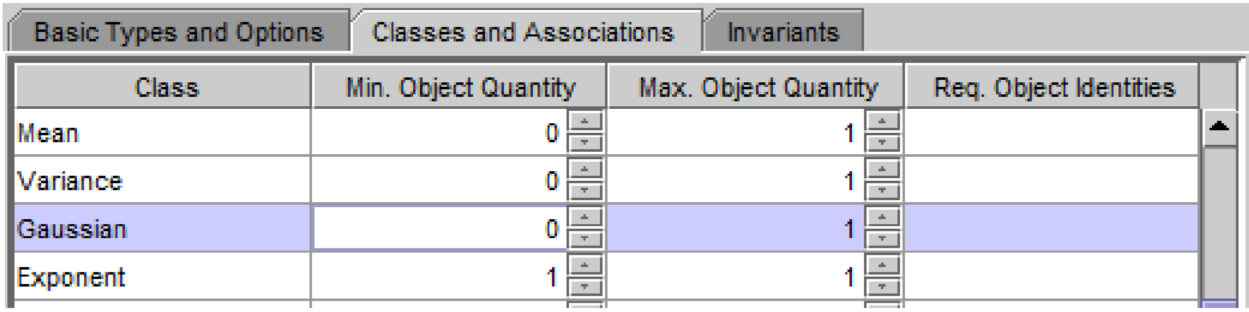

Figure A.16. The USE Model Validator Configuration - Classes and Associations tab with fix for the Gaussian Class error.

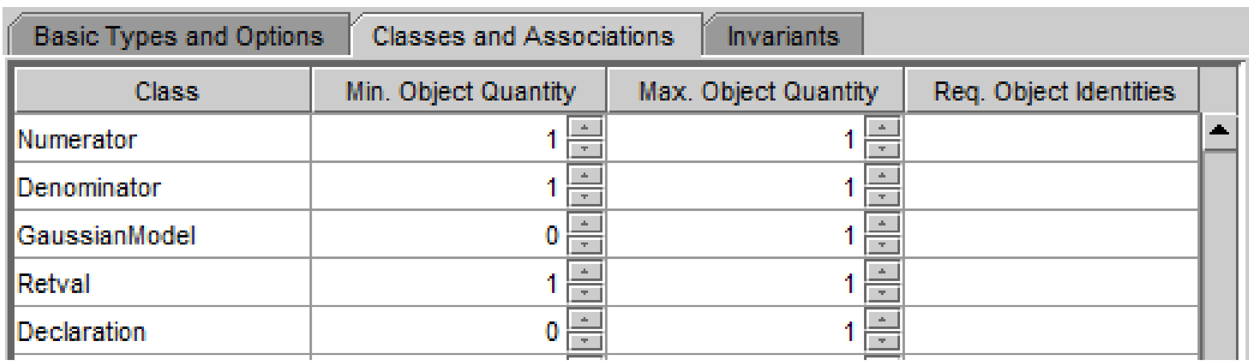

Figure A.17. The USE Model Validator Configuration - Classes and Associations tab with fix for the GaussianModel Class error.

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 autogenerated 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.