THE USE OF A THEOREM PROVER TO VERIFY A LIQUID FLOW CONTROL PROGRAM

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Erkan UÇAR
February, 1993

THE USE OF A THEOREM PROVER TO VERIFY A LIQUID FLOW CONTROL PROGRAM

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Erkan Uçar

February, 1993

Erkan Uçar

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Varol Akman (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Akif Eyler

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. İlyas Çiçekli

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of the Institute

ABSTRACT

THE USE OF A THEOREM PROVER TO VERIFY A LIQUID FLOW CONTROL PROGRAM

Erkan Uçar
M.S. in Computer Engineering and Information Science
Advisor: Assoc. Prof. Varol Akman
February, 1993

Program verification is an important task since it produces reliable software. Verification of real-time control programs needs special attention since these run in the real world and it is difficult to determine their mathematical properties. Besides, validating large real-time programs manually is impossible. Owing to these reasons, mechanical program verification systems have to be used. Boyer-Moore Theorem Prover (NQTHM) which, in fact, is a general-purpose automated theorem prover, is such a system. We corroborated the control programs of a simple real-time system, viz. a water-tank complex, using NQTHM. A useful simulator (called WATERWORKS) has been implemented for this purpose.

Keywords: Program Verification, Boyer-Moore Theorem Prover (NQTHM), Real-Time Control, Simulation, Commonsense Reasoning, Liquids

ÖZET

BİR TEOREM İSPATLAYICININ BİR SIVI AKIŞ KONTROL PROGRAMINI DOĞRULAMADA KULLANILMASI

Erkan Uçar
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Danışman: Doç. Dr. Varol Akman
Şubat, 1993

Güvenilir yazılım ürettiğinden dolayı program doğrulama önemli bir iştir. Uygulamalarının gerçek dünyada çalıştırılmasından ötürü gerçek-zamanlı kontrol programlarının doğrulanması özel dikkat ister ve bunların matematiksel özelliklerini bulmak zordur. Ayrıca, büyük gerçek-zamanlı programları elle doğrulamak imkansızdır. Bu nedenlerden dolayı, mekanik program doğrulama sistemleri kullanılması gerekir. Aslında genel-amaçlı bir otomatik teorem ispatlayıcı olan Boyer-Moore Teorem İspatlayıcısı (NQTHM) böyle bir sistemdir. Biz NQTHM'i kullanarak basit bir gerçek-zamanlı sistemin, yani bir su-tank kompleksinin, kontrol programlarını doğruladık. Bu amaca yönelik olarak WATERWORKS isimli kullanışlı bir benzetim sistemi gerçekleştirdik.

Anahtar Sözcükler: Program Doğrulama, Boyer-Moore Teorem İspatlayıcısı (NQTHM), Gerçek-Zamanlı Kontrol, Benzetim, Sağduyusal Akıl Yürütme, Sıvılar

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my supervisor Assoc. Prof. Varol Akman for his guidance, suggestions, and invaluable encouragement throughout the development of this thesis. (N.B. Both Akman and I gratefully acknowledge the fundamental contributions of Profs. Robert S. Boyer and J Strother Moore to NQTHM without which WATERWORKS would not be possible.)

I would also like to thank to Prof. Akif Eyler and Asst. Prof. İlyas Çiçekli for reading and commenting on the thesis.

I owe special thanks to Prof. Mehmet Baray for providing a pleasant environment for study.

I am grateful to the members of my family for their infinite moral support and patience that they have shown, particularly in times I was not with them.

Contents

List of Figures

Chapter 1

Introduction

A basic area of research in computer science is program verification whose goal is to help the production of reliable software. After observing a computer program, we can decide whether a mathematical proof of its correctness is needed. If such a proof is deemed necessary, one should convert the statements of the program to a more formal notation (e.g., well-formed formulas). Then, these formulas have to be proved using a formal (say, logical) framework. Mechanical program verification tools have been developed for this purpose.

In this thesis, we used the Boyer-Moore Theorem Prover (NQTHM) [3, 5] to verify the control of WATERWORKS, our implementation of a water-tank system. In this system, users can model and simulate liquid flow through various water tanks which are topologically coupled and/or cascaded to each other. Control is applied to the valves of the system to avoid liquid overflow in the tanks. With the help of NQTHM, an inductive formalization of the system is constructed to prove the correctness of such a control.

Chapter 2 briefly introduces the control of real-time systems by discussing a common example: a heater controlled by a thermostat. Analysis of another system which controls the behavior of a vehicle in a varying crosswind then follows. This last problem has been verified using NQTHM [6] and has largely motivated our research.

In Chapter 3, NQTHM (New Quantified THeoreM Prover) is discussed. The program is viewed from two directions: logical and mechanical. Section 3.1 explains the logical aspects including the programming language concepts and problem formalization. In Section 3.2, the mechanical system is introduced by

focusing on its organizational aspects and user interface.

Chapter 4 explains WATERWORKS, the water-tank system which is coded in the C programming language, and how its control is verified using NQTHM. Both informal and formal models of the system are given. After that, a conjecture about the system is proved. This chapter ends with a discussion of the relevance of our research to qualitative reasoning.

Chapter 5 contains the concluding remarks and directions for further research. In Appendix A, the basic commands of WATERWORKS are listed. Appendix B gives instructions for accessing WATERWORKS.

Chapter 2

Real-Time Control

In this chapter, we consider issues involving "real-time" control of one or more devices. For a better understanding what is meant by real-time control, consider the following simple situation which is adapted from [22]:

> Consider a workshop with a heater that is controlled by a thermostat. For the sake of discussion, the basic mechanism consists of *sensors* that detect the characteristics of the environment and *devices* which can be (de)activated. The only device is the heater and there are two sensors: a thermometer to measure the temperature and detects whether it is the desired temperature or not, and another sensor which sends a signal if the heater is on.

Here, it is only the thermostat which performs the "reasoning." (For more complex control programs, clearly a more sophisticated modeling is needed.) In this problem, the state of the system is determined by

- the actual temperature,
- the desired temperature, and
- whether or not the heater is on.

The needed information consists of the values of the sensors. All sensors are checked at regular intervals. Each time the sensors are checked, the reasoning component is invoked to determine whether or not any actions are required,

and if required the appropriate devices must be turned on/off. In this case, the reasoning component determines whether or not the temperature has fallen below the desired value. If it has, and if the heater is off, the heater is turned on. However, if the temperature is above the desired value, and the heater is on, the heater is turned off.

The construction of real-time control systems presents two fundamental problems. First, the construction of sensors and interfaces poses a variety of challenges. The second and more important problem involves the implementation of the required reasoning component. In many situations, such as this heater example, the required reasoning is so simple that almost all of the attention is directed to building the appropriate sensors. On the other hand, in complex situations, the rules needed for reasoning are more complicated than the design of sensors.

The reasoning component is a program in which, initially, the rules that determine which responses are necessary in any given situation are encoded as clauses in a general axiom list. The system sleeps between the periodic sensor checks. When the sensor is checked, a single clause giving the current state of the system is formed. Then, the reasoning component "wakes up" and starts deducing the actions to be taken. This process continues until the limit conditions of the system hold.

These ideas apply (in principle) to large systems that are monitored in real time. Some important applications include nuclear power systems, large assembly lines, manufacturing plants, and chemical production systems [22]. In such systems, a large number of sensors are required to keep track of the state of the system, and to report information to mechanisms for reacting against abnormal states. In many cases, inference is managed by an automated reasoning system.

A problem which has been verified using NQTHM [6] and motivated our research is to show that a certain control program keeps a vehicle on a straight-line course in a varying crosswind [6]. It is assumed that the desired course of the vehicle as the x-axis towards the increasing values. The vehicle carries a sensor that, in each sampling interval of time, reads either 1, 0, or -1, according to whether the vehicle is to the left of the course ($y > 0$), on the course ($y = 0$), or to the right of the course ($y < 0$). It also has an actuator which is used to change the y-component of the vehicle's velocity under the

control of some program which reads the output of the sensor. The problem is to state formally what it means to keep the vehicle on course and, for a particular control program, mechanically prove that this program satisfies the high level specification.

It is easy to see that this problem involves a specification of the environment with which the control program interacts. In addition to that, the effects of repeated interchanges over time are needed together with a description of input/output interchange between the program and its environment.

Since only the behavior in the y-direction is considered, the model represents a one-dimensional control problem. To avoid arbitrarily large values, the wind speed, w, is assumed to change by at most one unit from one sampling interval to the next. (Thus, the increment in w can be one of -1, 0, or 1.) At each sampling interval, the control program may increment or decrement the y-component of its velocity. If v is the accumulated speed in the y-direction measured as the number of units the vehicle would move in one sampling interval if there were no wind, then the y-coordinate of the vehicle at time $t+1$ is the y-coordinate at time t, plus the accumulated v at time t, plus the displacement due to the wind at time $t+1$:

$$w(t+1) = w(t) + dw(t+1),$$
$$y(t+1) = y(t) + v(t) + w(t+1).$$

The control program changes v at each sampling interval as a function of the current sensor reading $s1$ (at time $t+1$), and the previously obtained reading $s2$ (at time t):

$$v(t+1) = v(t) + deltav(s1, s2),$$

where *deltav* is the mathematical function specifying the output of the control program, e.g.:

$$deltav(s1, s2) = -s1 + 2(s2 - s1).$$

The authors [6] considered the following FORTRAN program for the implementation of this specification. If `SEN1` is the current sensor reading, $s1$,

and the value of SEN2 is the previous sensor reading, $s2$, then ANS is set to $deltav(s1, s2)$.

```
SUBROUTINE DELTAV(SEN1)
INTEGER SEN1, SEN2, ANS
COMMON /DVBLK/ SEN2, ANS
ANS = ((2 * SEN2) - (3 * SEN1))
SEN2 = SEN1
RETURN
END
```

Observing the behavior of the vehicle under different wind histories, two conjectures are stated:

1. No matter how the wind behaves, the vehicle never gets farther than 3 units away from the x-axis.
2. If the wind becomes constant for at least 4 sampling intervals, the vehicle returns to the x-axis and stays there until the wind changes direction.

Boyer, Green, and Moore formalized the problem with an interpreter (coded in NQTHM) which is essentially a simulator for this system. The interpreter takes two arguments. The first is the "state" of the vehicle and the environment—the vehicle position and velocity and the wind velocity. The second is a "wind history"—a list of wind velocity increments describing how the wind changes at discrete time units over some finite time interval. The interpreter determines the final state of the vehicle by "simulating" each state change, i.e., by calculating for each sampling unit the new wind velocity, the resulting position of the vehicle, the associated sensor reading, the response of the control program, and the effect of the vehicle's velocity.

Formally, they let the system states be triples, $< w, y, v >$, viz. the wind speed, the y-position of the vehicle, and the accumulated v. The function STATE returns such a triple, and the functions W, Y, and V are defined to return the respective components of such a triple. Thus, the expression (STATE 43 -2 -41) denotes a state where:

```
(W (STATE 43 -2 -41)) = 43,
(Y (STATE 43 -2 -41)) = -2,
(V (STATE 43 -2 -41)) = -41.
```

Given the current state and the increment to the wind speed, the next state of the vehicle, `NextState(dw, State)`, is returned considering the changes in w, y, and v from time t to $t+1$. `ArbitraryWind(Lst)` returns true or false according to whether each element of `Lst` is a valid change. Consequently, `FinalState(Lst, State)` returns the final state of the vehicle given the wind history, `Lst`, and the initial state, `State`. `FinalState` is recursively defined and can be thought of as simulating the state changes induced by each change in the wind.

Finally, the conjectures stated previously are formally defined as `Theorem1` and `Theorem2` below.

```
Theorem1.  VEHICLE-STAYS-WITHIN-3-OF-COURSE:
If ((ArbitraryWind(Lst)) and
    (State == FinalState(Lst, <0,0,0>)))
Then
    (3 >= Y(State) >= -3)
```

If `Lst` is an arbitrary wind history and `State` is the state of the system after the vehicle has traveled through the wind starting from the initial state $< 0,0,0 >$, then the y-coordinate of `State` is between -3 and 3.

```
Theorem2.  VEHICLE-GETS-ON-COURSE-IN-STEADY-WIND:
If ((ArbitraryWind(Lst1)) and
    (SteadyWind(Lst2))    and
    (Length(Lst2) >= 4)   and
    (State == FinalState(Append(Lst1, Lst2), <0,0,0>)))
Then
    (Y(State) == 0)
```

`SteadyWind` recognizes a wind sequence consisting of 0's, and `Append` concatenates two sequences. If `Lst1` is an arbitrary wind history and `Lst2` is a history of 0's at least 4 sampling intervals long, and `State` is the state of the

system after the vehicle has traveled through the concatenation of two wind histories, then the y-position of the vehicle in the final state is 0.

The key to the proof is that the state space of the vehicle can be partitioned into a finite number of classes. A close observation of the vehicle's behavior shows that any state $< w, y, v >$ reachable starting from $< 0, 0, 0 >$ can be put into one of the 13 classes according to y and $w + v$:

y	w + v
-3	1
-2	1 or 2
-1	2 or 3
0	-1, 0 or 1
1	-2 or -3
2	-1 or -2
3	-1

NQTHM is incapable of discovering this fact for itself. Instead, the user of the system may suggest it by defining the function `GoodStateP(State)` to return true or false according to whether `State` is in one of the 13 classes above, and then instruct NQTHM to prove:

```
Lemma.
If ((GoodStateP(State))  and
    ((dw == -1) or (dw == 0) or (dw == 1)))
Then
    (GoodStateP(NextState(dw, State))).
```

After proving this lemma, NQTHM can establish by induction on the number of sampling intervals that the final state of the vehicle is a good state. From that conclusion it follows that the y-position of the vehicle is within ± 3 of the x-axis. The proof of the second theorem is similar. The vehicle is in a good state after `Lst1` has been processed. But if the vehicle is in a good state and the wind remains steady for 4 sampling intervals, it is easy to show that the vehicle returns to the x-axis with $w + v = 0$. However, in this case it stays on the x-axis as long as w stays constant.

Chapter 3

The Boyer-Moore Theorem Prover

NQTHM was first described in [3]. At that time, the main purpose was to describe in detail how the theorem prover worked, its organization, proof techniques, heuristics, etc. Recently, the logic and the theorem prover have changed somewhat [5].

Two basic reasons for these changes were the efficient use of functions in the logic as new proof procedures (once they have been proved sound) and the introduction of bounded quantification and partial recursive functions. These have resulted in the integration of a linear arithmetic decision procedure and a facility with which the user can give hints to the theorem prover. The two most commonly used hints can be stated as "consider the following instance of the previously proved theorem n" and "do not consider using the previously proved theorem m."

3.1 The Boyer-Moore Logic

The logic is a quantifier-free first-order logic with equality. Its language is a relative of Pure Lisp and consists of variables and function names combined in prefix notation. Its axioms and rules of inference are obtained from the propositional calculus and function symbols by adding:

- axioms characterizing certain basic function symbols,
- two *extension principles* with which one can add axioms to the logic to introduce inductively defined *data types* and recursive functions, and
- mathematical induction as a rule of inference.

The mechanization of the logic is performed via a collection of Lisp programs which let the user define new data types, recursive functions, and proofs. One can view NQTHM as a programming language with which it is possible to define functions and execute them with reasonable efficiency without ever proving anything.

There is a database of rules that determines the prover's behavior. When the user submits axioms, definitions, and theorems, NQTHM creates this database of rules. Each proved result is converted into a rule and asserted in the database. The user can also provide a sequence of lemmas to help the system prove the newly defined conjectures by enabling the existing rules in the database.

3.1.1 Programming Language Aspects

The Boyer-Moore logic contains two "extension" principles under which the user can add new axioms to the system with the guarantee that the model for the logic can be extended to hold for these axioms. These principles allow the axiomatization of new data types and the definition of new recursive functions. Because of these extension principles it is more meaningful to speak of the *Ground Zero* (or *primitive*) logic and its extensions obtained by adding new data types, instead of the traditional logic which implies the choice of some fixed set of axioms.

Almost all of the functions which are axiomatized in the logic are *total recursive*. That is, when we apply a function to explicit values, we can determine the explicit value which is equal to the application. The mechanization of the logic also includes commands for evaluating functions on explicit values. Thus, the logic and its mechanization provide a *programming language* and an *execution environment*.

Syntax The statements in the logic are called "terms." A term is either a variable symbol or a function symbol of n arguments. When a term is defined, functions are required to be applied to the correct number of arguments. An example term containing comments is shown below:

```
(IF (ZEROP N)        ;If N is 0
    N                ;return N
    (ADD1 N))        ;otherwise, return N+1
```

Boolean Operators Below is the list of Boolean operators [5]:

(TRUE): A constant with the abbreviation T.

(FALSE): A constant with the abbreviation F.

(IF X Y Z): If X is F return Z; otherwise return Y.

(EQUAL X Y): Return T if X=Y and F otherwise.

(TRUEP X): Return T if X=T and F otherwise.

(FALSEP X): Return T if X=F and F otherwise.

(AND P Q): Return T if both P and Q are non-F; otherwise return F.

(OR P Q): Return T if either P or Q is non-F; otherwise return F.

(NOT P): Return F if P is non-F; otherwise return T.

(IMPLIES P Q): Return T if either P is F or Q is non-F; otherwise return F.

Data Types One of the ways that the user can extend the logic is by the addition of axioms to define a new class of inductively constructed objects. The *shell principle* provides this facility. This is a mechanism by which the user can create new data types. This principle is also used in the initial creation of the Ground Zero logic to axiomatize the natural numbers, ordered pairs, literal atoms, and the negatives. The principle permits the axiomatization of n-tuples with type restrictions on each of the n components. The axioms that are necessary to form the new type are generated from the axiom schemas by instantiating them with the names that are provided by the user. Here is the general form of a shell definition:

Shell Definition.
Add the shell `const` *of* `n` *arguments*
with (optionally, base function `base`)
recognizer function `r`,
accessor functions $\texttt{ac}_1, \cdots, \texttt{ac}_n$,
type restrictions $\texttt{tr}_1, \cdots, \texttt{tr}_n$, *and*
default functions $\texttt{dv}_1, \cdots, \texttt{dv}_n$.

Here `const`, `n`, `base`, `r`, $\texttt{ac}_i$, $\texttt{tr}_i$, and $\texttt{dv}_i$, have to be filled in by the user with particular function names, numbers, etc.

Objects of the new type are "recognized" by `r`, which returns `T` or `F` according to whether its argument is of the new type. `Base` takes no arguments and returns an object of the new type. `Const` takes `n` arguments and returns an n-tuple of the new type. $\texttt{Ac}_1, \cdots, \texttt{ac}_n$ "access" the respective components of constructed n-tuples of the new type. $\texttt{Tr}_1, \cdots, \texttt{tr}_n$, describe what types of objects are in the components of each constructed n-tuple. $\texttt{Dv}_1, \cdots, \texttt{dv}_n$, are constant functions that specify the values to be used when the supplied argument fails to satisfy its type restriction [3, 5].

Every object of the new type is either the base object (`base`), or else is constructed by `const` from smaller objects satisfying the type restrictions. When the shell principle is invoked, `const`, `base`, `r`, and each of the $\texttt{ac}_i$ must be new function symbols which are never used before. The shell principle defines these symbols. The default functions may be `T`, `F`, previously introduced base function symbols, or the new base function.

Each shell class is disjoint from others. In addition, `T` and `F` are different from all shell objects. The Ground Zero logic includes four shells with the shell principle:

NUMBERP – the natural numbers

Add the shell `ADD1` of one argument
with the base function `ZERO`,
recognizer function `NUMBERP`,
accessor function `SUB1`,
type restriction `(ONE-OF NUMBERP)`,
default function `ZERO`.

(ZERO) is a constant, abbreviated as 0. (ADD1 I) returns the next natural number after I. If I is not a natural number, 0 is used in its place. (NUMBERP X) returns T or F according to whether X is a natural number. (SUB1 I) returns the predecessor of the natural number I. If I is 0 or not a natural number, (SUB1 I) returns 0.

LISTP – the ordered pairs

Add the shell CONS of two arguments
with recognizer function LISTP,
accessor functions CAR and CDR,
default functions ZERO and ZERO.

(CONS X Y) returns the ordered pair whose first component is X and second is Y. (LISTP X) returns T or F according to whether X is an ordered pair constructed by CONS. (CAR X) returns the first component if X is an ordered pair; otherwise it returns 0. (CDR X) returns the second component if X is an ordered pair; otherwise it returns 0.

LITATOM – the words

Add the shell PACK of one argument
with recognizer function LITATOM,
accessor function UNPACK,
default function ZERO.

(PACK L) constructs and returns the literal atom with L as its "printed name." In general, L is a list of ASCII character codes. (LITATOM X) returns T or F according to whether X is a literal atom. (UNPACK X) returns L if X is a literal atom obtained by PACKing L; otherwise it returns 0.

NEGATIVEP – the negative integers

Add the shell MINUS of one argument
with recognizer function NEGATIVEP,
accessor function NEGATIVE-GUTS,
type restriction (ONE-OF NUMBERP),
default function ZERO.

(MINUS I) returns the negative of I if I is natural. (NEGATIVEP X) returns T or F according to whether X is a negative integer. (NEGATIVE-GUTS X) returns the absolute value of X if X is a negative integer; returns 0 otherwise.

Function Definitions The user can define new functions by just adding equations which allow calling of these new functions to be translated to the calls of the old ones. A typical function definition can be viewed as:

Definition.
(fn $x_1 \cdots x_n$) = term

The *formals* of fn are $x_1, \cdots, x_n$, and the *body* of fn is the term term. When the principle of definition is invoked, fn, the formals, and the body must satisfy these restrictions:

- fn must not be defined before and mentioned in any axiom,
- each of the formals must be a distinct variable,
- the body must be a term and cannot contain any variables other than the formals, i.e., no "global" variables, and
- there must exist a measure of the arguments of fn that can be proved to decrease ordinally in each recursive call of fn, e.g., a finite list in LISP.

3.1.2 Formalizing Problems

In converting the question "Is this program correct?" into the question "Is this formula a theorem of the logic?" we face two problems: how to use mathematics to formalize everyday concepts in computer science and how to use Boyer-Moore logic in particular. The first one is a hard problem since there is usually a philosophical gap between the user's intentions and any formal representation of them. Consider the example of appending two objects. It is easy to see that the value of

```
(APPEND NIL '(3 4))
```

is '(3 4), and the value of

```
(APPEND '(1 2) '(3 4))
```

is '(1 2 3 4). What is the value of (APPEND NIL X)? In general, we cannot answer this question unless we know the value of X. But, whatever the value of X, the value of (APPEND NIL X) is the same as that of X, i.e.,

```
(EQUAL (APPEND NIL X) X)
```

has the value T, regardless of the value of X. The question is "Can we prove it?" It is easy to *test* the assertions by executing the expressions under example assignments to the variables. But is it possible to demonstrate that the value must always be T? The answer is "no" if we only work with the logic as a programming language.

A *theorem* is a formula that is either an axiom or can be derived from theorems using the rules of inference. So, if we wish to determine whether a formula has the value T in all models, we can try to *prove* it. (There are formulas that are true in all models but that cannot be proved.) Consider the example of appending three objects: (APPEND X (APPEND Y Z)). Suppose we have already computed (APPEND X Y), and found out that it is the value of the variable P. Then we might use (APPEND P Z), i.e., (APPEND (APPEND X Y) Z), instead of (APPEND X (APPEND Y Z)). The reasoning is fine, *if* we know that APPEND is associative, i.e.,

```
(EQUAL (APPEND (APPEND V1 V2) V3)
       (APPEND V1 (APPEND V2 V3)))
```

always evaluates to T.

This is a simple example of the problem of *formalization.* It is not necessary to be able to prove theorems in order to formalize problems. What is necessary is to have a clear understanding of what formulas in the logic "mean." For the sake of definite answers to the questions stated above, consider the implementation of stacks:

Problem. Formalize stacks on which only numbers are pushed. A stack is an object and the operations on it are functions that return "new" objects with some functional relations to the input. Informally, we need the following five functions:

`PUSH`: Constructs the stack obtained by pushing a given a number onto a stack of numbers.

`EMPTY`: Returns the empty stack of numbers.

`TOP`: Returns the topmost element of a given stack of numbers.

`POP`: Returns the stack obtained by popping the topmost element of a given stack of numbers.

`STACKP`: Returns `T` or `F` according to whether the given argument is a stack of numbers.

This informal specification can be formalized by the following shell definition:

Shell Definition.
Add the shell `PUSH` of two arguments,
with the base object `EMPTY`,
recognizer function `STACKP`,
accessor functions `TOP` and `POP`,
type restrictions `(ONE-OF NUMBERP)` and `(ONE-OF STACKP)`,
default functions `ZERO` and `EMPTY`.

There are some cases which are included in the formalization. `(PUSH T STK)` produces the same stack as `(PUSH 0 STK)` because `T` is not a number and so `PUSH` translates it to the first default object `0`. Similarly, `(TOP (EMPTY))` is `0` because `TOP` returns the first default object when called on a non-stack or on the empty stack. Consequently, `(POP F)` is `(EMPTY)` because `POP` returns the second default object when called on a non-stack or the empty stack.

The expression of relationships and concept definitions are "mathematical" rather than "computational." Thus, the formal statement of some familiar mathematical relationships such as associativity, transitivity, etc. are stated below:

```
(EQUAL (PLUS A B)                    ;Commutativity of PLUS
       (PLUS B A))

(EQUAL (PLUS (PLUS A B) C)           ;Associativity of PLUS
       (PLUS A (PLUS B C)))

(EQUAL (PLUS A 0)                    ;Right Identity of PLUS
       A)

(EQUAL (TIMES P (PLUS A B))          ;Distributivity of TIMES
       (PLUS (TIMES P A)             ;over PLUS
             (TIMES P B)))

(IMPLIES (AND (LESSP A B)            ;Transitivity of LESSP
              (LESSP B C))
         (LESSP A C))
```

The first formula is a theorem and thus it is "always" true, no matter which values we choose for A and B. In other words, PLUS is a commutative function. If we write it in a more traditional notation:

$$\forall a \forall b \;\; a + b = b + a$$

In the system's notation – since the quantifiers are absent – it can be thought that all formulas are (implicitly) universally quantified over all variables. However, the third formula above is *not* a theorem. Normally, when we say "0 is the right identity of addition," we implicitly restrict our attention to numeric arguments. In general, we do not ask questions like "what is the sum of T and 0?" But in Boyer-Moore language such questions arise and they have answers: the sum of T and 0 is 0, not T. The version of right identity law in Boyer-Moore logic is:

Theorem.
`(IMPLIES (NUMBERP A) (EQUAL (PLUS A 0) A))`

3.2 The Mechanical System

The system includes the theorem proving program which is discussed in the previous section but also provides a large number of features having little to do with theorem proving. These include the mechanization of the shell and definition principles, a run-time environment for executing functions in the logic, and facilities for keeping track of the database of rules.

In this section, we will first introduce how to define functions, adding new data types, and managing the system's database of facts. Other than these, essential features like how to get the system started, how to stop, load, save, etc. will be explained. Next, we will discuss how experienced users interact with the system and how the theorem prover works, with particular attention to those aspects of its behavior under the control of the user.

3.2.1 Introduction

NQTHM uses a variety of high-level derived rules of inference to discover and describe its proofs. The proof techniques that the system uses are the following:

- *Simplification*: Combination of decision procedures, term rewriting, and "metafunctions" (user-supplied simplifiers that have been mechanically proved correct);
- *Destructor Elimination*: Choosing an appropriate representation for the objects being manipulated (to change "bad" terms with "good" ones);
- *Cross-fertilization*: Using heuristics and eliminating equality hypotheses;
- *Generalization*: Adopting a more general goal obtained by replacing terms in the given goal by new variables;
- *Elimination of Irrelevance*: Discarding apparently unimportant hypotheses; and
- *Induction*: Applying mathematical induction on arguments that are selected by the user. The selection of an "appropriate" induction is based on an analysis of the recursive functions mentioned in the conjecture.

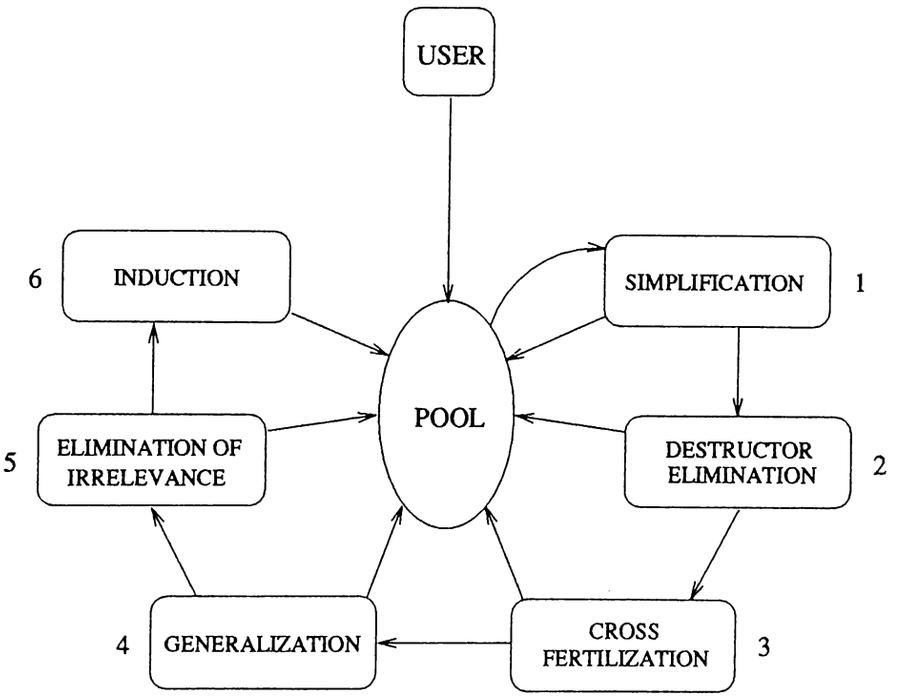

Figure 3.1: Organization of NQTHM

Each of these techniques is an implemented computer program that takes a formula as input and yields a set of formulas as output; the input is provable if each of the outputs is. From a logical point of view, each of these programs is a derived rule of inference that is "run backwards." From a problem solving point of view, each program "reduces" the input goal to a conjunction of subgoals.

The theorem prover is organized around a "pool" of goal formulas (Figure 3.1). Initially, the user places an input conjecture into the pool. Formulas are drawn out of the pool one at a time for consideration. Each formula is passed in turn to the six techniques (programs) in the order they are listed above until some program is applicable. When an applicable program produces a new set of subgoals, these is added to the pool. The consideration of goals in the pool continues until either the pool is empty and no new subgoals are produced, or one of several termination conditions is met.

The first two processes in Figure 3.1 preserve equivalence and are concerned primarily with simplifying formulas. The next three processes strengthen the formula to be used in the induction processes which is the last in the cycle. By the time a formula arrives at the induction process it has been simplified and

generalized as much as possible.

The system is organized as an interactive command interpreter which performs operations on a global database representing the theory. The database contains the axioms, user-defined shells, defined functions, and theorems proved by the system. Some commands add facts to the database. Other commands permit the user to inspect the database or remove facts from the database (with a consequent removal of the logically dependent facts). Finally, there are commands which write the database to a file and restore it from a file.

The prover is fully automatic in the sense that it requires no advice or directives from the user once a proof attempt has started (unless an abort occurs). However, when we view the proof techniques, we can also say that the prover is interactive since the database influences the behavior of the system.

The rules in the database are generated from the theorems formulated by the user and previously proved by the system. When a theorem is submitted for proof, the user attaches to it one or more *tokens* which are taken from the set {REWRITE, META, ELIM, GENERALIZE} that defines what classes of rules are to be generated from the theorem, if proved. For example, the theorem

```
(IMPLIES (NOT (MEMBER A L))
         (EQUAL (DELETE A L) L))
```

generates the REWRITE rule: replace (DELETE A L) by L provided (MEMBER A L) rewrites to F. The equivalent formula

```
(IMPLIES (NOT (EQUAL (DELETE A L) L))
         (MEMBER A L))
```

generates a different rule: replace (MEMBER A L) by T provided (EQUAL (DELETE A L) L) rewrites to F. Propositionally equivalent theorems may have different interpretations as rules.

The key role of the user is to guide the theorem prover to the proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of these theorems.

The act of adding a new rule to the database or changing the status of a rule in the database is called an *event*. Each event has a *name*, chosen by the user, and generally is either the name of a function introduced by the event or the name of the axiom or theorem introduced. The database is actually an acyclic directed graph with events stored at the nodes which are named by the event name. The arcs connecting nodes represent "immediate dependency." The system automatically keeps track of the logical dependencies among events.

3.2.2 How to Interact with NQTHM

Users of the system tend to arrange things so that most proofs are achieved either by induction followed by simplification of each case or simplification alone. Complicated proofs are broken down into lemmas and each lemma is proved independently and stored as a `REWRITE` rule. This is a powerful strategy because the system is more predictable and the user spends less time analyzing unexpected subgoals (to see if the system is on the right path or not). Another advantage is that the user can lead the system to produce the entire proof "automatically" in less time. In addition to these, once a subgoal has been converted to a rule in the database the user will not have to prove it again. The last and the most important advantage is that the resulting database is useful for future problems in the same domain.

To carry out a proof session, a user uses the Lisp interface and a text editor. In this way he manages to remember what he was doing when he aborted a proof and set out a subgoal. In most systems, it is possible to type text into an "edit buffer," evaluate part of the buffer in the Lisp system and send the output to both the terminal and another edit buffer [17].

Using these standard features, a session can be carried out by starting with an empty edit buffer. Then the user has to type in the `PROVE-LEMMA` command for `Example-Lemma`, execute that command, diagnose the need for `SubLemma1`, and abort the proof attempt. Then he types the `PROVE-LEMMA` command for `SubLemma1` into the edit buffer immediately before the command just tried. With consecutive execute-diagnose-abort-type cycles he reaches a point where the last executed lemma succeeds. Then he goes back—similar to pushing and popping from a stack of lemmas—executing each lemma in turn. The final state of the edit buffer contains a successful proof of `Example-Lemma`. This is called the *text editor approach.*

3.2.3 How NQTHM Works

Recall that NQTHM is built out of six processes: simplification, destructor elimination, cross-fertilization, generalization, elimination of irrelevance, and induction. Each process takes a formula as input and returns a set of formulas as output. When one process does not change the input formula, the next process in the list above is tried.

Simplification The simplifier is the most complex process of NQTHM. It is the only process that can return the empty set of formulas and combines many different proof techniques: decision procedures for propositional calculus and equality, a procedure for linear arithmetic over the natural numbers, a term rewriting system, and the application of metafunctions.

Input and output formulas of the simplifier are represented as clauses or lists of literals. The heart of the simplifier is "the rewriter," a term rewrite system which takes as input a term and a set of assumptions and returns a term equivalent to the input, under the assumptions. For example, the definitions `(NOT P) = (IF P F T)` and `(NLISTP L) = (NOT (LISTP L))` are used to rewrite each occurrence of `(NLISTP x)` to `(IF (LISTP x) F T)`.

To simplify a clause the simplifier rewrites each literal under the assumption that the remaining literals are F. In a sense, the simplifier replaces each literal by its rewritten counterpart. However, there are three special cases. If the rewritten literal is `T`, the clause is a theorem since one of its disjuncts is true. If the rewritten literal is `F`, the literal can be dropped from the disjunction. Lastly, if the rewritten literal contains an `IF`-expression, the answer clause is split into two clauses, according to the test of the `IF`.

Elimination of Destructors This process tries to exchange "destructive" terms for "constructive" ones, using rules derived from axioms and previously proved lemmas. Suppose we want to prove a formula of the form `(P A (SUB1 A))`. Let `A` be a positive integer so that it can be represented as `(ADD1 I)`, for some natural number `I`. But then `(SUB1 A)` is `I`. Thus, we can transform `(P A (SUB1 A))` into `(P (ADD1 I) I)`, exchanging the destructive term `(SUB1 A)` with the constructive term `(ADD1 I)`.

Heuristic Use of Equalities To use an inductive hypothesis—which is an equality—we substitute one side of it for selected occurrences of the other side and then "throw away" the hypothesis. This is called *cross-fertilization.* This is a way of generalizing the conjecture being proved. Thus, it is possible for the cross-fertilization process to take a theorem as input and produce a non-theorem as output, i.e., the system can be caused to allow a non-theorem as its subgoal. To avoid this mistake, the equality hypothesis is thrown away *only if* an induction is already done as a prior step. Generally, it is an inductive hypothesis which is thrown away.

Generalization This process replaces certain non-variable terms by variables. For example, the system can generalize

```
(EQUAL (APPEND (REVERSE C) NIL)
       (REVERSE C))
```

by replacing `(REVERSE C)` by the new variable `X`:

```
(EQUAL (APPEND X NIL)
       X)
```

If the output formula is a theorem, then the input formula is also a theorem by instantiation.

Elimination of Irrelevance Irrelevant hypotheses in a conjecture may make it difficult to prove the conjecture by induction. The system has some elementary heuristics for identifying irrelevant hypotheses. It partitions the literals of the input clause into groups whose variable symbols do not overlap and then deletes those literals in the groups that can be falsified by instantiation.

Induction The system attempts to find inductive arguments suitable for a conjecture by analyzing the recursive functions mentioned in it. Each recursion suggests a corresponding induction if the measured argument positions are occupied by variable symbols. For example, the term `(APPEND X Y)` suggests induction on `X` by successive `CDR`s with the base case `(NLISTP X)`, because

`APPEND` recurses in that way on its first argument. The term does not suggest induction on `Y`. Also note that the term `(APPEND (CAR X) Y)` does not suggest any induction since the term in the first argument position, i.e., `(CAR X)`, is not a variable and therefore cannot be instantiated. `Y` is irrelevant since `APPEND` does not recurse on this argument.

Chapter 4

WATERWORKS: The Water-Tank System

Real-time control tasks give rise to programs for which it is difficult to find mathematical specifications. To concentrate on such specification problems, we have chosen the problem of avoiding overflow in water-tank systems. In the next section, we will informally state the problem, and then convert it to formal statements obeying the requirements of the theorem proving system, NQTHM. The final section contains the steps taken to assure that the control programs of the tanks satisfy their high level specifications.

4.1 The Informal Model

In NQTHM, integers are provided, but not the reals. Thus, the quantities we use, i.e., time, amount, inflow, outflow, etc., are all integral units.

To model a water-tank system, we need to handle the problems of a two-tank system (Figure 4.1). In a cascaded tank system, the number of "levels" is two. The outflow of the first tank is an addition to the inflow of the second tank which is somewhat below. However, in a coupled tank system, both tanks are at the same level and connected with a pipe. (For simplicity, we assume that the pipe has no volume.)

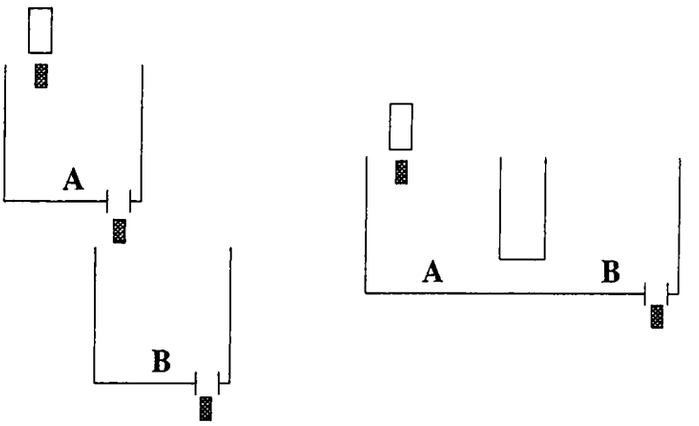

Figure 4.1: Two types of two-tank systems: cascaded and coupled

Figure 4.2 illustrates the architecture of a water-tank. The components are optional and user-supplied tools. Each tank consists of the following parameters:

`Name`	tank number,
`LeftTap`	value of left tap (0 if closed),
`RightTap`	value of right tap (0 if closed),
`LeftValve`	value of left valve (0 if closed),
`RightValve`	value of right valve (0 if closed),
`Separator`	indicator of a separator,
`CoupledTo`	number of the tank that is coupled with this tank,
`CascadedToLeft`	number of the tank that is cascaded below left,
`CascadedToRight`	number of the tank that is cascaded below right,
`CascadedFromLeft`	number of the tank that is cascaded above left,
`CascadedFromRight`	number of the tank that is cascaded above right,
`Inflow`	inflow,
`Outflow`	outflow,
`Amount`	amount.

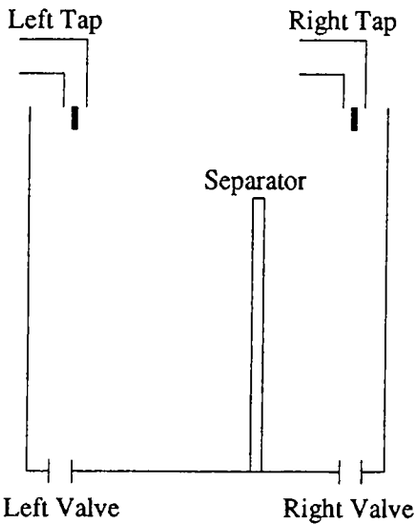

Figure 4.2: A single water-tank and its components

We measure the inflow, outflow, and amount of tank i as follows:

$$\begin{aligned} Inflow_i(t+1) &= LeftTap_i(t+1) + RightTap_i(t+1) + \\ &\quad LeftCascade_i(t+1) + RightCascade_i(t+1) \\ Outflow_i(t+1) &= LeftValve_i(t+1) + RightValve_i(t+1) \\ Amount_i(t+1) &= Amount_i(t) + Inflow_i(t+1) - Outflow_i(t+1) \end{aligned}$$

where $LeftCascade_i$ and $RightCascade_i$ are the outflows of left and right tanks that tank i is cascaded *from*, i.e.,

$$\begin{aligned} LeftCascade_i(t+1) &= Outflow_{CascadedFromLeft_i} \\ RightCascade_i(t+1) &= Outflow_{CascadedFromRight_i}. \end{aligned}$$

Due to the modeling primitives, and characteristics of two-tank systems, the inflow of the tanks in a coupled tank system is adjusted by an additional increment/decrement of 1 unit, if the following condition holds:

$$Amount_i(t+1) > Amount_{Coupled_i}(t+1).$$

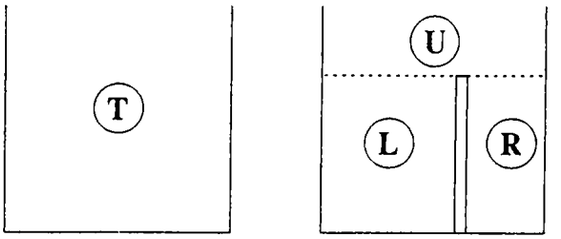

Figure 4.3: Volume of a tank without and with a separator

If the tank has a separator, then there are some situations that need more attention: if the tank is coupled to another one and the tank is filled from the opposite side of the separator, then it takes some time for the flow of water to exceed the separator and reach the pipe. In such a case, the comparison stated above is applied to the partial volumes, not to the total volumes of the tanks. Another case, similar to the previous one, is the one where a separator exists before a valve. Considering the regions that a separator divides a tank into, we modeled all tanks (at most 10 tanks can be used in a sample simulation) with the same example dimensions. Figure 4.3 shows the possible regions of a tank. If no separator exists, then the only volume considered is the total volume `T`, 20 units. Since we model the system in two-dimensions, what we mean by `volume` is actually the area of the region. If there is a separator, then left volume `L` is 9 units, right volume `R` is 6 units, and upper volume `U` is 5 units, adding up to 20 units. (We assume that the separator has no volume.)

At each sampling interval, the amounts of the tanks in the model are calculated concerning their environmental conditions. Controls of overflow are achieved by the valves of the tanks:

$$Valve_i(t+1) = Valve_i(t) + Control$$

where

$$Control = Amount_i(t+1) - OflowVal$$

if $Amount_i(t+1) \geq OflowVal$. $OflowVal$ is the desired value that we want the amount of water not to exceed; in our example it is 20 units.

When we observe the behavior of our system with different models, we can make a conjecture saying that *none of the tanks modeled with positive outflows, i.e., having at least one valve, is subject to an overflow.*

4.2 The User-Interface

The water-tank simulator is coded in C programming language (the source code is approximately 70 Kbytes) using the graphics facilities in an XView environment [16]. A snapshot of the simulator during a simulation is shown in Figure 4.4.

There are two drawing canvases in the main frame. The `TOOLS` canvas is drawn to introduce the user the components, and possible modeling structures of single and two-tank systems. This canvas has no dynamic characteristics. The `SIM-PAD` canvas is the SIMulation PAD, in which the modeling and the simulation take place. In the upper region the number of tanks initialized is shown (maximum 10). In the middle region the modeling and the simulation occur. The lower region is the instruction line. Modeling steps are written in this line in order to assist the user. Error messages also appear here.

Other than the canvases, there is a panel where each button in the panel represents a single command. They are activated by a single click with the left button of the mouse. The functions of these buttons are listed in Appendix A.

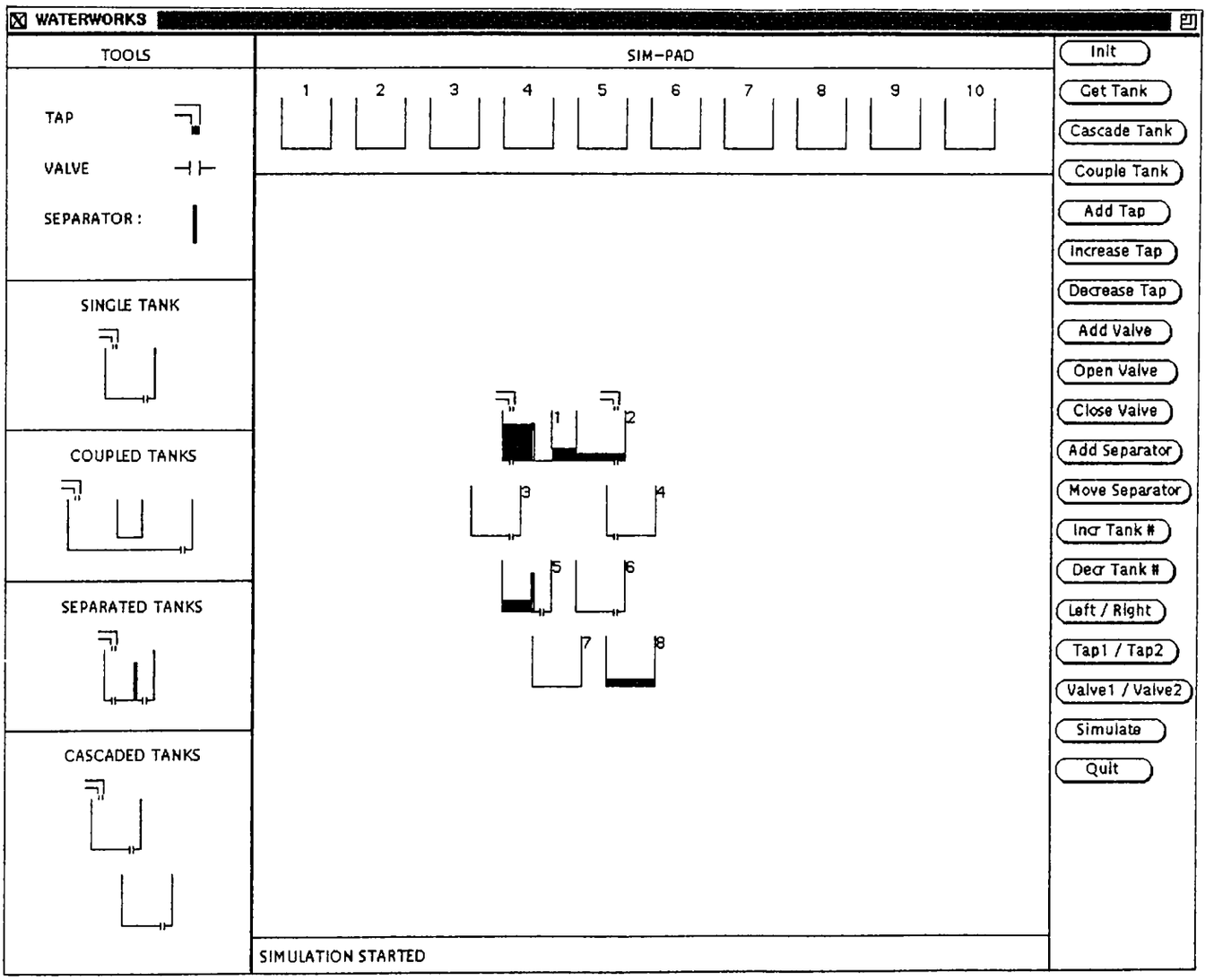

Figure 4.4: User interface of WATERWORKS

4.3 Example Runs of the Simulator

In this section, there are five basic examples. In each example, the behavior of the tanks in the modeled system are plotted. What we mean by "behavior" is the variation of inflow, outflow, and amount of water. The vertical axes show these values whereas the horizontal axes represent the time intervals. The scaling of the axes differ from one graph to the other since they are adjusted according to the variation of the values.

These graphs help the user of NQTHM to guide the theorem prover by strategically selecting the conjecture to prove and the sequence of lemmas required for this proof. So, with a careful analysis of these graphs we decide on the steps needed to take for a successful proof session, in our case the proof of the impossibility of liquid overflow.

The first example (Figure 4.5) contains a single tank. Note the amount of the tank after 10 sampling intervals.

In Example 2 (Figure 4.6), a coupled tank system is simulated. Since the outflow of the tap is greater than 1 (in coupled tank systems, water passing from one tank to the other is 1 unit), the amount of water in both tanks steadily increases with the amount in the first tank 1 unit greater than the second one (Figures 4.6, 4.7).

If the tank has a separator, the outflow becomes positive after the amount of water to the left of the separator exceeds the level of the separator (Figures 4.8).

In Example 4 (Figure 4.9), a cascaded tank system is simulated. The behavior of the tank parameters are given in Figures 4.9 and 4.10.

The last example contains four tanks. Tanks 1 and 2 are coupled and cascaded to tanks 3 and 4, respectively (Figures 4.11, 4.12, 4.13, 4.14).

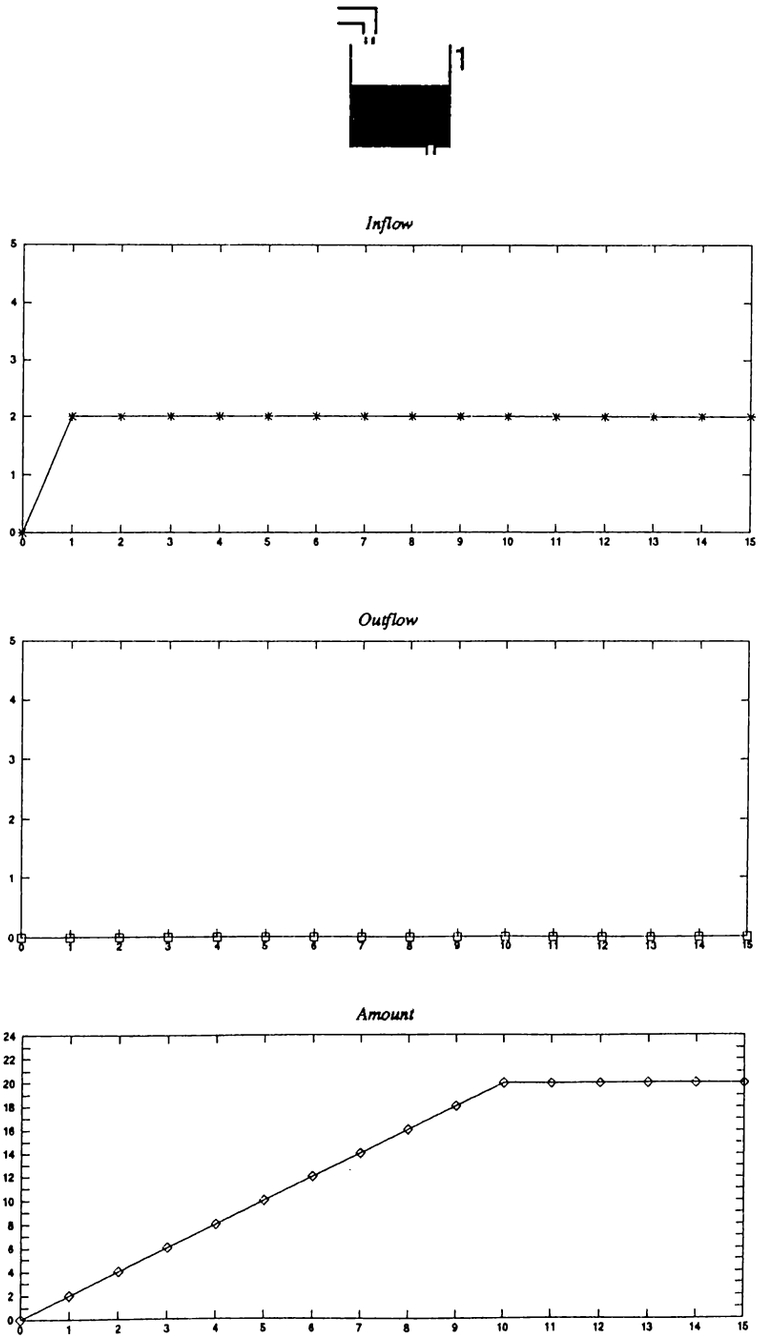

Figure 4.5: Example 1: A single-tank system and behavior of Tank 1

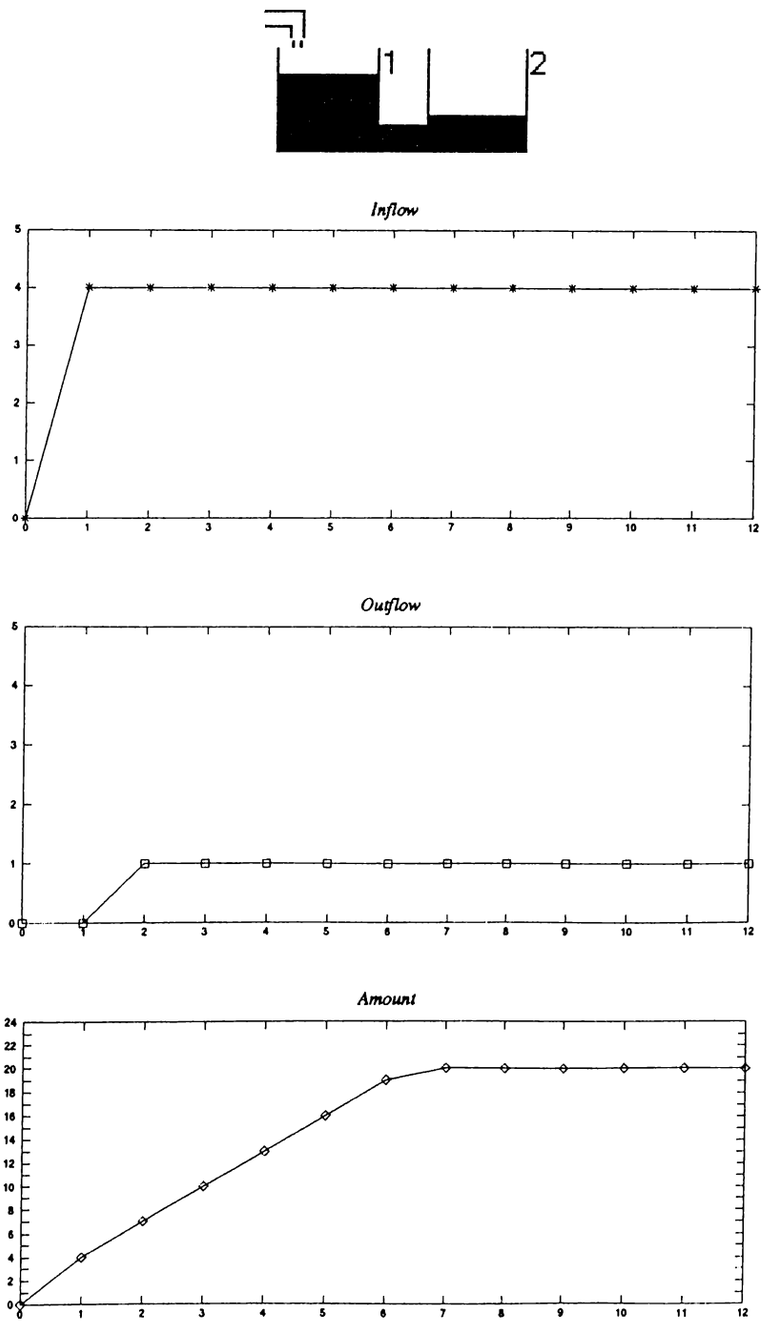

Figure 4.6: Example 2: A coupled two-tank system and behavior of Tank 1

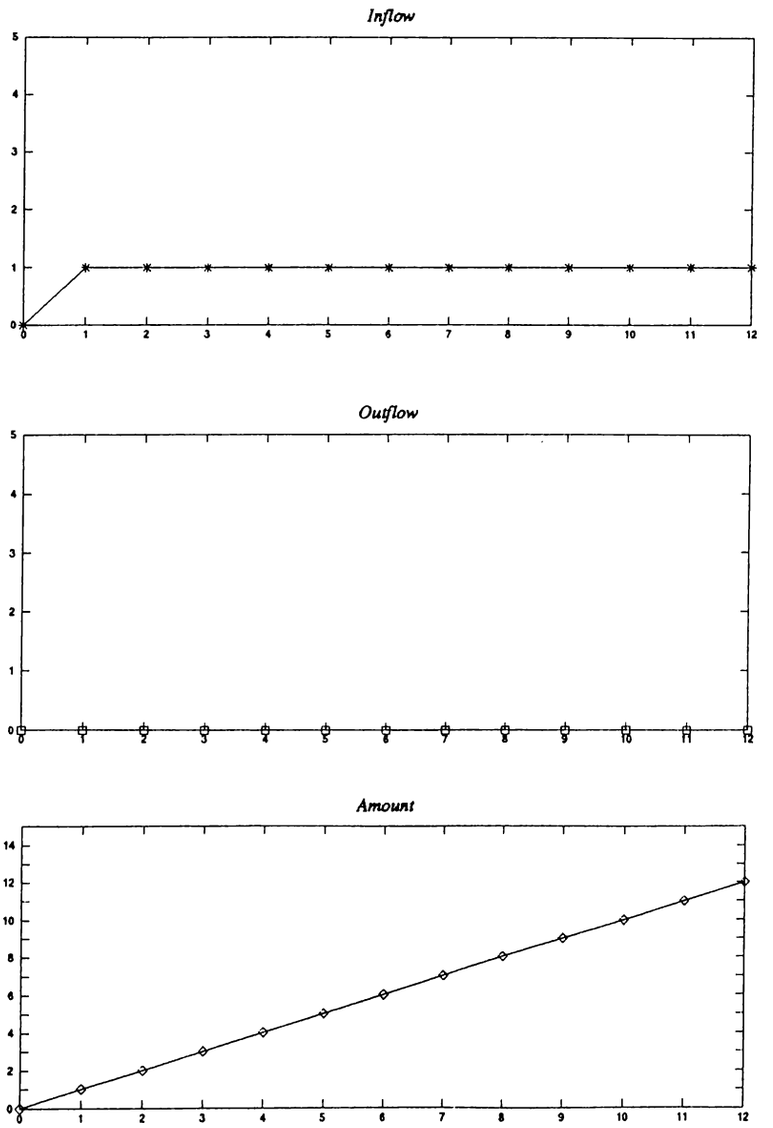

Figure 4.7: Behavior of Tank 2 in Example 2

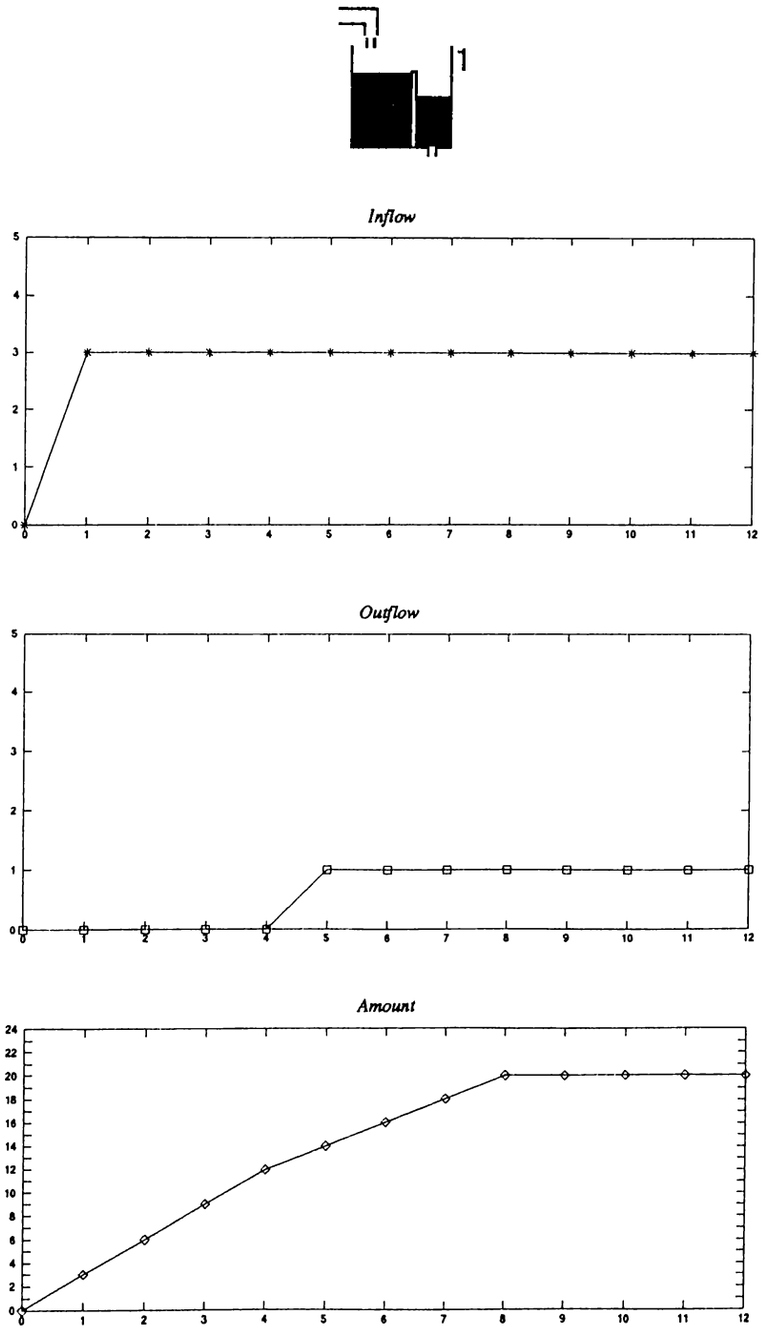

Figure 4.8: Example 3: A single tank with a separator and behavior of Tank 1

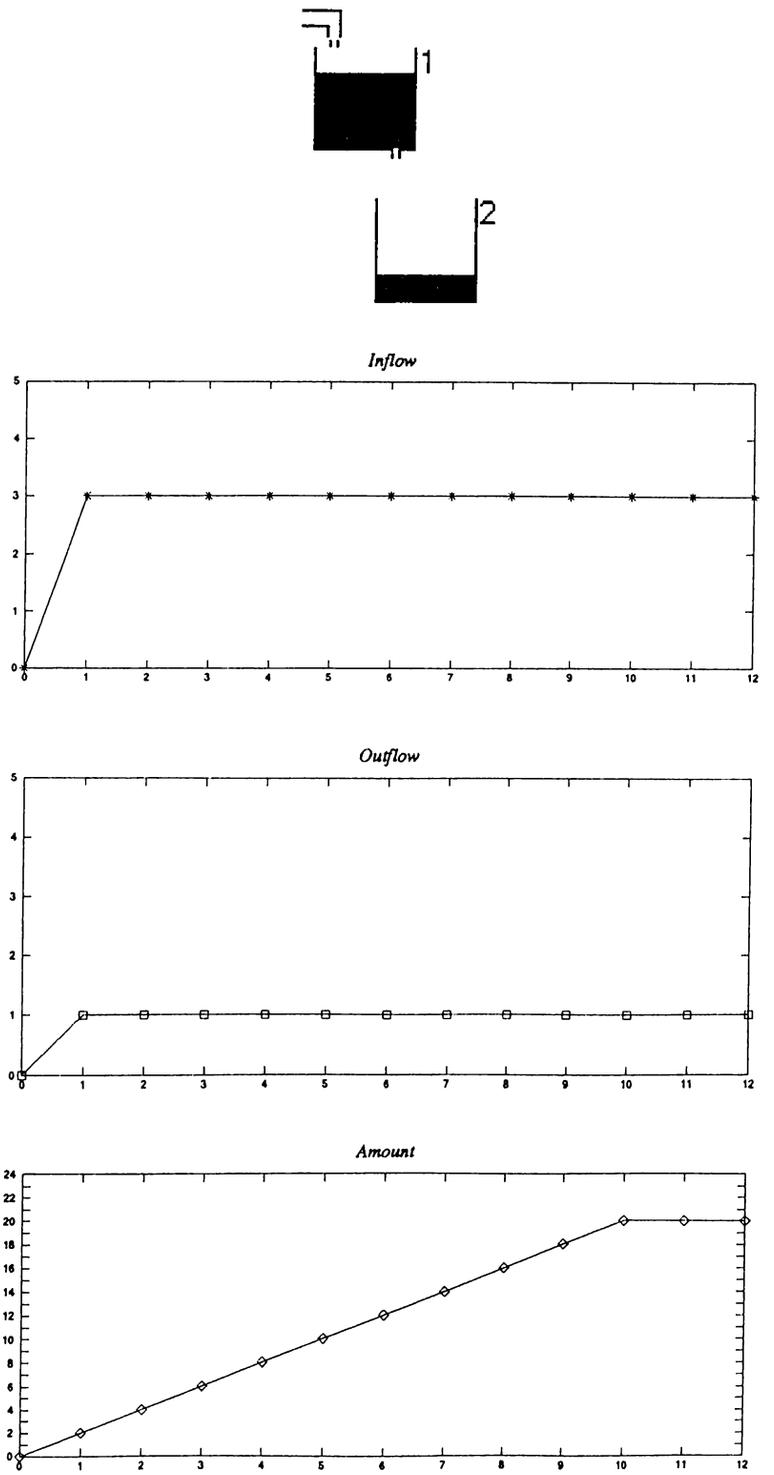

Figure 4.9: Example 4: A cascaded two-tank system and behavior of Tank 1

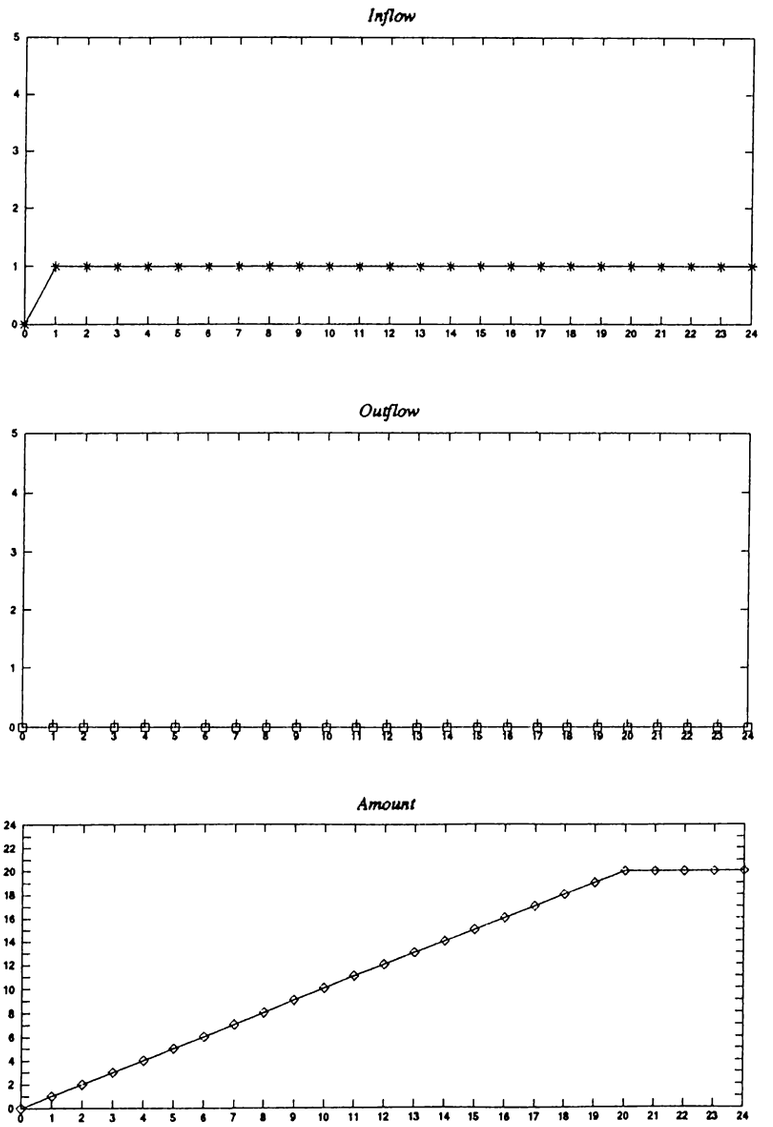

Figure 4.10: Behavior of Tank 2 in Example 4

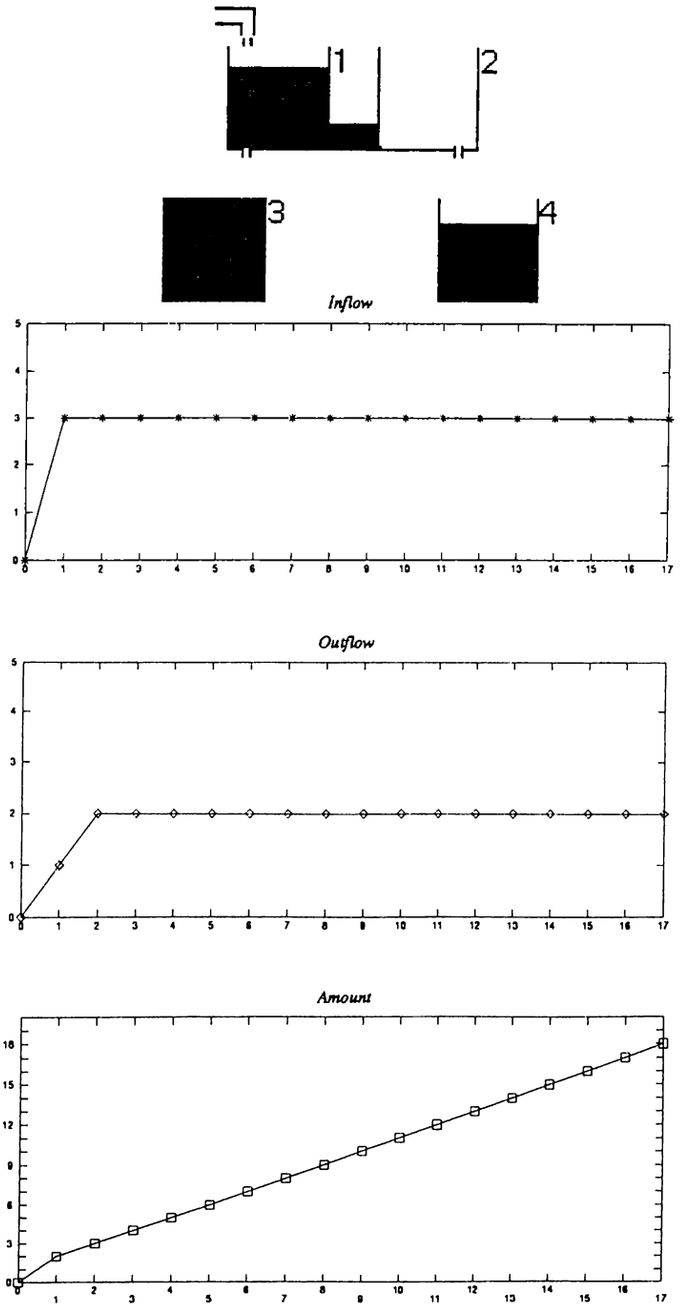

Figure 4.11: Example 5: A four-tank system and behavior of Tank 1

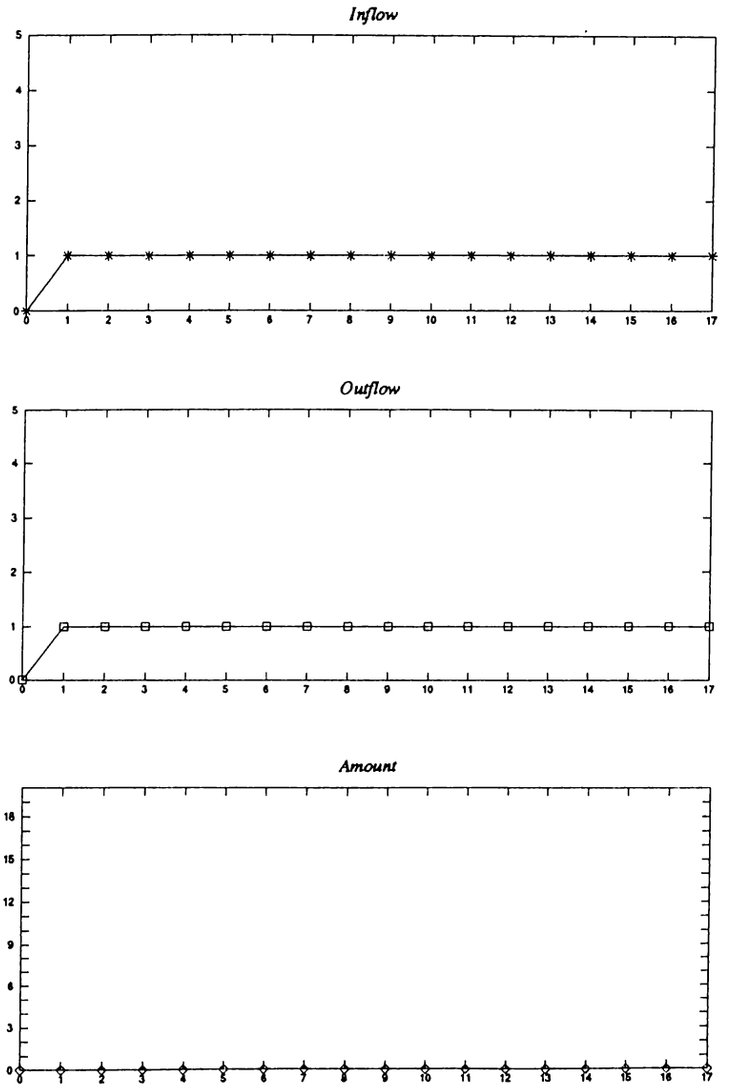

Figure 4.12: Behavior of Tank 2 in Example 5

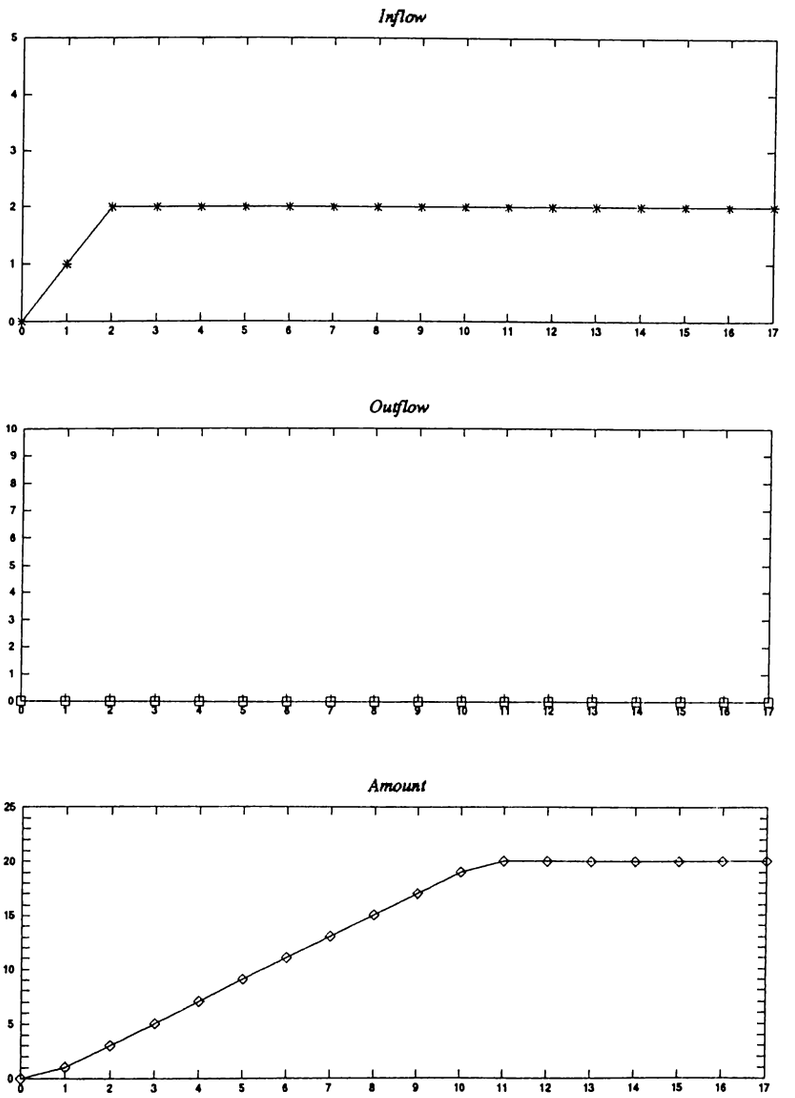

Figure 4.13: Behavior of Tank 3 in Example 5

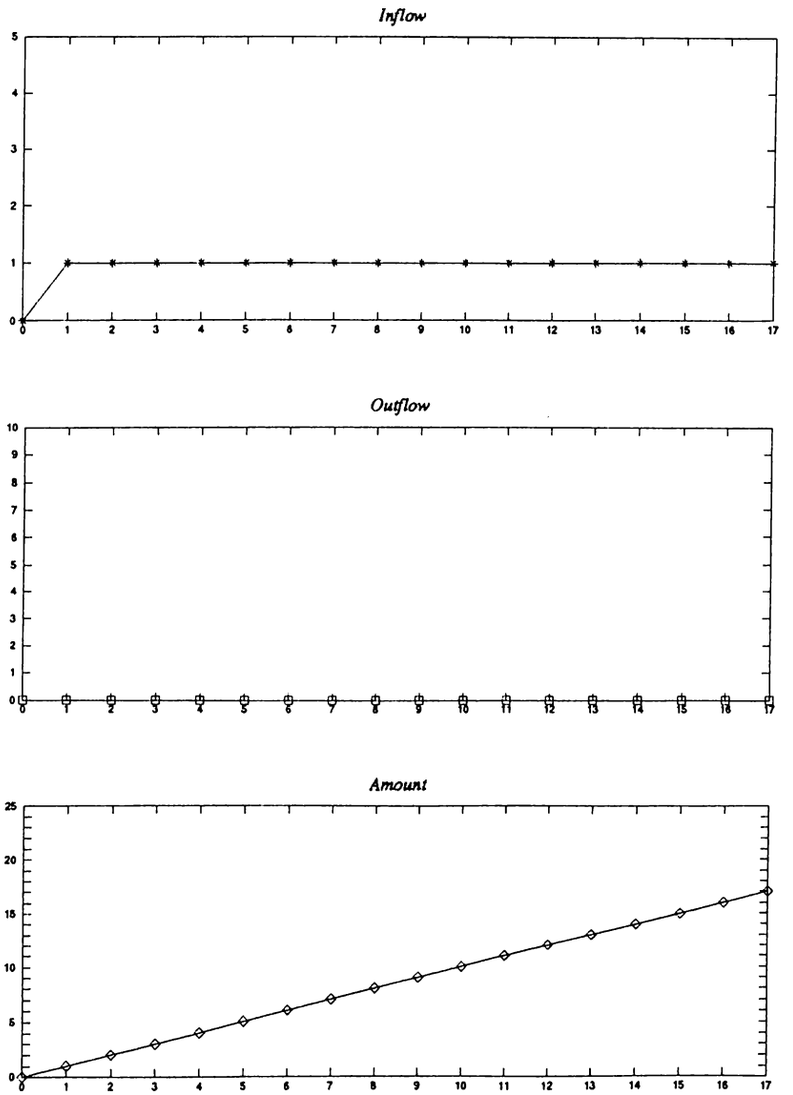

Figure 4.14: Behavior of Tank 4 in Example 5

4.4 Proving Our Conjecture

To state our conjecture that there can never be an overflow, we formalize the control program and the environment in NQTHM syntax. Similar to the introductory examples in which a vehicle is steered down a straightline course in a crosswind that varies with time, our formal model is expressed as a recursive function which takes three arguments: inflow of the tank, outflow of the tank, and an initial state of the tank. This function returns the final state of the tank after it is applied the sequence of $< inflow, outflow >$ pairs under the direction of the control program.

As stated before, we make use of the Shell Principle to define a new data type TANK-STATE which is viewed as the state of an individual tank at any time point:

```
ADD-SHELL TANK-STATE NIL TANK-STATEP
          ((AMOUNT (NONE-OF) ZERO)
           (LEFTVALVE (NONE-OF) ZERO)
           (RIGHTVALVE (NONE-OF) ZERO)))
```

A tank state consists of a triple $< amt, lvalve, rvalve >$ containing the amount of the water in the tank, outflow of the left valve, and outflow of the right valve of the tank, respectively. TANK-STATEP returns T if the input state fits to this declaration. The choice of the parameters are due to the control strategy of the formal simulator. Since the control of overflow is achieved via opening and closing the existing valves through which the liquid flow takes place, we decided to construct the state of a tank in this way. We also include the amount in the tank since the overflow check is applied to this parameter.

Next, we define a control function mapped from the informal model. Having two parameters, the current amount and the upper limit that which the amount should not exceed for an overflow not to occur, the control function is defined as:

```
(DEFN CONTROL (AMT OVERFLOW)
      (ZDIFFERENCE AMT OVERFLOW))
```

Considering the inductive nature of the proof strategy, we need to move

on consecutive tank states after incremental time instances. For this purpose, we define a function NEXT-STATE to return the next state of a tank, given the inflow, outflow, and the current state of that tank:

```
(DEFN NEXT-STATE (INFLOW OUTFLOW STATE)
      (TANK-STATE (ZDIFFERENCE (ZPLUS (AMOUNT STATE) INFLOW)
                               (OUTFLOW))
                  (ZPLUS (LEFTVALVE STATE)
                         (CONTROL (AMOUNT STATE) 20))
                  (ZPLUS (RIGHTVALVE STATE)
                         (CONTROL (AMOUNT STATE) 20)) ))
```

This definition follows from the equations of *Inflow, Outflow, Amount, Valve,* and *Control* which we have informally stated in the previous section.

The behavior of a tank over n time intervals is represented as a list of length n. Each component of this list is another list containing the inflow and outflow values of the tank at that time instance. Since, we formally simulate the system in finite time, i.e., with a finite list, we need to prove that there is no overflow when we reach the end of this list. This is achieved by reaching (computing) the final state of that particular tank. So, we define the function FINAL-STATE recursively so that it takes a list of behaviors and an initial state and returns the final state. With this recursive definition it simulates the state changes after each sampling interval:

```
(DEFN FINAL-STATE (BEHAVIOR-LIST STATE)
      (IF (EMPTY BEHAVIOR-LIST)
          STATE
          (FINAL-STATE (CDR BEHAVIOR-LIST)
                       (NEXT-STATE (CAR (CAR BEHAVIOR-LIST))
                                   (CAR (CDR
                                          (CAR BEHAVIOR-LIST)))
                                   STATE))))
```

BEHAVIOR-LIST is a list of sublists where each sublist consists of two values: inflow of water and outflow of water. These two parameters are enough to determine the others (cf. Section 4.1).

It is the user's responsibility to feed some heuristics to prove the conjectures. We take an approach similar to the heuristic applied in [6]. To prove the conjecture at hand, we divide the state space of a tank into two: valid and invalid states. A tank is in a valid state if the amount of water in the tank is less than or equal to 20. This is a user-supported heuristic to assist NQTHM for discovering whether a state is valid or not. We write the function `VALID-STATEP` for this purpose:

```
(DEFN VALID-STATEP (STATE)
      (IF (LEQ (AMOUNT STATE) 20)
          T
          F))
```

With the rewrite lemmas, we can form a suitable basis for our conjecture, since the rewriter works in such a way that once a rewrite lemma is successfully proved, it stores this lemma in the database for future use. So, we command the theorem prover a rewrite lemma to prove the situation in which if a state is valid, then the next will also be valid within the parameters of the model:

```
(PROVE-LEMMA NEXT-VALID-STATE (REWRITE)
             (IMPLIES (AND (VALID-STATEP STATE)
                           (NOT (EMPTY BLIST)))
                      (VALID-STATEP (NEXT-STATE
                                     (CAR (CAR BLIST))
                                     (CAR (CDR (CAR BLIST)))
                                     STATE)))))
```

As stated above, we need to induct on the behavior of a tank given the current state of the tank. Thus, we formally *simulate* our system at an individual tank level. Therefore, to step over the behaviors of the tank, we implement another recursive function:

```
(DEFN GET-NEXT (BEHAVIOR-LIST STATE)
      (IF (EMPTY BEHAVIOR-LIST)
          STATE
          (GET-NEXT (CDR BEHAVIOR-LIST)
                    (NEXT-STATE (CAR (CAR BEHAVIOR-LIST))
                                (CAR (CDR (CAR BEHAVIOR-LIST)))
                                STATE))))
```

The main problem for the proof of our conjecture is to construct a structural induction (we force the theorem prover for induction, since the nature of our problem requires such an induction) over the behavior list. This can be achieved by another rewrite lemma:

```
(PROVE-LEMMA ALL-STATES-VALID (REWRITE)
             (IMPLIES (AND (NOT (EMPTY BLIST))
                           (VALID-STATEP STATE))
                      (VALID-STATEP (FINAL-STATE BLIST STATE)))
             ((INDUCT (GET-NEXT LIST STATE))))
```

In other words, if the behavior list of the tank is not empty (the tank shows a behavior in time) and the current state is a valid state (the amount of water is less than or equal to 20) then the final state of the tank should be valid, required that we induct on its behaviors.

We can now formally state our conjecture:

```
(PROVE-LEMMA THERE-IS-NO-OVERFLOW NIL
             (IMPLIES (AND (NOT (EMPTY BLIST))
                           (EQUAL STATE
                                  (FINAL-STATE BLIST
                                               (TANK-STATE 0 0 0))))
                      (LEQ (AMOUNT STATE) 20)))
```

If BLIST is the history of behavior of a tank and tt STATE is the state of the system after the tank has faced this behavior history starting from the initial state <0,0,0>, then the amount of water in the tank in state STATE is less than or equal to 20 units.

After proving the rewrite lemma NEXT-VALID-STATE, NQTHM establishes that the final state of the tank is also valid. From this conclusion, THERE-IS-NO-OVERFLOW follows.

For the system modeled in water-tank simulator, each of the behavior list of the tanks can be subjected to this type of proof. This is an implemented example, but the use of NQTHM is not limited only to this type of examples. For different systems modeled in WATERWORKS, various conjectures can be formalized and proved. For instance, a typical one would be "the amount of water in the tank is 0" (refer to tanks 3, 4, 6, and 7 in Figure 4.4.)

4.5 Issues Related to Qualitative Reasoning

Qualitative Reasoning is an area of research aiming to provide a purely qualitative approach to problem solving in domains which have a standard quantitative treatment (such as physics) [10]. Most of the time, we express the problems we face in qualitative terms, even though the problem might traditionally be described using continuous real-valued parameters and differential equations which might then be solved analytically or numerically. However, it is clear that one does not need to have a knowledge of differential equations in order to be able to provide answers at the level of abstraction. Of course, one might be able to provide an answer purely on the basis of experience, i.e., inductively. But, to explain why the answer is correct would seem to require a deeper, structural model of the system and the ability to reason about such a model.

Describing a physical system by the exact quantitative values of its variables, although complete, fails to provide sufficient explanation of how the system functions [18]. *Qualitative physics* is an alternative physics in which the physical concepts are defined in a simpler yet formal basis [9, 10]. In classical physics, the characterization of physical change is defined within a nonmechanistic framework and thus is difficult to adapt to commonsense interpretation. Qualitative physics provides an alternative way of arriving, when possible, at the same conclusions and suggests a simpler basis for reasoning about physical mechanisms [11]. This commonsense theory of physics tries to describe and systemize physical phenomena as they appear and as they may most effectively be thought about for everyday purpose. The problems addressed in qualitative

physical reasoning include predicting the future history of a physical system, planning physical actions to carry out a task, and designing tools to serve a given purpose.

Broadly speaking, there are three main approaches to qualitative reasoning. Perhaps the simplest from an ontological view is the so-called *constraint-centered* approach in which an entire system is described by a homogeneous set of constraints; typical of such an approach is the work of Kuipers [14, 15]. A more structured approach is the *component-centered* approach where a system is modeled by instantiating components from a library which are then connected together explicitly. The work of de Kleer and Brown [8] is a good example of this approach. WATERWORKS may also be viewed as an example of this approach since the components (e.g., taps, valves, pipes) are used in a similar manner. Finally, the *process-centered* approach builds on the component-centered approach by modeling not only individual components explicitly but also processes which act on them. Forbus' work [9] examplifies this approach.

One of the typical issues addressed in physical reasoning systems is the modeling of liquids or systems concerning liquid flows. Liquids present a tricky ontological problem since they may not be individuated. Hayes [12] proposes to individuate liquids by their containers and develops a physics of liquids in these terms. The problem is hard because in a formal and propositional representation liquids are denoted by mass terms and the semantics of mass terms has been the subject of much philosophical debate. Moreover, liquids can divide, combine, and deform with incredible ease compared to solid objects.

In WATERWORKS, although a numerical simulation is run, in the level of abstraction qualitative notions are also used. For instance, the optional separator in the tanks is assumed to have no volume. Similarly, in coupled tank systems, the connecting pipes have no volume either. Such assumptions, although not so difficult to relax, are immaterial in proving that no overflow occurs. It causes no unsurmountable difficulty to assign say, a volume to a pipe.

WATERWORKS, concerning the modeling primitives, is based on Hayes' *contained space ontology* [12]. Thus, qualitative modeling issues play an important role. Moreover, the analysis and observation of the behavior plots enable us to state qualitative theorems. One can realize that both the strategy and

the formalization in NQTHM (for our application) consist of no numerical formulations. In addition to that, the structure and the proof steps can be viewed to run and conclude in a commonsense manner. All users of the system view the notions of liquid flow and overflow in the same way. No two people think in a different way upon observing the increase in the inflow of water (with a stable outflow) causes the amount of water in the tank to increase.

Qualitative reasoning seems to provide a promising approach especially because it gives one the ability to reason with incomplete information [1]. It can function with incomplete models or data whenever this information is not easily available. Qualitative simulation is more efficient than numerical simulation since it can be implemented with straightforward constraint satisfaction [15]. The need for qualitative simulation arises from the fact that the behavior of realistic models under varying conditions is complex, and an exhaustive simulation to detect all important behaviors can be expensive. Via qualitative simulation, useful approximations of model behaviors are obtained at a reduced cost. Moreover, these approximations can often be expressed and explained more easily.

There is certainly much work to do in qualitative reasoning, especially at a theoretical level, such as developing and extending qualitative reasoning methods to cope with a wider range of problem types. In addition to that, the interface between qualitative and quantitative reasoning paradigms needs to be researched since qualitative reasoning is seen as a complement rather than a replacement for quantitative reasoning. Finally, we might remark that further research must take into account shape and space in addition to time since many complex systems require sophisticated spatial reasoning.

Chapter 5

Conclusion

In this work, we showed how formal verification can be used to satisfy the high level specification of the control of a simple real-time system. We have also illustrated how such control programs can be mechanically proved to match the informal design of such a system.

Within the limits of our problem domain, we restricted the liquid that flows through the tanks to be water. Concerning application areas like fuel-oil tanks, chemical factories, etc., WATERWORKS might be viewed as a prototype and can be extended to have tanks containing various kinds of liquid with different viscosity parameters and other physical properties.

Development of a generator to obtain the verification conditions is crucial. This tool can be used to eliminate the work of the user who interfaces the behavioral facts of the simulator to the formal statements of the program verification system (in our case, a theorem prover).

Particular to our system, some heuristics can be developed to achieve more efficient runs. Such heuristics can then be generalized to be used in different systems.

In our system, the control is mainly applied vis-à-vis overflow situations. An extension might be to control the amount at a user-specified value or adjust the level of water in each tank so that the level of water never exceeds or gets lower than the specified boundaries.

A final remark is due to the user interface of the simulator. Instead of having a single message line and panel buttons as the only command activators, liberal

use of other facilities like command lines and pop-up menus would certainly help. A better idea might be to use an object-oriented tool for simulation so that each component of the water-tank system, i.e., tank, tap, valve, and separator, can be accessed and used in an easier fashion.

Appendix A

Command Summary of WATERWORKS

Below is a list of commands to model and simulate a water-tank system.

- INIT: initializes the available tanks together with the drawing tools including the frame, drawing canvases, and the simulation pad.
- GET-TANK: creates the water-tanks.

There are two commands for developing two-tank structures:

- CASCADE-TANK: cascades two tanks by appropriately checking the topological constraints. Automatically adds a valve to the first tank which hierarchically occurs at the upper level.
- COUPLE-TANK: couples two tanks by appropriately checking the topological constraints.

Commands related to the addition and removal of the tools:

- ADD-TAP: attaches a tap to the current tank with the default amount of inflow, i.e., 1 unit.
- ADD-VALVE: attaches a valve to the current tank with the default amount of outflow, i.e., 1 unit.
- ADD-SEPARATOR: attaches a separator to the current tank.

- MOVE-SEPARATOR: moves the separator from the current tank.

There are some commands that regulate the outcome of the tools, namely taps and valves:

- INCREASE-TAP: increments the inflow from a tap (upper bound 2 units).
- DECREASE-TAP: decrements the inflow from a tap (lower bound 0 equivalent to removing the tap).
- OPEN-VALVE: increments the outflow of a valve (upper bound 2 units).
- CLOSE-VALVE: decrements the outflow of a valve (lower bound 0 equivalent to removing the valve).
- SIMULATE: starts the simulation and stores the characteristics of each tank, viz. inflow, outflow, and amount.

In addition, there are some commands which allow the user to switch between tanks, to change the direction for the attachment of tools, and to adjust the order of tank coupling and tank cascading:

- INCR-TANK #: increments the current tank number.
- DECR-TANK #: decrements the current tank number.
- LEFT/RIGHT: changes the current direction (default Left).
- TAP1/TAP2: changes the current tap (default Tap1).
- VALVE1/VALVE2: changes the current valve (default Valve1).

Finally,

- QUIT: ends a session.

Appendix B

Availability of WATERWORKS

WATERWORKS is used to simulate liquid flow in simple water-tank systems that are modeled by forming various two-tank systems and connecting them via topological constraints to obtain larger systems.

It is coded in the C programming language using the drawing tools of XView [16] on a Sun SPARC. The source code is about 70 kbytes. To become familiar and run example simulations with the system, access the directory

`/home/usr3/erkan/WATERWORKS`

and follow the instructions in the file `README`. Similarly, interested users of NQTHM should access the directory

`/home/usr3/erkan/nqthm`.

References

[1] Akman, V., and ten Hagen, P. J. W. The Power of Physical Representations. *AI Magazine* **10**:49–65, 1989.

[2] Akman, V., and ten Hagen, P. J. W. Fronti Nulla Fides (No Reliance can be Placed on Appearance). Letter to the Editor. *AI Magazine* **11**:10–11, 1990.

[3] Boyer, R. S., and Moore, J S. *A Computational Logic*, New York, NY: Academic Press, 1979.

[4] Boyer, R. S., and Moore, J S. Proof-Checking, Theorem Proving, and Program Verification. *Contemporary Mathematics* **29**:119--132, 1984.

[5] Boyer, R. S., and Moore, J S. *A Computational Logic Handbook*, San Diego, CA: Academic Press, 1988.

[6] Boyer, R. S., Green, M. W., and Moore, J S. The Use of a Formal Simulator to Verify a Simple Real Time Control Program. *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra.* ed. W. H. J. Feijen, 54–66, New York, NY: Springer-Verlag, 1990.

[7] Davies, R., and O'Keefe, R. *Simulation Modeling with Pascal*, London, UK: Prentice-Hall, 1989.

[8] de Kleer, J., and Brown, J. S. A Qualitative Physics Based on Confluences. *Artificial Intelligence* **24**:7–83, 1984.

[9] Forbus, K. Qualitative Process Theory. *Artificial Intelligence* **24**:85–168, 1984.

[10] Forbus, K. Qualitative Physics: Past, Present, and Future. *Exploring Artificial Intelligence*, ed. H. Shrobe, 239–296, San Mateo, CA: Morgan Kaufmann, 1988.

[11] Hayes, P. The Second Naive Physics Manifesto. *Formal Theories of the Commonsense World.* eds. J. R. Hobbs and R. C. Moore, 1–36, Norwood, NJ: Ablex, 1985.

[12] Hayes, P. Ontology for Liquids. *Formal Theories of the Commonsense World.* eds. J. R. Hobbs and R. C. Moore, 71–107, Norwood, NJ: Ablex, 1985.

[13] Hayes, P. The Naive Physics Manifesto. *The Philosophy of Artificial Intelligence.* ed. M. A. Boden, 171–205, New York, NY: Oxford University Press, 1990.

[14] Kuipers, B. Commonsense Reasoning about Causality: Deriving Behavior from Structure. *Artificial Intelligence* **24**:169–203, 1984.

[15] Kuipers, B. Qualitative Simulation. *Artificial Intelligence* **29**:289–388, 1986.

[16] Nye, A. *XView Programming Manual*, Sebastopol, CA: O'Reilly & Associates. Inc., 1990.

[17] Steele, G. L. *Common Lisp: The Language*, Burlington, MA: Digital Press, 1990.

[18] Uçar, E., and Akman, V. Modeling an Oscillator: A Case Study. *Proceedings of Sixth International Symposium on Computer and Information Sciences (ISCIS-VI)*, eds. M. Baray and B. Özgüç, Volume II, 671–679, Amsterdam, Holland: Elsevier, 1991.

[19] Uçar, E., and Akman, V. Proof-Checking Process-Based Reasoning about Physical Systems. *Proceedings of Symposium on Advances in Simulation*, eds. A. R. Kaylan and T. İ. Ören, 47–58, İstanbul: Boğaziçi University Publications, 1992.

[20] Weld, D. S., and de Kleer, J. (eds.) *Readings in Qualitative Reasoning about Physical Systems*, San Mateo, CA: Morgan Kaufmann, 1990.

[21] Widman, L. E., Loparo, K. A., and Nielsen, N. R. (eds.) *Artificial Intelligence, Simulation, and Modeling*, New York, NY: John Wiley and Sons, 1989.

[22] Wos, L., Overbeek, R., Lusk, E., and Boyle, J. *Automated Reasoning: Introduction and Applications*, Englewood Cliffs, NJ: Prentice-Hall, 1984.