ANALYSIS OF QUANTITY SPACE HIERARCHY IN CC

By

SHASHI V. KOWDLE

Bachelor of Engineering

Bangalore University

Bangalore, India

1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1994

OKLAHOMA STATE UNIVERSITY

ANALYSIS OF QUANTITY SPACE HIERARCHY IN CC

Thesis Approved:

Blayne E. Mayfield

Thesis Advisor

[illegible]

J Chandler

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Blayne E. Mayfield for his guidance and advice throughout my research work. Without his guidance completion of this thesis would have been difficult.

My special thanks goes to my mother Kamala, father Viswamurthy, brother Ashok and Sujaya for their encouragement and support throughout my graduate studies.

I would also like to thank Drs. Paul D. Benjamin and John P. Chandler for their useful suggestions while serving on my committee. I would like to express my gratitude to Dr. Mansur H. Samadzadeh for his advice and encouragement.

Finally, I would like to thank my friends Sujatha and Raghu for their help. Last but not the least I wish to express my gratitude to my friend Abdul for his encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

Computers are used widely in science and engineering for data analysis and simulation [1]. Computer simulation is an expensive field of experimentation. It requires considerable system analysis, program development work, and long computer run times [1]. Nevertheless, application of computer simulation has grown extensively in the last two decades and covers a wide range of areas. An important factor for its extensive use is its flexibility. There are many other approaches to generate the behavior of a physical system, such as mathematical programming, analytical approach, and others. In mathematical programming problems derived from the real world have to be transformed into an idealized model with certain specific structure [1]. By using analytical approach, it is possible to construct such models without significant compromise from the true nature of actual systems. Analytical approach produces optimal solutions. The simulation approach, on the other hand, usually imposes less stringent constraints on modeling and the actual systems. One of the objectives of simulation is to develop functional relationships which use simulation to obtain insights into relationships among variables. Instead of looking for a solution to an individual numeric problem, it searches for general relationships among variables.

Computers can be exploited in a grander way. One such exploitation is to build systems capable of reasoning about the physical world, much as engineers or scientists do. This led to the model based diagnosis, which is used to develop symbolic computational methods for representing knowledge. In model based diagnosis, models have to be built for physical systems and computer simulation can be carried out on those models. In order to build models for physical systems, the following has to be considered [2]:

1. The model should express only the information known about the system.
2. The model should not require assumptions beyond what is known about the system.
3. From the model, it must be mathematically and computationally feasible to derive predictions.
4. From the model, it should be possible to match predictions against observations.

Computer simulation carried out on the physical system generates the system's behavior. The behavior describes the change in a variable's value over time. There are different types of simulations:

1. Quantitative Simulation
2. Analytical Simulation
3. Qualitative Simulation

One important representation for the model of a physical system is qualitative description of continuous variables, their directions of change, and constraints among them. The qualitative description of the variables led to the research on qualitative simulation of physical systems from the model based diagnosis [2]. Much research has been carried out on qualitative simulation.

Qualitative simulation solves certain drawbacks of quantitative and analytical simulation. In qualitative simulation the variables can be in symbolic form or linguistic form. In qualitative simulation, by first examining the physical structure, a set of differential equations (constraint equations) that describe the structural relationships among different variables of a system may be derived (variables can have values such as Tall, Very Tall, etc.). Then, the possible behaviors of the system can be predicted using the constraint equations and the initial state. QSIM is a software tool for using and exploring qualitative simulation [10].

Qualitative simulation may produce multiple solutions since qualitative reasoning schemes approximate and abstract variable values and constraint relationships. In this regard, Kuipers (Kuipers and his research group are working on qualitative simulation at University of Texas, Austin) proved that qualitative simulation may generate spurious (i.e., impossible); also, they are not suitable for all domain models. In certain physical systems it is necessary to know an individual component's behavior along with the behavior of the entire physical system (e.g., Simulation of Electrical Circuits). For simulation of such complex physical systems, it is necessary to extend Kuiper's method of qualitative simulation. Component-Connection modeling is one such extension.

In the Component-Connection modeling approach the physical system is modeled in terms of its components and their interconnections. Component-Connection modeling is very useful in certain simulations. For example, in the fault analysis of an electrical circuit, the behavior of each component must be known as well as the behavior of the entire circuit. CC is a software package that is used for the simulation of Component-Connection models [11], [12].

1.2 Problem

In the CC software package [12] quantity spaces (explained in the later chapters) are defined as a set of landmarks arranged in a partial order, where landmarks are the possible qualitative magnitudes for a variable. The CC documentation [12] by Franke and Dvorak mentions that the quantity spaces can be arranged in an inheritance hierarchy. Initially it appeared that in such hierarchies only the conservation correspondences (lists of landmarks whose sum is zero) were inherited and not the landmark lists. After running a few examples it was observed that both the conservation correspondences (explained later in Chapter II) and the landmark lists are inherited, but only a simple inheritance of landmark values is possible. Inheritance operations cannot be performed on the inherited values. Moreover, the landmark values and the conservation correspondences that are inherited override the previous landmark values and conservation correspondences of the quantity space.

1.3 Outline Of The Thesis

In this thesis a software program is developed which performs inheritance operations on the quantity spaces. Qualitative simulation can then be performed on the output obtained from the software program. Also, an analysis is carried out to obtain the advantages and disadvantages of performing the inheritance operations on the quantity space.

The software program was created in "C" programming language in Sequent Symmetry S/81 environment. The implementation was performed on QSIM and CC package which is written in common lisp. Finally, X window system was used to view the output behavioral plot obtained by QSIM.

CHAPTER II

LITERATURE REVIEW

2.1 Modeling and Simulation

Modeling primarily deals with the relationship between physical systems and creating models for the systems. The model for a physical system can be used to generate the behavioral data plot of state variable's value versus time [15] as shown in Fig. 1.

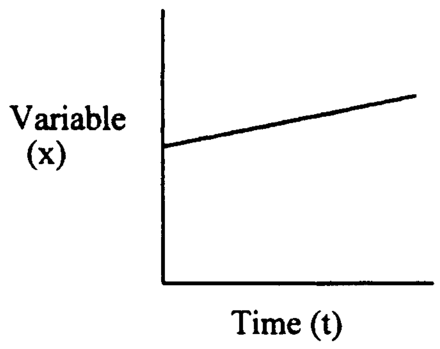


Figure 1: Simulation - Behavior of Variable V/S Time

The basic categorization of a model relates to the time base on which model events occur. A model is a *continuous time model* if time is specified to flow continuously, in that time advances smoothly through real numbers. The model is a *discrete time model* if time flows in a jump, in that time advances periodically. The continuous time models are further classified into *discrete event* and *differential equation* classes [15]. In a discrete event model, time flows continuously, but the event changes from one state to another at specific time intervals. A differential equation model is a continuous time, continuous state model in which state changes are continuous and the time derivatives (rate of change of time) are governed by differential equations. Models built for physical systems are often expressed by differential equations.

A second categorization of a model relates to the range of the model's descriptive variables [15]. The model is a *discrete state model* if its variables assume a discrete set of values. It is a *continuous state model* if their ranges can be represented by continuous numerical values. If the variables considered are qualitative in nature, the model is termed as *qualitative state model.* A qualitative state model can be a continuous time model or a discrete time model, where the variables can take on both numeric values and symbolic values. Simulation of some models can be performed using a software program which generates the behavior of the model.

2.1.1 Comparison of different simulations

There are various approaches to simulation of physical systems:

1. Quantitative Approach
2. Analytical Approach
3. Qualitative Approach

2.1.2 Quantitative Simulation

A variety of systems can be represented by a mathematical model in the form of differential equations. Reasoning of such physical systems can be performed by describing the structure of a physical system with differential equations and determining its behavior by solving the differential equations either analytically or quantitatively. Consider an example of a physical system

[7] (Fig. 2) consisting of a closed container filled with gas (at temperature T) receiving heat from a source (at temperature T_S).

A description of this system is "*the temperature of the gas increases until it is equal to the temperature of the source*".

$\Delta T = T_S - T$

inflow $= \Delta T/k$

$dT/dt =$ inflow

where: Ts = Source temperature

T = Gas temperature

ΔT = Temperature difference between source and gas

inflow = rate of flow of heat into gas

k = Cp (constant)

where: Cp = Coefficient of specific heat

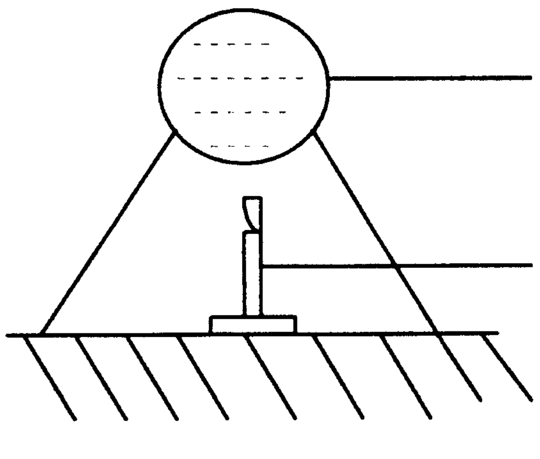


Figure 2. A Container of Gas Receiving Heat from a Source [7]. A Simple Heat-Flow System

Quantitative simulation of this physical system (Fig. 2) requires a complete description, in that the values of each parameter at each point of time considered must be given as a numerical value. That is, in the above example the values of Ts and T must be measured in the experiment in order to perform the quantitative simulation. If these values are not measured the value of ΔT cannot be calculated. The relationship between ΔT and *inflow* must also be specified precisely. The output of the quantitative simulation requires further interpretation to recognize and classify important events in the system's behavior (e.g. at what temperature the container breaks down etc.). The fundamental problem with this kind of simulation is the values of the parameters must be known. The simulation cannot run with incomplete knowledge of the system parameters. The behavioral description produced by quantitative simulation of a simple heat-flow system [7] is as shown in fig. 3:

t	0	1	2	3
T	300	370	433	490
Ts	1000	1000	1000	1000
Δt	700	630	567	510

Figure 3. Behavioral Description of Numerical Simulation [7]
(the value of constant k in the description is 10)

2.1.3 Analytical Simulation

In the case of analytical solution of a differential equation, the relationship between variables must be specified explicitly; then it can be solved analytically. Quantities can be represented as symbols instead of real numbers and a symbolic vocabulary of relationships can be asserted between quantities. In spite of this descriptive power, analytical solution of differential equations require global and knowledge-intensive operations such as indefinite integration.

The behavioral description produced by the analytical approach of simulation is shown in Fig. 4.

$$
\begin{aligned}
&d/dt\,T = \text{inflow} = k\Delta t = k\,(Ts - T) \\
&\int dT/Ts - T = \int k\,dt \\
&\ln(Ts - T) = \int kt + C \\
&Ts - T = C'\,e^{-kt} \\
&T = Ts - C'\,e^{-kt}
\end{aligned}
$$

Figure 4: Behavioral Description of Analytical Simulation [7]

Quantitative simulation treats quantities as real numbers but requires sophisticated interpretation to understand the structure of the system and derive the differential equations that describe the behavior of the systems. Analytical solution treats parameters as real-valued continuous function and yields an interpretable solution, but it requires sophisticated mathematical techniques to solve the differential equations. In quantitative simulation all the numerical values of the parameters must be known: only then can the behavior of the system be determined. However, in analytical simulation, even if the numerical values of some parameters are not present, the differential equations can be solved and the simulation can be carried out. Also in analytical simulation the variables can be symbolic (declarations such as Tall, Very Tall can be given to the variables), but in quantitative simulation only numeric values can be given.

2.1.4 Qualitative Simulation

The fundamental problem faced by quantitative simulation and solved by analytical simulation is that the variables in analytical simulation need not have a numeric value as in quantitative simulation; they can also be symbolic. However, analytical simulation requires immense computational and mathematical skills (as observed in the example in Fig. 4). Qualitative reasoning about physical systems is capable of using incomplete knowledge, such as a weakly specified functional relationship, or non numerical initial parameter values, and they do not require extensive mathematical skills.

Similar to the analytical approach qualitative reasoning of physical system operates on symbolic descriptions of real numbers and relations between them. But, the qualitative structure and behavioral descriptions offer simplicity of mechanism, in that the qualitative simulation process depends on the ability of the user to create and match simple assertions, rather than on arithmetic operations or symbolic integration. In qualitative simulation, the values of the variables and functional relationships among variables are constrained to lie in the qualitatively specified classes. The behavioral description of the simple heat-flow system (Fig. 2) produced by qualitative simulation is shown in Fig. 5.

Fig. 5 shows the behavior of the system when T, ΔT, and inflow vary according to the constraints specified until they reach their limits, i.e., T increases until it is equal to Ts. It also shows the behavior of the mechanism when the initial condition is T = Ts, $\Delta T = 0$, and inflow = 0; i.e., the system is steady.

2.1.5 Qualitative Modeling and Simulation

Qualitative reasoning methods provide more expressive power for states of incomplete knowledge than ordinary differential equations [2]. Research in reasoning with qualitative models has been motivated by such concerns as

```
1. initial condition:
       constant(Ts)
       T < Ts
   constraints (relationships among variables):
               ΔT > 0
               inflow > 0
               increasing (ΔT)
               decreasing (ΔT)
               decreasing (inflow)
2. initial condition:
       T = Ts
       ΔT = 0
       inflow = 0
   Constraints:
       steady (T)
       steady (ΔT)
       steady (inflow)
```

Figure 5: Qualitative Behavior of A Simple Heat-Flow System [7]

providing programs with the ability to reason about the physical world in the face of incomplete knowledge [3,4].

Model-building (necessary for building the qualitative models) starts with the description of a physical situation and builds an appropriate simplified model (as a qualitative differential equation described in the next section) [5]. Generally, the linear path through the model building is as shown in Fig. 6.

The various stages in building a model and predicting the behavior of a physical system can be described as

1. Model Selection:

   From a physical scenario of a system the elements that are not related to the specific qualitative behavior considered, are abstracted. This depends on the type of model that is being constructed.

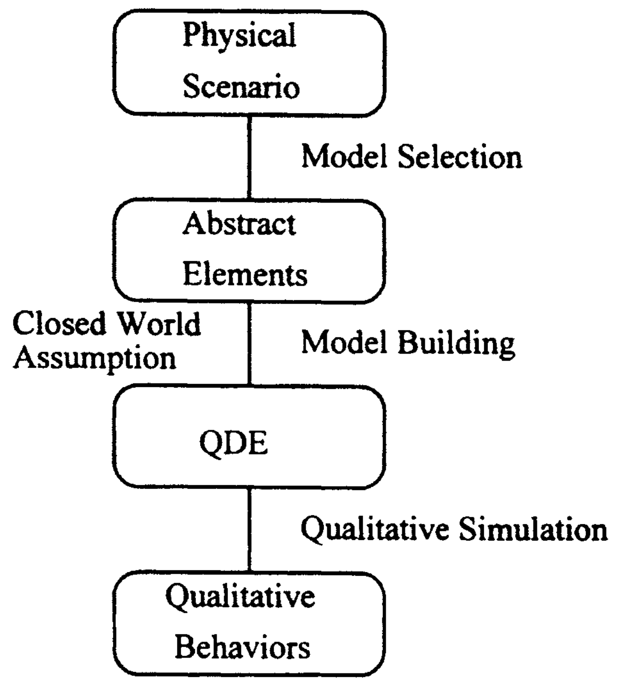


Figure 6. Steps Involved in Modeling and Simulation of a Physical System

2. Model Building:

   Using the closed world assumption the differential equation for the physical system can be obtained. CC and QPC are two of the software packages available for generating qualitative differential equations to describe the physical system. In turn this is used by QSIM to perform the qualitative simulation.

3. Qualitative Simulation:

   The qualitative simulation of the system produces its qualitative behaviors. QSIM is a software tool for performing qualitative simulation.

Three modeling ontologies are:

1. Device oriented approach (component-connection):
   The device oriented approach models a physical system in terms of its components and their interconnections. This approach of modeling models the physical systems directly.
2. Process-centered approach:
   The process-centered approach models a physical system in terms of a set of processes that govern the dynamic behavior of the system. This approach of modeling does not directly model the physical system, but describes it in terms of processes.
3. Constraint-based approach: The constraint-based approach models interactions among quantities that describes the behavior of the system. Interactions are described in terms of the qualitative constraint equations. This approach does not directly model the physical system.

Regardless of which approach is being used, *"system behavior"* is derived from the structure of the model using qualitative simulation [4]. *System behavior* is defined as changes in the system's state over a time interval. Qualitative models incorporate qualitative functional relations to specify constraints among quantities.

Using these modeling ontologies, the differential equation model for the physical system can be built. Once models (which are QDE i.e., qualitative differential equations) are obtained, qualitative simulation can be performed using QSIM. The differential equation model for QSIM can be given directly as QSIM input, or it can be obtained either from CC which produces a QDE model from a given set of components and connections or QPC which produces a QDE model from a given set of processes. CC and QPC are two compilers available for obtaining QDE models.

1. CC provides the necessary QDE structure suitable for QSIM input. It takes as input a physical system described in terms of explicit connections among instances of components defined in the component library.
2. QPC builds QDE models after the qualitative process theory by identifying sets of active views and processes in a view library and applying the closed world assumption to transform influences into constraints.

These approaches to model-building differ in the nature of the knowledge supplied by the modeler and in the way closed world assumption is applied. Specifically, when describing a device with a CC model, the modeler asserts that all relevant interactions between the components take place via explicit connections. In QPC, the system is responsible for determining the set of relevant interactions and deciding when to apply Closed World Assumption [5]. Given a qualitative model expressed as parameters, quantity spaces, and constraints, QSIM generates possible behaviors of the model.

Qualitative simulation incorporates qualitative representations of quantity values. Qualitative simulation starts with a set of qualitative constraints (a set of differential equations) and an initial state, and predicts the set of possible behaviors of the system. It applies in situations where knowledge of the system is imprecise or incomplete. Qualitative simulation requires two things:

1. Representations of the qualitative structure and behavior of the mechanism.
2. Algorithms for deriving behavior from structure and initial condition.

The structure of a mechanism is described in terms of sets of parameters and constraints among them. In a QDE the set of parameters are expressed as *quantity spaces* (as described below). The structural and behavioral representation for qualitative simulation could be shown as an abstraction of ordinary differential equations (ODE) and their solutions. A QDE is described as a set of ordinary differential equations with two essential abstractions.

First, *quantity space* is an abstraction of the real number line to an ordered set of *landmark values*. It is a collection of numbers which forms a partial order when compared to the real number line. *Quantity space* provides a means to partition numerical values and thus express boundary conditions for the behavior. A *landmark value* is a qualitatively interesting point in the range of variables. There are two distinct semantics for landmark values. They are temporally generic if they refer to a behavior in general and temporally specific if they refer to a single behavior [9]. Examples of temporally generic and temporally specific landmarks are: the script of a drama is temporally generic, because it describes the entire scenario of the drama and not any particular scene, and a specific scene in the drama is temporally specific, as it narrates the happenings in a particular scene.

Second, the arithmetic and differential constraints in the ordinary differential equations are augmented by a monotonic function constraint which describes a fixed but unknown function in terms of its direction of change
An example, of a monotonic function constraint :

1. M+ X Y   i.e., As X increases Y also increases.
2. M- X Y   i.e., As X decreases Y also decreases.

2.2 QSIM

QSIM provides the representations and algorithms necessary for deriving the behavior of a system from its structure and an initial state. QSIM uses the device oriented approach for modeling the physical system. QSIM takes as input a QDE and the description of its state, at time t_0 and then predicts a set of possible behaviors, which is interpreted as [2], [7]

QSIM: QDE, Qstate(t_0) -> or(QBeh$_1$ --- QBeh$_n$)

That is, starting in state (t_0), QSIM predicts that one of the behaviors QBeh$_1$...QBeh$_n$ will describe the actual behavior of the system.

The different tasks that can be performed by QSIM are[10]:

1. QSIM allows the user to define the structure of a mechanism as one or more QDE. The constraints necessary for determining the behavior of the system are defined here.
2. The user can specify the initial condition or state for the simulation.
3. QSIM generates the behavior tree of states from the initial state.
4. The user can use the behavior tools to explore these behaviors.

A QSIM model is expressed as a QDE. A QDE specifies the structure of a mechanism with a set of variables (continuously differentiable), quantity space (qualitative abstraction of a variable), and constraints. A QSIM variable takes on a qualitative value which includes a qualitative magnitude and a change of direction *qdir*, which can be increasing (inc), decreasing (dec), steady (std) or ignore (ign). When qdir is ignore it indicates that change in direction is unknown. In that case, simulation is performed with all possible change of directions.

2.2.1 Definitions and Terms in QSIM

The QDE is first created in QSIM, which generates a structure with the following specifications:

Quantity space: Defines the set of parameters used to describe the structure of the physical system.

Constraints: Consists of a set of parameters and a set of axioms stating relationships between the variables and derivatives of the parameters.

Layout: Specifies the format in which the output of the simulation is displayed.

The overall simulation is performed by using a built-in function. It performs the simulation from the initial state specified. The behavior tree is displayed from the initial state specified using a built-in function.

Given a model, QSIM generates the system's possible behaviors. Each behavior is represented as a sequence of states where each state describes the qualitative values of all the variables. Graphically, a behavior is shown as a collection of plots, one for each quantity in the model [11]. The plot depicts how a variable changes its value qualitatively from one time point to another. The value of a number or magnitude is described in terms of its quantity space. A quantity space is a collection of numbers which form a partial order. A quantity space of a number one point to the following interval and to the next time point and so on.

2.3 CC

CC is a software tool for compiling the Component-Connection model of a device into variables and constraints which are necessary to build a QSIM

QDE [11,12]. The output obtained from CC (a QDE model) is used as an input to QSIM to obtain the qualitative behavior of the physical system. CC uses the device oriented ontology to build the differential equation models.

1. The variables from the component instances provide the variables of the QDE.
2. The constraints from component instances provide the constraints of the QDE.
3. The connections imply additional constraints.

Two important concepts of model definition in CC are *component definition* and *component abstraction* [12]. Component definition permits components to be defined in terms of components and connections between these components; it also describes the relationships among QSIM constraints and other components. Component definition expressed in terms of other components is called *composed component definition,* and component definition expressed only in terms of QSIM constraints and variables is called *primitive component definition.*

Component abstraction defines the internal structure of the component. Component abstraction has two parts; *component-interface* and *component-implementation.*

Component-interface defines a component type (example: electrical, hydraulic, etc.) and an interface for that type. An interface description contains the terminals (input and output ports of a component) and parameters of the components. The terminals and parameters are used in both primitive and composed implementations.

Component-implementation provides the decomposition of components into simpler (sub) components. It also specifies the variables used by the components and the constraints among these variables. Component-implementation defined in terms of the QSIM constraint equations is called *primitive component implementation.* In primitive component implementation variables are declared either as terminal variables (variables associated with the terminals of a system) or component variables (additional variables to specify the behavior of a system).

On the other hand, *composed component implementation* is defined as connections among sets of primitive and composed components. The composed component implementation [11,12] describes a model of a physical system as being constructed from a set of components (variables with their constraints) and connections (which specifies relationship among terminals) among the terminals of these components.

It is possible to predict the behaviors of a system by compiling a CC input to QSIM QDE and simulating the behavior of QDE with QSIM.

During model building, the structures of component-interface, component-implementation and component-configuration are accessed. The hierarchy of component definitions is processed until a primitive implementation is reached. At this point model variables (list of variables in the current QDE) are generated for the component variables in the definition, and the list of constraints is added to the global list of constraints. When variable declarations are encountered, the quantity space for the variable is determined from any explicit quantity space in the variable declaration and default quantity spaces (if there are any such declarations) specified in the component implementation, component interface. The variables of the model and the constraints are passed on to the QDE that is built.

CHAPTER III

SOFTWARE AND IMPLEMENTATION

3.1 Background of the Research Problem

3.1.1 Inheritance and CC

When component models are constructed in CC, many variables in the model have identical or related quantity spaces. The landmark values and conservation correspondences of identical quantity spaces can be defined independently from the components in CC. These landmark values and conservation correspondences can be inherited by other quantity spaces, and various inheritance operations can be performed on them. The various inheritance operations are: generalization, specialization, and multiple inheritance.

Consider, for example, voltage variables in the component definitions of electronic components [12]. For a specific model, the quantity space for a voltage source can be represented as (0 Vhi), where Vhi represents some positive voltage. A resistor component defines voltage variables as the voltage difference between one terminal and the other. This voltage variable can range from (Vhi- .. Vhi), and its quantity space can be represented as (Vhi- 0 Vhi), where Vhi- = -Vhi. For a transistor component, an additional threshold voltage is required. The quantity space of the transistor can be represented as (-Vhi 0 Vth Vhi), where Vth is the threshold voltage. In the example, the quantity space of the electronic components (source, resistor and transistor) can be arranged in an inheritance hierarchy as shown in Fig. 7.

$$\begin{tikzcd}
\text{(0 Vhi)} \arrow[d, dash] & \text{(Voltage Source)} \\
\text{(-Vhi 0 Vhi)} \arrow[d, dash] & \text{(Resistor)} \\
\text{(-Vhi 0 Vth Vhi)} & \text{(Transistor)}
\end{tikzcd}$$

Figure 7. CC Quantity Space Hierarchy 1 [12]

The hierarchy above exhibits specialization operation on the quantity space of the electronic components. In specialization, a new quantity space is made to inherit the landmark values and conservation correspondences of an existing one. Then the new landmark values are added to the quantity space.

In the above hierarchy, the quantity space of the resistor can be obtained by inheriting the landmark values of the quantity space of the voltage source and adding the landmark value Vhi- to it. Since the quantity space of the voltage source does not contain any conservation correspondences, only the landmark values are inherited. Similarly, the quantity space of the transistor is obtained by first inheriting the landmark values and the conservation correspondences of the quantity space of the resistor and then adding the landmark value Vth to it. In this example the conservation correspondences are not added to the quantity space; only the landmark values are added to the quantity space.

In the hierarchy observed in Fig. 7, the quantity space of the resistor is modified in such a way that any voltage source with the same quantity space as the resistor can no longer exist; however, there can be some resistors with the same quantity space as the voltage source. Similarly, in the second and third level of the hierarchy there can be some transistors with the same quantity space as the resistor, but there cannot be any resistors with the same quantity space as the transistor. Also, in the hierarchy in the example above, as the level of hierarchy increases, the number of electronic components that can occur with the same quantity space as the element is restricted. That is, in the first level there can be some voltage sources or resistors or transistors with the same range of quantity space 0 Vhi (source quantity space). In the second level, only some resistors or transistors can exist with the same range of quantity space Vhi- 0 Vhi (resistor quantity space). In the third level only transistors can have the same range of the quantity space Vhi- 0 Vth Vhi (transistor quantity space).

Figure 8 shows another inheritance hierarchy of the quantity space of the

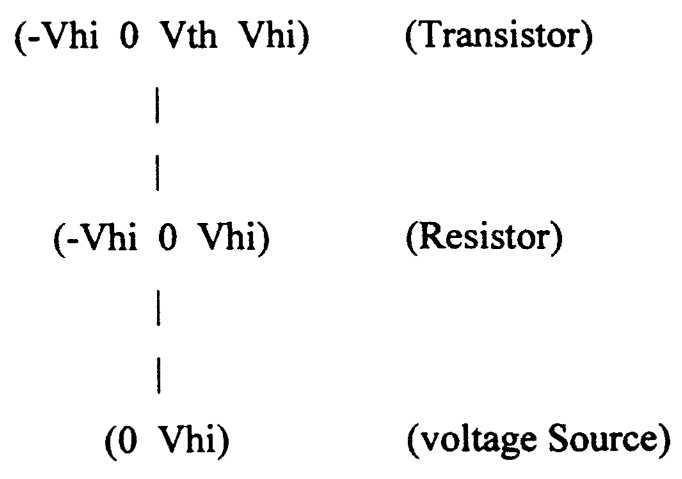
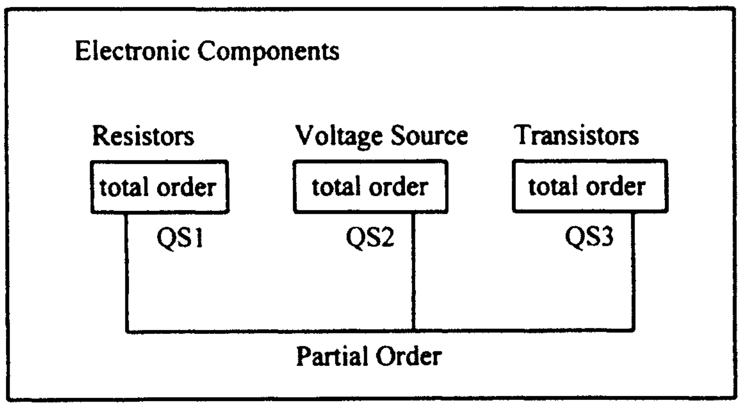


Figure 8. CC Quantity Space Hierarchy 2 [12]

electronic components. The hierarchy in Fig. 8 exhibits the generalization operation on quantity spaces. Generalization is exactly opposite to specialization.

Quantity spaces in CC and QSIM have two characteristics: conservation correspondences and landmark lists. Conservation correspondences are two or more landmark values enclosed in parentheses, whose algebraic sum is made to be zero. The landmark values are the totally ordered qualitatively interesting points enclosed in parentheses. If the landmark values of all electronic components are considered, the landmark values of a specific quantity space are partially ordered. Figure 9 shows the ordering of landmark values.



QS1 = Quantity Space of Resistor
QS2 = Quantity Space of Source
QS3 = Quantity Space of Transistor

Figure 9. Ordering of Landmark Values

In the example of electronic components considered in Appendix A (refer to appendix B also) the voltages (Vhi- Vhi) is a conservation correspondence in the quantity space of the resistor element. It was initially observed that during inheritance in CC only the landmark lists were inherited and the conservation correspondences were not inherited. But after running some examples (refer

Appendices A and B) it was observed that both the conservation correspondences and the landmark lists are inherited. However, the landmark value and conservation correspondences inherited override the earlier landmark values and conservation correspondences of the quantity space. The output obtained by the simulation is shown in the Fig. 10.

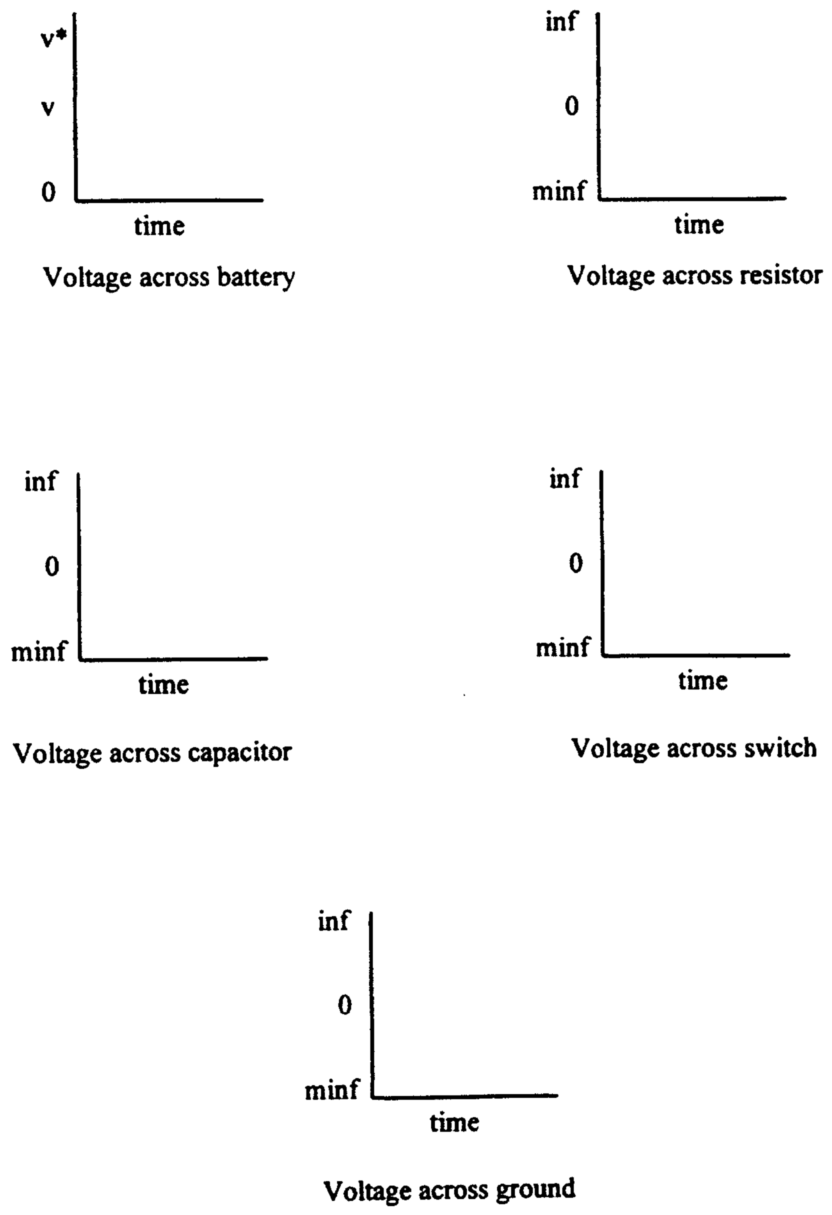


Figure 10. Qualitative Simulation of the Example in Appendix A

Initially the quantity space of the battery is (minf 0 inf). After inheriting the landmark values and conservation correspondences of the quantity space battery-volt (-V* 0 V*), the quantity space of the battery becomes (-V* 0 V*). This shows that the inherited landmark values and conservation correspondences override the earlier landmark values and conservation correspondences of the battery (minf 0 inf). If the new landmark values and the conservation correspondences do not override the old ones, the quantity space of the battery after inheritance must be the combination of the quantity spaces (minf 0 inf) and (-V* 0 V*). Figure 11 shows the quantity space of battery voltage before and after inheritance.

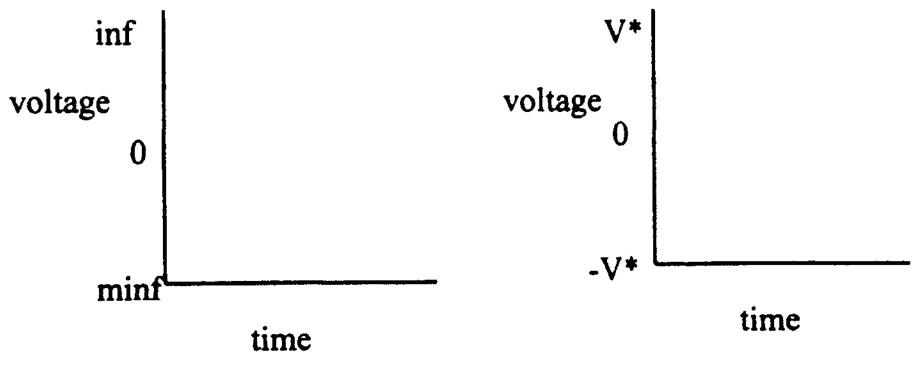


Figure 11. Voltage Across Battery Before Inheritance and Voltage Across it After Inheritance

3.2 Software Methodology

Inheritance represents a taxonomic hierarchy or **is-a** relationship. In CC the quantity space of the electronic or mechanical or hydraulic components can be arranged in an inheritance hierarchy. In order to arrange quantity spaces in a hierarchy the following inheritance operations are necessary.

3.2.1 Specialization

Assume that the quantity space of a voltage source is (0 Vhi). Consider a resistor element with the quantity space (Vhi- 0 Vhi). The quantity space of the resistor element can be obtained by inheriting the landmark values (0 Vhi) of the source and adding the landmark value Vhi- to it. After adding the new landmark value, the landmark values must be arranged in total order. Then the quantity space of the resistor becomes (Vhi- 0 Vhi). Adding a landmark value to the inherited landmark value is an inheritance operation.

3.2.2 Generalization

Assume that the quantity space of a resistor element is (Vhi- 0 Vhi). Consider a voltage source with the quantity space (0 Vhi). This quantity space can be obtained by inheriting the landmark values and conservation correspondences of the resistor element and deleting the landmark value Vhi- from it. Then the quantity space of the source becomes (0 Vhi). Here the conservation correspondence is also specialized. The resistor has the conservation correspondence (Vhi- Vhi); however, the inheritance operation the voltage source will not have any conservation correspondence. Deleting a landmark value from the inherited landmark values is an inheritance operation.

3.2.3 Multiple Inheritance

Multiple inheritance of quantity space is an inheritance operation where landmark values and conservation correspondences are inherited from two or more quantity spaces. Then the landmark values are arranged in total order. Assume the quantity space of two voltage sources are (0 Vhi) and (Vhi- 0), respectively. If a third voltage source with quantity space (Vhi- 0 Vhi) exists, its quantity space can be obtained by inheriting the landmark values and conservation correspondences from both the voltage sources (0 Vhi and Vhi- 0) and arranging them in total order.

3.3 Software Implementation

There are different approaches for implementing the inheritance operations on the quantity spaces. One such approach is to make modifications in the CC code to implement the inheritance operations. Another approach is to develop a preprocessor that takes a CC input file, with the inheritance operations to be performed on the quantity spaces, and generates an output CC file which contains the inheritance operations being performed. The focus of this thesis is the implementation of this sort of preprocessor and determining the advantages and disadvantages of implementing inheritance operations on the quantity spaces.

The function performed by the preprocessor is as shown in Fig. 12.

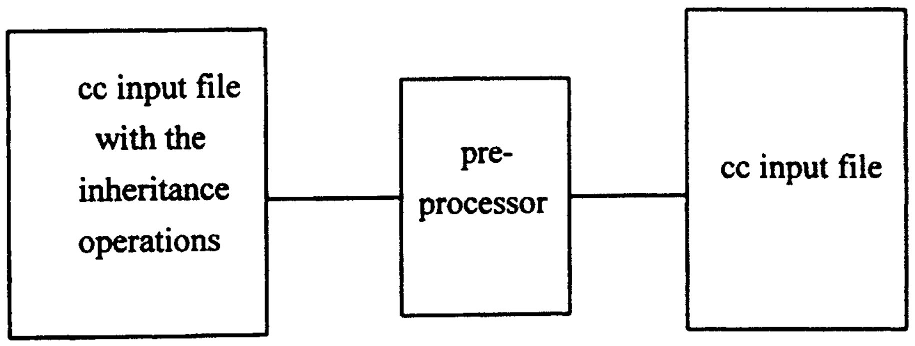


Figure 12. Function Performed by the Preprocessor

The code for the preprocessor is given in appendix C. The main functions performed by the preprocessor are:

3.3.1 Add a new landmark value

In the input file the syntax for adding a new landmark value is

(add new-landmark-value quantity-space variable new-quantity-space)

where the definition of the syntax is :

add	----	Function name to add a new landmark value
new-landmark-value	----	New landmark value to be added to the quantity space
quantity-space	----	Quantity space to which the new landmark value has to be added
variable	----	An electrical or mechanical or hydraulic variable
new-quantity-space	----	Name of the new quantity space to be created

Consider the following example which adds a new landmark value to a quantity space:

```
(define-quantity-space vs1 (0 v* inf))
(R resistor (add v1 vs1 voltage new-volt))
```

In the above example the quantity space of the voltage variable vs1 is (0 v* inf). A new voltage v1 has to be added to the quantity space vs1. When such an input is fed to the preprocessor, the output generated by the preprocessor is as follows:

```
The quantity space to which the new landmark value is to be added is
                  (0 v* inf)
Give the position of the new landmark value:
```

If the user specifies a position below 1, the new landmark value is added at the first position. If the user specifies a position above 3, the new landmark value is incorporated in the end. If the position of the new landmark value specified by the user is 2, the preprocessor creates a new quantity space new-volt with the landmark values (0 v1 v* inf). The quantity space of the resistor is made to inherit this new quantity space with the landmark values (0 v1 v* inf). Since the user specifies the position of the new landmark value, the landmark values will already be arranged in total order. The user could also have been given the flexibility of specifying the two landmark values between which the new landmark value has to be incorporated. A user-friendly environment is created by not giving this flexibility to the user. Also, if the user specifies the position of the new landmark value, the possibilities of the user giving erroneous values is greater.

3.3.2 Delete a Landmark Value

In the input file the syntax for deleting a landmark value is

(delete landmark-value quantity-space variable new-quantity-space)

where the definition of the syntax is :

delete	----	Function name to delete a landmark value
landmark-value	----	Landmark value to be deleted from the quantity space
quantity-space	----	Quantity space from which the landmark value has to be deleted
Variable	----	An electrical or mechanical or hydraulic variable
new-quantity-space	----	Name of the new quantity space to be created

Consider the following example which deletes a landmark value from a quantity space:

```
(define-quantity-space vs1 (0 v1 v* inf))
(R resistor (delete v1 vs1 voltage new-volt))
```

In the above example, the quantity space of the voltage variable vs1 is (0 v1 v* inf). The landmark value v1 has to be deleted from the quantity space vs1 so that the quantity space of the resistor R after inheritance operation is (0 v* inf). When such an input is fed to the preprocessor, a new quantity space new-volt with the landmark values (0 v* inf) is created. The quantity space of the resistor R is made to inherit the new quantity space new-volt.

3.3.3 Combine Landmark Values of Two Quantity Spaces

In the input file the syntax for combining two quantity spaces is

(combine quantity-space1 quantity-space2 variable new-quantity-space)

combine	----	Function name to combine two quantity spaces
quantity-space1	----	Name of the quantity space which has to combined with the other
quantity-space2	----	Name of the quantity space which has to combined with the other
variable	----	An electrical or mechanical or hydraulic variable
new-quantity-space	----	Name of the new quantity space to be created

There are different cases that can occur when combining landmark values and conservation correspondences of two quantity spaces.

The first case is inheriting from two quantity spaces that do not have any common landmark values between them. An example is:

```
(define-quantity-space vs1 (0 v* inf))
(define-quantity-space vs2 (v2 v1*))
(R resistor (combine vs1 vs2 voltage new-volt)
```

In the above example, quantity space for the voltage variable vs1 is (0 v* inf) and that of vs2 is (v2 v1*). These two quantity spaces do not have any landmark values in common. When such an input is fed to the preprocessor, the output generated by the preprocessor is as follows:

```
The two quantity spaces to be combined are
                    (0  v*  inf)
                    (v2  v1)
              M E N U
1. User sets the partial ordering
2. Preprocessor sets the partial ordering
Select your choice:
```

If the user sets the partial ordering the following output is generated by the preprocessor:

```
Give the ordering among the quantity spaces
      (0  v1  v*  inf)
      (v2  v1)
```

Once the user sets the order, a new quantity space new-volt is created with the ordering of landmark values specified by the user. If the ordering of landmark values set by the user is (0 v* v2 v1 inf), a new quantity space new-volt with the landmark value specified is created. The quantity space of the resistor is made to inherit this quantity space. If the preprocessor sets the order the following output is generated by the preprocessor:

```
The quantity spaces to be combined are
                    (0  v*  inf)
                    (v2  v1)
The ordering set by the preprocessor is (v2  v1  0  v*  inf)
Do you want to change the ordering among the quantity spaces:
```

If the user opts to change the ordering, the ordering of landmark values is queried and a new quantity space new-volt is created with the landmark ordering specified by the user. Otherwise, a new quantity space new-volt is created with the landmark ordering specified by the preprocessor.

The second case is inheriting from two quantity spaces that have some common landmark values. An example is:

```
(define-quantity-space vs1 (0 v1 v* inf))
(define-quantity-space vs2 (v2 v1 v1* inf))
(R resistor (combine vs1 vs2 voltage)
```

In the above example, quantity space for the voltage variables vs1 and vs2 are (0 v1 v* inf) and (v2 v1 v1* inf), respectively. These two quantity spaces have landmark values v1 and inf in common between them. When such an input is fed to the preprocessor, the output generated by the preprocessor is similar to the output generated when landmark values are inherited from two quantity spaces, which do not have any common landmark values between them.

However, here if the user sets the ordering, the user is queried to set the ordering of landmark values between 0 and v2 and then the preprocessor adds the common landmark value v1. Again the user is queried to set the total order between the landmark values v1* and v*. Then the preprocessor adds the common landmark value inf in the end. Finally if the order set by the user between 0 and v2 is (v2 0), and between v* and v1* is (v* v1*), the landmark values of the new quantity space created is (v2 0 v1 v* v1* inf). If the preprocessor sets the ordering the following action takes place:

The ordering set by the preprocessor is

(0 v2 v1 v* v1* inf)

The user can again change the ordering of landmark values if necessary. The quantity space of the resistor R is made to inherit the quantity space new-volt.

3.4 Results and Discussion

The example shown in appendix D was run on the package and the following results were observed:

In the example a resistor-capacitor circuit is considered (appendix B). In the circuit there are two voltage quantity spaces vs1 (v1 V v*) and vs2 (-v* V v2). The battery voltage is considered as a combination of the quantity spaces vs1 and vs2. A new quantity space newbvolt is created by combining the landmark values and conservation correspondences of vs1 and vs2. At the output file generated by the preprocessor, a new quantity space newbvolt, with the landmark values (-v* v1 V v2 v*) (this is the order set by either the user or the preprocessor) is created . If it is assumed that the capacitor charges only on the positive side of the battery voltage, its quantity space can be obtained by inheriting the battery voltage and deleting the landmark value -v* from it. The voltage across the capacitor then becomes (v1 V v2 v*). The voltage across the resistor will be similar to the battery voltage (assuming negligible resistance). The resistor voltage can be directly obtained by inheriting the battery voltage. In order to show the adding operation, the capacitor voltage is inherited and the landmark value -v* is added to the inherited quantity space. The voltage across the resistor then becomes (-v* v1 V v2 v*). The following example demonstrates the multiple inheritance, specialization and generalization operations performed by the preprocessor. The output obtained by simulating the example with the default quantity space (minf 0 inf) across the battery, resistor and capacitor is shown in Fig. 13.

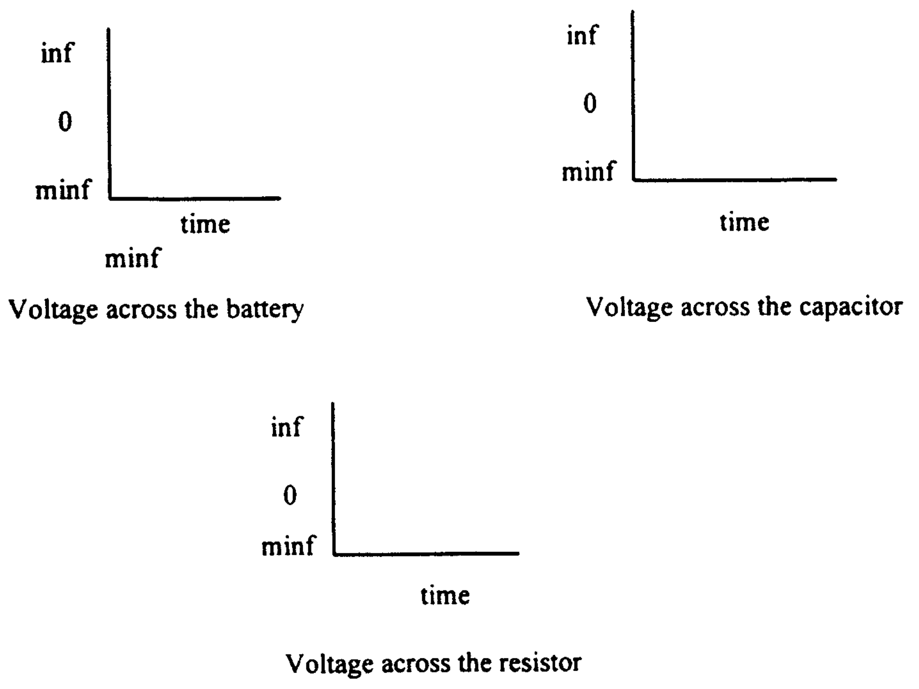


Figure 13. Qualitative Simulation of Example Shown in Appendix D with Default Quantity Space Values Across the Components.

The output obtained by simulating the same electrical circuit with the inheritance operations mentioned in the example given in appendix D is shown in Fig. 14.

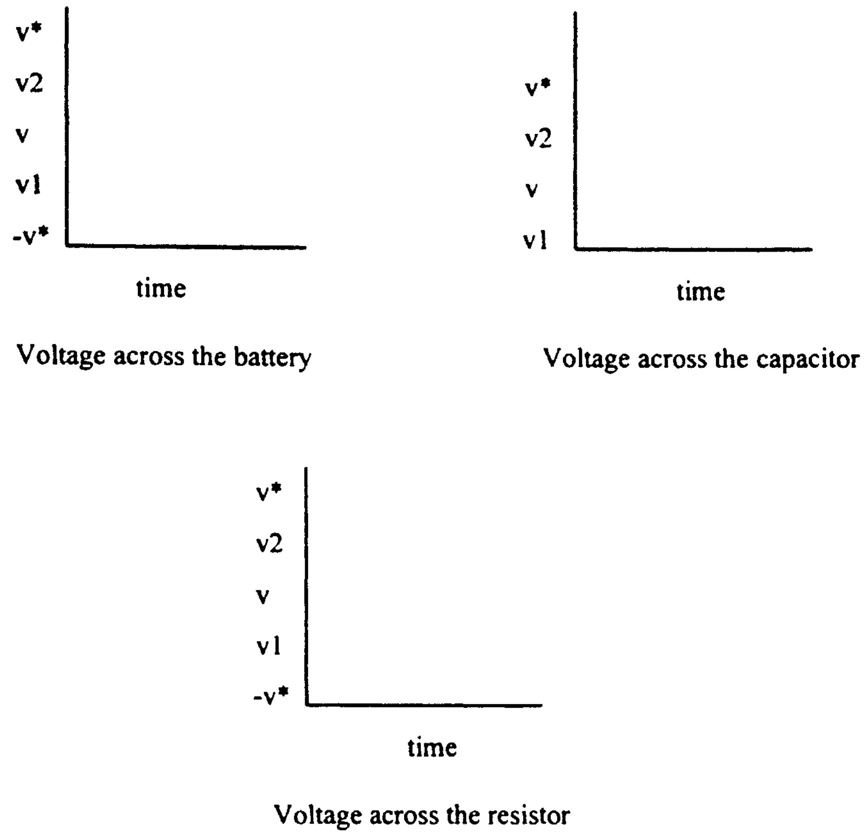


Figure 14. Qualitative Simulation of Example Shown in Appendix D with Inheritance Operations.

3.5 Advantages and Disadvantages of Inheritance

Some of the Advantages of Inheritance are:

Reusability is the ability of software products to be reused. In CC there are many variables that have common quantity spaces. Instead of declaring the same quantity spaces repeatedly, they can be inherited from the declared quantity spaces. Since the code will be reused, the reliability increases (the likelihood of discovering the errors will be greater), and therefore the maintenance cost is reduced.

In CC some quantity spaces are similar to the declared quantity spaces with a few added or deleted landmark values. In such cases, instead of declaring a new quantity space, the landmark values and conservation correspondences are inherited from another quantity space. Thus extendibility of software is possible.

Since the identical quantity spaces can be constructed by inheriting the landmark values and conservation correspondences of the quantity spaces that are already defined, development time can be spent on understanding the portion of the CC package that is new or unusual.

Some of the Disadvantages of Inheritance are:

1. One of the disadvantages is the overhead of software code.
2. The compilation of an external program increases the total compilation time and thereby decreases the execution speed.
3. Since the input file to the preprocessor requires the user to specify some inputs, the output generated depends on the accuracy of the input provided.
4. In qualitative simulation of the output file, the number of possible states produced increases as the number landmark values of the quantity spaces increase. Because of this reason the run time also increases.

3.6 Advantages of the Package

1. It provides the various inheritance operations: generalization, specialization, and multiple inheritance.
2. The user need not specify the ordering among landmark values during multiple inheritance. The user is given the flexibility of changing the ordering of the landmark values if necessary.
3. The program complexity is reduced. During multiple inheritance the user

need not specify all details of the new quantity space.

4. The user need not know the details of creating a new quantity space if it is identical to a quantity space already existing; only the differences in the landmark values or conservation correspondences between the already existing quantity space and the new quantity space must be specified.

3.7 Limitations of the Package

1. When adding a new landmark value, the user has to specify the position at which it has to be incorporated. If the user specifies an incorrect position, the simulation result will also be erroneous.
2. When combining two quantity spaces in multiple inheritance the user can specify the ordering of landmark values. Since input is specified by the user, if the partial order is specified incorrectly, the output produced will vary.
3. Only one landmark value can be added or deleted. Also, only two quantity spaces can be combined in multiple inheritance.

CHAPTER IV

CONCLUSIONS

Systems are not born into an empty world. Almost always, new software expands on previous developments. Inheritance provides the extendibility of software. In the CC package, inheritance of quantity spaces is possible. This thesis focuses on an approach to provide inheritance operations on the quantity spaces in CC and determining the advantages and disadvantages of performing inheritance operations on the quantity spaces.

The software package, developed in C, acts as a preprocessor. An input file similar to a CC input file is fed to the preprocessor. The input file contains the specialization operations necessary for the CC input. The preprocessor performs these inheritance operations and writes the result to an output file. This file is then fed as an input file to CC. The different inheritance operations performed are: addition of a new landmark value, deletion of a landmark value and combination of two quantity spaces (multiple inheritance). In addition to the inheritance operations, the advantages and disadvantages of performing such operation is analyzed.

4.1 Future Work

In CC the concept of inheritance is a relatively new field. Although much research is done on CC and QSIM, not much research has been done in the field of inheritance of quantity space. This package provides the basic inheritance operations on the landmark values and conservation correspondences of quantity spaces. The inheritance operations are limited to adding or deleting only one landmark value at a time. The preprocessor can be developed to add or delete more than one landmark value at a time. Multiple inheritance also can be carried on more than two quantity spaces. As the preprocessor uses object oriented programming concepts, it can be implemented using "C++" instead of "C".

Instead of using a preprocessor, the inheritance of quantity spaces can be incorporated into the CC package itself. This reduces the software overhead and also decreases the compilation time.

BIBLIOGRAPHY

1. Nabil R. Adam, Ali Dogramaci, (1979). Current Issues in Computer Simulation, Academic Press, 101-107.

2. Benjamin Kuipers, (1989). Qualitative Reasoning: Modeling and Simulation with incomplete Knowledge, Automatica, 25(4), 571-585.

3. Biswas G., Manganaris S., and Yu X., (1992). Extended Component Connection Modeling for analyzing Complex Physical Systems, Technical Report, CS-92-02, Vanderbilt University, 1-27.

4. Biswas G., Manganaris S., and Yu X., (1993). Extended Component Connection Modeling for analyzing Complex Physical Systems, IEEE Expert, 8(1), 48-57.

5. Benjamin J. Kuipers, (1993). Qualitative Models, Artificial Intelligence, 59(1-2), 125-132.

6. Paul A. Fishwick, Paul A. Luker, (1991). Qualitative Simulation Modeling and Analysis, Springer-Verlag, 51-71.

7. Kenneth D. Forbus, (1985). Commonsense Reasoning about Casuality, Qualitative Reasoning about Physical System edited by Daniel G. Bobrow, The MIT Press Cambridge, Massachesetts, 169-190.

8. Benjamin Kuipers, (1993). Qualitative Simulation now and then, Artificial Intelligence, 59(1-2), 133-140.

9. Kenneth D. Forbus, (1989). Qualitative Physics: Past Present and Future, Qualitative Reasoning Group, Department of Computer Science, University of Urbana Campaign, 242-260.

10. Farquhar A., Kuipers B., Rickel J., and Throop D., QSIM: The Program and its Use, Internal Documentation, Department of Computer Science, University of Texas at Austin, 1-128.

11. Franke D. W., and Dvorak D. L., (1993). CC: Component-Connection Models for Qualitative Simulation, A User's Guide, Internal documentation, Department of Computer Science, University of Texas at Austin, 6-47.

12. Franke D. W., and Dvorak D. L., (1991). CC: Component-Connection Models for Qualitative Simulation, A User's Guide, Internal documentation, Department of Computer Science, University of Texas at Austin, 5-32.

13. Kuipers Benjamin, (1986). Qualitative Simulation, Artificial Intelligence, 29, 289-306.

14. Kuipers Benjamin, (1992). Component-Connection Models, Draft, 1-23.

15. Bernard P. Zeigler (1976) Theory of Modeling and Simulation, A Wiley Interscience Publication, 1-51.

APPENDIX A

EXAMPLE OF INHERITANCE OF QUANTITY SPACE

EXAMPLE OF INHERITANCE OF QUANTITY SPACES

```
;;; -*- Syntax: Common-Lisp; Package: QSIM -*-
(in-package :qsim)
;;; Copyright (c) 1991, Benjamin Kuipers.
;;; Assumes that /u/kuipers/cc/lib/simple-interface.lisp already loaded.

;;; defining the quantity space
(define-quantity-space battery-voltage (0 V V* ))

;;; component definition of resistor-capicitor circuit
(define-component-interface RC "R-C" electrical
  (terminals t1 t2))

;;; component implementation of the resistor capacitor circuit
;;; in the component definition the battery initially has the default quantity
;;; space voltage i.e, (minf  0  inf).  The battery voltage is made to inherit the
;;; voltage defined at volt i.e, (0 V V*)
(define-component-implementation Basic RC
  "Resistor-Capacitor Circuit"
   (components (B battery (quantity-spaces (voltage battery-voltage)))
            (R resistor)
            (C capacitor)
            (S switch)
            (G ground))

;;; the connections among various components and the terminals
   (connections (n1 (R t1) (S t1))
               (n2 (R t2) (C t1))
               (n3 (C t2) (B t2) (G t))
               (n4 (B t1) (S t2))))


;;; Battery - one of the electronic components used in the electrical circuit
(define-component-interface battery "battery" electrical
  (terminals t1 t2))
;;; component implementation of the battery with the voltage across it
;;; (minf 0 inf)
```

```
(define-component-implementation Basic
 battery "battery"
 (terminal-variables (t1 (v1 voltage independent)
                         (i1 current))
                     (t2 (v2 voltage independent)
                         (i  current display)))
 (component-variables (voltage voltage independent display))
 (constraints ((ADD voltage v2 v1))            ; Voltage measured across
terminals
          ((MINUS i i1) (0 0) (inf minf)))     ; Battery current has opposite
sign of outflow at t1
 )

;;; Capacitor - one of the components used in the resistor-capacitor circuit
(define-component-interface capacitor "capacitor" electrical
 (terminals t1 t2))
;;; component implementation of the capacitor with the voltage across it
;;; (minf 0 inf)
(define-component-implementation Basic
 capacitor  "capacitor"
 (terminal-variables (t1 (v1 voltage)
                         (i  current display))
                     (t2 (v2 voltage)
                         (i2 current)))
 (component-variables (voltage voltage display)
                      (c capacitance independent display)
                      (q charge display))
 (constraints ((ADD voltage v2 v1))            ; Voltage measured across
terminals
          ((MULT voltage c q))                 ; Charge is product of
Voltage and Capacitance
          ((d/dt q i))               ; Current is first derivative of Charge
          ((MINUS i i2) (minf inf) (0 0) (inf minf))      ; Current at termainls
has opposite sign
          )
 )

;;; Resistor - one of the components used in the resistor-capacitor circuit
```

```
(define-component-interface resistor "resistor" electrical
  (terminals t1 t2))
;;; component implementation of the resistor with the voltage across it
;;; (minf 0 inf)
(define-component-implementation basic
  resistor  "resistor"
  (terminal-variables (t1 (v1 voltage)
                          (i  current display))
                      (t2 (v2 voltage)
                          (i2 current)))
  (component-variables (voltage voltage display)
                       (r resistance independent display))
  (constraints ((ADD voltage v2 v1))             ; Voltage measured across
terminals
           ((MULT i r voltage))              ; Ohm's Law
           ((MINUS i i2) (minf inf) (0 0) (inf minf))          ; Current at
terminals has opposite sign
           )
  )

;;; Switch - one of the components used in the resistor-capacitor circuit
(define-component-interface switch "switch" electrical
  (terminals t1 t2))
;;; component implementation of the switch
(define-component-implementation basic Switch
  "Switch:  externally opened or closed"
  (mode-variables (mode open closed))
  (terminal-variables (t1 (v1 voltage)
                         (i1 current))
                      (t2 (v2 voltage)
                         (i  current)))
  (component-variables (v voltage))
  (constraints ((add v v2 v1))
           ((minus i i1))
           ((mode open)   -> ((constant i 0)))
           ((mode closed) -> ((constant v 0)))))

;;; Ground- one of the components used in the resistor-capacitor circuit
```

```
(define-component-interface ground "ground" electrical
 (terminals t))

;;; component implementation of the ground
(define-component-implementation basic Ground
 "Ground:  constant voltage (current sink)"
 (terminal-variables (t (v voltage)
                 (i current)))
 (constraints ((constant v 0))
          ((constant i 0))))

;;; This is a function used for getting an initial state and running the
;;; simulation with the initial state.  Also the parameters are initialised
(defun test-rc()
       (declare (special rc_basic))
  (let ((init (make-initial-state rc_basic
       (translate-cc-name-alist rc_basic
                 '(((rc B v) ((0) std))
                      ((rc r v ) ((0)std))
                  )))))
        (qsim init)
        (qsim-display init)
        ))
```

APPENDIX B

RESISTOR-CAPACITOR CIRCUIT

Resistor-Capacitor Circuit

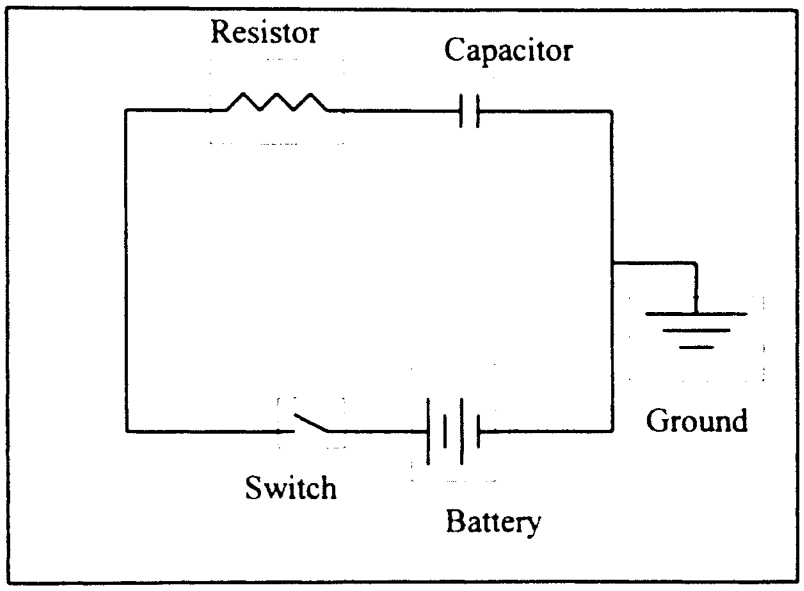


Figure 15. Resistor Capacitor Circuit

APPENDIX C

PROGRAM LISTING

VITA 2

Shashi V. Kowdle

Candidate for the Degree of

Master of Science

Thesis: ANALYSIS OF QUANTITY SPACE HIERARCHY IN CC

Major Field: Computer Science

Biographical:

Personal Data: Born in Mysore, India, January 4, 1969, daughter of K.S.Viswamurthy and H.K. Kamala.

Education: Recieved Bachelor of Engineering Degree from B.M.S. College of Engineering, Bangalore University, India in August 1990; completed the requirements for the Master of Science at Oklahoma State University in May 1994.

Professional Experience: Software Engineer, Hindustan Computer Limited, India, August 1989 to March 1991

PROGRAM LISTING

```
/******************************************************************************

      This is a program in C to implement the hierarchy of quantity spaces in CC.
      In the CC documentation by Franke and Dvorak it is mentioned that the quantity spaces can
      be arranged in a hierarchy. Quantity spaces can be made to inherit landmark values and conservation
      correspondences of other quantity spaces. But in such inheritance the landmark values inherited
      overrides the land mark values that existed before in the quantity spaces considered. More over
      no inheritance can be performed on the inherited quantity spaces. This program takes in a CC input
      file with the specialization operations that has to be performed and gives out a final CC input file with the
      specialization operations performed. The three different operations that can be performed are
                  1. Add a new landmark value to the quantity space specified
                  2. Delete a landmark value from the quantity space specified.
                  3. Combine landmark values from two different parent
                     quantity spaces.
      the input format of this file is

                  qsimconvert  <input-file> <output-file>

      The output file can be used as a CC input file.

******************************************************************************/

#include<stdio.h>

#define FALSE 0
#define TRUE 1

main(argc, argv)
int argc;
char *argv[];
{
  FILE *inp,*out;
  char qspace[200];

  /*opening of input and output files*/
  inp = fopen(argv[1],"r");
  out = fopen(argv[2],"w");

  if(argc==1){
    printf("Input and Output Files Are Not Specified \n");
    exit(1);
    }
  else if(argc==2){
    printf("Output File Not Specified \n");
    exit(1);
    }
  else{
    process_file(inp,out,qspace);
    printf("\n\n\t\tThe Output is written to the file %s\n\n",argv[2]);
    fclose(inp);
  }
}

/*****************************************************************************

Function: process_file

Purpose:

This is the  main function that is used to add delete and combine land
mark values of quantity spaces specified.  This function calls three main
functions for this purpose. The new quantity spaces that are obtained are
written onto a new file and the other contents of the input file are written
to a different file.  Finally the two files are combined to obtain the output
file.

*****************************************************************************/
process_file(inp,out,qspace)
FILE *inp;
FILE *out;
char qspace[];
{
  char line[82],fword[40],word[40],qs1[40],qs2[40],qs[40];
  char *new_qs,word1[40];
  char w1[40],w2[40],w3[40],new_q[15],qname[15],var[40],oper[40];
  char components[][40] = {"battery","resistor","capacitor","inductor",
               "INDUCTOR-B","TANK","SOURCE","SINK"};
  char new_qspace_name[10],qspace_name[82],c;
  int i=0,j=0,k=0,l=0,flag = FALSE,m=0,n=0,pos,a=0;
  FILE *out1,*out2;


  /*opening of two files to write the output*/
  out1 = fopen("main_file","w");
  out2 = fopen("temp","w");

  strcpy(qspace,"");
  /*initially put 10 blank lines to be filled by quantity-space*/
  fgets(line,82,inp);

  /*while end of file is not reached*/
  while (!(feof(inp))){

    /*if line begins with ;;; or if it is an empty line copy to op file*/
    if((line[i] == ';')||(strcmp(line," ")==0)&&(!feof(inp))){
      fputs(line,out1);
      fgets(line,82,inp);
      }

    /*if the line begins with any other characters do this .......*/
    else{
      i=0;
```

```
 strcpy(fword," ");
 strcpy(oper," ");
 sscanf(line,"%s",fword);

/*if the line has quantity space in it do this ......*/
 if((strcmp(fword,"(define-quantity-space")==0)&&(!feof(inp))){
    while(line[i++] != ' ');
    while(line[i++] == ' ');
    i--;
    k=0;

    /*get the quantity spaces and put them in a string qspace*/
    while(line[i] != ' ')
      qs[k++] = line[i++];
    qs[k] = '\0';
    strcat(qspace,qs);
    strcat(qspace,"#");
    while(line[i++] == ' ');
    i--;
    k=0;

    /*each landmark value is ended with a $ and each quantity space
      is ended with an @*/
    while(line[i] != ')'){
      while(line[i] != ' '){
        if(line[i] != '(')
          qs1[k++] = line[i++];
        else i++;
        }
      qs1[k++] = '$';
      while(line[i++] == ' ');
     i--;
      }
    qs1[k-1] = '@';
    qs1[k] = '\0';
    strcat(qspace,qs1);
    fgets(line,82,inp);
    strcpy(fword," ");
    }

/*if the input line is define component-implementation do this ....*/
else if((strcmp(fword,"(define-component-implementation")==0)&&(!feof(inp))){
     fputs(line,out1);
     fgets(line,82,inp);
     fputs(line,out1);
     fgets(line,82,inp);
     flag = FALSE;

     /*check to see if the implementation is not of a component
       already defined*/
     while((flag == FALSE) && (m!=8)){
       if(strstr(line,components[m])!=NULL)
         flag = TRUE;
       m++;
 }

/*if the implementation is of a component already defined do
 this ...........*/
if(flag == TRUE){
  fputs(line,out1);
  fgets(line,82,inp);
  strcpy(fword," ");
  }

/*else do this .....*/
else{
  fputs(line,out1);
  fgets(line,82,inp);
  fputs(line,out1);
  fgets(line,82,inp);
  sscanf(line,"%s",fword);

  /*if the line begins with the component definition scan the
    various operands from the line*/
  while((strcmp(fword,"(connections") != 0)&&(!feof(inp))){
    strcpy(oper," ");
    if(strcmp(fword,"(components")== 0)
      sscanf(line,"%*s%*s%*s%s",oper);

    /*if the line doesnot begin with the component do
      this ...........*/
    else
      sscanf(line,"%*s%*s%s",oper);

    /*if an add,delete or combine operation is to be
      performed do this*/
    if((strcmp(oper,"(add")==0)||(strcmp(oper,"(delete")==0)
      ||(strcmp(oper,"(combine")==0)){

      /*if the line begins with the component definition*/
      if(strcmp(fword,"(components") == 0){
        sscanf(line,"%*s%*s%*s%s",oper);

        /*if it is needed to add a landmark value to a quantity space do....*/
        if(strcmp(oper,"(add") == 0)
                sscanf(line,"%s%s%s%s%s%d%s%s%s",w1,w2,w3,oper,new_q,
                &pos,var,qname,new_qspace_name);
        /*if it is needed to delete a landmark value from a quantity space do....*/
        else if(strcmp(oper,"(delete") == 0)
            sscanf(line,"%s%s%s%s%s%s%s%s",w1,w2,w3,oper,new_q,var,qname,
                new_qspace_name);

        /*if it is necessary to combine landmark values of two quantity spaces do....*/
        else
          sscanf(line,"%s%s%s%s%s%s%s%s",w1,w2,w3,oper,qs1,qs2,var,new_qspace_name);
        fputs(w1,out1);
        fputs(" ",out1);
        }
  /*if the line doesnot begin with the component defn*/
  else{
    /*(B battery (add V 2 voltage vs)*/
    if(strcmp(oper,"(add") == 0)
      sscanf(line,"%s%s%s%s%d%s%s%s",w2,w3,oper,new_q,&pos,var,qname,new_qspace_name);
    /*(B battery (delete V  voltage vs)*/
    else if(strcmp(oper,"(delete") == 0)
      sscanf(line,"%s%s%s%s%s%s%s",w2,w3,oper,new_q,var,qname,new_qspace_name);
    else{
      sscanf(line,"%s%s%s%s%s%s%s",w2,w3,oper,qs1,qs2,var,new_qspace_name);
      }
    }

  /*write the output file*/
  fputs(w2,out1);
  fputs(" ",out1);
  fputs(w3,out1);
  fputs(" ",out1);
  fputs("(quantity-spaces(",out1);
  fputs(var,out1);
  fputs(" ",out1);

  a=strlen(new_qspace_name)-1;
  while(new_qspace_name[a] == ')')
  new_qspace_name[a--] = '\0';
  new_qspace_name[a+1]= '\0';
  /*call the add function to add the land mark value*/
  /*(add V 2 voltage vs)*/
  if(strcmp(oper,"(add") == 0)
    add(qspace,new_q,qname,new_qspace_name,out1);

  /*call the delete function to delete the land mark*/
  /*(delete V voltage vs)*/
  else if(strcmp(oper,"(delete") == 0)
      delete(qspace,new_q,qname,new_qspace_name,out1);

  /*call the combine function to combine the land mark*/
  else
    combine(qspace,qs1,qs2,out1,new_qspace_name);

  /*write the result to the output file*/
  fputs(")))",out1);
  fputs("\n",out1);
  fgets(line,82,inp);
  strcpy(fword," ");
  }

/*if no operation has to be performed do this ......*/
 else{
  if(!feof(inp)){
    fputs(line,out1);
    fgets(line,82,inp);
    sscanf(line,"%s",fword);
```

```
                }
              }
            }/*end of while*/
          }
      }
      else{
        if(!feof(inp)){
          fputs(line,out1);
          fgets(line,82,inp);
          strcpy(fword," ");
          }
        }
      }
  }

/*close the output file*/
fclose(out1);
i=strlen(qspace);
qspace[i-1] = '%';
i=0;

/*writing the new quantity spaces and the old quantity spaces in the
  output file*/
while(qspace[i] != '\0'){
  if(qspace[i] != '%'){
  j=0;
  fputs("(define-quantity-space ",out2);
  while((qspace[i] != '#')){
    if(qspace[i] == '@')
      i++;
    else
      qspace_name[j++] = qspace[i++];
    }
  qspace_name[j++] = ' ';
  qspace_name[j++] = '(';
  qspace_name[j] = '\0';
  fputs(qspace_name,out2);
  j = 0;
  while((qspace[i] != '@')&&(qspace[i] != '\0')){
    if(qspace[i] == '$'){
      qspace_name[j++] = ' ';
      i++;
      }
    else{
      if((qspace[i] == '#')||(qspace[i] == '%'))
        i++;
      else
      qspace_name[j++] = qspace[i++];
      }
    }
   qspace_name[j] = '\0';
   fputs(qspace_name,out2);
   fputs(")))",out2);
   fputs("\n",out2);
```

```
      }
    fputs("\n",out2);
    }
    fclose(out2);

  /*combining the contents of the two output files*/
  out1 = fopen("main_file","r");
  out2 = fopen("temp","r");
  while(!(feof(out2))){
    fgets(line,82,out2);
    fputs(line,out);
    }
  while(!(feof(out1))){
    fgets(line,82,out1);
    fputs(line,out);
    }

  /*closing the opened files*/
  fclose(out);
  fclose(out1);
  fclose(out2);
}

/****************************************************************************

Function: add

Purpose:

This is a function used to add a new landmark value to the quantity space
specified at the specified position.  qspace is a string that contains all the
quantity space names and landmark values for the quantity spaces.  The landmark
values of the specified quantity spaces is obtained from the qspace string.Then
the new landmark value is added to the quantity space.  Finally a new quantity
space name is given to the quantity space for which the new landmark value is
added.

        The syntax for adding the new landmark value is:

   (add new-landmark-value quantity-space variable new-quantity-space-name)


****************************************************************************/
add(qspace,new_q,qname,new_qspace_name,out)
char qspace[],new_q[],qname[],new_qspace_name[];
FILE *out;
{

  int i,j,k,l,flag,m,n=0,counter=0,pos;
  char a,check_qs[80],new_qspace[80],qname2[40];

  k=0;
  flag = FALSE;
  l=0;
```

```
for(i=0;(qname[i] != '\0');i++)
  qname2[l++] = qname[i];
qname2[l] = '\0';
strcpy(qname,qname2);

/*obtain the landmark values for the specified quantity space*/
for(l=0;((qspace[l]!='\0')&&(flag!=TRUE));l++){
  while(qspace[l] != '#'){
   if(qspace[l] != '@')
     check_qs[k++] = qspace[l];
   l++;
   }
  check_qs[k] = '\0';
  k=0;
  if(strcmp(check_qs,qname) == 0)
    flag = TRUE;
  if(flag != TRUE){
  while(qspace[l] != '@')
    l++;
  }
 }
i = 0;
system("tput clear");
printf("\n\n\n\n");
printf("Add a Landmark value to the Quantity space:\n\n\t");
print_qspace(qspace,qname);
printf("\n\n");
printf("Give Position Of the New Land Mark Value %s: ",new_q);
scanf("%d",&pos);

/*find no of landmark values*/
counter = find_no_of_lmarks(qspace,l);

/*if the position specified by the user is lesser than 1 change
  position of the new landmark value to 1*/
if(pos < 1)
  pos = 1;

/*if the position specified by the user is greater than the no of landmark
  values change position of the new landmark value as the last position*/
else if(pos > (counter + 1))
  pos = counter+1;
printf("\n");

/*obtain the position to put the new quantity space*/
if(flag == TRUE){
  while(n < (pos-1)){
    while((qspace[l] != '$')&&(qspace[l] != '\0')){
      new_qspace[i++] = qspace[l++];
      }
    new_qspace[i++]=qspace[l++];
    n++;
    }
  new_qspace[i] = '\0';
```

```
    /*add the new landmark value*/
    strcat(new_qspace,new_q);
    i = strlen(new_qspace);
    new_qspace[i++] = '$';
    while((qspace[l] != '@')&&(qspace[l] != '\0'))
      new_qspace[i++] = qspace[l++];
    new_qspace[i++] = '@';
    new_qspace[i] = '\0';
    while(qspace[l] != '\0') l++;

    /*put the new quantity space with the new landmark value added in the
      qspace string*/
    strcat(qspace,new_qspace_name);
    strcat(qspace,"#");
    strcat(qspace,new_qspace);
    fputs(new_qspace_name,out);
    }
  else{
    printf("Error in the input file\n");
    exit(0);
    }
}

/*****************************************************************************

Function: delete

Purpose:

        This is a function to delete the landmark value from the quantity space
specified at the specified position.  qspace is a string that contains all the
quantity space names and landmark values for the quantity spaces.  The landmark
values of the specified quantity spaces is obtained from the qspace string.Then
the specified landmark value is deleted from the quantity space.Finally the new
quantity space name is given to the quantity space for which the specified land
mark value is deleted and the new quantity space is added to the qspace string.

        The syntax for deleting a landmark value is:

   (delete landmark-value quantity-space variable new-quantity-space-name)

*****************************************************************************/

delete(qspace,new_q,qname,new_qspace_name,out1)
char qspace[];
char new_q[];
char qname[];
char new_qspace_name[];
FILE *out1;
{
  int i,j,k,l,flag,m,n=0,flag1,p;
  char qname2[40],check_qs[80],new_qspace[80],check[80];
```

```
k=0;
flag = FALSE;
l=0;
for(i=0;(qname[i] != '\0');i++)
  qname2[l++] = qname[i];
qname2[l] = '\0';
strcpy(qname,qname2);

/*obtain the landmark values of the quantity space specified*/
for(l=0;((qspace[l]!='\0')&&(flag!=TRUE));l++){
  while(qspace[l] != '#'){
    if(qspace[l] != '@')
      check_qs[k++] = qspace[l++];
      }
  check_qs[k] = '\0';
  k=0;
  if(strcmp(check_qs,qname) == 0)
    flag = TRUE;
  if(flag != TRUE){
    while(qspace[l] != '@')
     l++;
    }
  }
i = 0;
p = 0;
flag1 = FALSE;

/*obtain the landmark values until the landmark vAlue specified is obtained*/
if(flag == TRUE){
  while((qspace[l] != '@') && (flag1 != TRUE)){
    while(qspace[l] != '$'){
     new_qspace[n++] = qspace[l];
     check[p++] = qspace[l++];
     }
    new_qspace[n++] = qspace[l];
    check[p] = '\0';
    if(strcmp(check,new_q) == 0)
      flag1 = TRUE;
    else{
      p =0;
      l++;
      }
    }
  }
  n=n-2;
  l++;

  /*delete the landmark value specified*/
  if(flag1 == TRUE){
    while((new_qspace[n]!= '$')&&(new_qspace[n] != '\0'))
      n--;
    n++;

  /*copy the landmark values after deleting the specified landmark value*/
```

```
      while(qspace[l] != '@')
         new_qspace[n++] = qspace[l++];
      new_qspace[n++] = '@';
      new_qspace[n] = '\0';
      }

   /*obtain a new quantity space name*/
   fputs(new_qspace_name,out1);

   /*put the new quantity space name in the qspace string*/
   strcat(qspace,new_qspace_name);
   strcat(qspace,"#");
   strcat(qspace,new_qspace);
}
/*****************************************************************************
```

Function: combine

Purpose:

This is a function used for combining quantity spaces from 2 specified quantity spaces. The land mark values of the two quantity spaces which has to be combined is obtained from the qspace string. the ordering of the landmark values during the combination is done with the user preference. The user is given the preference of the landmark values. The landmark values that are common to both the specified quantity spaces is taken once and put in a new string. This function calls another function common_qs that gives all the quantity spaces that are common to both the quantity spaces.

The syntax for combining landmark values from two quantity spaces is:

(combine quantity-space-1 quantity-space-2 variable new-quantity-space-name)

```
*****************************************************************************/

combine(qspace,qs1,qs2,out,new_qspace_name)
char qspace[];
char qs1[];
char qs2[];
char new_qspace_name[];
FILE *out;
{
  char common[20][40],qspace1[80],qspace2[80],temp1[20][50],temp2[20][50],
     word[20][80];
  char temp_qspace[100],check_qs[40],arrange[40][40];
  int a=0,i=0,j=0,k=0,l=0,m=0,n=0,flag,flag2,flag1,o=0,p=0,b=0,choice;

  flag = FALSE;

  /*obtain the landmark values of the first quantity space that has to
    combined with the other quantity space from the qspace string which
    has the entire quantity spaces with their names and landmark values*/
  for(l=0;((qspace[l]!='\0')&&(flag!=TRUE));l++){
    while(qspace[l] != '#'){
```

```
    if(qspace[l] != '@')
      check_qs[k++] = qspace[l++];
      }
  check_qs[k] = '\0';
  if(strcmp(check_qs,qs1) == 0)
    flag = TRUE;
  else{
    while(qspace[l] != '@')
      l++;
    k=0;
    }
  }
if(flag == TRUE){
  while(qspace[l]!= '@')
    qspace1[p++] = qspace[l++];
  qspace1[p] = '\0';
  }
else printf("error\n");

/*obtain the landmark values of the second quantity space that has to
  combined with the other quantity space from the qspace string which
  has the entire quantity spaces with their names and landmark values*/
flag = FALSE;
for(l=0;((qspace[l]!='\0')&&(flag!=TRUE));l++){
  while(qspace[l] != '#'){
    if(qspace[l] != '@')
      check_qs[k++] = qspace[l++];
      }
  check_qs[k] = '\0';
  if(strcmp(check_qs,qs2) == 0)
    flag = TRUE;
  else{
    while(qspace[l] != '@')
      l++;
    k=0;
    }
  }
 p = 0;
 if(flag == TRUE){
   while(qspace[l]!= '@')
     qspace2[p++] = qspace[l++];
   qspace2[p] = '\0';
   flag = FALSE;
   }
 else printf("error\n");

 strcat(qspace1,"$");
 strcat(qspace1,"end");
 strcat(qspace2,"$");
 strcat(qspace2,"end");
 /*get the landmark values common among the 2 quantity quantity spaces*/
 common_qs(common,qspace1,qspace2);
 flag1 = FALSE;
 n=0;
```

```
/*take the landmark values that are before the common landmark values
  in the first quantity space and put in a temperory string temp1*/
k=0;
m=0;
while(strcmp(common[n],"\0") != 0){
while((qspace1[i] != '\0')&&(flag1 == FALSE)){
  l=0;
  while((qspace1[i] != '$')&&(qspace1[i]!='\0')){
    check_qs[l++] = qspace1[i++];
    }
  check_qs[l] = '\0';
  if(strcmp(check_qs,common[n]) == 0){
    flag1 = TRUE;
   if(qspace1[i] !='\0')
     i++;
   }
  else{
    strcpy(temp1[k++],check_qs);
   if(qspace1[i] !='\0')
     i++;
    l = 0;
    }
  }
  strcpy(temp1[k++],common[n]);
  l=0;

/*take the landmark values that are before the common landmark values
  in the second quantity space and put in a temperory string temp1*/
flag2 = FALSE;
if(flag1 == TRUE){
  while((qspace2[a] != '\0')&&(flag2 == FALSE)){
    while((qspace2[a] != '$')&&(qspace2[a] != '\0')){
      check_qs[l++] = qspace2[a++];
      }
    check_qs[l] = '\0';
    if(strcmp(check_qs,common[n]) == 0){
      flag2 = TRUE;
      if(qspace2[a] !='\0')
      a++;
      }
    else{
      strcpy(temp2[m++],check_qs);
      l = 0;
      if(qspace2[a] !='\0')
      a++;
      }
    }
   }
  strcpy(temp2[m++],common[n]);
  n++;
  flag1 = FALSE;
  flag2 = FALSE;
  }
```

```
strcpy(temp1[k],"\0");
strcpy(temp2[m],"\0");


  /*if there are no landmark values before the common land mark value
    in the first quantity space put the landmark values before the
    common value from the second string into the new quantity space*/
  if(k==0){
    for(o=0;(strcmp(temp2[o],"end") != 0);o++){
      strcat(temp_qspace,temp2[o]);
      strcat(temp_qspace,"$");
      }
    strcat(temp_qspace,common[n]);
    strcat(temp_qspace,"$");
    }

  /*if there are no landmark values before the common land mark value
    in the second quantity space put the landmark values before the
    common value from the first string into the new quantity space*/
   else if(m==0){
     for(o=0;(strcmp(temp1[o],"end") != 0);o++){
       strcat(temp_qspace,temp1[o]);
       strcat(temp_qspace,"$");
       }
     strcat(temp_qspace,common[n]);
     strcat(temp_qspace,"$");
     }

   /*if there are no landmark values before the common land mark value
     in the first and second quantity space put the common landmark value
     into the new quantity space*/
    else if((k==0)&&(m==0)){
     strcat(temp_qspace,common[n]);
     strcat(temp_qspace,"$");
     }

   /*if there are landmark values before the common land mark value
     in the first and second quantity space the user is given a choice
     to give the order of preference among the landmark values*/
    else{
    a = 0;
     j=0;
    system("tput clear");
    printf("\n\n\n\n\n");

    printf("\t\tThese are the two quantity spaces to be combined\n\n");
    printf("\t\t\t\t(");
    while(strcmp(temp1[j],"end") != 0)  {
      printf("%s  ",temp1[j]);
      j++;
      }
     printf(")");
     printf("\n");
     printf("\t\t\t\t(");
    k=0;
    while(strcmp(temp2[k],"end") != 0) {
      printf("%s ",temp2[k]);
      k++;
      }
    printf(")");
    printf("\n\n\n");

    /*A menu where either the user can specify the ordering of
      landmark values or the preprocessor can specify the ordering*/
    printf("\t\t\t\tM E N U\n\n");
    printf("\t\t1. User Specifies Ordering Of Landmark Values\n");
    printf("\t\t2. Preprocessor Specifies Ordering Of Landmark Values.\n");
    printf("\n\t\tSelect Your Choice: ");
    do{
      scanf("%d",&choice);
    }while((choice!=1) && (choice != 2));
    switch(choice){
      /*user specifies the ordering of landmark values*/
      case 1: user_spec(common,temp_qspace,temp1,temp2);
              break;

      /*preprocessor specifies the ordering of landmark values*/
      case 2: pre_spec(common,temp_qspace,temp1,temp2);
              break;
      default: break;
      }
    }


/*if there are no common land mark values among the quantity spaces
  specified the user is given a choice to give the order of preference
  among the landmark values of the 2 quantity spaces*/
if(n==0){
   printf("Arrange the Quantity Spaces in the pattern  a b c d \n");
   printf("Give the order of preference among the quantity spaces:\n");
   strcat(qspace1,"$");
   strcat(qspace1,qspace2);
   i=0;l=0;
   while(qspace1[i] != '\0'){
     k=0;
     while((qspace1[i] != '$')&&(qspace1[i] != '\0'))
       check_qs[k++] = qspace1[i++];
     check_qs[k]='\0';
     if(strcmp(check_qs,"end") != 0)
     strcpy(temp1[l++],check_qs);
     i++;
     }
     strcpy(temp1[l],"end");
  for(i=0;(strcmp(temp1[i],"end")!=0);i++)
    printf("%s ",temp1[i]);
    scanf("%s",word);
    while(strcmp(word,"$") != 0){
      strcat(temp_qspace,word);
```

```
            strcat(temp_qspace,"$");
            scanf("%s",word);
            }
      }

   /*put the new landmark values to the new quantity space that is created*/
   fputs(new_qspace_name,out);
   strcat(qspace,new_qspace_name);
   strcat(qspace,"#");
   strcat(qspace,temp_qspace);
   i=strlen(qspace)-1;
   qspace[i] = '@';
}

/*****************************************************************************

Function: common-qs

Purpose:

This function is used to obtain the landmark values that are common among the
2 quantity spaces that has to be combined to obtain the new quantity space
with the combination of landmark values from both the parents.

*****************************************************************************/
common_qs(common,qspace1,qspace2)
char common[][40];
char qspace1[];
char qspace2[];
{
   int i=0,j=0,k=0,l=0,n=0,flag;
   char temp[40],temp1[40];

   l=0;
   /*compare the landmark values of the first quantity space with that
     of the second*/
   while(qspace1[i] != '\0'){
     while((qspace1[i] != '$')&&(qspace1[i] != '\0'))
       temp[j++] = qspace1[i++];
     temp[j] = '\0';
     k=0;
     flag = FALSE;
     while((qspace2[k] != '\0') && (flag == FALSE)){
       j=0;
       while((qspace2[k] != '$')&&(qspace2[k] != '\0'))
         temp1[j++] = qspace2[k++];
       temp1[j] = '\0';
       if(strcmp(temp,temp1) == 0){
         flag = TRUE;
         j=0;
         }
       else{
         if(qspace2[k] != '\0')
         k++;
```

```
        j=0;
        }
      }

    /*put the common land mark values to an array*/
    if(flag == TRUE){
      strcpy(common[l++],temp1);
      flag = FALSE;
      }
    if(qspace1[i] != '\0')
      i++;
    }
}
/***************************************************************************

Function: print-qspace
Purpose:

This is a function for printing the quantity spaces specified.
***************************************************************************/
print_qspace(qspace,qname)
char qspace[];
char qname[];
{
   char check_qs[40];
   int i = 0,flag = FALSE,k = 0;


   while((qspace[i] != '\0')&&(flag == FALSE)){
     k = 0;
     while(qspace[i++] != '#')
       check_qs[k++] = qspace[i-1];
     check_qs[k] = '\0';
     if(strcmp(check_qs,qname) == 0)
        flag = TRUE;
     else while(qspace[i++] != '@');
     }

   if(flag == TRUE){
   printf("( ");
   while((qspace[i] != '@') && (qspace[i] != '\0')){
     k = 0;
     while((qspace[i++] != '$')&&(qspace[i] != '\0'))
        check_qs[k++] = qspace[i-1];
     check_qs[k] = '\0';
     printf("%s ",check_qs);
     }
   printf(")");
   }
 }
```

```
/***************************************************************************
Function: user-spec

Purpose:

The user specifies the ordering of the landmark values. This function is
used when combing landmark values from two quantity spaces.  This displays
the landmark values that have to be arranged in a quantity space and the
user is queried for the ordering of those landmark values.  Then a new
quantity space is created with the ordering of landmark values specified
by the user.
***************************************************************************/
user_spec(common,temp_qspace,temp1,temp2)
char common[][40];
char temp_qspace[],temp1[][50];
char temp2[][50];
{
  int n=0,j=0,b=0,a=0,k=0,flag = FALSE,l=0;
  char word[30][40],arrange[30][40];

  system("tput clear");
  printf("\n\n\n\n\n");

  printf("These are the two quantity spaces to be combined\n\n");
  printf("\t(");
  while(strcmp(temp1[j],"end") != 0)  {
    printf("%s  ",temp1[j]);
    j++;
  }
  printf(")");
  printf("\n");
  printf("\t(");
  k=0;

  /*print the quantity spaces to be combined*/
  while(strcmp(temp2[k],"end") != 0) {
    printf("%s  ",temp2[k]);
    k++;
  }
  printf(")\n");
  printf("Arrange Quantity Spaces:\n\n ");
  n=0;
  j = 0;

  /*Query the ordering of landmark values*/
  while(strcmp(common[n],"\0")!=0){
    b = 0;
    printf("\t\t(");
    while(strcmp(temp1[j++],common[n])!=0)
      strcpy(arrange[b++],temp1[j-1]);
    while(strcmp(temp2[a++],common[n])!=0)
      strcpy(arrange[b++],temp2[a-1]);
    strcpy(arrange[b],"\0");
    for(k = 0; k < b; k++)
```

```
          printf("%s ",arrange[k]);
        printf(")");
        flag = FALSE;

        /*Until; the user specifies the correct input do ....*/
        while(flag == FALSE){

        /*accept the new ordering of landmark values*/
        for(k=0;k<b;k++)
          scanf("%s",word[k]);
        strcpy(word[k++],"$");
        strcpy(word[k],"\0");
        l=0;
        k = 0;

        /*check for wrong input specified by the user*/
        while(l<b){
          flag = FALSE;
          while((strcmp(word[k],"$")!=0) && (flag == FALSE)){
            if(strcmp(arrange[l],word[k])==0){
              k=0;
              flag = TRUE;
              }
            else  k++;
            }
            l++;
          }
        }

        /*once the input is correctly specified by the user update the
          qspace string*/
        if(flag == TRUE){
          for(k = 0;k < b;k++){
            strcat(temp_qspace,word[k]);
            strcat(temp_qspace,"$");
            }
          }
      if(strcmp(common[n],"end") != 0){
        strcat(temp_qspace,common[n]);
        strcat(temp_qspace,"$");
        }
      n++;
      }
}

/*****************************************************************************
Function: pre-spec

Purpose:

The pre-processor specifies the ordering of the landmark values. This function
is used when combing landmark values from two quantity spaces.  This displays
the landmark values that have to be arranged in a quantity space and the
preprocessor specifies the ordering of the landmark values. The user is queried
```

```
for the ordering of the landmark values if necessary. if the user opts to
change the ordering of landmark values the new ordering of the landmark values
is accepted by the program and a new quantity space is created with the
ordering of landmark values specified by the user.  Else a new quantity space
is created with the ordering of landmark values specified by the preprocessor.
******************************************************************************/
pre_spec(common,temp_qspace,temp1,temp2)
char common[][40];
char temp_qspace[],temp1[][50];
char temp2[][50];
{
  int i=0,n=0,j=0,b=0,a=0,k=0,flag = FALSE,l=0,t;
  char word[30][40],arrange[30][40],temp3[30][40],ans[4];

  system("tput clear");
  printf("\n\n\n\n\n");

  /*print the two quantity spaces to be combined*/
  printf("\tThese are the two quantity spaces to be combined\n\n");
  printf("\t\t(");
  while(strcmp(temp1[j],"end") != 0)  {
    printf("%s  ",temp1[j]);
    j++;
  }
  printf(")");
  printf("\n");
  printf("\t\t(");
  k=0;
  while(strcmp(temp2[k],"end") != 0) {
    printf("%s  ",temp2[k]);
    k++;
  }
  printf(")\n\n");
  printf("\tThis is the ordering of the Landmark values:\n\n");
  printf("\t\t");
  l = 0;
  n = 0;
  j = 0;
  t=0;
  i=0;

  /*print the ordering of landmark values specified by the preprocessor*/
  while(strcmp(common[n],"\0") != 0){
    while(strcmp(temp1[i++],common[n])!= 0)
      strcpy(temp3[l++],temp1[i-1]);
    while(strcmp(temp2[j++],common[n])!= 0)
      strcpy(temp3[l++],temp2[j-1]);
      strcpy(temp3[l++],common[n]);
    n++;
    }
  strcpy(temp3[l],"\0");
  printf("( ");
  for(l=0;(strcmp(temp3[l],"end") != 0);l++)
    printf("%s ",temp3[l]);
printf(")\n\n");
n = 0;
j = 0;

/*query if the user needs to change the ordering of landmark values*/
while(strcmp(common[n],"\0")!=0){
   b = 0;
   printf("\tDo You Want To Change The Ordering Among:\n");
   printf("\t\t\t[Press y or n]: \n");
   printf("\t\t\t\t(");
   while(strcmp(temp3[j++],common[n])!=0)
     strcpy(arrange[b++],temp3[j-1]);
   strcpy(arrange[b],"\0");
   for(k = 0; k < b; k++)
     printf("%s ",arrange[k]);
   printf(")");
   scanf("%s",ans);

   /*if the user nedds to change the ordering of landmark values*/
   if(strcmp(ans,"y")== 0){
     printf("\n\t\tGive Landmark Values:\n");
     flag = FALSE;
     while(flag == FALSE){

     /*accept the ordering of landmark values*/
     for(k=0;k<b;k++)
       scanf("%s",word[k]);
     strcpy(word[k++],"$");
     strcpy(word[k],"\0");
     l=0;
     k = 0;

     /*check for wrong input specified by the user*/
     while(l<b){
       flag = FALSE;
       while((strcmp(word[k],"$")!=0) && (flag == FALSE)){
         if(strcmp(arrange[l],word[k])==0){
           k=0;
           flag = TRUE;
           }
         else  k++;
         }
         l++;
       }
     }

     /*if correct input is specified by the user create a new
       quantity space with the ordering specified by the user*/
     if(flag == TRUE){
       for(k = 0;k < b;k++){
         strcat(temp_qspace,word[k]);
         t++;
         strcat(temp_qspace,"$");
         }
```

```
          t++;
          }
      if(strcmp(common[n],"end") != 0){
        strcat(temp_qspace,common[n]);
        strcat(temp_qspace,"$");
        }
      }
      /*else create a new quantity space with the ordering specified
        by the preprocessor*/
      else{
        while(strcmp(temp3[t++],common[n])!=0){
          strcat(temp_qspace,temp3[t-1]);
          strcat(temp_qspace,"$");
          }
        if(strcmp(common[n],"end") != 0)
        strcat(temp_qspace,common[n]);
        strcat(temp_qspace,"$");
        }
      n++;
      }
}
/**********************************************************************
Function: find_no_of_lmarks

Purpose:

Find the number of landmark values in the quantity space to which
a new landmark value is to be added. This is used to specify
the position of the new landmark value if it is wrongly specified
by the user. This function is used by the add function.
**********************************************************************/
find_no_of_lmarks(qspace,l)
char qspace[];
int l;
{
  int count=0;

  while((qspace[l] != '@')&&(qspace[l] != '\0')){
    if((qspace[l] != '$') &&(qspace[l] != '@') && (qspace[l] != '\0'))
       l++;
    else{
       l++;
       count++;
       }
    }
  return(count);
}
/**********************************************************************/
```

APPENDIX D

EXAMPLE SHOWING INHERITANCE OPERATIONS

EXAMPLE SHOWING INHERITANCE OPERATIONS

```
;; -*- Syntax: Common-Lisp; Package: QSIM -*-
(in-package :qsim)
;;; Copyright (c) 1991, Benjamin Kuipers.
;;; Assumes that /u/kuipers/cc/lib/simple-interface.lisp already loaded.
(define-quantity-space vs1 (0 V v1  ))
(define-quantity-space vs2 (V* V v2  ))

(define-component-interface RC "R-C" electrical
  (terminals t1 t2))
(define-component-implementation
Basic
RC
  "Resistor-Capacitor Circuit"
   (components (B battery (combine vs1 vs2 voltage newbvolt))
            (C capacitor (delete 0  voltage newbvolt newcvolt)
            (R resistor (add 0 1 voltage newcvolt newrvolt)
            (S switch)
            (G ground))
   (connections (n1 (R t1) (S t1))
               (n2 (R t2) (C t1))
               (n3 (C t2) (B t2) (G t))
               (n4 (B t1) (S t2))))
```

Refer Appendix A for the component definitions of battery, resistor, capacitor and ground.