ABSTRACT

Title:	Modeling, Validation and Verification of Concurrent Behavior in the Panama Canal System Using LTSA and UPPAAL
	Maliheh Poorfarhani. Master of Science, 2005
Directed By:	Associate Professor Mark Austin, Department of Civil and Environmental Engineering and Institute for Systems Research

A complex system is composed of subsystems. Common sense dictates complex systems in the physical world are reactive and concurrent in nature. The procedure to model a concurrent system and the procedure to validate its performance are complex because of the presence of the interactions between and among the subsystems, calling for an integrated approach from the viewpoint of systems engineering. In this thesis, a canal system is the object of study. The main objectives are to achieve a deadlock free and safe architectural model of a canal system, as measured by transportation criteria. Specifically, the Panama Canal is used as a case study where a procedure has been developed to model this waterway. This thesis models the scenario-based specifications, system behavioral model, animated verification and validation of the Panama Canal with LTSA and with the help of UPPAAL brings time constraints into the canal behavioral model.

MODELING, VALIDATION AND VERIFICATION OF CONCURRENT BEHAVIOR IN THE PANAMA CANAL SYSTEM USING LTSA & UPPAAL

By

Maliheh Poorfarhani

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2005

Acknowledgements

I would like to express my most sincere gratitude to Dr. Mark Austin, my academic advisor under whose supervision this thesis was written.

I would also like to thank Noosha Haghani, my friend and fellow systems engineering graduate student at the University of Maryland and Evangelos Kaisar for their sincere help, collaboration, and support.

I would also thank my defense committee members, Dr Eyad H Abed and Dr. Guangming Zhang for their helpful comments and feedbacks.

Special thanks to my friends Nargess Memarsadeghi, Mehdi Kalantari Khandani, Banafsheh Keshavarzi, Pedram Hovareshti, Shabnam Tafreshi, Anahit Samadi, Banafsheh Sadeghi, Parvin Asadzadeh, Shabnam Shafiee, Nima Mesgarani, Nasim Vakili, Behnam Nikzad, Amiali Sharifi, Maryam Farbodi, Maria Striki, Mohammad Haeri Kermani, and all the people who helped and supported me finish this thesis.

Especially, I would like to give my thanks to my beloved husband, Houman whose help; patience, love and support enabled me to complete this work.

Finally I want to thank my lovely parents and my family back home, who always supported me to continue my education, especially my lovely Mom whose weekly calls always motivated me to make my best effort. I dedicate this thesis to her without whom I would not be where I am, mersi Maamaan!

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Thesis overview

This thesis is concerned with the modeling, validation and verification of concurrent behavior in canal systems through use of the labeled transition system analyzer (LTSA)[2] and UPPAAL, a formal model checking software developed in Denmark. During the past decade, accidents [1] in narrow canal system have created a crucial need for behavior modeling procedure that can accurately predict behavior of the ship movement and can control the canal operations. In the following sections we will give a brief introduction to the characteristic of the concurrent systems and then we talk about the canal system structure and behavior.

Highly Concurrent Systems

Concurrency is the execution of two or more independent, interactive tasks over the same period of time; their execution can be interleave or even simultaneous. [2] Concurrency is used in many kinds of systems, both small and large. Concurrency in large systems causes a lot of parallel executions, which makes the system highly concurrent [3]. The typical home computer is a great example. In separate windows, a user runs a web browser, a text editor and music player all at once. The operating system interacts with each of them. The operating system waits for the user to start and stop programs, while also handling underlying tasks such as resolving what information is transmitted to or from the internet and how that information goes to which specific programs. Modeling of highly concurrent systems is challenging because it requires time management (Scheduling and coordination) many information flows at once [4].

Canal System

To better understand the work of this thesis, a description of the canal operation and structure must be given. The canal system is a two-lane waterway mechanism used to allow the transportation of ships. Figure 1 shows top and elevation views of the Panama Canal; our canal system also is very similar to Panama Canal. As shown there are two series of locks at both sides of the lake [9] and ships have to pass through the narrow waterway between the locks. The Panama Canal is approximately 80 kilometers long between the Atlantic and Pacific Oceans [9]. This waterway was cut through one of narrowest saddles of the isthmus that joins North and South America. The Canal uses a system of locks -compartments with entrance and exit gates. The locks function as water lifts: they raise ships from sea level to the level of Lake (26 meters above sea level); ships then sail the channel through the Continental Divide [9]. Lock chambers -steps-- are 33.53 meters wide by 304.8 meters long. The maximum dimensions of a ship that can transit the Canal are: 32.3 meters in beam; draft -their depth reach- 12 meters in Tropical Fresh Water; and 294.1 meters long (depending on the type of ship)[9].

The water used to raise and lower vessels in each set of locks comes from the lake by gravity; it comes into the locks through a system of main culverts that extend under the lock chambers from the sidewalls and the center walls.

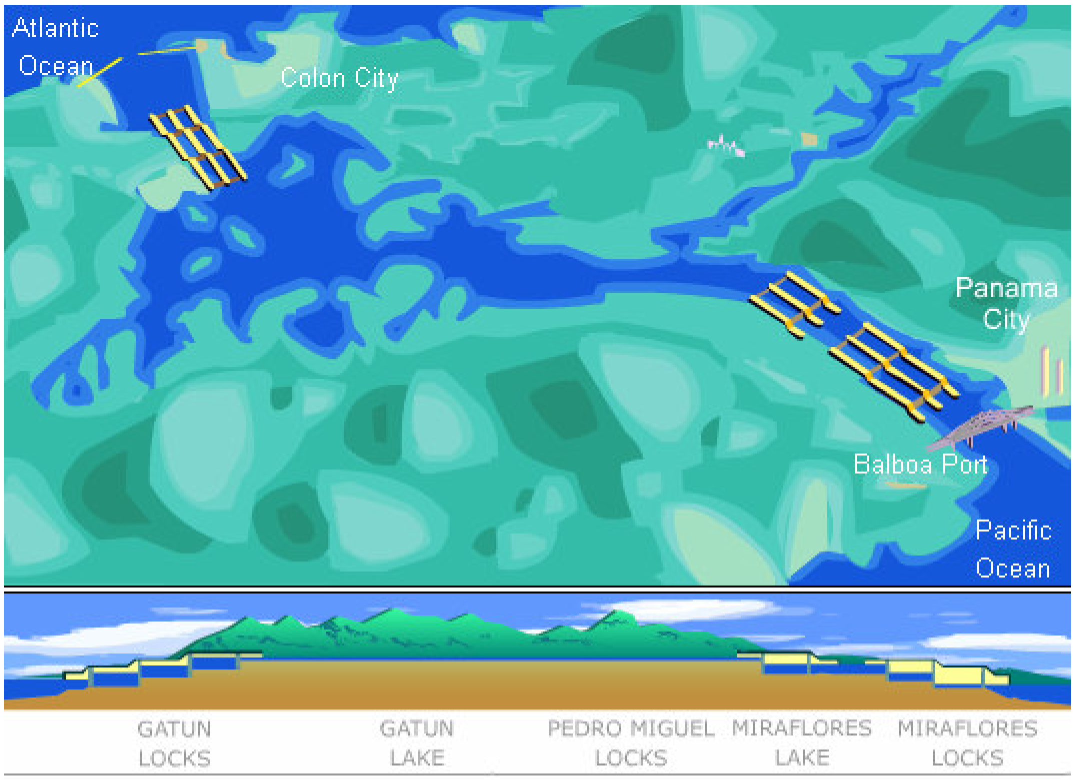


Figure 1. Plan and elevation views of the Panama Canal System[9].

The main components of the canal system are shown in Figure 2. Apart from ships, the canal system is composed of locks, which in turn, have gates and valves for raising and lowering water level. The entrance and exit of the canal contains three sets of locks that hold the ships. The gates between locks separate them from each other, also to control the water level of each lock there is a valve assign to each lock. As a ship approaches the first chamber, valves below the compartment are released and the water level reaches that of the outside of the canal. The gates then open and the ship moves into the first chamber. After the gates lock, valves of the first and second chamber are opened to allow the water level of the first chamber to rise and match the second chamber. The gates are again opened and the ship moves to the second chamber. This process continues in the third chamber until the ship approaches the lake. The ship will then travel through the lake to the opposite side of the canal, making sure not to crush into oncoming ships. Again, the process of raising the water level and unlocking the gates is repeated through three sets of locks/compartments until the ship exits the canal completely. While it is certainly feasible to operate the canal system with one ship passing through the lock system at any time, economic issues dictate that the canal system be operated with multiple ships. Coordinating and scheduling behaviors of the ships, gates and valves needs to be carefully designed to avoid the accidents (e. g. a gate opens before the water level is lowered) and guarantee that ships will pass through the canal system in a reasonable time .The control center of the canal system controls the scheduling and coordination of concurrent behaviors through the canal system.

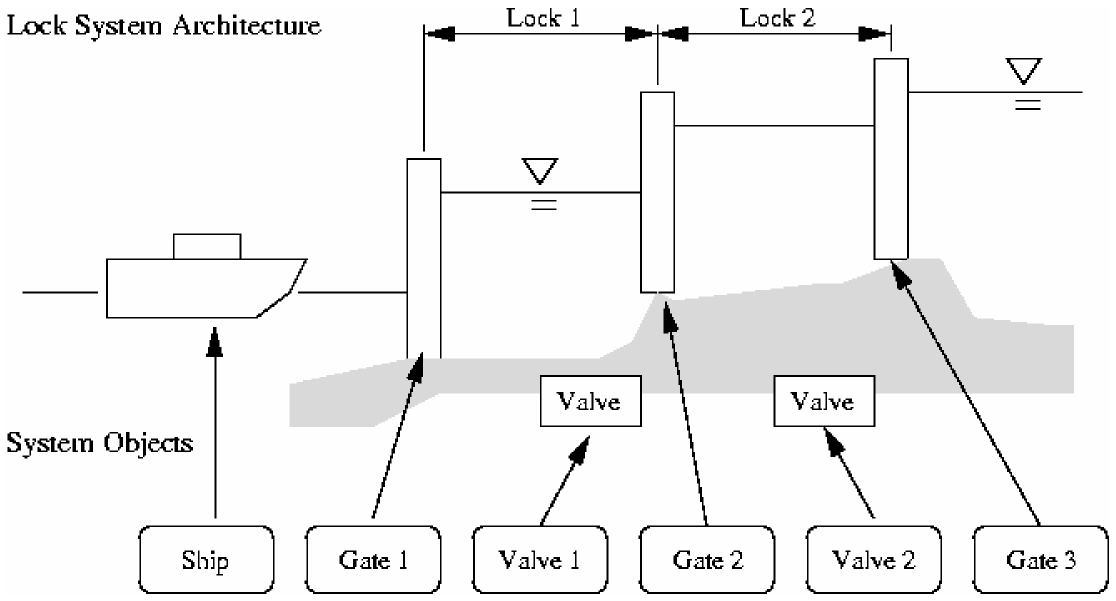


Figure 2. Main Components of the Canal System.

<u>Historical statistics of the Panama Canal Operation</u>

The Panama Canal Authority (ACP) is an autonomous agency of the Government of Panama. Since the handover of the Panama Canal on December 31, 1999, the ACP has shifted its operations from a profit-neutral utility to a market-oriented business model – one that is totally focused on customer service and reliability [9].

The importance of the Panama Canal continues to grow because of increased trade Asia, particularly in goods transported in containers (many exports are containerized these days). The "All-Water Route" (from Asia through the Panama Canal to the East Coast of the United States and back) is one of the fastest-growing worldwide trade routes and is promoted through Memorandums of Understanding (MOU) between the Canal and U.S. East Coast port authorities (recently renewed this year [9]). More Panamax-sized vessels (the largest vessel that can transit the Canal) are using the Canal today, carrying more cargo than ever. As these trends continue, the ACP remains committed to its customers and stakeholders, the Panamanian people, working every day to enhance the waterway's safety, efficiency and reliability but they are still big issues in terms of money and time.

In the following we list the important issues in Panama Canal management system:

Reliable: Canal Waters Time (CWT), the average time it takes a vessel to navigate the Canal, including waiting time for passage, decreased 20 percent from 32.9 hours in 1999 to 26.7 hours in 2004.

Managing demand: Panama Canal/Universal Measurement System (PC/UMS) tonnage increased 17 percent from 227.5 million PC/UMS tons in for year1999 to 266.9 million PC/UMS tons in 2004.

Safety: Accidents can be attributed to:

(1) A steady increase in the size and number of ships passing through the canal;

(2) Increased work hours of the ship pilots, and

(3) A lack of maintenance on the canal infrastructure.

Lanes in the Pedro Miguel, Miraflores and Gatun locks are 110 feet wide, and it is not unusual for a vessel to have only two feet of clearance on either side. As shown in Figure 3, the number of official accidents decreased from 28 in 1999 to 10 in 2004. The actual number of accidents is significantly higher. For an accident to be classified as "official" and for the vessel to collect damages, it must stay for an investigation. A typical investigation takes 24 hrs. So, in practice, this only happens if the cost of the incident significantly exceeds the cost of delay.

Panamax vessel transits: Panamax vessel transits increased 27 percent, from 4,198 transits in for year1999 to 5,329 in 2004.

Direct contributions to Panamanian Government: In five years the ACP has contributed a total of $1332.3 million.

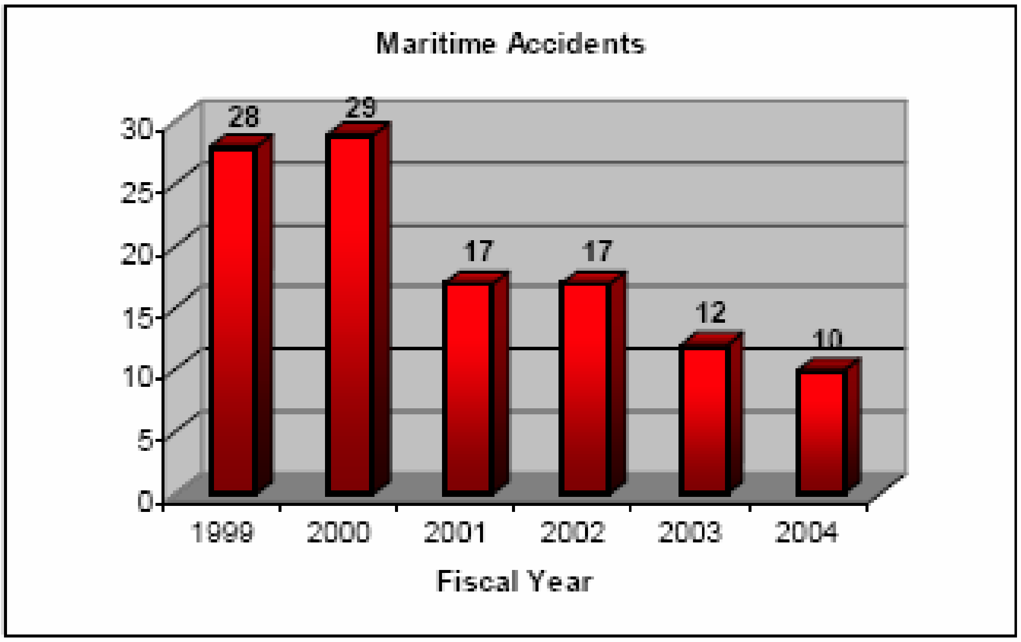


Figure 3. The number of accidents in the Panama Canal system in fiscal year.

Motivation (LTSA, UPPAAL)

Due to the growing complexity of canal system operation, there is a strong need for formal approaches to verification and validation of concurrent system behaviors in canal. The labeled transition system analyzer (LTSA) and UPPAAL are two modeling, synthesis and verification tools for behavioral modeling [5, 6]. Not only are their purposes complementary but also they employ different approaches to accomplish their goals. LTSA checks that the specification of a concurrent system satisfies the properties required of its behavior. LTSA also supports specification animation, a pathway to visual validation and interactive exploration of system behavior. LTSA has built features that help a designer to fulfill the gaps between the behavioral and architectural aspects of specifications (such as message sequence charts) and the actual capabilities of proposed system architecture.

UPPAAL, in contrast, is a tool for modeling, synthesis and verification of the real time systems. The common application areas are real time controllers [7]. Canal control systems are also a soft real time system. In traditional canal models, similar to the Panama Canal [9], time is not that crucial to correct operation. From an economic standpoint, however, the time that a ship takes to pass through canal system is proportioned to cost. Hence, there is a strong economic need for behavior models that explicitly account for the time-behavior of individual and groups of ships passing through the canal system.

Behavioral Modeling

Behavior models are accurate and abstract descriptions of the intended behavior of a system. They organize systems as concurrent blocks and describe how they interact.

Behavior models have solid mathematical foundations that can be used to support precise analysis, verification and validation of properties [2]. The valuable techniques and tools that have been developed for this purpose have shown that behavior modeling and analysis are successful in uncovering the slight errors that can appear in designing of concurrent systems. Summarizing, it is important and beneficial to balance the potential benefits and shortcomings of approaches based on behavior models.

Figure 4 shows the step-by-step procedure that we will follow to formulate concurrent behavior models. Individual use cases will be elaborated the scenarios, which in turn, will be expended into fragments of behavior expressed as activity and sequence diagrams. Composition techniques can formulate models of component-behavior and architecture-level behavior.

Overview of the Validation and Verification Process

This thesis employs message sequence charts (MSC) to describe models of system level behavior in canal system. LTSA can construct finite labeled transition system that precisely captures its specified set of traces [8] from a given MSC specification.

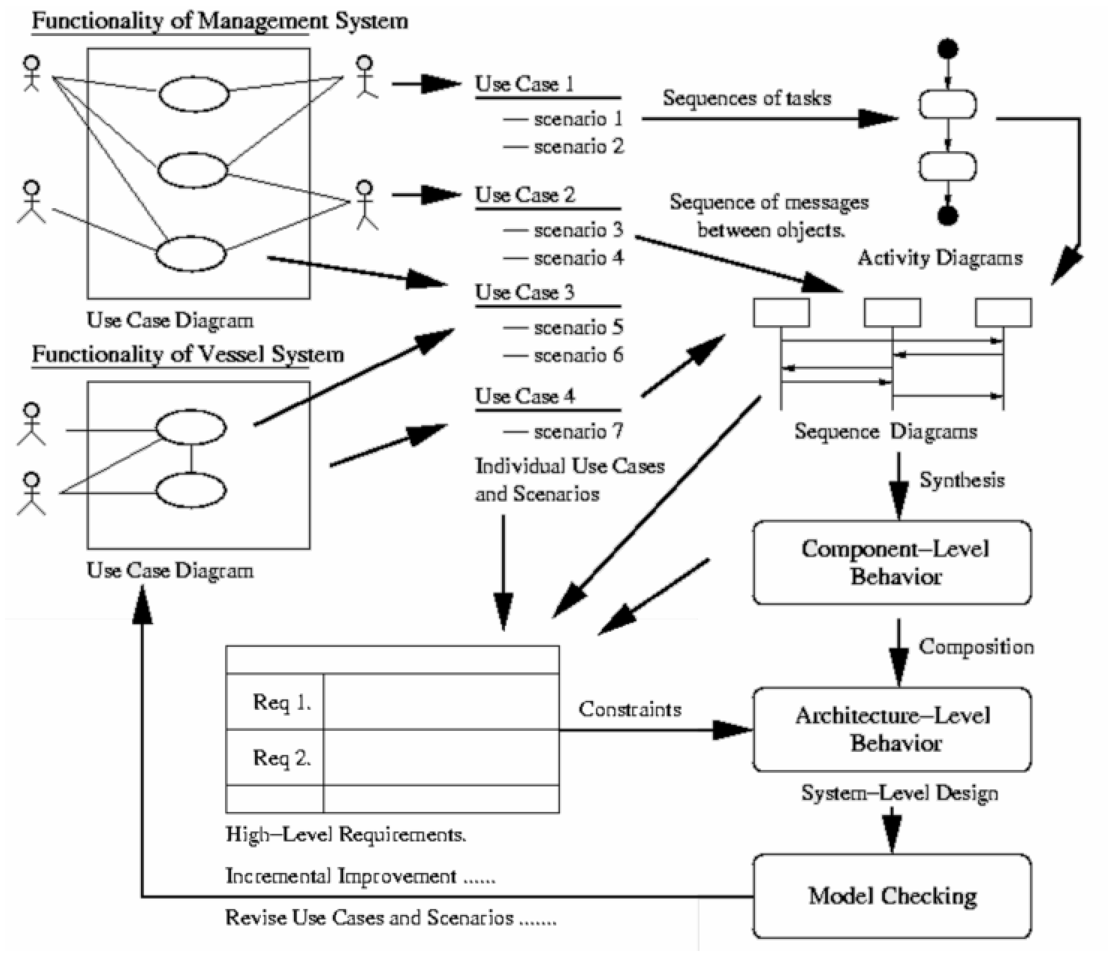


Figure 4. Incremental Development and Behavior Modeling: From Uses Cases to Model Checking [1].

Expecting stakeholders to produce a complete set of scenarios with complete coverage in one go is unrealistic. To help designers articulate what must be provided by the system and what must be prevented, the methodology employs a combination of positive, negative and implied scenarios to guide incremental improvement of system-level descriptions [8]. See Figure 5. The detailed model partitions scenarios into three categories. Positive scenarios specify the intended system behavior. Negative scenarios specify undesirable behaviors the system is expected not to exhibit (e.g., operations that are unsafe). Implied scenarios correspond to gaps in the scenario-based specification.

These gaps can occur in two ways. First, when models of architecture-level behavior are composed from component-level behaviors, gaps in the scenario description will occur when individual component-level behavior has an inadequate view of intended system-level behavior. A second possibility is that the system architecture may contain feasible of traces of behavior that are not detailed in the scenario specification (i.e., the system architecture might do something that the user is unaware of). An implied scenario may simply mean that an acceptable scenario has been overlooked and that the scenario specification needs to be completed [1]. In simulation part, as shown in Figure 5, all the gaps between the desirable architectural model and MSC specifications are filled so the simulation output is the intended architectural model. Implied scenarios, which are shown in Figure 5 are the result of a mismatch between the required behavior and the required architecture of a system: it is not always possible to provide an arbitrary set of traces with fixed system architecture. However, more importantly, implied scenarios indicate gaps in a system specification. An implied scenario may be an acceptable system trace that has been overlooked by stakeholders. This is not surprising as scenario-based specifications are inherently partial [2]. It is reasonable to assume that stakeholders do not provide a comprehensive set of examples of how the system is expected to behave. In this case, the implied scenario should be added to the specification in order to extend it covering a case that had not been clearly described in the original system description. It is also possible, however, that the implied scenario represents unwanted system behavior. This means that the implied scenario is a result of a partial description of the system architecture. As shown in Figure 5 the architectural model which, is the model generated by LTSA according our specifications, will be compared with the trace model, trace model is our desired model. In model checking process the differences turn into the implied scenario, the implied scenario can be accepted as negative implied scenario or positive implied scenario, this process will be done over and over till there would not be any implied scenario that means the architectural model is the same as our desirable model. By detecting and validating implied scenarios it is possible to drive the elaboration of scenario-based specifications and behavior models and possibly converge to a state where there are no more implied scenarios to be validated:

1. If a positive scenario is added as the result of accepting an implied scenario, then the specification for “acceptable system behavior” is extended.
2. If a negative scenario is added as the result of rejecting an implied scenario, then the specification is strengthened.

The decision to accept or reject a scenario depends on the problem domain at hand and will require consultation with the project stakeholders.[1]

Systems requirements correspond to constraints on system functionality, system interfaces, and nonfunctional concerns, such as safety and reliability. They are generated, in part, from features in the activity and sequence diagrams. For example, sequence diagrams also imply component interfaces needed to support the passing of message between components.[1]

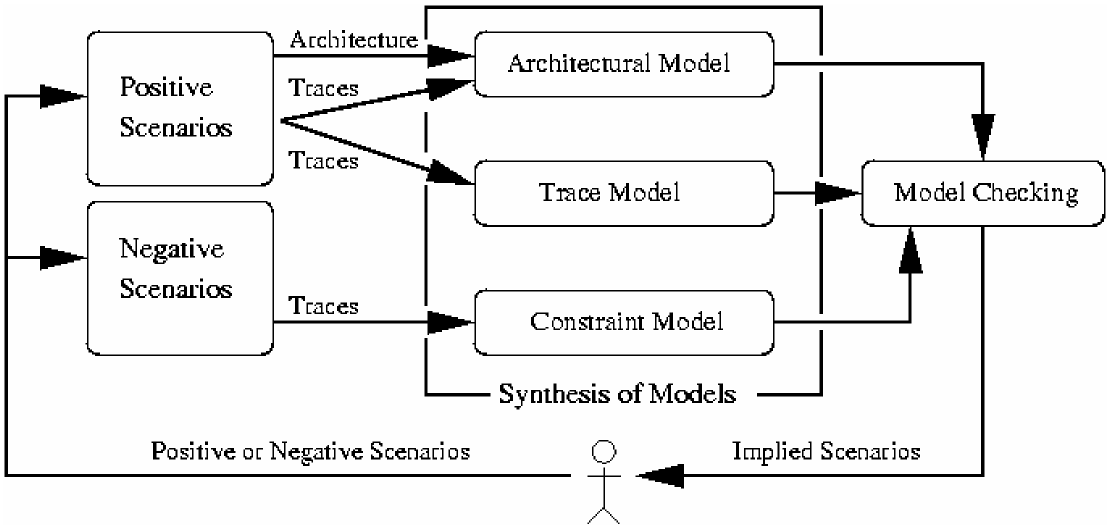


Figure 5. Simplified View of the Whole Model Checking Process in LTSA.[1]

For simulation and verification with LTSA, this thesis uses finite state process (FSP) code to define the behavior of the system. We describe FSP code more in detail in Chapter 3.

Mechanical System Verificaton with the Animator Box

Finite labeled transition starts for the canal system are generated by LTSA both after simulating the FSP code and MSCs. As shown in Figure 6, the animator box in LTSA can be used to mechanically verify behavior in the labeled transition states (LTSs)

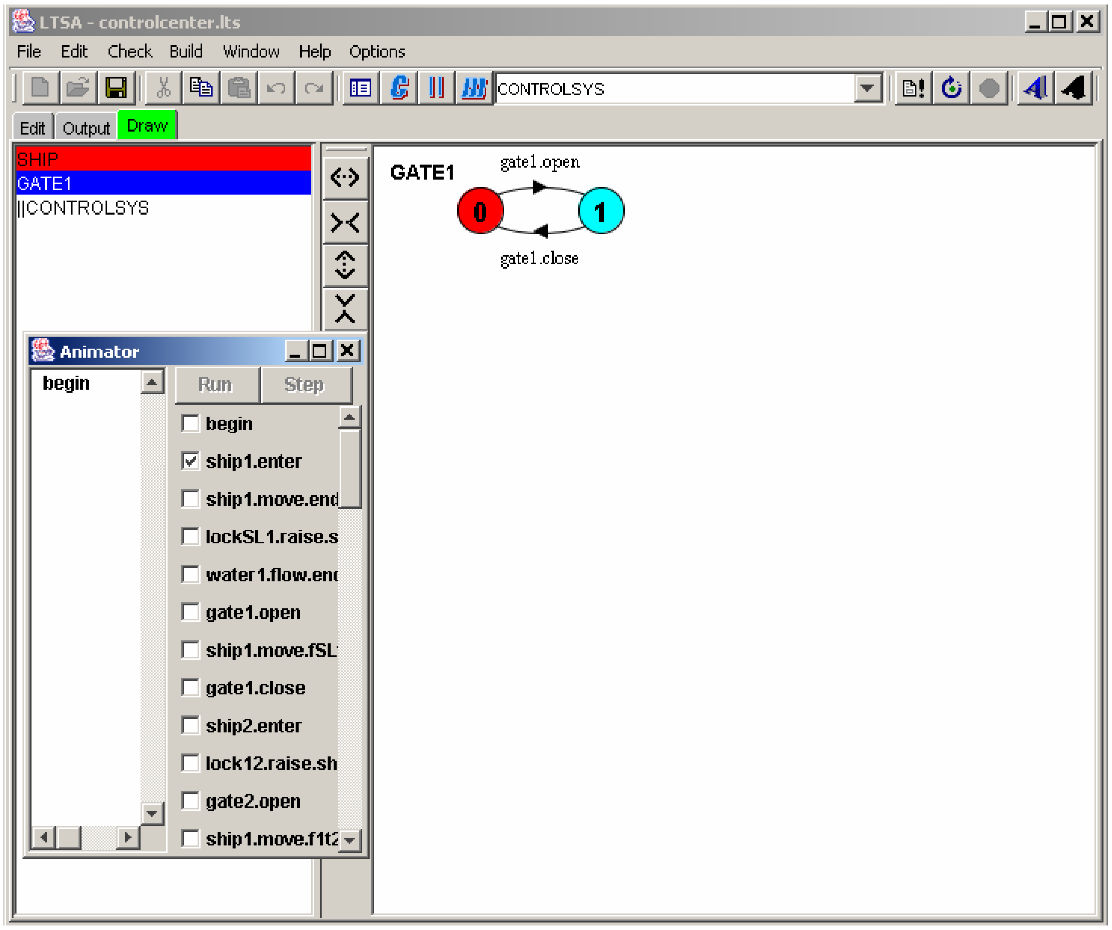


Figure 6. Animator box for mechanically system verification

All the transitions are traversed mechanically to check the system safety and liveliness. System safety means that in finite labeled transition system it should not be any error states or unwanted transitions and system liveliness means that something good eventually should happens. Finally, a visualized verification is done to show the animation of the canal system behavior.

LTSA does not have the ability to simulate real time systems. However, in this thesis we model a real time canal system to reduce the waiting time in the line approaching the canal system. Modeling of the real time canal system is done with UPPAAL. After the desirable architectural model is obtained in LTSA, the time issue is added to the transitions of finite state machines, simulated and verified

Contributions

In traditional canal systems, a ship can pass through the canal system very slowly, if permitted. From an economic standpoint, however, time is money and in present day canal system (e g. Panama Canal) costs are greater than they should be simply because of wasted time [9]. Therefore, there is a strong need for behavior models that incorporate time.

This thesis has two important contributions. In the approach of canal modeling; firstly, it gives an architectural model of the canal, which allows for formal representation and evaluation of expressing for safety, liveliness and deadlock. The second contribution is incorporation of the “time issue” in to canal system modeling. The result is a canal system with higher efficiency and higher reliability

Thesis Outline

In this thesis we present step by step of the modeling and verification approach with two behavioral modeling and verification tools, LTSA and UPPAAL. Both developments include theory and practice. We also present the strength and weakness of each tool and suggest an approach for combining these two tools to use their best abilities and achieve the best and the most reliable architectural model. In Chapter 2 we describe development of the canal system from use cases and scenarios through the requirements. In Chapter 3 we present a methodology to achieve the system architecture. In Chapter 4 we implement the canal system with LTSA and UPPAAL and present the system validation and verification. Discussion, evaluation and conclusion are in Chapter 5.

Chapter 2: Synthesis of Canal System Operation using UML

This chapter describes the pathway of development from use cases to scenarios to requirement and simplified models of behavior.

Use Cases

A use case is a collection of possible sequences of interactions between the system under discussion and its Users. The collection of Use Cases should define all system behavior relevant to the users to assure them that their goals will be carried out properly.

Use cases are used during the analysis phase of a project to identify and partition system functionality; in other words, a use case is a set of actions that a system performs that yields an observable result of value to a particular actor. This formalism separates the system into actors and use cases. An actor is a person, organization, or external system that plays a role in one or more interactions with a system (actors are typically drawn as stick figures on UML Use Case diagrams). Actors represent roles that can be played by users of the system. Those users can be humans, other computers, pieces of hardware, or even other software systems. The only criterion is that they must be external to the part of the system being partitioned into use cases. They must supply stimuli to that part of the system, and they must receive outputs from it. Use cases describe the behavior of the system when one of these actors sends one particular stimulus. This behavior is described textually as the form of scenarios that we will describe later in this chapter.

Figure 7 shows the Use case diagram of the canal system:

The five actors are: Operator (Control Center), Shipmovement sensor, Gatesensor, Ship Captain and Valve sensor.

The three high-level use cases are: Control Up Hill Move, Control Down Hill Move and Control Lake Movement.

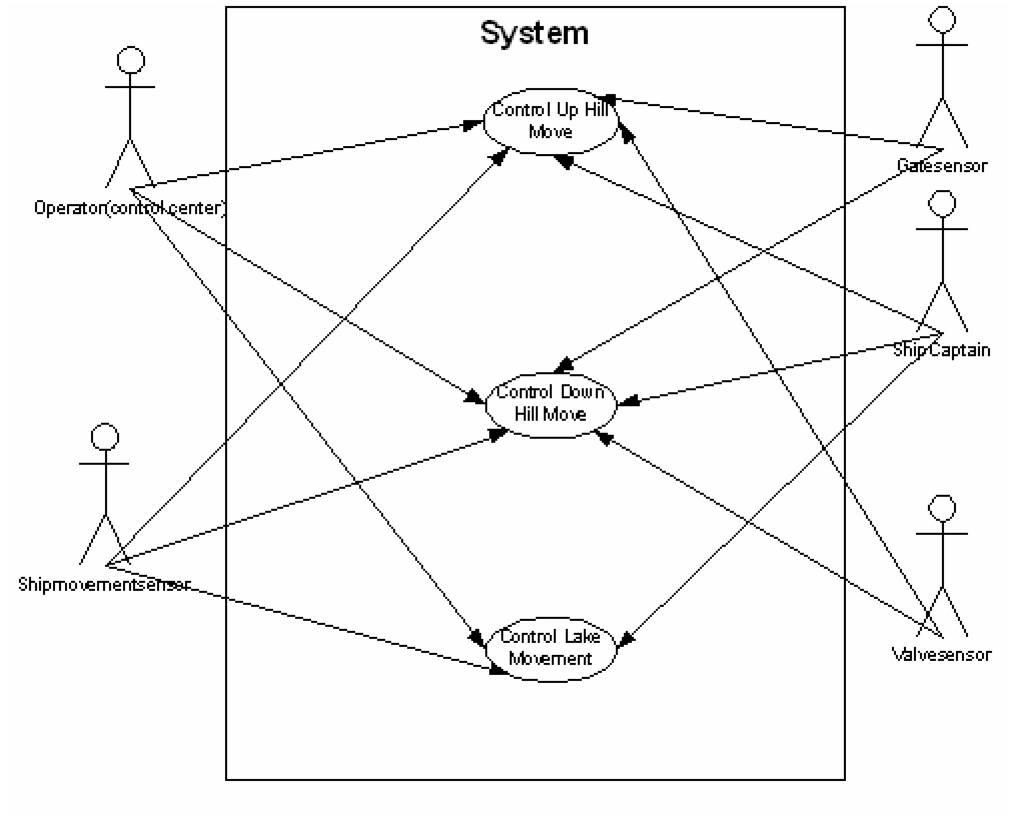


Figure 7. Use case diagram of the Canal system.

The Use case diagram of the Canal system is shown on very top-level base.

We will show the control architecture in detail in Chapter 3.

Scenarios

Scenarios present detailed textual description of the system behavior, in other words they are sequences of actions that may not be observable to the system environment, but they are embedded in the system design [10]. In the following we present the scenarios of the Canal system according to the Canal system Use Case diagram:

The initial amount of n and v are equal to 1.

Use Case 1: Control Up Hill Move

- Actors: Operator (Control Center), Shipmovementsensor, Gatesensor, Shipcaptain, Valvesensor.
- Preconditions:
  1. The connection between the operator/control center and ship captain should be active.
  2. Ship should be running properly.
- Basic Flow of events:
  1. Ship approaches the canal system.
  2. Ship captain sends the arrival request to the ship control center.
  3. Ship move in to the queue behind the canal system.
  4. Control center sends the acknowledgement that it received the signal.
  5. Control center checks the lock system.
- Alternative Flow of Events 1:
  6. Control center does not find empty first lock.
  7. Control center sends the order to the ship to stay in the queue
  8. Everything repeat from step 5.
- Alternative Flow of Events 2:
  6. Control center finds empty first lock.

7. Control center checks the water level of lock n and lock n+1

- Alternate Flow of Events 1:
  8. Water level of lock n and lock n+1are not equal.
  9. Open valve in lock (n+1)

- Alternate Flow of Events 2: Water level of lock n and lock (n+1) is equal
    8. Control center sends the closevalve order to the valve of lock (n+1)
    9. Valve of lock (n+1) sends the acknowledge to the control center.
  10. Control center sends the “open gate” order to the closest gate to ship of lock n.
  11. Control center sends the “allow passage” message to the ship captain.
  12. Ship passes through the lock
  13. Ship movement sensor sends the ship movement completion message to the control center.
  14. Control center order the ship to stop.
- Post condition:
  1. Ship completely is located in a lock
  2. Ship completely comes out of another lock.

Use Case 2: Control Down Hill Move

- Actors: Operator (Control Center), Shipmovementsensor, Gatesensor, Shipcaptain, Valvesensor.
- Preconditions:
  1. The connection between the operator/control center and ship captain should be active.
  2. Ship should be running properly.
- Basic Flow of Events
  1. Ship approaches the canal system.
  2. Ship captain sends the arrival request to the ship control center.
  3. Ship move in to the queue behind the canal system.

4. Control center sends the acknowledgement that it received the signal.
5. Control center checks the lock system.

- o Alternative Flow of Events 1:
    6. Control center does not find empty first lock.
    7. Control center sends the order to the ship to stay in the queue
    8. Everything repeats from step 5.
- o Alternative Flow of Events 2:

6. Control center finds empty first lock.

7. Control center checks the water level of lock n and (n-1).

- o Alternative Flow of Event1:

8. Water level of lock n and lock n-1are not equal.
9. Open valve in lock (n)

- o Alternate Flow of Event 2: Water level of lock n and lock (n-1) is equal

8. Control center sends the closevalve order to the valve of lock (n)
9. Valve of lock (n) sends the acknowledge to the control center.
10. Control center sends the “open gate” order to the closest gate to ship of lock n.
11. Control center sends the “allow passage” message to the ship captain.
12. Ship passes through the lock
13. Ship movement sensor sends the ship movement completion message to the control center.
14. Control center order the ship to stop.

Use Case 3. Control Lake Movement

Actors: Operator (control center), Shipmovementsensor, Shipcaptain

- Precondition:
  1. Ship is towards the lake direction and located in the most adjacent lock to the lake.
  2. Ship is running properly.
  3. The connection between the ship and control center is active.
- Basic Flow of Events:
  1. Ship captain sends the request
  2. Control center put the request in the queue.
  3. Control center check the queue.(control center gives the passage permit according First In First Out order).
- Alternate Flow of Events1:
  4. Control center gives the passage permit to the ship.
  5. Ship enters the lake.
- Alternative Flow of Events 2:
  4. Control center put the ship on the queue.
  5. Repeat from step
- Post condition:
  1. Ship one turn gets closer to entering the lake.
  2. One movement in the queue has happened.

Activity Diagrams

Activities diagrams provide a visual reference for documenting sequences of tasks showing a single activity. The following diagrams show the canal subsystems activity diagrams consistent with the canal system use case diagram, Figure 8 shows the activity diagram of ship passing through the canal in uphill direction and it is consistent with the flow of events in the use case one, each alternative flow of events is correspondence to the diamond boxes shown in Figure 8, after the first set of alternative flow of events the water level of two adjacent locks is checked and then another alternative flow of events happens, in the second round of the alternative flow of events, the water level is controlled by the opening or closing the valve of the lock with higher water level, after matching the water level of two adjacent locks the gate of the lock n gets open and ship can pass through the gate. Figure 9 shows the activity diagram of the ship passing through the lake between the two sets of lock. The flow of events is consistent with flow of events in use case three. When ships are ready to pass the lake in both directions, to avoid the collision of the ships, only one ship can pass through the lake at a time. For this purpose if there are two ships waiting to pass the lake at the same time we should give the priority to one of the to pass the lake because as we mentioned only one ship can pass through the lake at the same time. The priority rule will obey the first ship in first ship out. The more efficient strategy will give the priority to the smaller and faster ships but implementing this fairer model is not in the focus of this thesis. Figure 10shows the ship movement through the canal in down hill direction, the flow of events is consistent with the use case two flow of events. We will talk about separating the up hill movement and down hill movement in chapter 3.

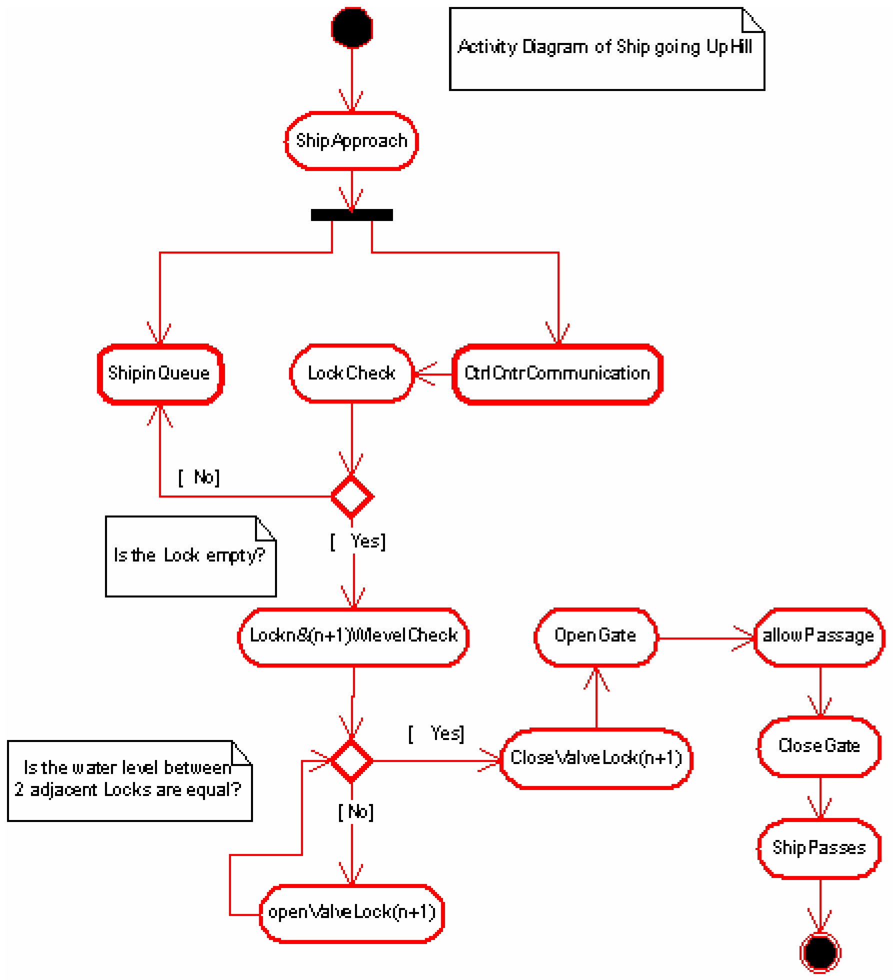


Figure 8. Activity diagram for a Ship Passing through the Canal (Up Hill)

As it is shown in Figure 8 in the first alternative flow of events if the lock that ship wants to enter is full the control center put the ship in a queue behind the canal, however, in real world the control center just send the wait signal to the ship, but here to keep track of the ships waiting to enter the canal and keep them in order we create a queue behind the canal.

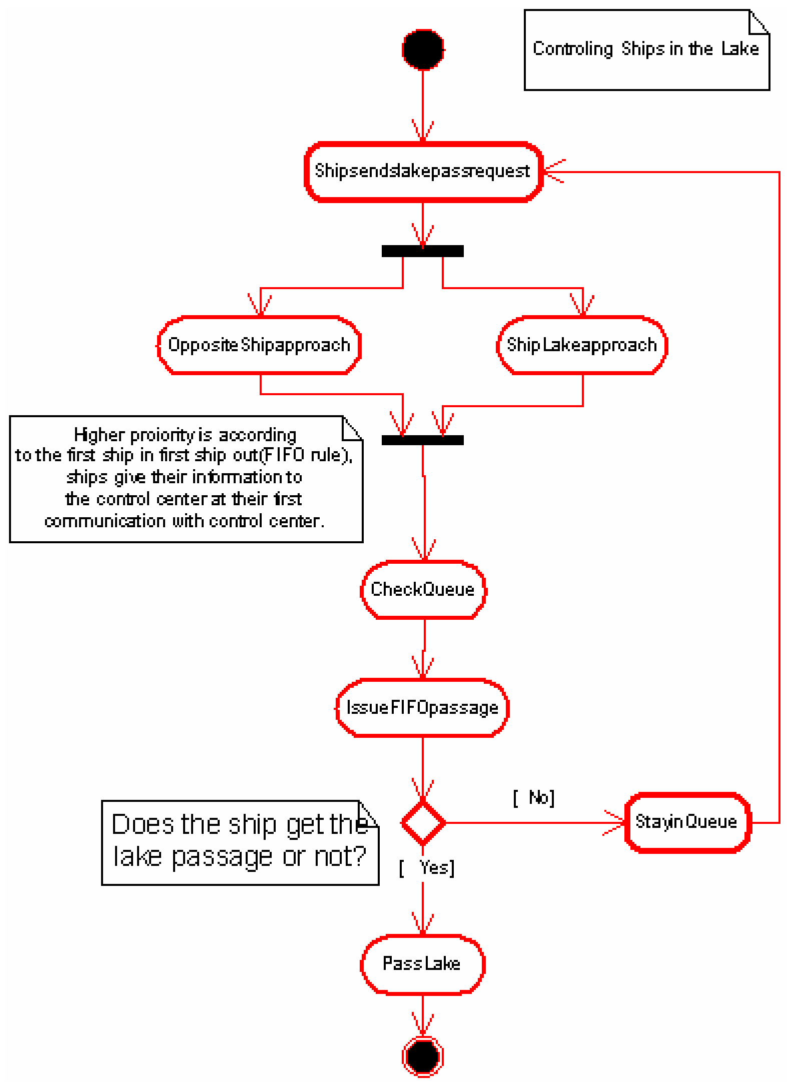

Controling Ships in the Lake
Shipsendslakepassrequest
OppositeShipapproach
ShipLakeapproach
Higher proiority is according to the first ship in first ship out(FIFO rule), ships give their information to the control center at their first communication with control center.
CheckQueue
IssueFIFOpassage
Does the ship get the lake passage or not?
[No]
StayinQueue
[Yes]
PassLake

Figure 9. Activity diagram for control of ship movement passing through the lake.

In Figure 9, control center may put ships approaching the lake through the canal system, coming from opposite directions may, into a queue. This queue is similar to the queue behind the canal when ships want to enter the canal for the first time.

As we described it, the priority to passing the lake is according the FIFO rule.

After ship completes its movement through the lake, it should pass through another set of locks. At this point control center treats ships as they want to enter the canal system from the ocean, from this point concerning which side of the lake ship is its movement will be treated like in use case one (control up hill move) or like use case two (control down hill move).

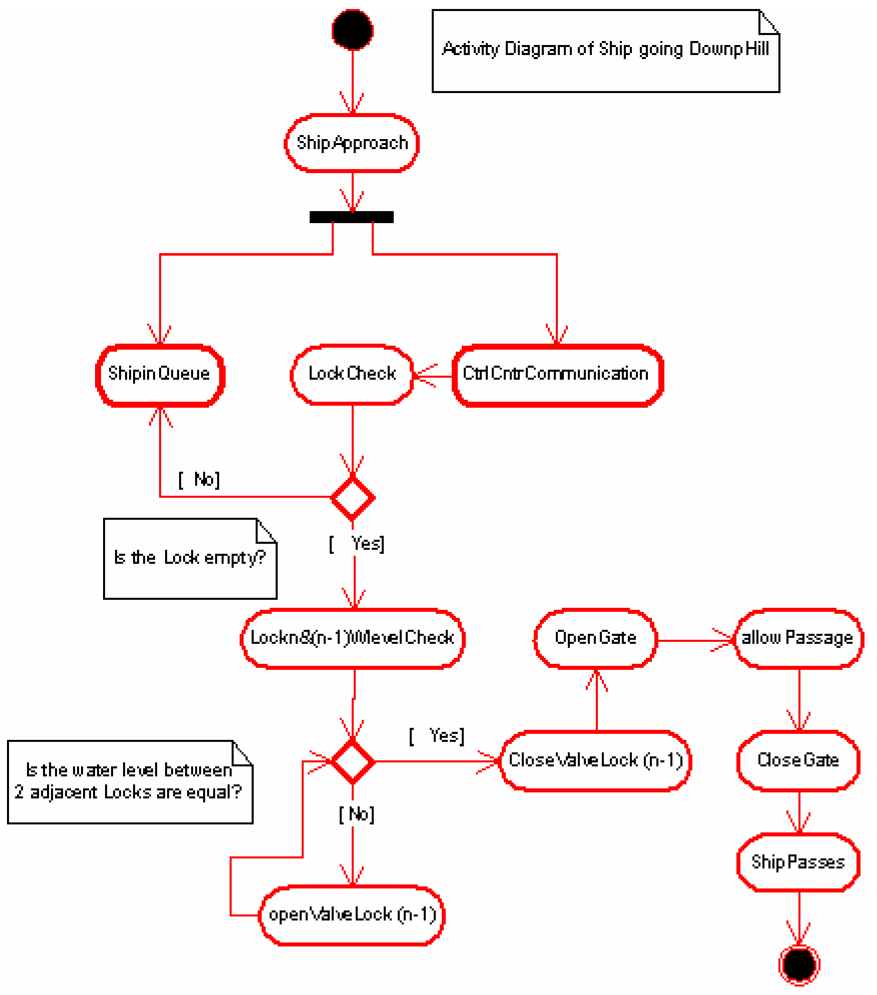


Figure 10. Activity diagram for a Ship Passing through the Canal (Down Hill)

Message Sequence Charts (MSCs)

MSCs allow the developer to describe patterns of interaction among sets of components; these interaction patterns express how each individual component behavior integrates into the overall system to establish the desired functionality. Typically, one such pattern covers the interaction behavior for one particular service or scenario of the system. For example in the following figures some of the MSCs of the canal system are shown, they were developed by LTSA. We will describe the canal system architecture in detail in Chapter 3. Figure 11 shows the interaction between control center, ship movement control and gates.

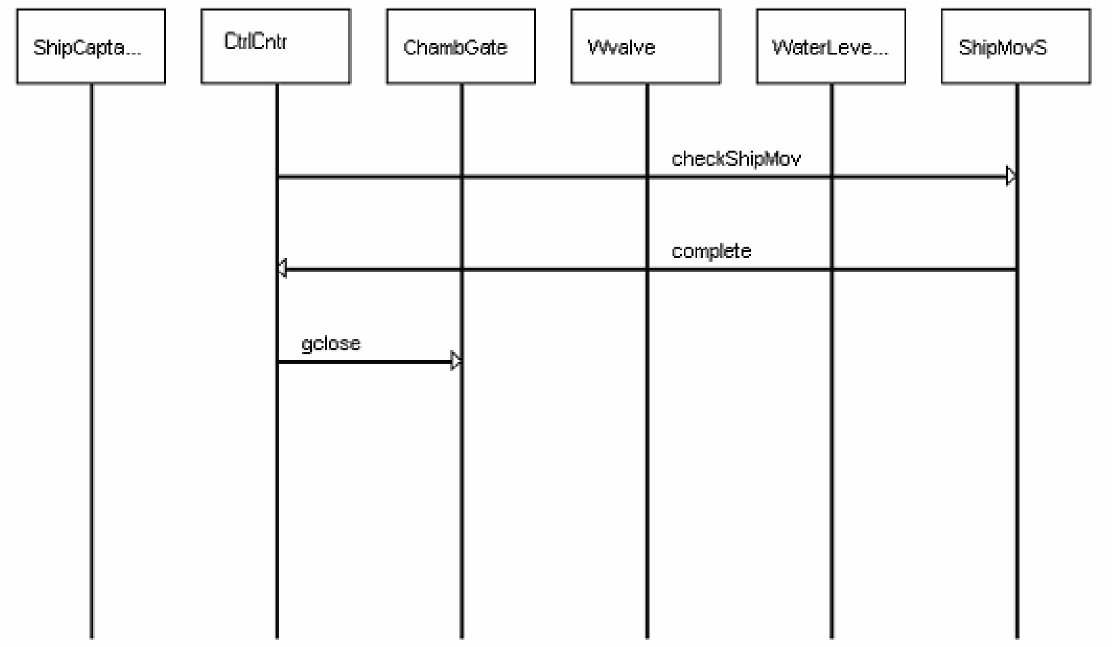


Figure 11. A Basic Message Sequence Chart of Interaction Between the Control Center, Ship Movement Sensor and the Gate.

Figure 12shows the interaction between control center, water valve, gate and ship captain. Each set of message sequence charts shape a component of a system which in canal system Figure 11 shape the GATESENSOR component and Figure 12 shows the SHIPMOVEMENT control component. The relationships that all the system components have make the system architecture, which we described it in detail in Chapter 3 in “system architecture” section.

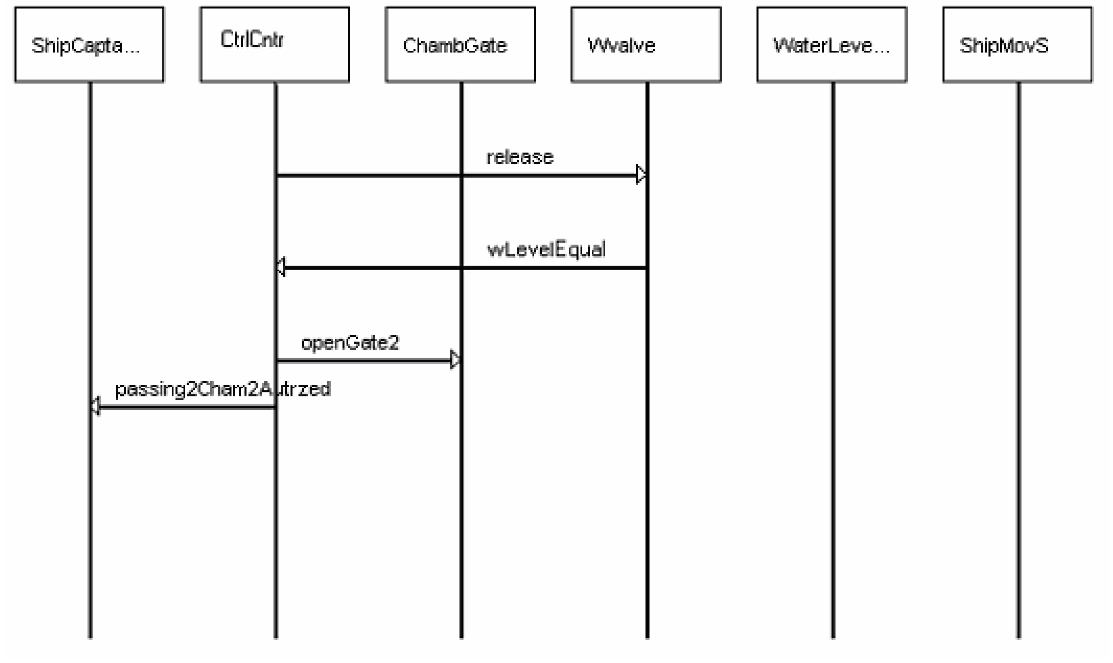


Figure 12. A Basic Message Sequence Chart of Interaction Between the Control Center , Water Valve and Ship Captain.

Requirements

Use Case one: Control Up Hill Move

1. Ship shall approach the canal by an appropriate speed that it can stop in distance not shorter than its length.
2. Ship captain and ship movement control in the control center shall be able to contact by wireless communication devices.
3. Ship shall stop in a queue behind the canal
   3.1. Ship shall stop at the end of the last ship in the queue or first gate of the canal.
   3.2. Ship shall move through the queue when control center send the movement order.

4. Ship captain and control center operator shall send an acknowledgement to each other after receiving any message from each other. (This requirement is for avoiding the deadlock and having liveliness in the system)
5. Control center and lock occupation sensor shall be able to communicate with each other.
   - 5.1. Control center shall check the locks before any passage permit message.(for safety issue)
   - 5.2. Control center shall check the closest lock to the ship.
   - 5.3. Control center shall not issue the passage permit message as long as the closest lock to the ship asking for passage is full.
     - 5.3.1. Control center shall order the ship "stay in the queue" while the closest lock to the ship is full.
6. Control center and water level sensor shall be able to communicate with each other.
   - 6.1. Control center shall check the water level of the two adjacent locks, n and n+1, where the ship will enter the lock n.
7. Control center shall not issue the passage permit through two adjacent locks as long as their water levels are equal. (Safety)
8. Control center and valve sensor shall be able to communicate with each other.
   - 8.1. Control center shall send the "close valve" message to the valve (n+1) when the water levels are about to be equal.
9. Control center and gate sensor shall able to communicate with each other.
   - 9.1. Control center shall issue the permit passage for a ship through a lock after the closest gate of the lock to the ship be completely opened. (Safety)

10. Ship shall pass through the locks one forth of their average speed in the sea.

Uses case two: Control Down Hill Move

1. Ship shall approach the canal by a specific speed that it can stop in distance not shorter than its length.
2. Ship captain and ship movement control in the control center shall be able to contact by wireless communication devices.
3. Ship shall stop in a queue behind the canal
    - 3.1. Ship shall stop at the end of the last ship in the queue or first gate of the canal.
    - 3.2. Ship shall move through the queue when control center send the movement order.
4. Ship captain and control center operator shall send an acknowledgement to each other after receiving any message from each other. (This requirement is for avoiding the deadlock and having liveliness in the system)
5. Control center and lock occupation sensor shall be able to communicate with each other.
    - 5.1. Control center shall check the locks before any passage permit message. (For safety issue)
    - 5.2. Control center shall check the closest lock to the ship.
    - 5.3. Control center shall not issue the passage permit message as long as the closest lock to the ship asking for passage is full.

5.3.1. Control center shall order the ship “stay in the queue” while the closest lock to the ship is full.

6. Control center and water level sensor shall be able to communicate with each other.

6.1. Control center shall check the water level of the two adjacent locks, n+1 and n, where the ship will enter the lock n.

7. Control center shall not issue the passage permit through two adjacent locks as long as their water levels are equal.(safety)

8. Control center and valve sensor shall be able to communicate with each other.

8.1. Control center shall send the “close valve” message to the valve (n) when the water levels are about to be equal.

9. Control center and gate sensor shall able to communicate with each other.

9.1. Control center shall issue the permit passage for a ship through a lock after the closest gate of the lock to the ship be completely opened.(safety)

10. Ship shall pass through the locks one forth of their average speed in the sea.

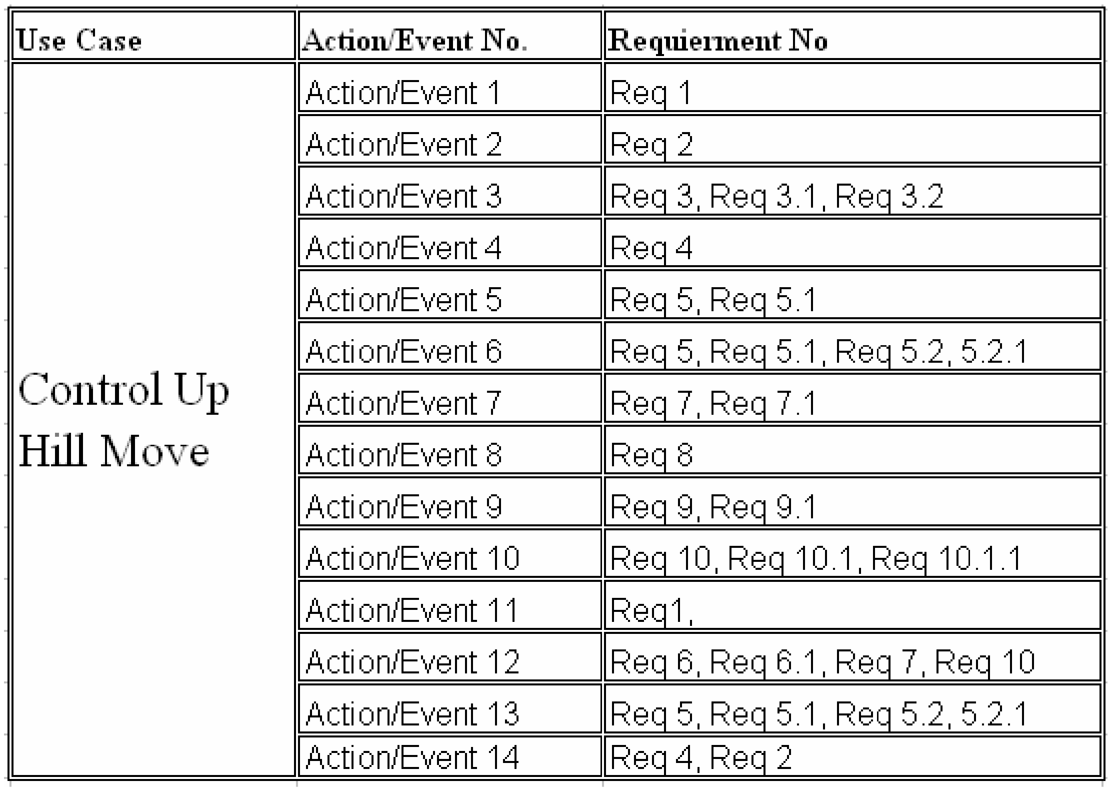
Use cases three: Control Lake Movement

1. Ship must not enter the lake without the control center permission.
2. Control center shall give the lake passage permit according the first In First Out order (FIFO). (Safety)

   2.1. Control center should apply FIFO order regardless of the different sides of the lake. (Deadlock prevention)

Traceability

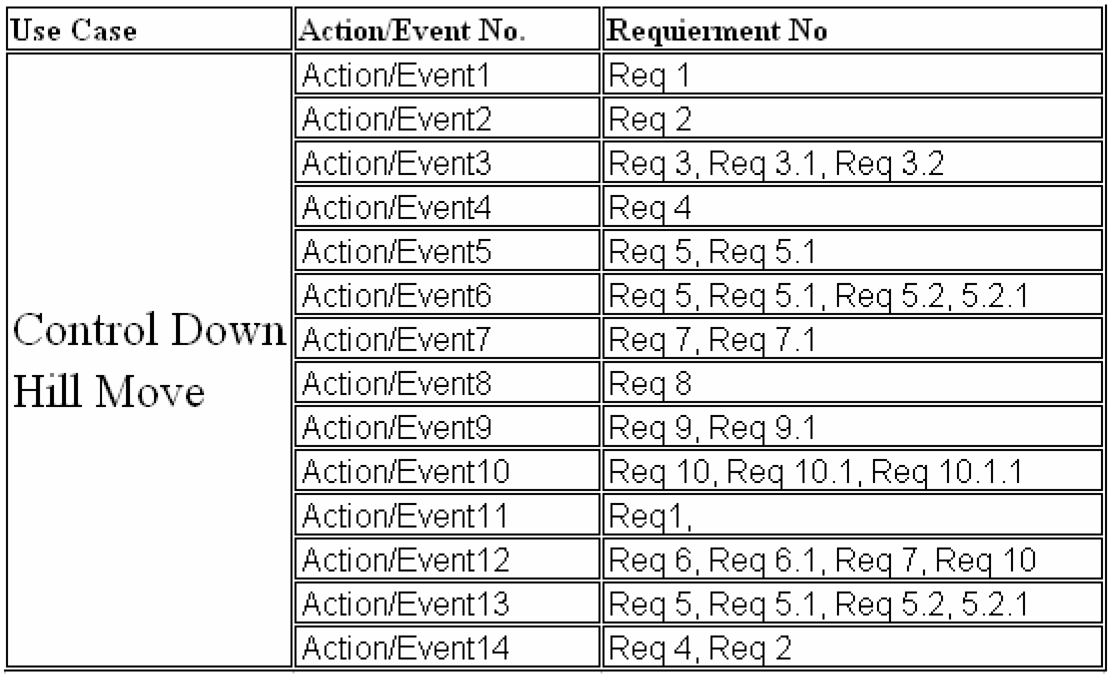
Traceability is managed throughout the development cycle. Each traceability link is stored as a model precision in the repository, and can be associated with either a UML element introduced during analysis or design (use case, scenario), with the algorithmic code complements of an operation, or with the definition of tests linked to the scenario or to the operation. In Table 1 we show the trace of the requirements and scenarios of use case one.



Use Case	Action/Event No.	Requierment No
Control Up Hill Move	Action/Event 1	Req 1
	Action/Event 2	Req 2
	Action/Event 3	Req 3, Req 3.1, Req 3.2
	Action/Event 4	Req 4
	Action/Event 5	Req 5, Req 5.1
	Action/Event 6	Req 5, Req 5.1, Req 5.2, 5.2.1
	Action/Event 7	Req 7, Req 7.1
	Action/Event 8	Req 8
	Action/Event 9	Req 9, Req 9.1
	Action/Event 10	Req 10, Req 10.1, Req 10.1.1
	Action/Event 11	Req1,
	Action/Event 12	Req 6, Req 6.1, Req 7, Req 10
	Action/Event 13	Req 5, Req 5.1, Req 5.2, 5.2.1
	Action/Event 14	Req 4, Req 2

Table 1. Traceability of requirements and scenarios of each use case one

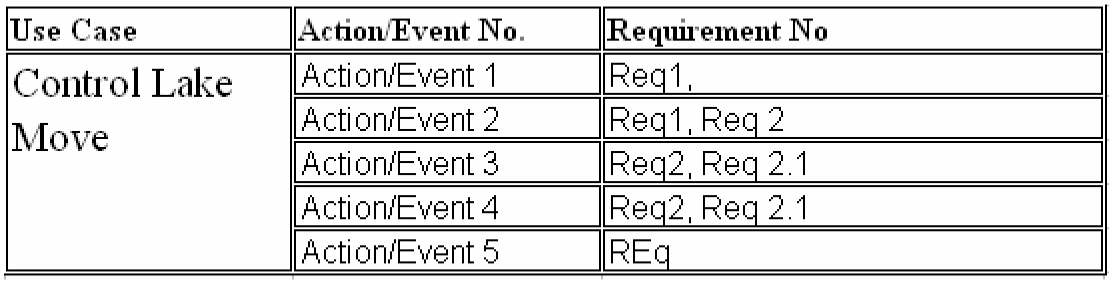
In Table 2 the traceability of between use case two ,scenarios and requirement is show:



Use Case	Action/Event No.	Requierment No
Control Down Hill Move	Action/Event1	Req 1
	Action/Event2	Req 2
	Action/Event3	Req 3, Req 3.1, Req 3.2
	Action/Event4	Req 4
	Action/Event5	Req 5, Req 5.1
	Action/Event6	Req 5, Req 5.1, Req 5.2, 5.2.1
	Action/Event7	Req 7, Req 7.1
	Action/Event8	Req 8
	Action/Event9	Req 9, Req 9.1
	Action/Event10	Req 10, Req 10.1, Req 10.1.1
	Action/Event11	Req1,
	Action/Event12	Req 6, Req 6.1, Req 7, Req 10
	Action/Event13	Req 5, Req 5.1, Req 5.2, 5.2.1
	Action/Event14	Req 4, Req 2

Table 2. Traceability of requirements and scenarios of each use case two.

In Table 3 also the traceability between the scenarios, requirements and use case three is shown.



Use Case	Action/Event No.	Requirement No
Control Lake Move	Action/Event 1	Req1,
	Action/Event 2	Req1, Req 2
	Action/Event 3	Req2, Req 2.1
	Action/Event 4	Req2, Req 2.1
	Action/Event 5	REq

Table 3. Traceability of scenarios, requirements and use case three.

Summary and Discussion

UML [11] diagrams help to show both behavioral and structural model of a complex system in a organized graphical representation which helps to understand the system design better, but the question is that, how we can find out if the behavioral diagrams we draw by UML work properly or not. The result of verification/validation of the system will answer this question, which UML is not capable of answering this question by synthesis and validation of the model. In the Chapter 4 we propose a methodology to model and verify the canal system with the help of two modeling and verification tools, LTSA and UPPAAL.

Chapter 3: Proposed Methodology

In this chapter we formulate a methodology for modeling and evaluating concurrent behaviors in canal system through the use of LTSA and UPPAAL.

Behavior Models

Overall system-level behavior corresponds to a combination of behavior within each subsystem, plus interactions among the subsystems. In moving the analysis and design of a system to its implementation, system behavior has a widely varying critical role, aspects of which can be represented in multiple formats (e. g, activity diagrams as we saw in the Chapter one, state charts, Message sequence charts, finite state machines or automatons). MSCs and similar description techniques have gained significant popularity over the past decade as a means for illustrating scenarios of component interaction in concurrent systems. In this chapter our main focus is to model the behavior of the canal system with state charts. State charts extend the capabilities of state transition (systems also called Automata) to include hierarchical organization of states, concurrent processes and broadcast mechanisms. They are the favorite graphical description technique for state-oriented component specifications; they stress the state change and the input/output relationship that defines an individual component's behavior.

Canal System Behavior

Consider the canal system, which typically covers several processes (such as controlling the water levels, gate movements, ship movements). Each of these services may result in several hundred lines of executable code. By means of scenarios we can illustrate key patterns of the canal operation behaviors of the components without having to present all the details of what else happens before, during, and after the time interval covered by the scenario. From an analysis perspective a good system: (1) exhibits safety and liveliness, and (2) avoids deadlocks. A safety property asserts that nothing bad will happen during the system execution. A liveliness property asserts that something good eventually happens (e.g., suppose that ships are approaching a narrow passageway. Liveliness would assert that, eventually, all of them will be able to pass through the passageway safely)[1]. A system state is deadlocked when there are no eligible actions that a system can perform

Behavioral Modeling with FSP and LTSA

LTSA is a behavioral modeling and verification tool [2], that can generate finite state machines representing of system behavior from MSCs. In other words LTSA drives state machines representing of individual processes and overall system process assembled from bMSCs (basic message Sequence charts) and hMSC (high level Message Sequence chart). bMSCs represent the interaction between the system components in an individual system process and hMSCs represents the relations between the processes in a system. To achieve the desirable, safe and deadlock free architectural model through modeling with MSCs, LTSA composes all the bMSCs together and run them in parallel. LTSA bridges the gaps between the system specifications and architectural model by following the relations in hMSCs and bMSCs. If any conflicts happen LTSA gives you an implied scenario representing the difference, the implied scenarios also shows the dead locks and unsafe conditions in the system design. The designer should decide how to accept the implied scenario, either as a negative implied scenario or as a positive implied scenario. The appeal of this top-down procedure is the systematic way of identifying gaps in the design either by omission or error. This approach has limited functionality however. The Message Sequence chart approach cannot implement a scalable model, for example in canal system, we cannot model a canal system with any number of ships entering the system but LTSA provide another approach to model a scalable system.

The manual specification of finite state process (FSP) is a second approach to system design and verification with LTSA. FSP is a textual description of the scenario model. Safety and liveliness issues should be written as the FSP code. In FSP code models of architecture-level behavior are obtained through the parallel composition of concurrent processes at the component level. Given two labeled transition systems (LTSs) P1 and P2, we denote the parallel composition P1||P2 as the LTS that models their joint behavior. By extension, the architectural-level behavior model is defined by:

Architecture-Level Behavior Model = P1||P2||P3|| · · · Pn

Example 1: Figure 13 shows a simple example of three switches working concurrently.

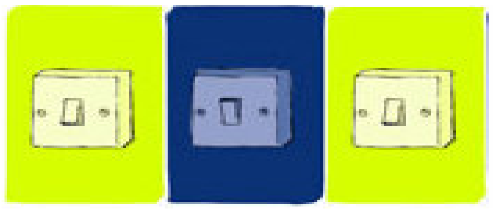


Figure 13. The simple switch example.

To show the concurrent behavior of three switches working together, the FSP model will be:

```
SWITCH =  (on->off->SWITCH).

||TWO_SWITCH = (a:SWITCH || b:SWITCH).

||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
```

The first line of the code shows the operation of a switch whose state is either on or off. The second line shows the parallel composition of the operation of two switches and line three shows the parallel composition of the operation three switches. Figure 14 shows the LTS model of Switch example. As shown in Figure 14 for each switch there is one LTS, labeled: `s.1:SWITCH, s.2:SWITCH` and `s.3:SWITCH`. There is one LTS model labeled: `SWITCHES`, which is the parallel composition of the three switches.

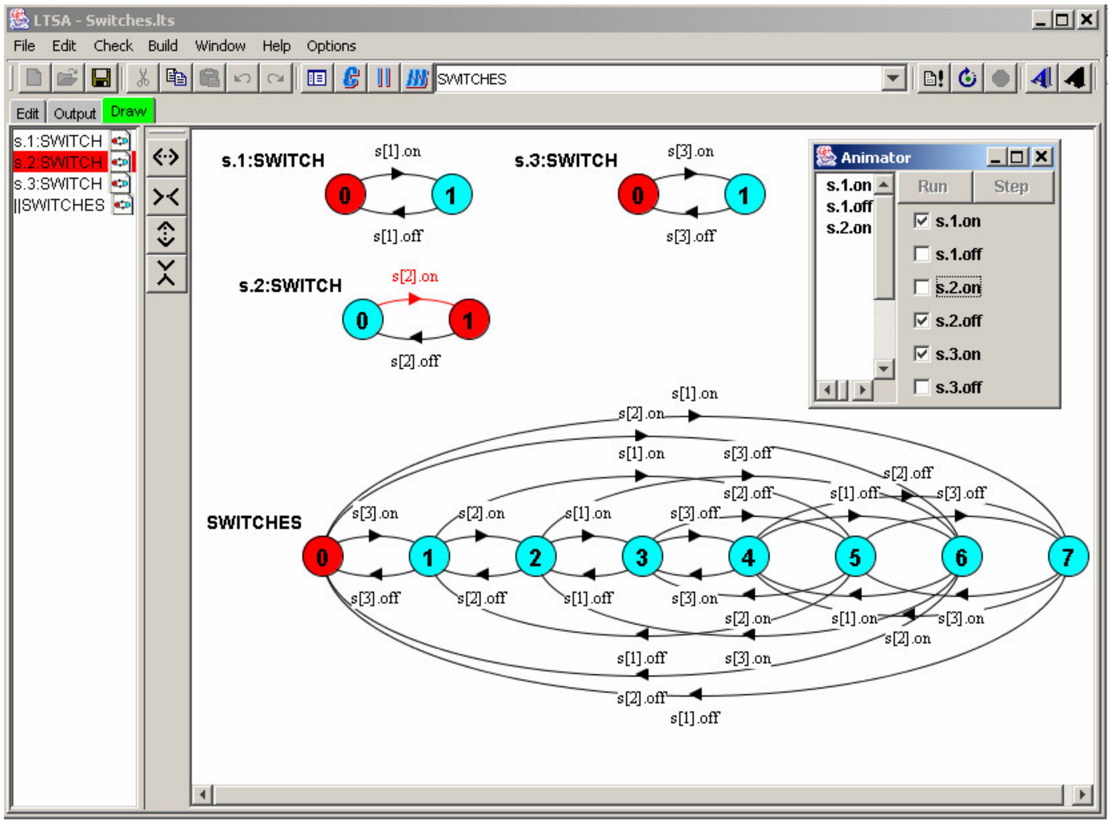


Figure 14. LTS model of Switch example.

Example 2: As we saw in the switch example, we write each process as an independent process and then we compose all of them to gather to shape the whole system, in the canal case we also follow the same rule to code the behavior of the system, for example the common gate between two adjacent locks should not be open until the water level of these two locks are equal:

```
                    While Wlevel2 <>Wlevel1
                           (Wlevel2 -> Wraise)
```

LTSA composes all the processes written in FSP together in parallel.

To verify that the system is correct, we can use the Animator box as we described it in Chapter 2. A second forms of visual verification or animation of the system model maybe built by linking the XML file to the FSP file. The generated animation does not have any nobility by itself; this is the FSP code that controls the animation movements.

Describing UPPAAL

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata based on temporal logic [12]. A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All the clocks progress synchronously. In UPPAAL, a system is modeled as a network of several such timed automata in parallel. We use UPPAAL to insert the time issue in our system and verify the real time canal system with UPPAAL.

System Architecture-Level Processes

We employ UML state charts for the representation of individual components behaviors and overall architectural. To simplify the difficulty in model formulation, Figure 15 shows the very high level architecture processes of the canal system. Overall control of a ship passing through the canal is divided into three important stages: the "UpHillMovCtrl" state does all of the control actions when a ship is moving up hill through the canal system. The “LakeMovCtrl” state does all the control actions while ship is passing through the lake, between two series of locks. The lake is a narrow passageway and, as such, ships can not move in both direction through the lake at the same time, (in other words, lake is like a one way water way.) The “DownHillMovCtrl” state is similar to the “UpHillMovCtrl”state except that the ship is moving down hill through the lock system. It is important to note that the strategy for matching water levels in two adjacent locks in the uphill movement is different than for the down hill movement. In uphill between two adjacent locks (n and n+1) we should raise the water level of the lock n but for the down hill we should raise the water level of lock n+1. These processes are illustrated in Figure 8 and Figure 9. The second reason for separation of control strategies is that we can keep track of ship entering the lake and exiting the lake, which helps to control the ship movement through the lake. Up hill ship movement should apply when ship enters the canal lock system and passes through three lock systems, the lake ship movement control should apply when the ship passes the first three locks and enter the lake, the down hill ship movement should apply when a ship enters the lock system from the lake and passes through three locks.

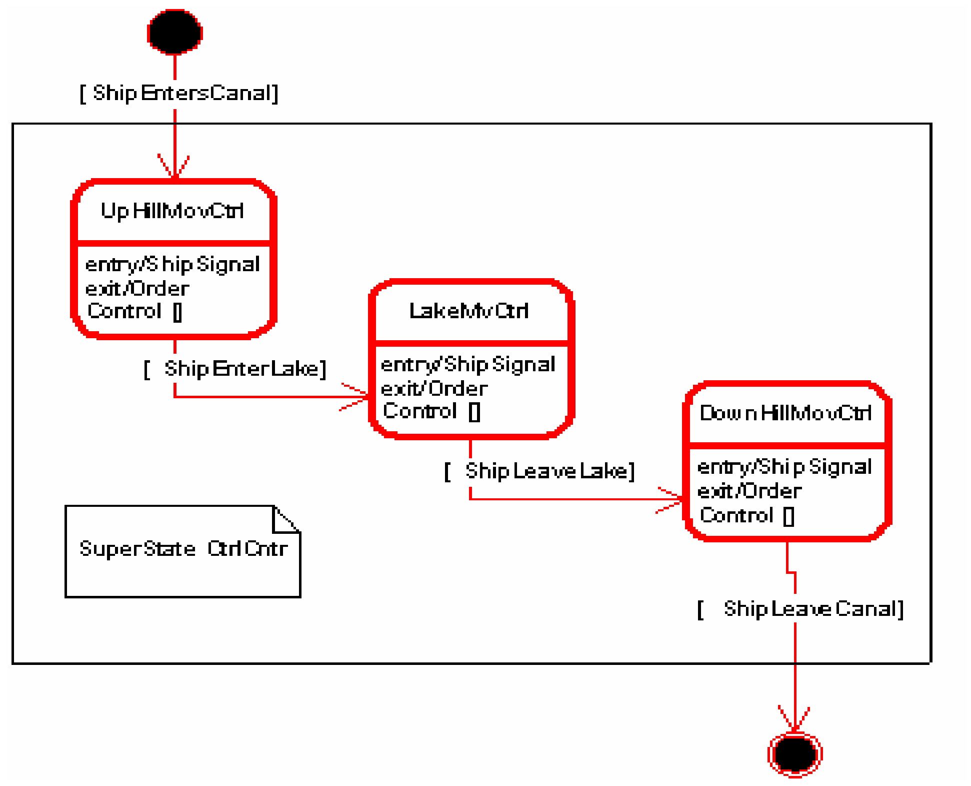


Figure 15.High level state chart diagram of the Canal system.

As shown in Figure 15 the overall architecture is named " Ctrl Cntr" super state, which is the whole canal system control centerThere are five main independent tasks that are executing concurrently:

1. "ShipMovUHCtrl" state communicate with the"Ship1,Ship2,Ship3…"to get the information from the Ships and issue the movement order for them according to what condition that Ships are in.
2. "ShipMovUHCtrl" and "QueueCtrl" are interacting with eachother to control the queue of the ships that are waiting to pass the canal or lake "
3. "ShipMovUHCtrl" and LkOccpCtrl" are communicating to check each lock occupancy. When a ship requests the passage lock occupancy will be done for the closest lock to the ship in the same direction of the ship movement.

Avoiding collision of ships in the lock system is the first safety issue to be checked.

4. “ShipMovUHCtrl”, “WlevelS”, “ValveSCtrl” and “ValveCtrl” stands for controlling ship movement in up hill direction, water level sensor indicator, valve sensor controller and valve controller, they interact with each other to check the other safety issues with the water level. After the measure of water level of the locks that ship is located in and the lock that ship want to move in, sent to the “ShipMovUHCtrl”, it compares the measurments and if the numbers do not match, “ShipMovUHCtrl” sends the necessary order to the “ValveSCtrl”. The order indicate the opening of the valve in the lock with higher water level. “ValveSCtrl” sends the order to the “ValveCtrl”, the valve of the lock with higher level of water will be opened until the water levels get match, for this reason “WlevelS” must constantly check and measure the water level of the two locks.
5. In this process “ShipMovUHCtrl”, “GateSCtrl” and “GateCtrl” interact with each other so that the gate of the lock that ship is moving into gets open on time. In Up hill movement the water level of lock n+1 is higher than the water level of lock n, so when ship is located in lock n and request movement into the lock n+1 the gate between these two locks should be open only and only when the water level of these two adjacent locks are equal, other wise the ship in lock n will be drowned.

Figure 16, 16 and 17show details of concurrent processes inside each of the three main state charts. Figure 16 is a break down of concurrent control tasks in the “UpHillMovCtrl” state.

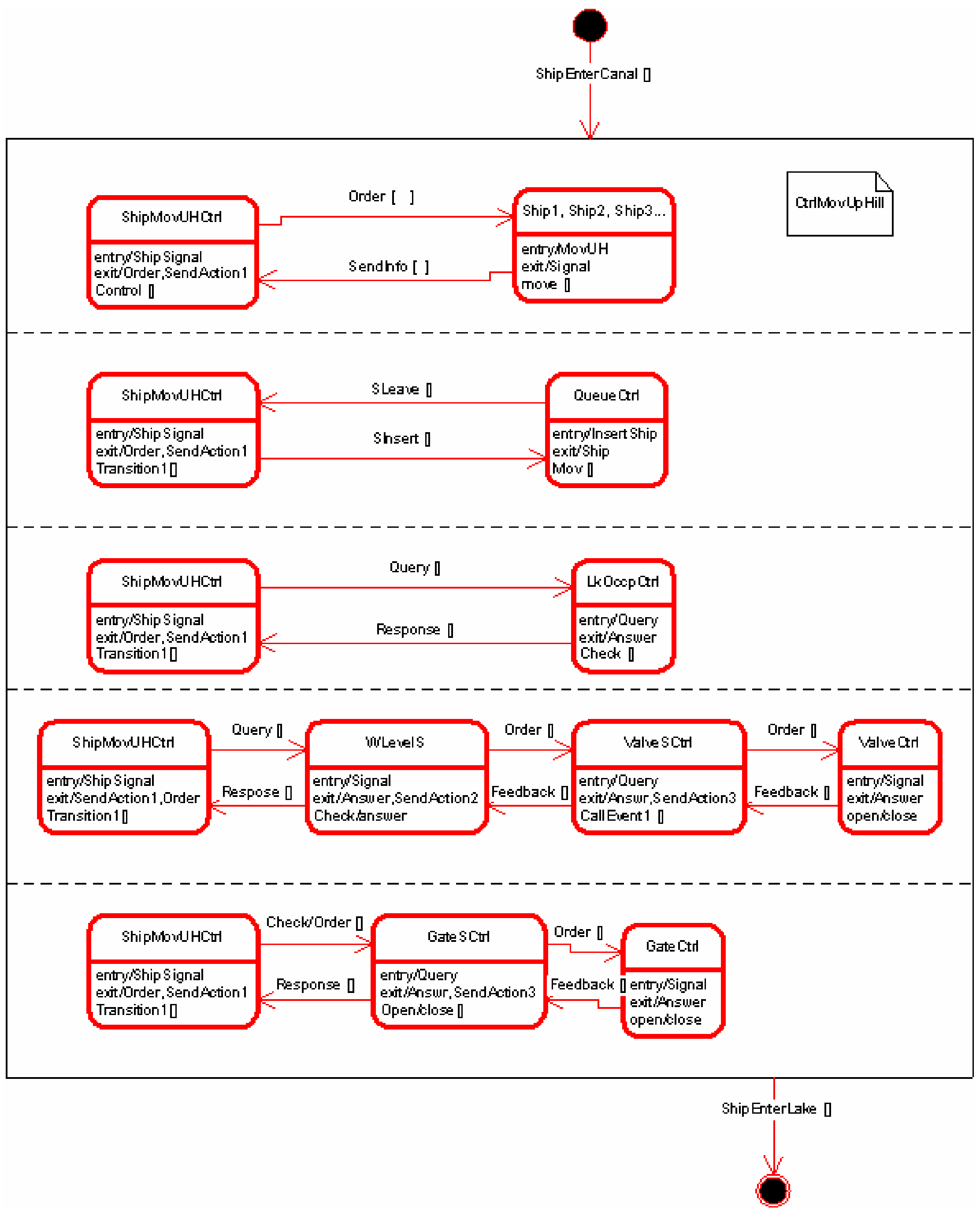


Figure 16. State chart of concurrent processes in the “UpHillMovCtrl” movement.

Figure 17 shows the independent concurrent behaviors in the ship movement control through the lake:

1. "ShipMovLakCtrl" and "Ship1, Ship2, Ship3…" are communicating to get the information like lake passage request, status of the ship and sending the orders to the ships.
2. In the second process "ShipMovLakCtrl" and "QueueCtrl" communicate with each other in order to control the queue of the ships waiting to pass through the lake. As we mentioned before, the lake between the two series of locks [9] is a oneway narrow water way which means that ships traveling in opposite directions are not allowed to pass in the lake at the same time. When Ships request the passage from both sides of the lake, "ShipMovLakCtrl" puts them in a queue and the priority is with the ship that request "lake passage" before the others, basically the priority rule is FIFO (First In First Out).

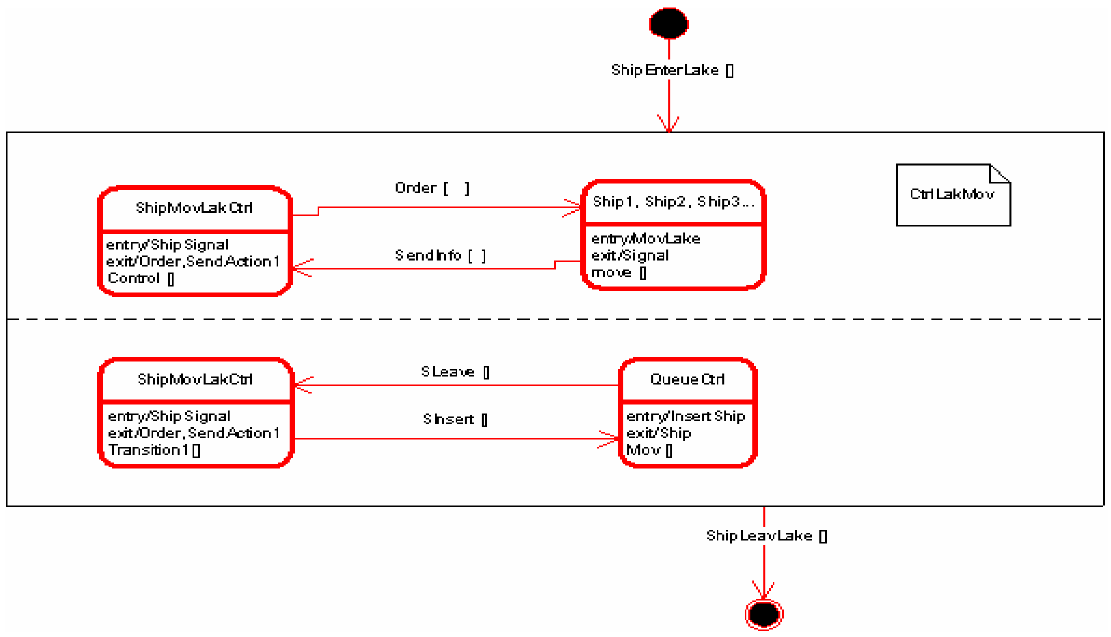


Figure 17. The state chart of all the concurrent processes in the "LakeMovCtrl" state chart.

Figure 18 is a break down of concurrent control tasks in the “DownHillMovCtrl” state. There are five main independent tasks that are executing concurrently:

1. “ShipMovDHCtrl” state communicate with the“Ship1,Ship2,Ship3…”to get the information from the Ships and issue the movement order for them according to what condition that Ships are in.
2. “ShipMovDHCtrl” and “QueueCtrl” interact with each other to control the queue of the ships that are waiting to pass the lake “
3. “ShipMovDHCtrl” and LkOccpCtrl” communicate to check the occupancy of each lock. When a ship requests the passage, lock occupancy will be checked for the closest lock to the ship and of course in the same direction of the ship movement. This is one of the safety issues that should be checked in order to not have a collision in the lock system.
4. “ShipMovDHCtrl”, “WlevelS”, “ValveSCtrl” and “ValveCtrl” interacte with each other to check safety issues associated with water level. After the measure of water level of the lock that ship is located in and the lock that ship want to move in, sent to the “ShipMovDHCtrl”, it compares the numbers and if the water levels do not match “ShipMovDHCtrl” sends an order to the “ValveSCtrl”, the order will be like for example:” open valve of the lock with higher water level” which in the down hill case is the water level of the lock that ship is located in. “ValveSCtrl” sends the order to the “ValveCtrl”, the valve of the lock with higher level of water will be opened until the water levels get match, for this reason “WlevelS” must constantly check and measure the water level of the locks.

5. In this process “ShipMovDHCtrl”, “GateSCtrl” and “GateCtrl” interact with each other so that the gate of the lock that ship is moving into gets open on time. In down hill movement the water level of lock n+1 is lower than the water level of lock n, so when ship is located in lock n and request movement into the lock n+1 the gate between these two locks should be open only and only when the water level of these two adjacent locks are equal, other wise the ship in lock n will be drowned.

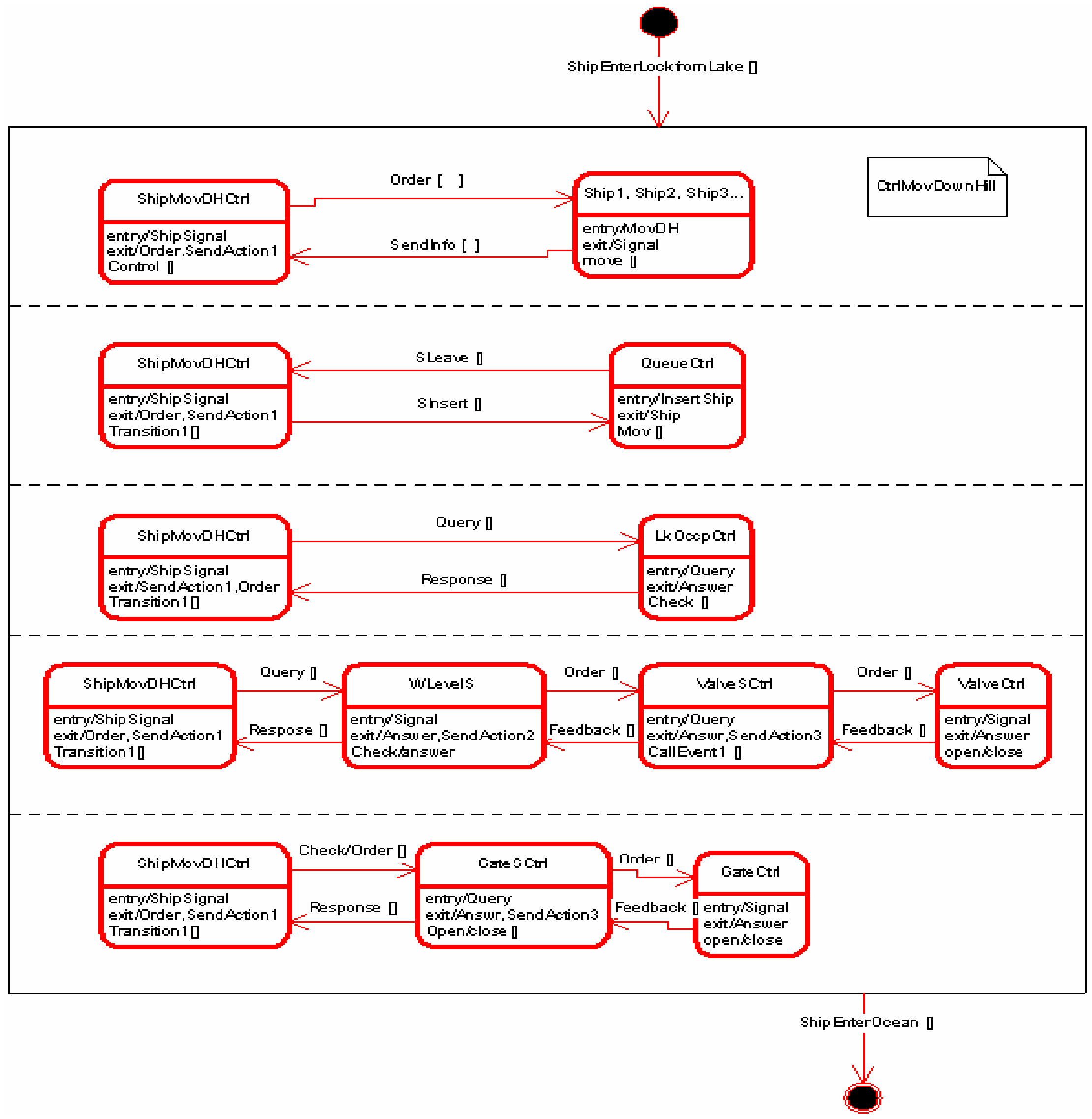


Figure 18. State chart of the concurrent processes in the “DownHillMovCtrl” movement.

Eventually after ship completes its down hill movement through the locks, it enters the ocean.

Detailed Representation of System Objects

Now that the high-level processes and their communication details are in place, we can expand the detail to account for the specific features (e.g. Number of locks and sensors of waterway) in the control system. The relationship of the water level control, valve sensors and actual valve is shown briefly in Figure 16, we show their relationship for each set of lock in Figure 19. The dashed line separates each independent process and the independent processes are executing in parallel with each other. As shown in Figure 19 shows that “ShipMovUpHillCtrl” state interacts with six valve sensor control and valve control. Although the “ShipMovUpHillCtrl” is the common state between all the processes in Figure 19 but each process is independent from the other because the “ValveSCtrl” and “ValveCtrl” states in each process are different from the other.

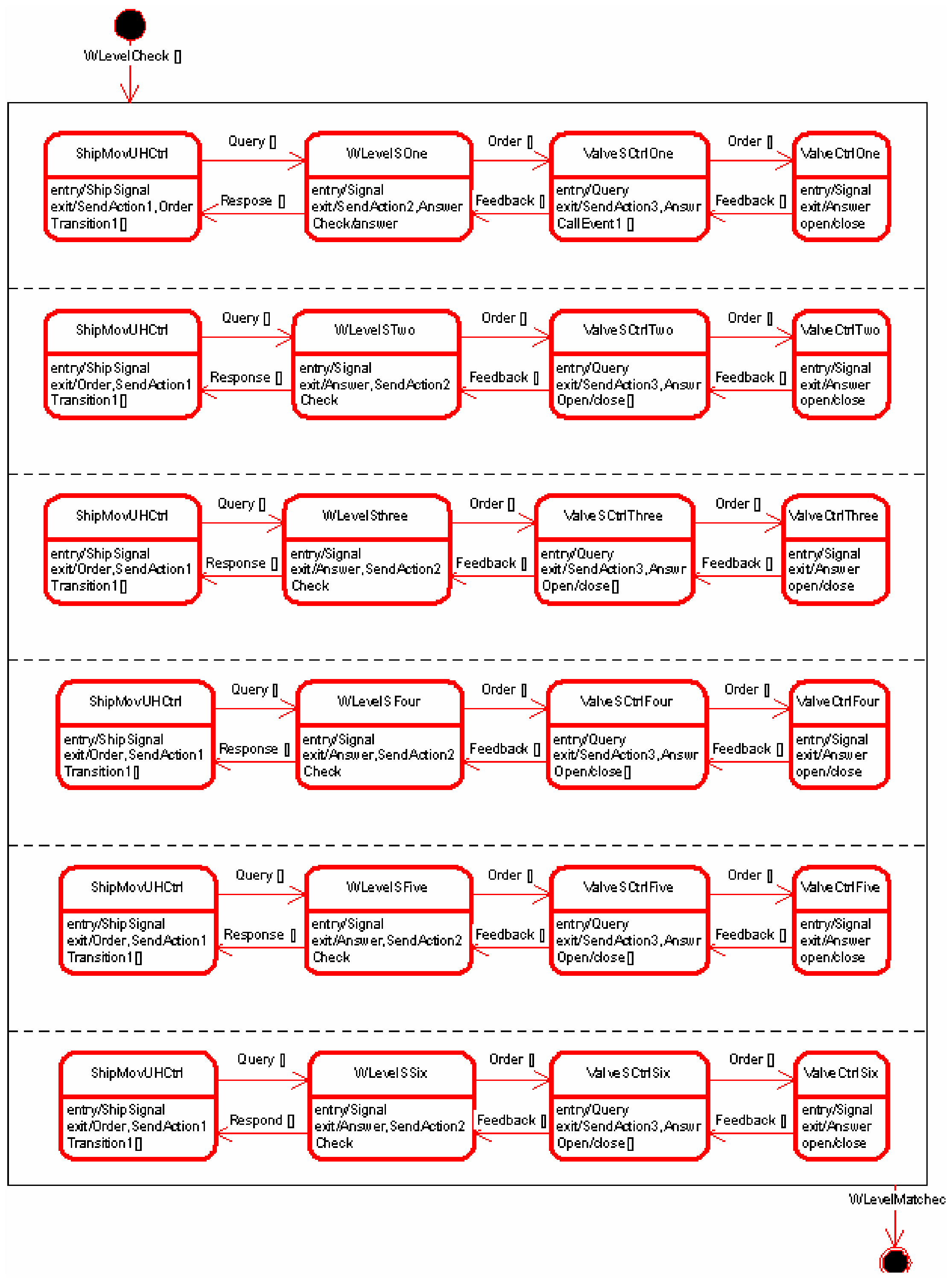
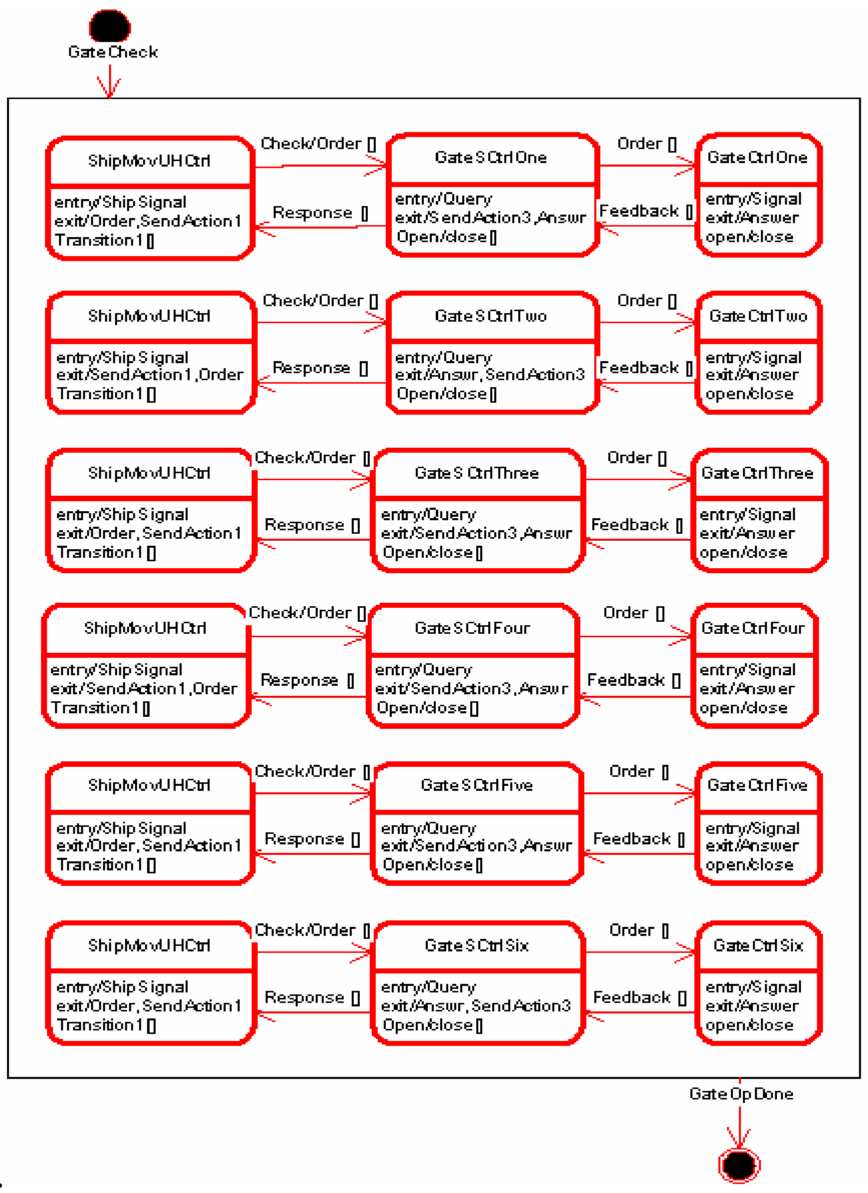


Figure 19. State chart of water Level control in each lock system .

The relationship of the water level control, gate sensors and actual gates is shown briefly in Figure 16, we show their relationship for each set of lock in Figure 20.



.

Figure 20. State chart of water gate control in each lock system

Strategy of implementation

The strategy of implementation is as follows: we translate state charts into behaviorally equivalent finite state process algebra code (FSP code) representation. We then use model-checking techniques in LTSA to detect violations of liveliness properties. We show how LTSA can be applied to the generated process algebra for the detection of deadlocks. Finally, we demonstrate how model-checking results can be related back to the original state charts model.

In the following fragments of code, we will briefly map the FSP code to the System architecture and show the FSP code for each state of the architecture of the canal control system shown in Figure 16, Figure 17 and Figure 18. We don't go much into detail of the code and only illustrate the part of it. You can find the complete code in Appendix A.

"**LakeMvCtrl":**

"`UPH`" stands for UPHILL,"`DNH`" stands for DOWNHILL.

```
property WATERWAY = (UPH[Ships].enter  -> UPH[1]
                      |DNH[Ships].enter -> DHS[1]),
UPH[i:Ships] = (UPH[Ships].enter -> UPH[i+1]
            |when(i==1)UPH[Ships].exit  -> WATERWAY
            |when( i>1)UPH[Ships].exit  -> UPH[i-1]),
DNH[i:Ships] = (DHS[Ships].enter -> DNH[i+1]
             |when(i==1)DNH[Ships].exit  -> WATERWAY
             |when( i>1)DNH[Ships].exit  -> DNH[i-1]).
```

Safety check: if both ships coming from opposite direction wants to pass the waterway, the unsafe situations will happen:

```
||UNSAFE = (UPH[Ships]:SHIP|DNH[Ships]:SHIP ||WATERWAY).
```

```
||SHIPS = (UPH[Ships]:SHIP||DNH[Ships]:SHIP ||WATERWAY||ONEWAY ).
```

Waterway is a common resource for the ships coming from different directions,

Dead Lock Check

If ships of both groups want to use it through a specific logic, for example ship 1 should pass when water way is empty and also ship2 at the opposite direction has the same logic and both of them have entered the water way so concerning their logic they will be there for ever, this is called deadlock:

```
||DEADLOCK = SHIPS>>{{UPH,DNH}[Ships].exit}.
```

Queue Ctrl:

When the lock system is full, ships are put in a queue and at the first moment that the lock system is ready to accept another ship they will be retrieved. The retrieval will be in FIFO (First In First Out) format:

```
property QUEUE = (Ships.enter  -> QUEUE[1]),
QUEUE[i:Ships] = (QUEUE[Ships].enter -> QUEUE[i+1]
    |QUEUE[Ships].exit  -> QUEUE[i-1]),
```

ShipMvUHControl:

This state plays a very critical role in the architecture because all the control actions will be done according to the ship situation an also ship location will be updated in this state:

```
SHIPMOVCTRL = (shippassrequest[i]->LOCKOCCUPANCYCHECK->WATERLEVELCHECK-
>SHIP[i+1]|SHIP[i+1]->GATECTRL).
```

“ShipMvDHCtrl” will be the same, although the functionality of these two control modules are the same but they control different movement

Ship:

At all times, ships should have an established links of communication with the ship movement controller. A ship should report any location change and request.

```
SHIP[x] = (approach->sendrequest->stop|move->end ->
SHIP[x]&&LOCKCHECK&&SHIPMOVCTRL).
SHIP[x] = SHIP[x+1],
```

Also the canal system is scalable in the terms of ship numbers, the scalability make the canal system more similar to the existing real canal systems like Panama Canal.

LkOccpCtrl:

Lock occupation should be done to issue a passage permit or locate the ship in a queue behind the canal.

```
LOCKCHECK[k] = (shiprequest ->lockcheck[j] ->response -
>SHIP&&GATECTRL&&QUEUE)
LOCKCHECK[k] = LOCKCHECK[k+1],
```

<u>GateSCtrl:</u>

With the help of this control sensor the state of the gate will be clear, this state gives the signal to the ship movement and the gate.

```
GATESCTRL[l] = (SHIPMOVCTRL->gatecheck->GATE&&SHIPMOVCTRL)

GATESCTRL[l] = GATESCTRL[l+1]
```

<u>GateCtrl:</u>

This state shows the physical actions of the gates.

```
GATE = (GATECTRL->open->close->GATE)
```

<u>WLevelCtrl:</u>

This state will show the water level state in each lock.

```
WLEVELCTRL = (GATECTRL->wlevel[n]->wleveleq[1])
```

wleveleq[1]: if wlevel[n] = wlevel[n+1]->wlevel[0]

<u>ValveSCtrl:</u>

Water level control state and Valve sensor control state have a mutual communication until the water level of two adjacent locks becomes equal. This procedure can happens for several locks at the same time.

```
VALVESCTRL = (wlevel[0]->openvalve[n+1]->WLEVELCTRL)

openvalve[n+1]: if UPH[i]->openvalvel[n]
```

<u>ValveCtrl:</u>

This state shows the physical actions of the valves.

```
VALVECTRL = (VALVESCTRL->open|close->VALVE)

VALVECTRL[p] = VALVECTRL[p+1]
```

Then we compose all the states together as they should compile concurrently:

```
CANALSYSTEM = ||
WATERWAY||QUEUE||SHIP||LOCKCHEK||SHIPMOVCTRL||WLEVELCTRL||GATESCTRL
||GATECTRL||VALVESCTRL||VALVECTRL
```

In the following chapter we discuss validation and verification of the Canal system that we design so far with LTSA and UPPAAL. The design with UPPAAL is based on the timed automaton, which are almost the same as our state charts except that they also carry the time issue.

Chapter 4: Verification and Validation

In this chapter we describe tool support for validation of canal system behavior:

1. Mechanically via LTSA.
2. Visually through XML and Scenebeans.
3. Mechanically through the use of timed automaton and UPPAAL.

We conclude this chapter with a summary of the complementary features of LTSA and UPPAAL.

Application of Canal System Operation

The canal system operation examined in this case study closely matches operations in the Panama Canal. For a detailed description, the interested reader can refer to Chapter 1.

Tool Support

The canal system consists of six locks. Water movement between locks is achieved by the canal valves and by the help of gravity. When a ship is exiting one lock and entering another lock, the water level of the adjacent locks should be the same in other words, the gate between two adjacent locks should not be opened until the water level of the adjacent locks are equal. The most common form of concurrent behavior occurs when more than one ship is passing through the canal. Water levels should check for more than one ship at the same time, and valve and gate operations for more than one lock should be checked at the same time. The control center should communicate with more than one ship captain at the same time.

In the next section we implement and test the highly concurrent canal system with LTSA and UPPAAL.

We use two behavioral modeling and verification tools, which cover all aspects of behavior described in this thesis. With LTSA, we model the canal system both with MSCs and with FSP code, and validate the system properties mechanically. Validation of system behavior through graphic animation is not in itself novel. The novelty of the approach is the firm semantic foundation on which animations are constructed, and the ease and flexibility with which an animation can be described and associated with the LTS model that drives it. We use part of our canal system to illustrate the approach to animation.

<u>Gate sensor communication with control center</u>:

The gate sensor in the canal system takes an input message from control center and outputs a message. The gate sensor is modeled below as an FSP process. FSP is the input notation for the LTSA tool. It is a simple process algebra used as a concise way to specify labeled transitions systems.

```
GATE = (ctrlctrmsg -> close-> GATE
| ctrlctrmsg -> open -> GATE).
```

In the above, "->" denotes action prefix and "|" choice. The choice between open or close is nondeterministic. In addition to the FSP code, we build a XML file indicating the speeds, directions, dimensions and shapes of components in the canal system. Then the XML file is linked to the FSP file in the FSP code. It follows that animation we see means the FSP code. If details of the FSP code implementations are correct, then this will be reflected through a visual verification of the canal system.

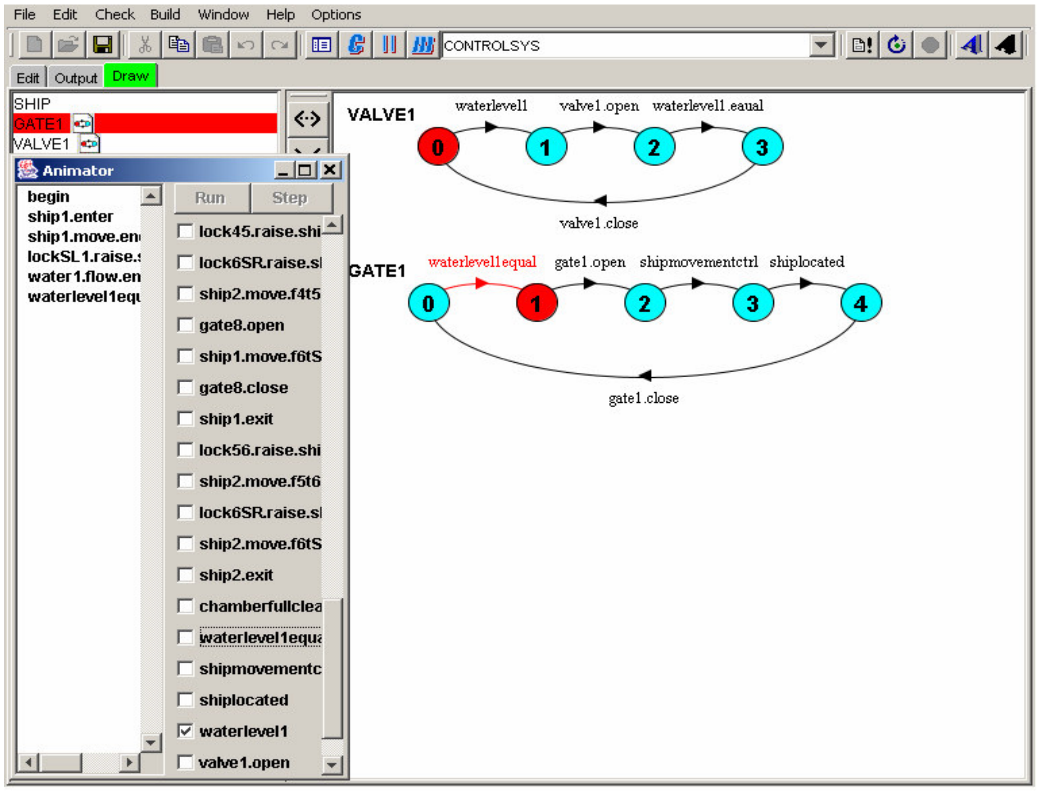
With help of UPPAAL we enter the time issue to the canal model and make it a real time system modeled with automata, then we verify the real-time canal model in UPPAAL, which is based on temporal logic.

Part 1. Mechanical Validation/Verification of the canal system with LTSA

We focus on the validation/verification of the canal system that we design in Chapter1 through 3.

Mechanical Validation

The first step of validation by LTSA is checking the system mechanically by the animator box (as described it in Chapter 1.) Figure 20 shows a typical screen shot of the LTSA analyzer in action.



```
begin
ship1.enter
ship1.move.en
lockSL1.raise.
water1.flow.en
waterlevel1equ
```

```
☐ lock45.raise.shi
☐ lock6SR.raise.s
☐ ship2.move.f4t5
☐ gate8.open
☐ ship1.move.f6tS
☐ gate8.close
☐ ship1.exit
☐ lock56.raise.shi
☐ ship2.move.f5t6
☐ lock6SR.raise.s
☐ ship2.move.f6tS
☐ ship2.exit
☐ chamberfullclea
☐ waterlevel1equa
☐ shipmovementc
☐ shiplocated
☑ waterlevel1
☐ valve1.open
```

Figure 21. Mechanically checking the canal system by Animator box.

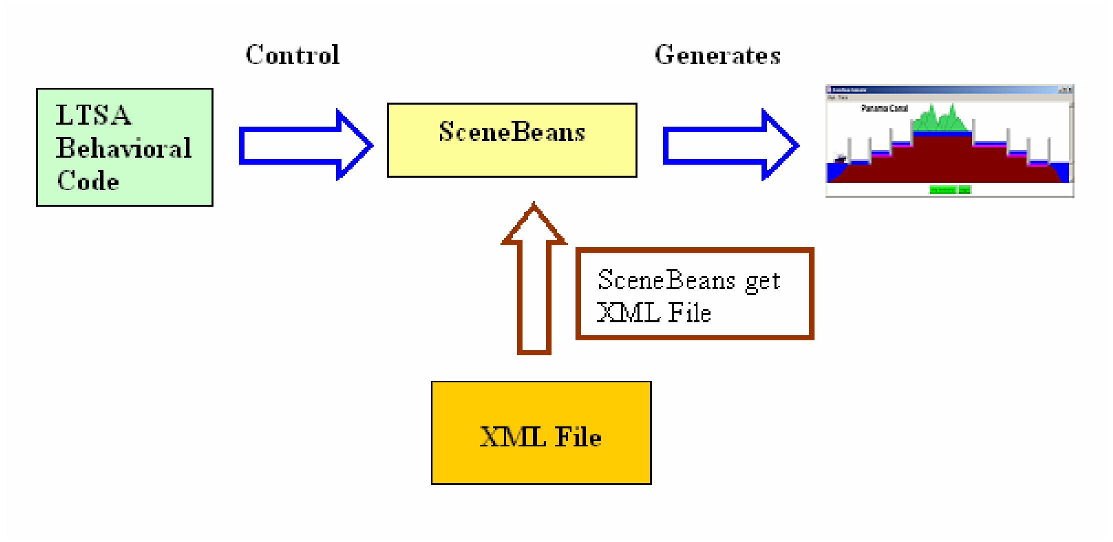


Figure 22. Animation Production Procedure Outline

While we are checking the system behavior with the animator box, LTSA also generates the associated state diagram to the component behavior that we check with Animator box.

Part 2. Visual Validation through XML and Scenebeans

Visual validation is an easy-to-understand means of verifying that behavior of the model is consistent with our intuitive feel for what should happen. If the visualization does not proceed as expected, then there is a strong likelihood that the behavior model will contain mistakes. After passing this stage successfully the next verification will be animation that is a visual verification of the canal system, it shows the behavior of the actual system and interaction between the components so before building the actual system we know how the designed canal system works. Figure 22 shows the step-by-step procedure for creating an animation production whose behavior is controlled by the labeled transition system in LTSA.

All the behavior of the system component are linked to the XML file following rules that will be explained in a moment. You can find the complete code of the FSP and XML file of the canal system in the Appendix. SceneBeans graphs transmit the combination of XML and FSP file into an animation.

SceneBeans is a Java framework for building and controlling animated graphics. It removes the hard work of programming animated graphics, allowing programmers to focus on what is being animated, rather than on how that animation is played back to the user. SceneBeans is based on Java Beans and XML. Its component-based architecture allows application developers to easily extend the framework with visual and behavioral components. It is used in the LTSA tool to animate formal models of concurrent systems. Linking the FSP code to XML file, which describe the animation components, creates animation. The behaviors discussed in the system architecture in Chapter 3 will control the animation components in XML file [13]. We load the XML file into the FSP code by:

```
animation CANAL = "xml/controlcentertest.xml"
```

There are two important syntaxes to link the behavior of the components in the FSP code to the animation components in the XML file: "`commands`" and "`event`". Commands are what start and control animations in the xml file. We associate actions with commands in the FSP code. Then, whenever an action occurs, it will call its associated command in the XML file. The command in the XML file usually resets and starts some behavior, announces events, sets values of objects, etc.

Syntax:

```
actions {<action_name>/<command_name>}
```

There should be a <command> tag in the xml file with the same name as the one in FSP code. Here is an example of the canal system:

```
actions{ begin / begin,
         SHIP1.enter/SHIP1.enter,…
```

SHIP1 will communicate with the SHIP process in FSP code that we described in the architecture model in chapter3. The related code in XML is: --

```
  <forall var="p" values="1 2 3 4 5 6 7 8 n">
    - <command name="SHIP${p}.enter">
```

As we mentioned another important syntax to link the behavior of the components in the FSP code to the animation components in the XML file is" event". Events can be thought of as conditions for the purposes of the LTS animation. Like commands, we associate events with actions as well. However, in this case, it means that the associated action cannot take place until that condition is true, i.e. until that event is announced.

```
    Syntax:controls {<action_name>/<event_name>}
```

All conditions are initialized to true to when the animation starts. We set them to false by announcing their negations. We first command should announce negations of all events. In the xml file, we announce events and their negations as follows:

```
    <announce event="<event_name>" />
```

There should be a <command> tag in the xml file with the same name as the one in FSP code. Here is an example of the canal system:

```
Controls {SHIP1.exit/SHIP1.exit,…
```

SHIP1 will communicate with the SHIP process in FSP code that we described in the architecture model in chapter3. The related code in XML is:

```
<forall var="p" values="1 2 3 4 5 6 7 8 n">

- <event name="SHIP${p}.exit">
```

Visual Experiments

The LTSA generated animation for one ship passing through the canal is shown in Figure 23 and for two ships is shown in Figure 24. In Figure 23 the ship is moving through the third chamber/lock, the third gate is still open waiting for the ship to finish its movement, the second chamber/lock is shown full because of the safety issues. As long as one the gates of lock are open the lock considered as full to prevent any probable accident. In Figure 23 water level sensor shows that the water level between the second lock and the third lock are equal. Figure 24 shows two ships passing through the canal. The ship in the first lock is waiting for the water level in the first lock and the second lock become equal, the valve in the second lock is open to move the excess water in the second lock to the first lock to match the water level between the two locks.

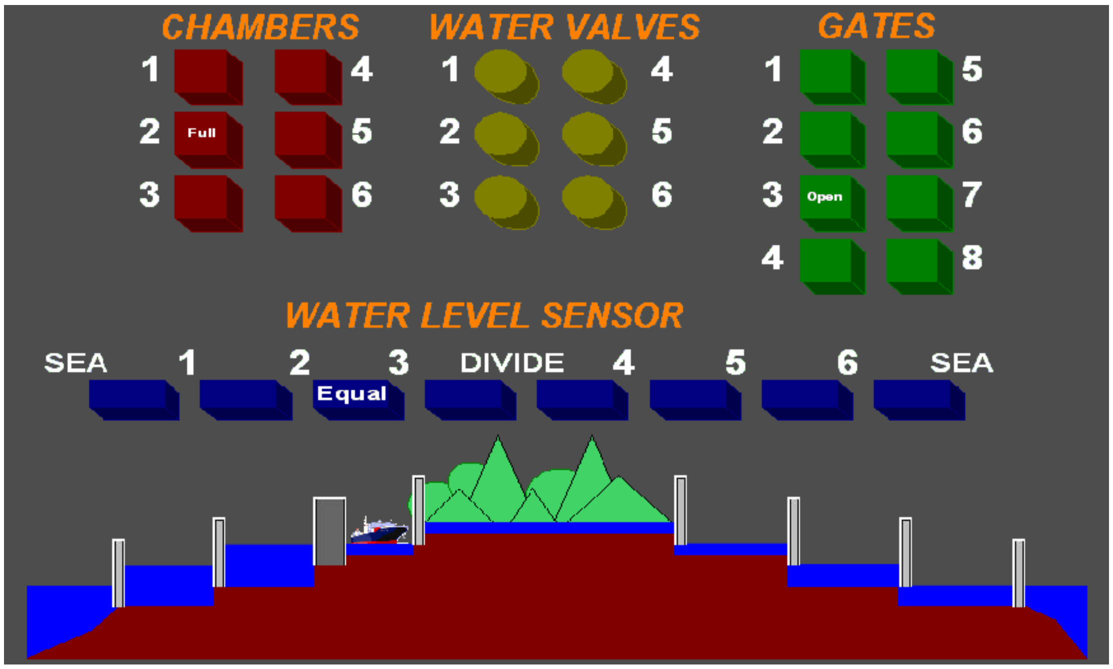


Figure 23. One Ship Passing through the Canal

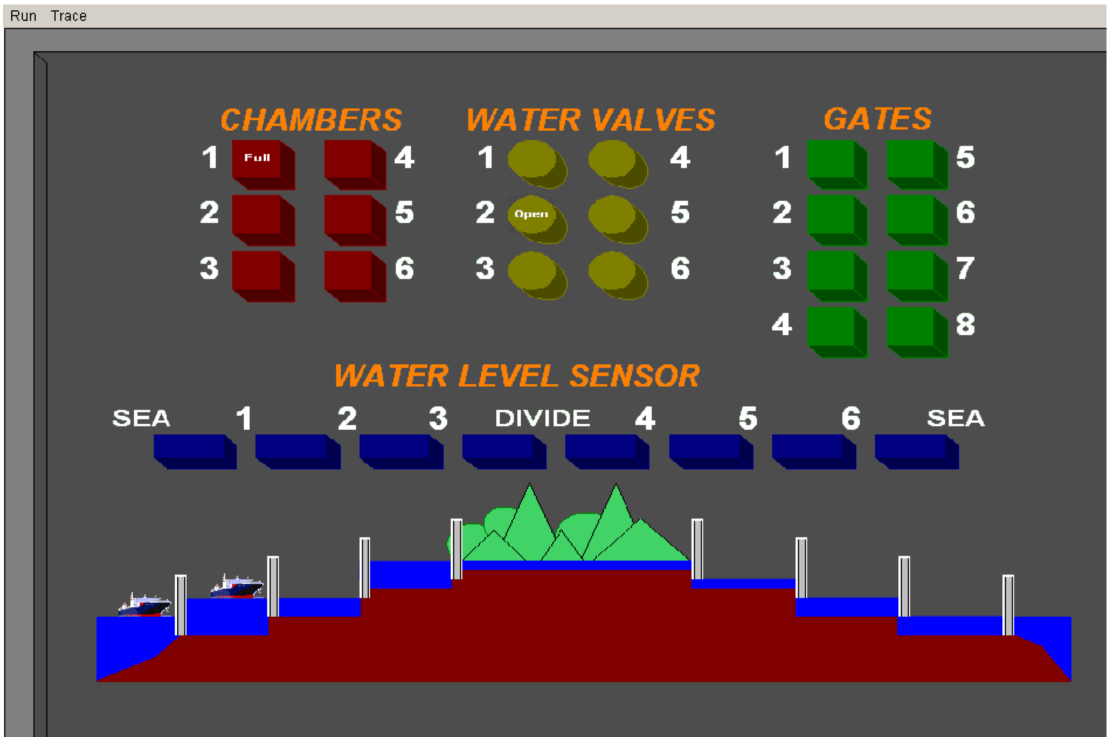


Figure 24. Two Ships Passing through the Canal

The visual validation extracted from the behavioral design shows that how the actual system works even before making any prototype.

Part 3. Validation/Verification of the Canal System with UPPAAL

As we mentioned in Chapter 3, we use UPPAAL to model and verify the canal system to insert the time issue into the system. A canal system like Panama Canal [14] works in a traditional way that time issue is sacrificed to ensure safe operation which means the ships wait in each lock more than it is needed to make sure all the conditions are satisfied to reach a safe state to start moving through the canal, but, with the help of UPPAAL we can enter time constraints into the system to make it more efficient.

UPPAAL Modeling Language

A timed automaton is a finite-state machine extended with clock variables. UPPAAL uses a dense-time model, where a clock variable evaluates to a real number. All the clocks progress synchronously. In UPPAAL, a system is modeled as a network of several such timed automata operating in parallel. The model is further extended with bounded discrete variables that are part of the state. These variables are used as in programming languages: they are read, written, and are subject to common arithmetic operations. A state of the system is defined by the locations of all automata, the clock constraints, and the values of the discrete variables. Every automaton may fire an edge (sometimes misleadingly called a transition) separately or synchronize with another automaton, which leads to a new state.

<u>Example. A Simple Lamp</u>

Figure 25 shows a timed automaton model for a simple lamp. The lamp has three locations: off, low, and bright. If the user presses a button, i.e., synchronizes with press?, then the lamp is turned on. If the user presses the button again, the lamp is turned off. However, if the user is fast and rapidly presses the button twice, the lamp is turned on and becomes bright. The user model is shown in Fig. 1(b). The user can press the button randomly at any time or even not press the button at all. The clock y of the lamp is used to detect if the user was fast (y < 5) or slow (y >= 5).

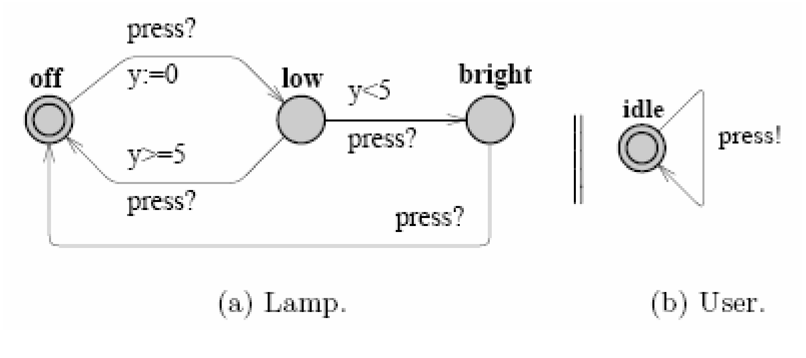


Figure 25. The simple lamp example.[15]

Model Formulation in UPPAAL

To begin design of the canal system, we draw the state chart diagrams for each component in the system. We assume bogus time constraints introduced as clock, as shown in Appendix B, for opening or closing the gates, matching the water level of two adjacent locks and complete movement of a ship from one lock to the other one.

Ship Operation

We first defined the ship operation or path, which will determine the liveliness of the system. Please refer to Figure 26. The ship requests passage as it approaches the canal. Then the ship will continue to pass through 8 gates before it has reached the end of the canal. The clock variable used in this state diagrams is x. The transition between ShipMoving and ShipPassedGate will not occur until x>=3. This indicates that it takes the ship at least 3 clock events before it is completely in a chamber.

Gate Operation

Figure 27 shows the operation of each gate in the canal. There are a total of 6 gates in the canal design. Each gate is identified with the g_num variable that classifies the gate id. The clock is defined by variable y. This design shows that the gate will open and close in at most 6 clock events.

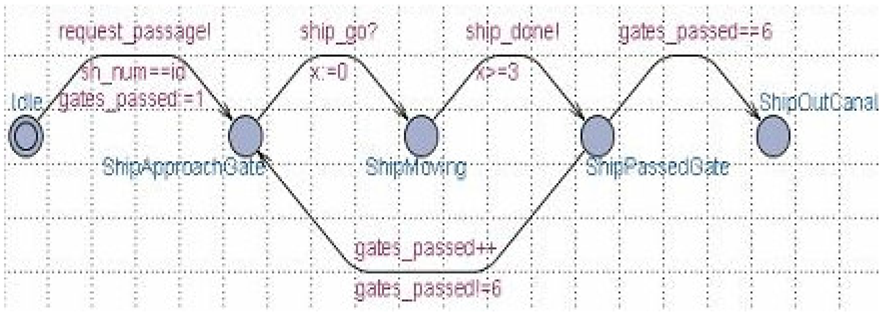


Figure 26. Ship Operation through the canal system.

Chamber/Lock Operation

The operation of each chamber in the canal is shown in Figure 28. There are a total of 6 chambers in the canal design identified through the ch_num or channel number variable. This state diagram does not contain any clocks. Rather, synchronizations are used to trigger transitions between states.

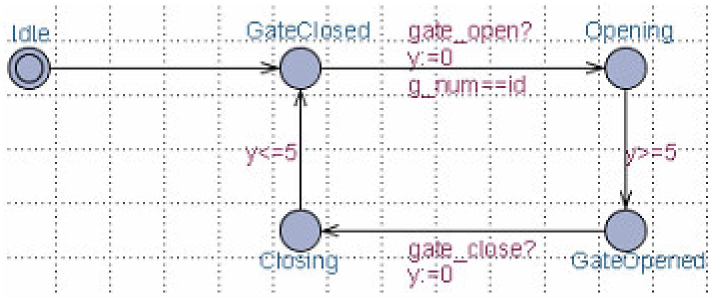


Figure 27. Gate operation through the canal system.

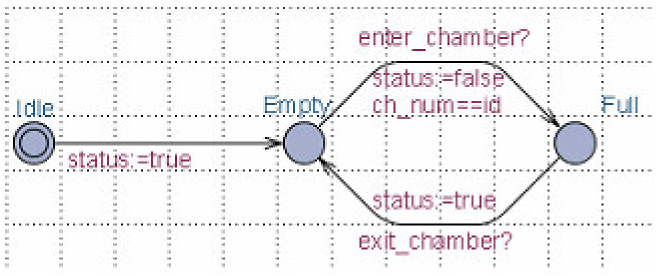


Figure 28. Chamber/Lock operation through the canal system

Canal Control Center Operation

Figure 29 shows the state diagram of the control center. After initializations of several variables the Control Center will remain in the state WaitForNextShip. Once a request for passage has occurred then the states will cycle through repeatedly until 6 gates have passed. The control center gives the commands (synchronizations) to open and close gates and water valves along with commanding the ship movement.

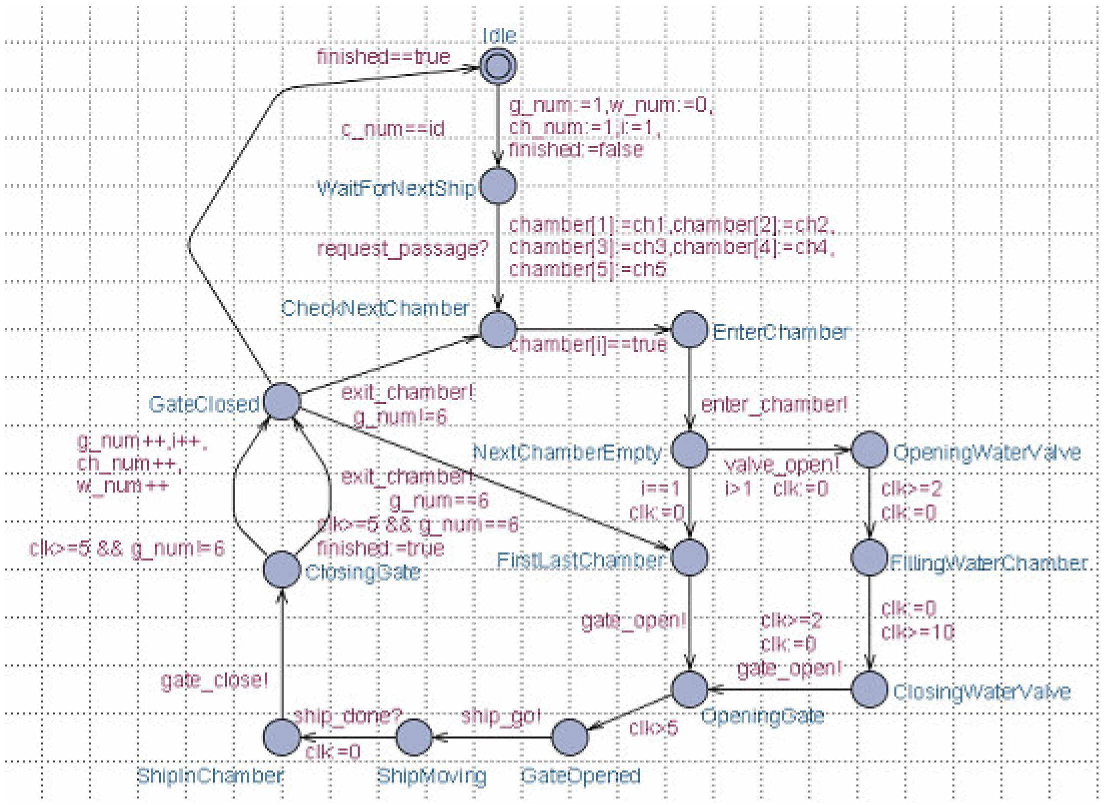
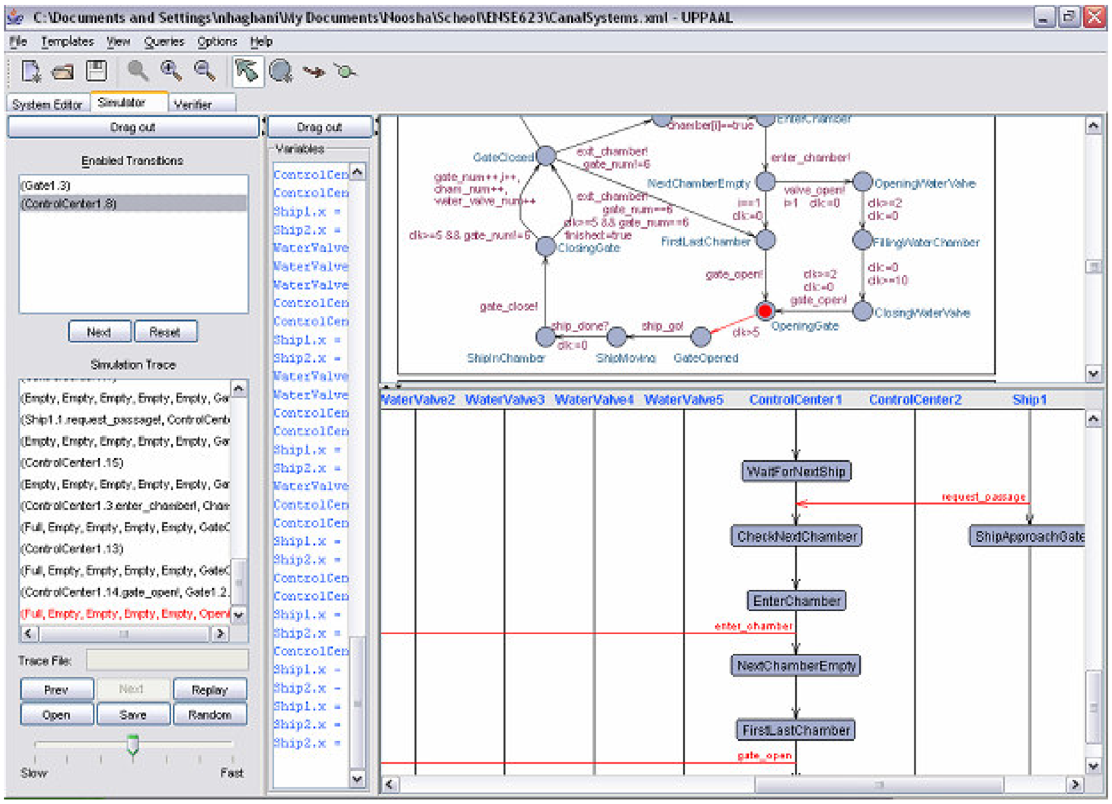


Figure 29. Canal control center operation.

Compilation and Simulation in UPPAAL

Following the completion of the finite state charts, the diagrams are compiled and then simulated. The simulation, traverses through each state chart in relation to one another, and automatically generates sequence diagram of the behavior of the components, as shown in Figure 30. During the simulation, we determined that the Ship could reach the state ShipOutCanal found in Figure 30 indicating the liveliness of the system. The validation as shown in Figure 30 is similar to the animator box validation in LTSA. Each transition that can happens between the message sequence charts corresponds to a valid transition



C:\Documents and Settings\nhaghani\My Documents\Noosha\School\ENSE623\CanalSystems.xml - UPPAAL

Enabled Transitions

(Gate1.3)
(ControlCenter1.8)

Simulation Trace

(Empty, Empty, Empty, Empty, Empty, Ga
(Ship1.1.request_passage!, ControlCent
(Empty, Empty, Empty, Empty, Empty, Ga
(ControlCenter1.15)
(Empty, Empty, Empty, Empty, Empty, Ga
(ControlCenter1.3.enter_chamber!, Cham
(Full, Empty, Empty, Empty, Empty, GateC
(ControlCenter1.13)
(Full, Empty, Empty, Empty, Empty, GateC
(ControlCenter1.14.gate_open!, Gate1.2
(Full, Empty, Empty, Empty, Empty, Open

Figure 30. Simulation and verification screen of the canal operation.

The designed system is scalable, as shown in Figure 29 and the number of ships that canal can handle can be entered to the model.

Complementary Features of LTSA/UPPAAL

In LTSA we start our design from the subsystem of the whole system and after all we define how the subsystems interact with each other, according to our definition of the interaction between the subsystems and also subsystems behaviors LTSA synthesizes the overall system architecture. The subsystems might have common components so their internal behavior interactions affects the overall system behavior, LTSA helps us to revise the system design as far as there is not any harmful interaction between the system components and the overall system behavior by checking for all the possible deadlocks and ensuring the safety in the system. In UPPAAL we model each subsystem as a real time system based on timed automata, the UPPAAL simulator simulate all the parallel subsystems. In UPPAAL there is no system overall architecture so the deadlock check only is done for the internal interactions between the subsystems, it means that if the subsystems of a canal do not have deadlock so the overall architecture of the canal might not have dead lock.

To have a more secure and deadlock free system we suggest to design and simulate a system with LTSA then if the system is a real time system apply the simulated design into the UPPAAL to achieve a real time deadlock free and secure system. By this time LTSA is not capable of simulating and verifying the real time systems. LTSA gives a visual verification, which shows the actual system behavior.

Chapter 5: Conclusions and Future Work

Summary of Contributions

Waterways such as canal systems are significant source of traffic congestion [16]. To mitigate these problems we applied the synthesis and verification approach using LTSA and UPPAAL tool to analyze the safety and liveliness/deadlock of the concurrent behavior of the canal system and entering the time issue into the canal system to make a more efficient system. The design is scalable which means it works for a number of ships that can possibly occupy the canal system. In Panama Canal seven ships occupy the canal system in one direction at the same time.

The key benefit of this approach is the formal modeling of concurrent behaviors in the lock system and verifying that unsafe operations will not occur in a reasonable amount of time with the number of ships that can possibly occupy the canal system. In the follow we bring a list of the thesis contributions:

1. **Scalability**: close to the actual Canal system.
2. **Deadlock free and safe model**: save money and time
3. **Soft real time canal system**: save time, we are not sure about money?

Future Work

We created two queues behind the canal and before the lake, the passage priority was according to the FIFO rule, however it is not really an efficient way for giving the priority to the ships, for example when a small and fast ship is behind a big and slow ship according to our priority rule the slower ship enters the lake or lock system before the smaller and faster ship, but it is more efficient if we give the priority to the smaller ship.

Our future research can be described as: Analyzing the fairness of the “ship movement control” while ships are passing through the lake between the two set of locks.

Appendices A

This appendix contains XML code of the behavioral modeling of the canal system.

//This module defines the dimension of the animation window and ship movement algorithm.//

```
<?xml version="1.0" ?>
- <animation width="1027" height="803">
  - <forall var="p" values="1 2 3">
      <behaviour id="ship${p}.move" algorithm="movePoint"
          event="ship${p}.move.end" />
//This module defines the movement algorithm of valve, gate,
chamber and water level//
  </forall>
  <behaviour id="valveopen.move" algorithm="movePoint"
      event="valveopen.move.end" />
  <behaviour id="gateopen.move" algorithm="movePoint"
      event="gateopen.move.end" />
  <behaviour id="chamberfull.move" algorithm="movePoint"
      event="chamberfull.move.end" />
  <behaviour id="waterlevel.move" algorithm="movePoint"
      event="waterlevel.move.end" />
  //clearing all gates status//
  - <forall var="i" values="1 2 3 4 5 6 7 8">
    - <define id="gate${i}">
      - <compose id="gate${i}.composition" type="switch">
        - <forall var="status" values="closed open">
          - <primitive type="sprite">
              <param name="src"
                  value="image/${status}gate.jpg" />
              <param name="hotspot" value="(0,66)" />
            </primitive>
          </forall>
        </compose>
      </define>
  </forall>
  //Control flow of each water level//
  - <behaviour id="water1.flow" algorithm="constantSpeedMove"
      event="water1.flow.end">
      <param name="speed" value="0.25" />
  </behaviour>
  - <behaviour id="water2.flow" algorithm="constantSpeedMove"
      event="water2.flow.end">
      <param name="speed" value="0.25" />
  </behaviour>
  - <behaviour id="water3.flow" algorithm="constantSpeedMove"
      event="water3.flow.end">
      <param name="speed" value="0.25" />
  </behaviour>
```

```
- <behaviour id="water4.flow" algorithm="constantSpeedMove"
     event="water4.flow.end">
   <param name="speed" value="0.25" />
</behaviour>
- <behaviour id="water5.flow" algorithm="constantSpeedMove"
     event="water5.flow.end">
   <param name="speed" value="0.25" />
</behaviour>
- <behaviour id="water6.flow" algorithm="constantSpeedMove"
     event="water6.flow.end">
   <param name="speed" value="0.25" />
</behaviour>
//Coloring the water levels//
- <define id="water1">
  - <style type="RGBAColor">
      <param name="alpha" value="1" />
      <param name="blue" value="1" />
      <param name="green" value="0" />
      <param name="red" value="0" />
    - <transform type="translate">
        <param name="translation" value="(230,630)" />
      - <transform type="rotate">
          <param name="angle" value="pi" />
        - <primitive type="rectangle">
            <animate param="height"
//Location of each water level//
            behaviour="water1.flow" />
            <param name="width" value="80" />
            <param name="height" value="20" />
          </primitive>
        </transform>
      </transform>
    </style>
</define>
- <define id="water2">
  - <style type="RGBAColor">
      <param name="alpha" value="1" />
      <param name="blue" value="1" />
      <param name="green" value="0" />
      <param name="red" value="0" />
    - <transform type="translate">
        <param name="translation" value="(310,610)" />
      - <transform type="rotate">
          <param name="angle" value="pi" />
        - <primitive type="rectangle">
            <animate param="height"
                behaviour="water2.flow" />
            <param name="width" value="80" />
            <param name="height" value="20" />
          </primitive>
        </transform>
      </transform>
    </style>
  </define>
- <define id="water3">
  - <style type="RGBAColor">
      <param name="alpha" value="1" />
      <param name="blue" value="1" />
      <param name="green" value="0" />
      <param name="red" value="0" />
    - <transform type="translate">
        <param name="translation" value="(390,590)" />
      - <transform type="rotate">
          <param name="angle" value="pi" />
        - <primitive type="rectangle">
            <animate param="height"
                behaviour="water3.flow" />
            <param name="height" value="20" />
            <param name="width" value="80" />
          </primitive>
        </transform>
      </transform>
    </style>
  </define>
- <define id="water4">
  - <style type="RGBAColor">
      <param name="alpha" value="1" />
      <param name="blue" value="1" />
      <param name="green" value="0" />
      <param name="red" value="0" />
    - <transform type="translate">
        <param name="translation" value="(690,590)" />
      - <transform type="rotate">
          <param name="angle" value="pi" />
        - <primitive type="rectangle">
            <animate param="height"
                behaviour="water4.flow" />
            <param name="width" value="90" />
          </primitive>
        </transform>
      </transform>
    </style>
  </define>
- <define id="water5">
  - <style type="RGBAColor">
      <param name="alpha" value="1" />
      <param name="blue" value="1" />
      <param name="green" value="0" />
      <param name="red" value="0" />
    - <transform type="translate">
        <param name="translation" value="(780,610)" />
      - <transform type="rotate">
          <param name="angle" value="pi" />
        - <primitive type="rectangle">
```

```
              <animate param="height"
                    behaviour="water5.flow" />
              <param name="width" value="90" />
            </primitive>
          </transform>
        </transform>
      </style>
    </define>
  - <define id="water6">
    - <style type="RGBAColor">
        <param name="alpha" value="1" />
        <param name="blue" value="1" />
        <param name="green" value="0" />
        <param name="red" value="0" />
      - <transform type="translate">
          <param name="translation" value="(870,630)" />
        - <transform type="rotate">
            <param name="angle" value="pi" />
          - <primitive type="rectangle">
              <animate param="height"
                    behaviour="water6.flow" />
              <param name="width" value="90" />
            </primitive>
          </transform>
        </transform>
      </style>
    </define>
    //Valve and gate detail descriptions//
  - <define id="valveopen">
    - <transform type="translate">
        <param name="translation" value="(0,0)" />
        <animate param="translation" behaviour="valveopen.move"
            />
      - <primitive type="sprite">
          <param name="src" value="image/valveopen.gif" />
          <param name="hotspot" value="(0,40)" />
        </primitive>
      </transform>
    </define>
  - <define id="gateopen">
    - <transform type="translate">
        <param name="translation" value="(0,0)" />
        <animate param="translation" behaviour="gateopen.move"
            />
      - <primitive type="sprite">
          <param name="src" value="image/gateopen.gif" />
          <param name="hotspot" value="(0,40)" />
        </primitive>
      </transform>
    </define>
  - <define id="chamberfull">
    - <transform type="translate">
```

```
        <param name="translation" value="(0,0)" />
        <animate param="translation"
            behaviour="chamberfull.move" />
      - <primitive type="sprite">
          <param name="src" value="image/chamberf.gif" />
          <param name="hotspot" value="(0,40)" />
        </primitive>
      </transform>
    </define>
  - <define id="waterlevel">
    - <transform type="translate">
        <param name="translation" value="(0,0)" />
        <animate param="translation" behaviour="waterlevel.move"
            />
      - <primitive type="sprite">
          <param name="src" value="image/sensoreq.gif" />
          <param name="hotspot" value="(0,25)" />
        </primitive>
      </transform>
    </define>
    //This module describes ship movement//
  - <forall var="p" values="1 2 3">
    - <define id="ship${p}">
      - <transform type="translate">
          <param name="translation" value="(0,155)" />
          <animate                                param="translation"
              behaviour="ship${p}.move" />
        - <primitive type="sprite">
            <param   name="src"   value="image/shipsmgb.gif"
                /> 
            <param name="hotspot" value="(50,38)" />
          </primitive>
        </transform>
      </define>
    </forall>
  - <draw>
  //This module locates each gate into the animation window//
    - <transform type="translate">
        <param name="translation" value="(150,650)" />
        <paste object="gate1" />
      </transform>
    - <transform type="translate">
        <param name="translation" value="(230,630)" />
        <paste object="gate2" />
      </transform>
    - <transform type="translate">
        <param name="translation" value="(310,610)" />
        <paste object="gate3" />
      </transform>
    - <transform type="translate">
        <param name="translation" value="(390,590)" />
        <paste object="gate4" />
    </transform>
  - <transform type="translate">
      <param name="translation" value="(600,590)" />
      <paste object="gate5" />
    </transform>
  - <transform type="translate">
      <param name="translation" value="(690,610)" />
      <paste object="gate6" />
    </transform>
  - <transform type="translate">
      <param name="translation" value="(780,630)" />
      <paste object="gate7" />
    </transform>
  - <transform type="translate">
      <param name="translation" value="(870,650)" />
      <paste object="gate8" />
    </transform>
  - <forall var="p" values="1 2 3">
      <paste object="ship${p}" />
    </forall>
    <paste object="valveopen" />
    <paste object="gateopen" />
    <paste object="chamberfull" />
    <paste object="waterlevel" />
    <paste object="water1" />
    <paste object="water2" />
    <paste object="water3" />
    <paste object="water4" />
    <paste object="water5" />
    <paste object="water6" />
  - <primitive type="sprite">
      <param name="src" value="image/control center.gif" />
      <param name="hotspot" value="(0,0)" />
    </primitive>
  </draw>
  //Gate operaton interacting with FSP code
- <forall var="i" values="1 2 3 4 5 6 7 8">
  - <command name="gate${i}.open">
      <set object="gateopen.move" param="from"
           value="(700+70*(${i}>4?1:0),160+60*((${i}-
           1)%4))" />
      <set object="gateopen.move" param="to"
           value="(700+70*(${i}>4?1:0),160+60*((${i}-
           1)%4))" />
      <set object="gateopen.move" param="duration"
           value="0.001" />
      <start behaviour="gateopen.move" />
      <set object="waterlevel.move" param="from"
           value="(40+90*${i},460)" />
      <set object="waterlevel.move" param="to"
           value="(40+90*${i},460)" />
    <set object="waterlevel.move" param="duration"
        value="0.001" />
    <start behaviour="waterlevel.move" />
    <set object="gate${i}.composition" param="current"
        value="1" />
  </command>
- <command name="gate${i}.close">
    <set object="gateopen.move" param="from" value="(0,0)"
        />
    <set object="gateopen.move" param="to" value="(0,0)" />
    <set object="gateopen.move" param="duration"
        value="0.001" />
    <start behaviour="gateopen.move" />
    <set object="gate${i}.composition" param="current"
        value="0" />
  </command>
</forall>
- <forall var="p" values="1 2 3">
  //Locating the ship//
- <command name="ship${p}.enter">
    <set object="ship${p}.move" param="from"
        value="(100,630)" />
    <set object="ship${p}.move" param="to"
        value="(150,630)" />
    <set object="ship${p}.move" param="duration" value="3"
        />
    <start behaviour="ship${p}.move" />
  </command>
- <command name="ship${p}.move.fSLt1">
    <set object="valveopen.move" param="from" value="(0,0)"
        />
    <set object="valveopen.move" param="to" value="(0,0)" />
    <set object="valveopen.move" param="duration"
        value="0.001" />
    <start behaviour="valveopen.move" />
    <set object="ship${p}.move" param="from"
        value="(150,630)" />
    <set object="ship${p}.move" param="to"
        value="(230,630)" />
    <set object="ship${p}.move" param="duration" value="3"
        />
    <start behaviour="ship${p}.move" />
  </command>
- <command name="ship${p}.move.f1t2">
    <set object="valveopen.move" param="from" value="(0,0)"
        />
    <set object="valveopen.move" param="to" value="(0,0)" />
    <set object="valveopen.move" param="duration"
        value="0.001" />
    <start behaviour="valveopen.move" />
    <set object="ship${p}.move" param="from"
        value="(230,610)" />
```

```
   <set object="ship${p}.move" param="to"
        value="(310,610)" />
   <set object="ship${p}.move" param="duration" value="3"
        />
   <start behaviour="ship${p}.move" />
</command>
- <command name="ship${p}.move.f2t3">
   <set object="valveopen.move" param="from" value="(0,0)"
        />
   <set object="valveopen.move" param="to" value="(0,0)" />
   <set object="valveopen.move" param="duration"
        value="0.001" />
   <start behaviour="valveopen.move" />
   <set object="ship${p}.move" param="from"
        value="(310,590)" />
   <set object="ship${p}.move" param="to"
        value="(390,590)" />
   <set object="ship${p}.move" param="duration" value="3"
        />
   <start behaviour="ship${p}.move" />
</command>
- <command name="ship${p}.move.f3tD">
   <set object="valveopen.move" param="from" value="(0,0)"
        />
   <set object="valveopen.move" param="to" value="(0,0)" />
   <set object="valveopen.move" param="duration"
        value="0.001" />
   <start behaviour="valveopen.move" />
   <set object="ship${p}.move" param="from"
        value="(390,570)" />
   <set object="ship${p}.move" param="to"
        value="(600,570)" />
   <set object="ship${p}.move" param="duration" value="6"
        />
   <start behaviour="ship${p}.move" />
</command>
- <command name="ship${p}.move.fDt4">
   <set object="valveopen.move" param="from" value="(0,0)"
        />
   <set object="valveopen.move" param="to" value="(0,0)" />
   <set object="valveopen.move" param="duration"
        value="0.001" />
   <start behaviour="valveopen.move" />
   <set object="ship${p}.move" param="from"
        value="(600,570)" />
   <set object="ship${p}.move" param="to"
        value="(690,570)" />
   <set object="ship${p}.move" param="duration" value="3"
        />
   <start behaviour="ship${p}.move" />
</command>
- <command name="ship${p}.move.f4t5">
```

```
  <set object="valveopen.move" param="from" value="(0,0)"
      />
  <set object="valveopen.move" param="to" value="(0,0)" />
  <set object="valveopen.move" param="duration"
      value="0.001" />
  <start behaviour="valveopen.move" />
  <set object="ship${p}.move" param="from"
      value="(690,590)" />
  <set object="ship${p}.move" param="to"
      value="(780,590)" />
  <set object="ship${p}.move" param="duration" value="3"
      />
  <start behaviour="ship${p}.move" />
</command>
- <command name="ship${p}.move.f5t6">
  <set object="valveopen.move" param="from" value="(0,0)"
      />
  <set object="valveopen.move" param="to" value="(0,0)" />
  <set object="valveopen.move" param="duration"
      value="0.001" />
  <start behaviour="valveopen.move" />
  <set object="ship${p}.move" param="from"
      value="(780,610)" />
  <set object="ship${p}.move" param="to"
      value="(870,610)" />
  <set object="ship${p}.move" param="duration" value="3"
      />
  <start behaviour="ship${p}.move" />
</command>
- <command name="ship${p}.move.f6tSR">
  <set object="valveopen.move" param="from" value="(0,0)"
      />
  <set object="valveopen.move" param="to" value="(0,0)" />
  <set object="valveopen.move" param="duration"
      value="0.001" />
  <start behaviour="valveopen.move" />
  <set object="ship${p}.move" param="from"
      value="(870,630)" />
  <set object="ship${p}.move" param="to"
      value="(930,630)" />
  <set object="ship${p}.move" param="duration" value="3"
      />
  <start behaviour="ship${p}.move" />
</command>
- <command name="ship${p}.exit">
  <set object="ship${p}.move" param="from"
      value="(930,630)" />
  <set object="ship${p}.move" param="to"
      value="(1130,630)" />
  <set object="ship${p}.move" param="duration" value="3"
      />
  <start behaviour="ship${p}.move" />
</command>
```

```
</forall>
- <forall var="p" values="1 2 3">
  - <command name="lockSL1.raise.ship${p}">
      <set object="waterlevel.move" param="from" value="(0,0)"
          />
      <set object="waterlevel.move" param="to" value="(0,0)" />
      <set object="waterlevel.move" param="duration"
          value="0.001" />
      <start behaviour="waterlevel.move" />
      <set object="valveopen.move" param="from"
          value="(440,160)" />
      <set object="valveopen.move" param="to"
          value="(440,160)" />
      <set object="valveopen.move" param="duration"
          value="0.001" />
      <start behaviour="valveopen.move" />
      <set object="water1.flow" param="from" value="20" />
      <set object="water1.flow" param="to" value="0" />
      <start behaviour="ship${p}.move" />
      <start behaviour="water1.flow" />
    </command>
  - <command name="lock12.raise.ship${p}">
      <set object="waterlevel.move" param="from" value="(0,0)"
          />
      <set object="waterlevel.move" param="to" value="(0,0)" />
      <set object="waterlevel.move" param="duration"
          value="0.001" />
      <start behaviour="waterlevel.move" />
      <set object="valveopen.move" param="from"
          value="(440,220)" />
      <set object="valveopen.move" param="to"
          value="(440,220)" />
      <set object="valveopen.move" param="duration"
          value="0.001" />
      <start behaviour="valveopen.move" />
      <set object="chamberfull.move" param="from"
          value="(200,160)" />
      <set object="chamberfull.move" param="to"
          value="(200,160)" />
      <set object="chamberfull.move" param="duration"
          value="0.001" />
      <start behaviour="chamberfull.move" />
      <set object="ship${p}.move" param="from"
          value="(230,630)" />
      <set object="ship${p}.move" param="to"
          value="(230,610)" />
      <set object="ship${p}.move" param="duration" value="5"
          />
      <start behaviour="ship${p}.move" />
      <set object="water1.flow" param="from" value="0" />
      <set object="water1.flow" param="to" value="20" />
      <start behaviour="water1.flow" />
      <set object="water2.flow" param="from" value="20" />
```

```
   <set object="water2.flow" param="to" value="0" />
   <start behaviour="water2.flow" />
</command>
- <command name="lock23.raise.ship${p}">
   <set object="waterlevel.move" param="from" value="(0,0)"
        />
   <set object="waterlevel.move" param="to" value="(0,0)" />
   <set object="waterlevel.move" param="duration"
        value="0.001" />
   <start behaviour="waterlevel.move" />
   <set object="valveopen.move" param="from"
        value="(440,280)" />
   <set object="valveopen.move" param="to"
        value="(440,280)" />
   <set object="valveopen.move" param="duration"
        value="0.001" />
   <start behaviour="valveopen.move" />
   <set object="chamberfull.move" param="from"
        value="(200,220)" />
   <set object="chamberfull.move" param="to"
        value="(200,220)" />
   <set object="chamberfull.move" param="duration"
        value="0.001" />
   <start behaviour="chamberfull.move" />
   <set object="ship${p}.move" param="from"
        value="(310,610)" />
   <set object="ship${p}.move" param="to"
        value="(310,590)" />
   <set object="ship${p}.move" param="duration" value="5"
        />
   //ship movement interaction with FSP code regarding
        water level status//
   <start behaviour="ship${p}.move" />
   <set object="water2.flow" param="from" value="0" />
   <set object="water2.flow" param="to" value="20" />
   <start behaviour="water2.flow" />
   <set object="water3.flow" param="from" value="20" />
   <set object="water3.flow" param="to" value="0" />
   <start behaviour="water3.flow" />
</command>
- <command name="lock3D.raise.ship${p}">
   <set object="waterlevel.move" param="from" value="(0,0)"
        />
   <set object="waterlevel.move" param="to" value="(0,0)" />
   <set object="waterlevel.move" param="duration"
        value="0.001" />
   <start behaviour="waterlevel.move" />
   <set object="valveopen.move" param="from"
        value="(440,280)" />
   <set object="valveopen.move" param="to"
        value="(440,280)" />
   <set object="valveopen.move" param="duration"
        value="0.001" />
```

```
    <start behaviour="valveopen.move" />
    <set object="chamberfull.move" param="from"
          value="(200,280)" />
    <set object="chamberfull.move" param="to"
          value="(200,280)" />
    <set object="chamberfull.move" param="duration"
          value="0.001" />
    <start behaviour="chamberfull.move" />
    <set object="ship${p}.move" param="from"
          value="(390,590)" />
    <set object="ship${p}.move" param="to"
          value="(390,570)" />
    <set object="ship${p}.move" param="duration" value="5"
          />
    <start behaviour="ship${p}.move" />
    <set object="water3.flow" param="from" value="0" />
    <set object="water3.flow" param="to" value="20" />
    <start behaviour="water3.flow" />
  </command>
- <command name="lockD4.raise.ship${p}">
    <set object="valveopen.move" param="from"
          value="(510,160)" />
    <set object="valveopen.move" param="to"
          value="(510,160)" />
    <set object="valveopen.move" param="duration"
          value="0.001" />
    <start behaviour="valveopen.move" />
    <set object="water4.flow" param="from" value="0" />
    <set object="water4.flow" param="to" value="20" />
    <start behaviour="water4.flow" />
  </command>
- <command name="lock45.raise.ship${p}">
    <set object="waterlevel.move" param="from" value="(0,0)"
          />
    <set object="waterlevel.move" param="to" value="(0,0)" />
    <set object="waterlevel.move" param="duration"
          value="0.001" />
    <start behaviour="waterlevel.move" />
    <set object="valveopen.move" param="from"
          value="(510,160)" />
    <set object="valveopen.move" param="to"
          value="(510,160)" />
    <set object="valveopen.move" param="duration"
          value="0.001" />
    <start behaviour="valveopen.move" />
    <set object="chamberfull.move" param="from"
          value="(280,160)" />
    <set object="chamberfull.move" param="to"
          value="(280,160)" />
    <set object="chamberfull.move" param="duration"
          value="0.001" />
    <start behaviour="chamberfull.move" />
  <set object="ship${p}.move" param="from"
      value="(690,570)" />
  <set object="ship${p}.move" param="to"
      value="(690,590)" />
  <set object="ship${p}.move" param="duration" value="5"
      />
  <start behaviour="ship${p}.move" />
  <set object="water4.flow" param="from" value="20" />
  <set object="water4.flow" param="to" value="0" />
  <start behaviour="water4.flow" />
  <set object="water5.flow" param="from" value="0" />
  <set object="water5.flow" param="to" value="20" />
  <start behaviour="water5.flow" />
</command>
//interacting with FSP code to adjust the water level//
- <command name="lock56.raise.ship${p}">
  <set object="waterlevel.move" param="from" value="(0,0)"
      />
  <set object="waterlevel.move" param="to" value="(0,0)" />
  <set object="waterlevel.move" param="duration"
      value="0.001" />
  <start behaviour="waterlevel.move" />
  <set object="valveopen.move" param="from"
      value="(510,220)" />
  <set object="valveopen.move" param="to"
      value="(510,220)" />
  <set object="valveopen.move" param="duration"
      value="0.001" />
  <start behaviour="valveopen.move" />
  <set object="chamberfull.move" param="from"
      value="(280,220)" />
  <set object="chamberfull.move" param="to"
      value="(280,220)" />
  <set object="chamberfull.move" param="duration"
      value="0.001" />
  <start behaviour="chamberfull.move" />
  <set object="ship${p}.move" param="from"
      value="(780,590)" />
  <set object="ship${p}.move" param="to"
      value="(780,610)" />
  <set object="ship${p}.move" param="duration" value="5"
      />
  <start behaviour="ship${p}.move" />
  <set object="water5.flow" param="from" value="20" />
  <set object="water5.flow" param="to" value="0" />
  <start behaviour="water5.flow" />
  <set object="water6.flow" param="from" value="0" />
  <set object="water6.flow" param="to" value="20" />
  <start behaviour="water6.flow" />
</command>
- <command name="lock6SR.raise.ship${p}">
  <set object="waterlevel.move" param="from" value="(0,0)"
      />
```

```
        <set object="waterlevel.move" param="to" value="(0,0)" />
        <set object="waterlevel.move" param="duration"
            value="0.001" />
        <start behaviour="waterlevel.move" />
        <set object="valveopen.move" param="from"
            value="(510,280)" />
        <set object="valveopen.move" param="to"
            value="(510,280)" />
        <set object="valveopen.move" param="duration"
            value="0.001" />
        <start behaviour="valveopen.move" />
        <set object="chamberfull.move" param="from"
            value="(280,280)" />
        <set object="chamberfull.move" param="to"
            value="(280,280)" />
        <set object="chamberfull.move" param="duration"
            value="0.001" />
        <start behaviour="chamberfull.move" />
        <set object="ship${p}.move" param="from"
            value="(870,610)" />
        <set object="ship${p}.move" param="to"
            value="(870,630)" />
        <set object="ship${p}.move" param="duration" value="5"
            />
        <start behaviour="ship${p}.move" />
        <set object="water6.flow" param="from" value="20" />
        <set object="water6.flow" param="to" value="0" />
        <start behaviour="water6.flow" />
      </command>
    </forall>
  - <command name="chamberfullclear">
      <set object="chamberfull.move" param="from" value="(0,0)" />
      <set object="chamberfull.move" param="to" value="(0,0)" />
      <set object="chamberfull.move" param="duration"
          value="0.001" />
      <start behaviour="chamberfull.move" />
    </command>
</animation>
```

Appendix B

This Appendix contains UPPAAL code: In UPPAAL there isn't any coding like in LTSA, the base of the design is on the drawing automata state charts and defining the variables on the finite state charts edges, as shown in chapter 4.
// Here is the very top level system definition that UPPAAL requires.
Int n
Int x

```
Define ship system: ship1(1:n)
Define ship system: ship2(1:n)
Clock x, x<5
Define valve sensor system: valvesensor1(1:6)
Define valve sensor system: valvesensor2(1:6)
Define valve sensor system: valvesensor3(1:6)
Define valve sensor system: valvesensor4(1:6)
Define valve sensor system: valvesensor5(1:6)
Define valve sensor system: valvesensor6(1:6)
Clock x, x<3
Define valve system: Valve1(1:6)
Define valve system: Valve2(1:6)
Define valve system: Valve3(1:6)
Define valve system: Valve4(1:6)
Define valve system: Valve5(1:6)
Define valve system: Valve6(1:6)
Clock x, x<3
Define gate sensor system: gatesensor1(1:8)
Define gate sensor system: gatesensor2(1:8)
Define gate sensor system: gatesensor3(1:8)
Define gate sensor system: gatesensor4(1:8)
Define gate sensor system: gatesensor5(1:8)
Define gate sensor system: gatesensor6(1:8)
Define gate sensor system: gatesensor7(1:8)
Define gate sensor system: gatesensor8(1:8)
Clock y, y<3
Define gate system:Gate1(1:8)
Define gate system:Gate2(1:8)
Define gate system:Gate3(1:8)
Define gate system:Gate4(1:8)
Define gate system:Gate5(1:8)
Define gate system:Gate6(1:8)
Define gate system:Gate7(1:8)
Define gate system:Gate8(1:8)
Clock y, y<3
Define Lock system: lock1(1:6)
Define Lock system: lock2(1:6)
Define Lock system: lock3(1:6)
Define Lock system: lock4(1:6)
Define Lock system: lock5(1:6)
Define Lock system: lock6(1:6)
Clock x, x<5
```

Bibliography

1. Synthesis and Validation of High-Level Behavior Models for Narrow Waterway Management Systems, Mark Austin, Evangelos Kaisar, submitted to the Journal of Computing in Civil Engineering, ASCE, for possible publication

2. Concurrency: State Models & Java Programs, Jeff Magee & Jeff Kramer,2002

3. Highly Concurrent Shared Storage (2000), Khalil Amiri, Garth Gibson, Richard Golding,1998, International Conference on Distributed Computing Systems

4. Highly concurrent shared storage, Khalil Amiri, Garth A. Gibson, Carnegie Mellon University, Pittsburgh, PA, Richard Golding, Hewlett-Packard Laboratories, Palo Alto, CA, 2001

5. http://www.doc.ic.ac.uk/ltsa

6. http://www.uppaal.com

7. Closed World Specification of Embedded Real-Time Controllers, K. Brink, L. Bun, J. van Katwijk, W.J. Toetenel, Fac. of Tech. Math. & Inf., Delft Univ. of Technol., Netherlands,2000

8. S. Uchitel, J. Kramer and J. Magee. *Detecting Implied Scenarios in Message Sequence Chart Specifications.* In proceedings of the 9th European Software Engineering Conferece and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'01). Vienna, Austria. September 2001.

9. http://www.pancanal.com/eng/index.html

10. UML Use Case Diagrams, Engineering Notebook, C++ Report, Nov-Dec 98

11. Designing Software Product Lines With UML, Gomaa Hassan, 1998

12. Interval Temporal Logic, Antonio Cau, Ben Moszkowski and Hussein Zedan, Software Technology Research Laboratory, 1997

13. Graphical Animation of Behavior Models, Jeff Magee, Nat Pryce, Dimitra Giannakopoulou and Jeff Kramer {jnm, np2, dg1, jk}@doc.ic.ac.uk, Department of Computing, Imperial College of Science, Technology and Medicine, 2003

14. http://www.eclipse.co.uk/~sl5763/panama.htm

15. A Tutorial on Uppaal, November 2004, Gerd Behrmann, Alexandre David, and Kim G. Larsen Department of Computer Science, Aalborg University, Denmark{behrmann,adavid,kgl}@cs.auc.dk.

16. Kaiser E., Austin M.A., and Haghani A., "An Object-Oriented Approach for the Architecture Design of the Management of Narrow Passageways," International Workshop on Harbour, Maritime and Multinodal Logistics, Modeling and Simulation, Copacabana, Rio de Janeiro, Brazil, September 16-18, 2004