THE INSTITUTE FOR SYSTEMS RESEARCH

ISR TECHNICAL REPORT 2011-04

Compositional Behavior Modeling and Formal Validation of Canal System Operations with Finite State Automata

Mark Austin
John Johnson

Contents

Chapter 1

Introduction

1.1 Problem Statement

For more than two hundred years, canal systems have been designed to provide an economic and reliable way of transporting of high-bulk goods and commodities over long distances. During the 1800s, for example, inland waterways were built in North America and Europe to support new streams of commerce enabled by the industrial revolution. Today, modern canal systems act as critical links in world trade routes.

While it is certainly feasible to operate a canal system one ship at a time, economic pressures nearly always dictate that canals deal with the concurrent real-time behavior of many ships. Accordingly, the key objectives of canal traffic flow management systems are to: (1) prevent overloading of the canal, and facilities and services that might affect safety, (2) develop strategies to relieve these overloads, (3) oversee implementation of these strategies, and (4) minimize economic penalties on shipping operators due to traffic congestion. Industrial-age canal systems have kept these traffic management concerns in check through a reliance on human involvement in day-to-day operations. However, now that traffic demands in modern canal systems have far exceeded early expectations – Figure 1.1 shows, for example, crowding of ships in the Panama Canal – it is evident that constraints on capability (e.g., maximum throughput, minimizing delays due to accidents and maintenance) are increasingly due to a limited ability to sense the surrounding environment, control system responses, and look ahead and anticipate events [7, 17, 26, 33]. Therefore, in an effort to expand system functionality and improve performance canal management systems are moving toward an information-age implementations where automation handles some operations once handled by humans. Early indicators of this trend can be found in the Panama Canal, Turkey (Bosporus Straights) and Korea (Tsushima Strait), where large investments have already been

Figure 1.1: Collage of ship activity in the Panama Canal (Left-hand graphic shows transiting ships crowding Miraflores Locks and Lake; Right-hand graphic illustrates of limitations of present-day canal capacity).

made to develop traffic management systems for narrow waterways. In each of these cases, decision making is guided by GIS and data collected by land (sensors) centers [4, 14, 28].

We observe that as canal management systems become progressively diverse in their functionality, and solutions increasingly reliant on automation, the challenge in creating good system-level designs will steadily increase unless new approaches are developed. The traditional role of engineering analysis of waterway systems has been to focus on performance, where sophisticated techniques [9, 11, 41, 44] are justified by the adverse economics of poor system throughput (perhaps caused by adverse weather conditions or accidents). However, now that there is a general trend towards canal management systems relying on automation to achieve required levels of functionality and performance, there is a need for new models that can capture correctness of system functionality with respect to system goals. From the standpoint of system functionality, the natural question to ask is: how do we know that an automated canal management system will always do the right thing? The search for good answers to this question is complicated by several factors, including: (1) Concurrent behaviors in both the operation of the vessels and elements of the canal system (e.g., the pumps and lock gates) itself, and (2) A general trend toward the use of control strategies that are partially or fully decentralized. A third source of difficulty is due to the increased role of software necessary to the handle automation/control. Functionality for software systems is defined by logic (not differential equations). As as result, a small fault in the software implementation can trigger system level failures that are very costly and sometimes, even catastrophic [13, 16]. Lessons

learned from industry [16, 24, 37] indicate that almost all grave software problems can be traced back to conceptual mistakes made before the system implementation even started.

Transition from Industrial- to Information-Age Capability

The transition from industrial- to information-age capability brings with it a strong need to understand and overcome several important challenges. First, an implicit assumption in industrial-age system implementations is that humans will do the right thing at the right time. Not only do no such assurances exist for an automated system, but automated systems can sometimes fail in new and unexpected ways. Perhaps the automation will do more than required?

Lessons learned from industry [16, 24, 37] indicate that there are now many automated engineering systems with complexity approaching the point where validation of design correctness will be impossible without mechanisms for pre-deployment reasoning about system requirements and design built into the design process itself. These mechanism include [5, 36]:

1. **Formal Models.** To help prevent serious flaws in design and operation, design representations and validation/verification procedures need to be based on formal languages having precise semantics. The key problem with semi-formal validation procedures (e.g., UML and SysML diagrams) is that they lack the precise interpretation of scenarios needed for rigorous analysis and formal verification of system compliance. This can lead to system failure rates that are unacceptably high.

2. **Abstraction.** Abstraction mechanisms eliminate details that are of no importance when evaluating system performance and/or checking that a design satisfies a particular property. Figure 1.2 shows, for example, a strategy of simplifying assembly of multi-layer systems through repeated application of abstraction. While the "areas of concern" are likely to change from one layer to the next, we need the overall complexity of models (i.e., no of states and transitions) to remain independent of the underlying model fidelity. Otherwise, model size will eventually grow to a point of being computationally intractable.

3. **Decomposition.** Decomposition is the process of breaking a design at a given level of hierarchy into subsystems and components that can be designed and verified almost independently.

4. **Composition.** Composition is the process of systematically assembling a system from sub-systems and components.

In established approaches to system design, and as shown along the left-hand side of Figure 1.3, present-day procedures for "system testing" are executed toward the end of system development.

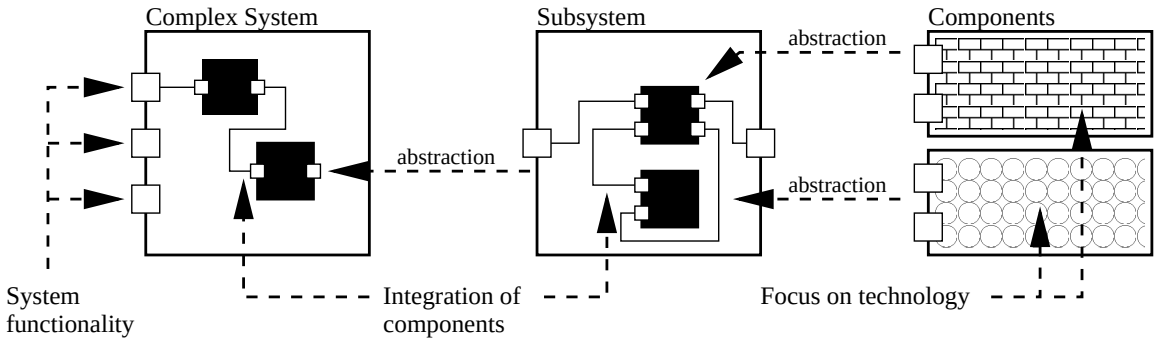

Figure 1.2: Simplifying system assembly through the use of abstraction (Adapted from Austin [3]).

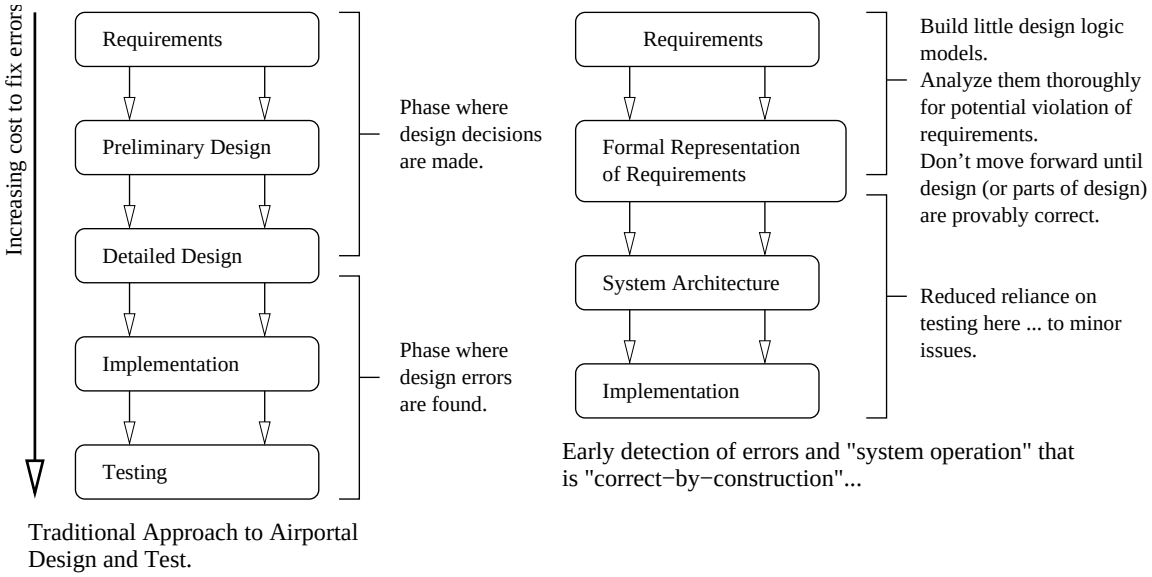

Figure 1.3: Pathways of traditional and model-based system development (Adapted from Sidorova [38]).

The well-known shortcoming of this approach is the excessive cost of fixing errors. Emerging approaches to system design [24, 38, 43] are based upon formal methods and selective use of design abstractions. The new approach benefits system design in two ways:

1. Concepts and notations from mathematics can provide methodological assistance, facilitating the communication of ideas and the thinking process, and
2. Formal methods allow us to calculate some properties of a design. requirements and using formal models for synthesis of architecture-level representations.

As illustrated along the right-hand side of Figure 1.3, the goal is to move design processes forward to the point where early detection of errors is possible and system operations are correct-by-construction [38].

1.2 Compositional Software Development

In each decade since the 1960s, remarkable advances in computer hardware and networking technology have allowed for expanded expectations of computing; in turn, ever expanding expectations have driven the need for new programming languages, new software development environments, and new application programs. As a case in point, prior to the early 1980s structured approaches to engineering software development with the Fortran and C programming languages were common place. Their primary purposes were to support development of numerical software and operating system applications respectively. Then beginning in the mid 1980s and with much fanfare, object-oriented approaches to engineering software development came into vogue. Early advocates promised: (1) improved mechanisms for the organization of software systems (e.g., through class hierarchies and inheritance mechanisms), and (2) improved economics through better reuse of software. After more than twenty years of use, it is now clear that early promises on ease of reuse, ease of software assembly, and associated productivity improvements have fallen short. One very simple problem is that you cannot create a new object-oriented program without having to actually design and implement a new class. A second problem with class hierarchies and inheritance mechanisms is that relationships among conceptual entities tend to be strongly coupled and quite inflexible. When a software system's requirements change, object-oriented software implementations do not naturally lend themselves to object removal, replacement and reconfiguration without significant perturbations to other parts of the software.

During the past 20-30 years, a parallel development in computing has focused on the design of computer languages for the modeling of systems having concurrent behaviors. Concurrency is a

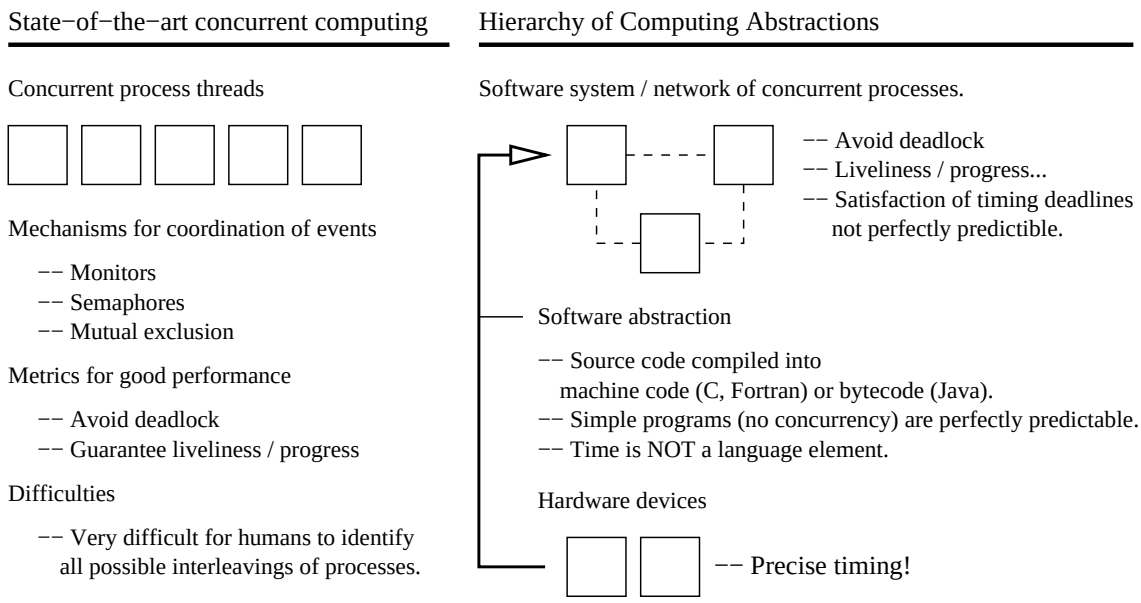

Figure 1.4: State-of-the-art computer modeling of systems having concurrent behaviors.

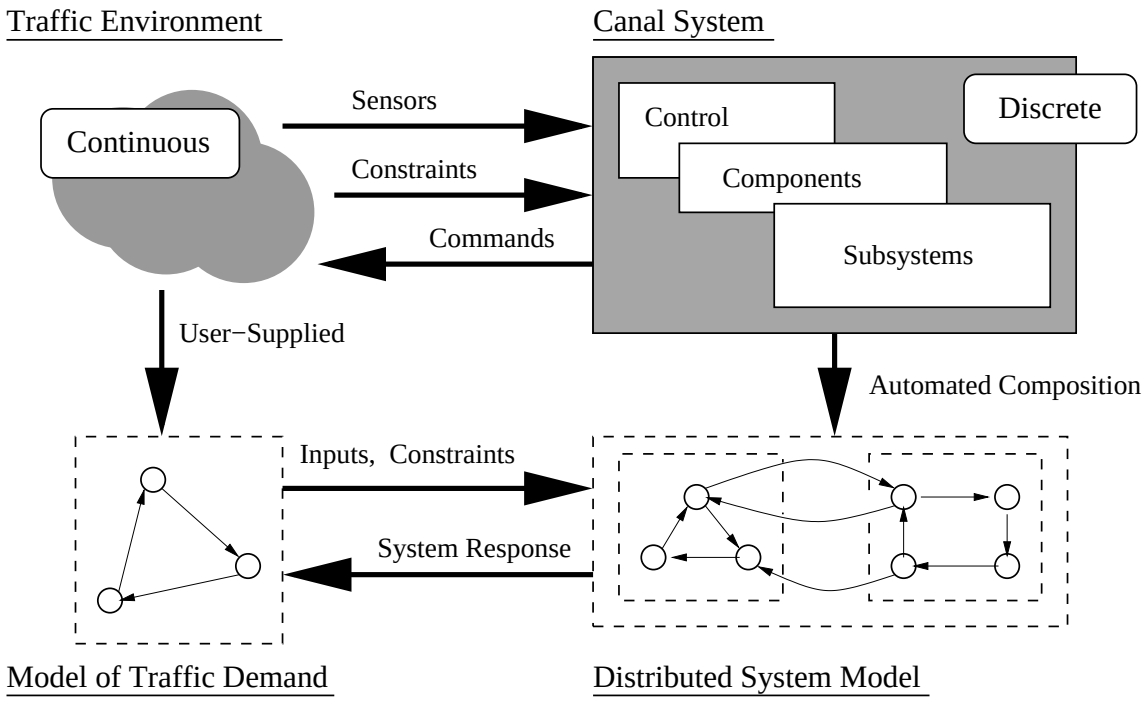

Figure 1.5: Automated assembly and behavior modeling of reactive processes.

fundamental aspect of life in a physical world. In addition to being logically correct, models need to balance the desire for autonomy (i.e., allowing for independent, concurrent, actions) against the need for scheduling/synchronization of events in order to get things done. State-of-the art procedures – see Figure 1.4 – rely on "threads of control" and operating system mechanisms (e.g., semaphores, monitors, mutual exclusion) for the synchronization and constrained scheduling of dependent processes. A good system design will achieve the dual objectives of avoiding deadlock and guaranteeing that "something good" will eventually happen.

The central problem with traditional approaches to software validation is that they rely on testing for the detection of errors. Although humans are quite adept at reasoning about small numbers of concurrent physical processes in their day-to-day life, identifying all of the possible interleavings among many concurrent system processes can be exceedingly difficult [39]. As a consequence, deadlocks can also be extremely difficult to find, sometimes going undetected for years [20]. Moving forward, we need methodologies for the synthesis and formal evaluation of safety-critical systems whose operations must be both reliable and predictable.

These weaknesses have led to the design of a new generations of languages that can efficiently handle the bottom-up dynamic assembly of component and software systems through scripting and composition mechanisms [29, 30, 31]. Instead of starting with highly nondeterministic mechanisms (i.e., top-down specification of threads) and relying on the system developer to prune nondeterminacy through the addition of constraints, the basic idea is to start with deterministic composable mechanisms, and introduce nondeterminism only where it is needed. This is a bottom-up approach to program assembly. Figure 1.5 shows, for example, a process modeling framework for the systematic assembly (composition of concurrent lower-level processes) of environmental (traffic demand) and distributed systems models (canal system) in a transportation systems application. Each component/subsystem will have behavior that can be modeled as finite state machine and will be implemented as a finite state process. Architecture-level models of behavior will be viewed as a network of interacting finite state machine processes, and will be synthesized through a bottom-up composition of subsystem- and component-level behaviors.

1.3 Scope and Objectives

Our first steps toward understanding the above-mentioned issues focused on the use of finite state automata for the top-down synthesis and analysis of behavior models for waterway management systems. We have proposed a methodology for the incremental transformation of informal operations concepts into system-level behavior models that are formal enough for automated validation [18, 19].

This report, in contrast, approaches the behavior modeling and validation problems from a combination of top-down and bottom-up perspectives. We explore the extent to which the principles of composition, hierarchical decomposition, and targeted abstraction can be applied to behavior modeling and formal validation of a real-world system such as the Panama Canal. System- and lockset-level behavior will be driven by bi-directional streams of ship traffic. At the lockset level, localized control will be employed for the safe, fair, and efficient scheduling of ship transit operations. At the system level, the primary design concerns include provision for: (1) adjustment of operations during emergency/maintenance events and, (2) cooperation of asynchronous lockset-level behaviors to ensure efficient transit of ships through the canal system. Because emergency/maintenance events will be detected at the lockset level, but controlled by a system-wide manager process, system-level canal behavior will be defined by a network of partially synchronized processes.

This step forward brings with it several important challenges. How, for example, should the behavior models and validation procedures be formulated so that lockset- and canal-system design concerns can be considered separately? Second, it is well known that safety properties are compositional, but that progress properties are not [24]. This study seeks to understand the consequences of the latter in a real-world application. We will see that naive approaches to composition lead to process models that quickly become computationally intractable. Therefore, to keep the size and complexity of the models in check, a second research challenge is one of combining formal models with strategies for process decomposition and abstraction in such a way that system properties may be formally evaluated with behavior models containing minimal detail.

The scope of investigation will be restricted to development of the two-layer behavior model and sets of functional/test requirements, as shown in Figures 1.6 and 1.7. We will use the labeled transition system analyzer (LTSA), a general-purpose tool designed for the high-level behavior modeling of concurrent and distributed (software) systems [23, 43]. The primary purpose of LTSA is to capture key interactions among systems and, thus, is most useful for modeling and validating system functionality. Architecture-level models of canal behavior will be created through the systematic assembly of lockset- and component-level behaviors. Design requirements for safety and liveliness will be formally expressed as finite state processes. We will demonstrate how safety validations can be accomplished through composition of finite state process models for canal behaviors and requirements. Key questions include: (1) how can LTSA be used to organize system behaviors into hierarchies of processes? (2) how can the overall size of process models be controlled through the selective abstraction of actions in lower level process models? and (3) how should validation work in conjunction with abstraction of process actions?

Chapter 2 covers the use of LTSA for process modeling of systems having concurrent behaviors, model checking, and procedures for the systematic organization of processes into layers. A new

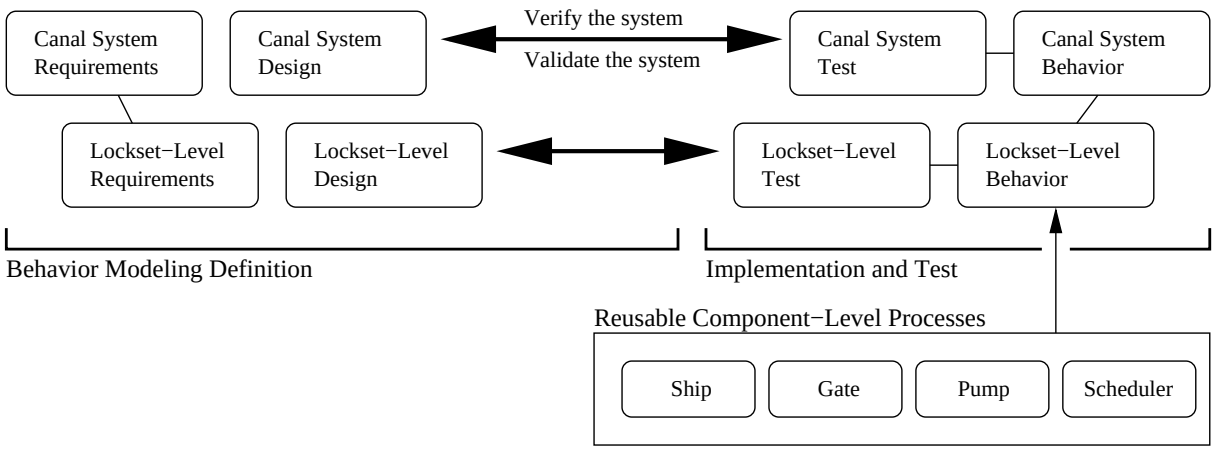

Figure 1.6: Step-by-step procedure for behavior model development, implementation, and testing/validation.

```
-------------------------------------------------------------------------------------
Canal System Concerns

Functional Requirements
=====================================================================================

1. When an emergency or maintenance occurs, all canal traffic at the event must be
   halted. Outbound traffic may clear the canal system. Traffic can resume after the
   emergency has been cleared and/or maintenance has been repaired.

2. All east- and west-bound ships must be guaranteed to reach the Atlantic and Pacific
   Oceans respectively.

-------------------------------------------------------------------------------------
Lockset Level Concerns

Functional Requirements
=====================================================================================

1. At any point in time, the scheduler must assign no more than one ship to a lockset.

2. All ships must acquire access to a lockset before they can depart.

3. Flooding must be prevented. A gate must not open until water levels on
   both sides of the gate have been equalized.

4. All ships that request passage through the lockset must eventually depart the
   lockset.

=====================================================================================
```

Figure 1.7: Canal and lockset-level requirements.

approach to validation based upon viewpoint-action-process traceability connections is proposed. Details of the Panama Canal and its past- and present-day operation are described in Chapter 3. Chapter 4 covers behavior modeling and formal validation of a two-stage lockset operation. In Chapter 5, behavior modeling and validation for system-level concerns (e.g., emergency and maintenance) is covered. Chapter 6 covers conclusions and future work.

Chapter 2

Process Modeling and Validation with LTSA

The labeled transition system analyzer (LTSA) is a tool for validating communication and sequencing among entities in systems containing concurrent behaviors [25, 42, 43]. In LTSA, processes correspond to sequences of actions. The power of LTSA lies in its ability to link processes through shared actions and compose processes side by side to create systems running concurrently on many levels. Spatial and temporal design concerns are not captured in LTSA (i.e., they are abstracted from modeling consideration). The textual representation is the finite state process (FSP) language. Labeled transition systems (LTSs) are the graphical representation. This tool runs as an applet on JAVA Runtime Environment 1.3 or higher.

2.1 From Requirements to Behavior with LTSA

Behavior in the canal system is a hybrid of functional and event-driven system behavior. From a vessel's point of view, behavior corresponds to sequences of tasks/functions that need to be completed, from arrival at the canal entrance through departure. When the tasks are complete, the corresponding processes terminate. From a canal management perspective, however, control/sensor processes run continuously, detecting and responding to events relevant to planned and unforeseen events.

As indicated in Figure 2.1, the transformation from requirements to high- level design occurs with the following activities: (1) identify main events, actions and interactions, (2) identify main processes, (3) identify and define properties of interest, and (4) structure processes into an

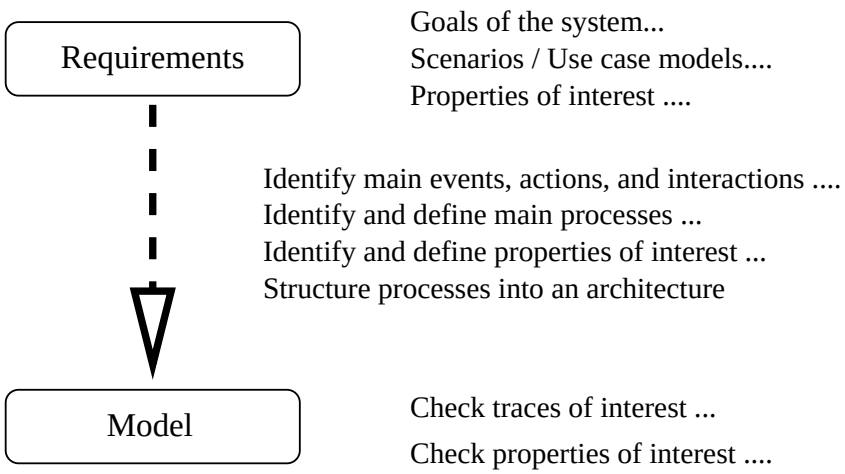

Figure 2.1: From requirements to architecture-level design.

architecture. A top-down specification of required behavior for components can be specified through the use of visual modeling languages such as UML. Each canal component or subsystem will have behavior that can be defined by a finite state machine. Architecture-level models of behavior will be viewed as a network of interacting finite state machine processes, and will be synthesized through a bottom-up composition of component-level behaviors (details below).

2.2 Finite State Processing (FSP) Language Features

The purpose of this section is to briefly introduce features of the FSP (Finite State Processing) language that will be used in development of behavior models for canal system operations. Process modeling of component-level behavior can be specified directly through FSP code. If concurrent behaviors have common elements, then there will be an interleaving of behaviors linked at common actions. Models of architecture-level behavior are obtained through the parallel composition of concurrent processes at the component level (details to follow below).

Relevant concepts include actions (including shared and guarded actions), parallel composition, choice, tagged processes (including action renaming), and modeling of shared resources. We also show how UML sequence diagrams can be guide the specification of component-level processes and their interaction.

Formal Definition of a Process. A labeled transition system (LTS) process contains: (1) all of the states that a process may reach, and (2) all of the transitions it may perform. In mathematical terms, a LTS process consists of a quadruple $(S, A, \triangle, q)$ where,

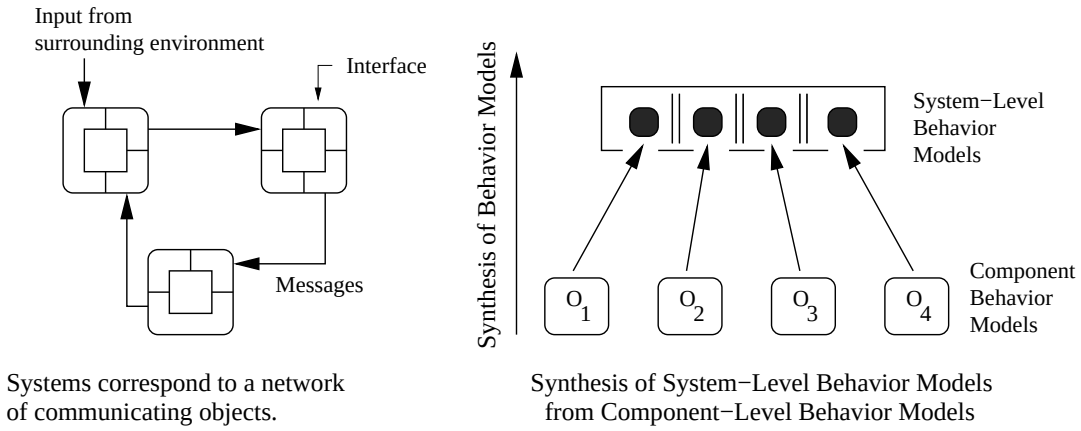

Figure 2.2: Two key elements of object-oriented development. Systems are modeled as networks of communicating sub-systems and objects. Models of system-level behavior are synthesized from component- and subsystem-level behaviors.

1. S is the set of states;

2. A = $\alpha P \cup \{\tau\}$, where αP is the communication alphabet of P which does not contain the internal action τ.

3. $\triangle \subseteq S \times A \times S$ denotes a transition relation.

4. q is a state in S which indicates the initial state of P.

The set of actions relevant to a behavioral description of a process P is called its alphabet. And it is denoted αP. We use the symbol π to represent an error state against which safety property violations may be tested (details to follow). A process that transitions into an error state may participate in no further transitions (i.e., the process deadlocks). The labeled transition system (LTS) for process P = $(S, A, \triangle, q)$ transits into another LTS of $P^{'} = \left(S, A, \triangle, q^{'}\right)$ with an action αA if an only if $\left(q, \alpha, q^{'}\right) \in \triangle$ and $q^{'} \neq \pi$ where π is an error (or deadlock) state. Mathematically we can state:

$$(S, A, \triangle, q) \quad \stackrel{a}{\rightarrow} \quad \left(S, A, \triangle, q^{'}\right) \tag{2.1}$$

if and only if $\left(q, \alpha, q^{'}\right) \in \triangle$ and $q^{'} \neq \pi$.

Rules on Restriction

$$\frac{P \xrightarrow{a} P'}{P \uparrow L \xrightarrow{a} P' \uparrow L} \qquad \left(a \in L, P' \neq \Pi\right) \tag{2.2}$$

$$\frac{P \xrightarrow{a} \Pi}{P \uparrow L \xrightarrow{a} \Pi} \qquad (a \in L) \tag{2.3}$$

$$\frac{P \xrightarrow{a} P'}{P \uparrow L \xrightarrow{\tau} P \uparrow L} \qquad \left(a \notin L, P' \neq \Pi\right) \tag{2.4}$$

$$\frac{P \xrightarrow{a} \Pi}{P \uparrow L \xrightarrow{\tau} \Pi} \qquad (a \notin L) \tag{2.5}$$

Rules on Parallel Composition

$$\frac{P \xrightarrow{a} P'}{P \| Q \xrightarrow{a} P' \| Q} \qquad \left(a \notin \alpha Q, P' \neq \Pi\right) \tag{2.6}$$

$$\frac{P \xrightarrow{a} \Pi}{P \| Q \xrightarrow{a} \Pi} \qquad (a \notin \alpha Q) \tag{2.7}$$

$$\frac{Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P \| Q'} \qquad \left(a \notin \alpha P, Q' \neq \Pi\right) \tag{2.8}$$

$$\frac{Q \xrightarrow{a} \Pi}{P \| Q \xrightarrow{a} \Pi} \qquad (a \notin \alpha P) \tag{2.9}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P' \| Q'} \qquad \left(a \in \alpha P \cap \alpha Q, P' \neq \Pi, Q' \neq \Pi\right) \tag{2.10}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} \Pi} \qquad \left(a \in \alpha P \cap \alpha Q, P' = \Pi \vee Q' = \Pi\right) \tag{2.11}$$

Table 2.1: Translational Semantics for Restriction and Composition Operators

2.2.1 Actions in LTSA

Process behavior is defined through sequences of actions/transitions a process may perform. If `x` is an action and P is a process, then the action prefix `x->P` describes a process that initially engages in the action `x` and then behaves exactly as prescribed by P. In practical terms, an action might be a communication, a signals, or perhaps, traditional execution of a task.

From a modeling and design standpoint, the alphabet of a model needs to be chosen to support required decision making while also keeping models tractable. Thus, a key aspect of alphabet design is to ignore actions and properties not immediately relevant to a particular activity (or set of activities).

Observability of Actions. Observability of actions in a process can be controlled by a restriction operator $\uparrow$. In more detail, the notation $P \uparrow L.$ represents the process projected from P in which only the actions in the set L are observable. Equations 2.2 through 2.5 in Table 2.1 define the transitional semantics of the restriction operator. As a case in point, equation 2.2 should be read as follows: Suppose that process P transitions into $P^{'}$ through the application of action a. If action a belongs to the set of observable actions in L (i.e., a $\in L$) and $P^{'}$ is not a deadlock state (i.e., $P^{'} \neq \pi$), then action a will transition $P \uparrow L$ into $P^{'} \uparrow L$.

2.2.2 Parallel Composition in LTSA

Given two labeled transition systems (LTSs) P_1 and P_2, we denote the parallel composition $P_1 \| P_2$ as the LTS that synchronizes actions common to both processes and interleaves the remaining actions. By extension the architectural-level behavior model is defined by:

$$\text{Architecture-Level Behavior Model} = P_1 \| P_2 \| P_3 \cdots P_n \tag{2.12}$$

where P_i is the finite state model for the i-th component among n interacting components. A symbolic representation of this process is shown on the right-hand side of Figure 2.2. Joint behavior is the result of all LTSs executing asynchronously, but synchronizing on all shared message labels. At the component level, the nodes of a labeled transition system represent states the component can be in. At the architecture level, labeled transition system nodes represent system-level states, which, in turn, correspond to specific combinations of component-level states. Transitions are labeled with messages sent between component processes.

Algebraic Properties of Parallel Composition. Table 2.1 provides a formal definition of parallel composition of processes and their corresponding algebraic properties. The alphabet for $P_1 \| P_2$ is given by the union of alphabets for P_1 and P_2. Equations 2.6 through 2.11 define the transitional semantics of the parallel composition operator. As indicated in equations 2.10 and 2.11 the composition operator is both commutative and associative. Finally, it is important to note that the rules require that a composite process be trapped in an error state π if any of its constituent processes is trapped.

Example 1. Modeling Behavior of a Door and Door Handle. In the fragment of code:

```
HANDLE      = ( down -> up    -> HANDLE).
DOOR        = ( open -> close -> DOOR ).
```

defines processes for a handle and a door, each having behavior defined by transitions between two states. The alphabet for the HANDLE process is $(down, up)$. The DOOR process has an alphabet $(open, close)$. Accordingly, for the handle and door, transitions are `up` and `down` and `open` and `close`, respectively. Behavior of the handle is constrained to follow the action sequence: `down -> up -> down -> up -> down` ..... and so on. And behavior of the door is constrained to follow the action sequence: `open -> close -> open -> close -> open` ..... and so forth.

Behavior of the doorway system is defined via the parallel composition of `HANDLE` and `DOOR` processes, i.e.,

```
||DOORWAY = ( HANDLE || DOOR ).
```

Composite process definitions (e.g., DOORWAY) are prefixed by a double vertical bar to distinguish them from primitive process definitions. The composition result is illustrated in Figure 2.3.

The doorway process moves among four states having the following interpretation:

Component behaviors		Composed behavior	
Handle System	Door System	System Tuple	Simplified Notation
up	close	(up, close)	State 0
up	open	(up, open)	State 1
down	open	(down, open)	State 2
down	close	(down, close)	State 3

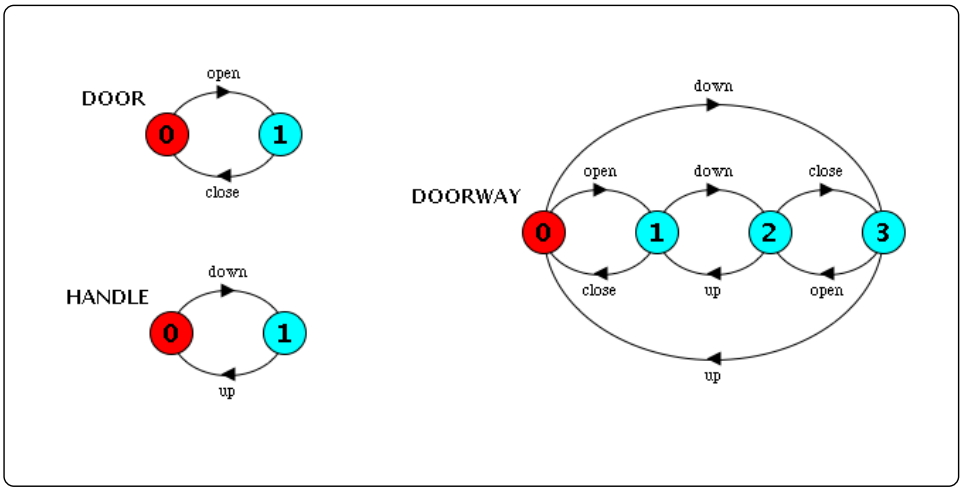

Figure 2.3: Composition of a doorway process from Door and Handle processes.

In the composite doorway system, the actions of HANDLE and DOOR may advance in any order, subject to the action-sequence constraints within each of the individually specified processes. States in the composite process are given by tuples where the first and second fields refer to the state of processes HANDLE and DOOR respectively. However, to simplify notation, LTSA simply labels the composite states with an integer (e.g., 0, 1, 2, 3 .. and so forth).

2.2.3 Role of Shared Actions and Action Relabeling

In the previous example, the alphabet of actions in door (i.e., `open`, `close`) is disjoint from the alphabet of actions in handle (i.e., `up`, `down`). As a result, the door and handle processes can advance in any order, subject to the action-sequence constraints within each process.

When action names in a process composition are common, these actions are said to be shared. LTSA uses shared actions as the mechanism to synchronize interactions among processes. While the shared actions may be arbitrarily interleaved, the shared actions must be executed simultaneously by all participating processes.

Example 2. Shared action in Doorway Process Model. A reasonable constraint would be to synchronize the handle `down` with door `open`, and handle `up` with door `close`, thereby reducing the doorway process model to two states (see States 0 and 2) This is accomplished through relabeling of the actions in `HANDLE`, followed by composition, i.e.,

```
||DOORWAY1 = ( HANDLE || DOOR )/{down/open,up/close}.
||DOORWAY2 = ( HANDLE || DOOR )/{open/down,close/up}.
```

In `DOORWAY1` the action names `close` and `open` are changed to `up` and `down`, respectively. And as illustrated in Figure 2.4, in `DOORWAY2` the action name are switched.

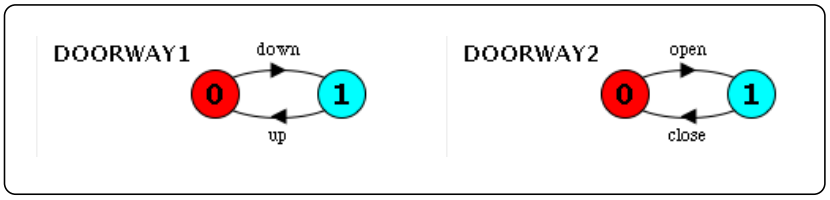

Figure 2.4: Illustrate action relabeling in doorway system.

2.2.4 Deterministic and Non-Deterministic Choice

The FSP language provides mechanisms for deterministic and non-deterministic choice. Their definitions are as follows (for details, see the LTSA book [24], pg. 15):

1. **Deterministic Choice.** If "x" and "y" are actions, then `( x->P | y->Q )` describes a process which initially engages in either the actions `x` or `y`. The execution of action `x` will have subsequent behavior described by `P`. Similarly, the execution of `y` will have subsequent behavior described by `Q`.

2. **Non-deterministic Choice.** The process `( x->P | x->Q )` is non-deterministic because after the action `x`, behavior may be described by either process `P` or process `Q`.

Example 3. Suppose that a sensor switches between two modes. Either it is polling for data, or it is engaged until it released. This scenario can be easily implemented via,

```
SENSOR      = (   engaged -> released -> SENSOR
              |  poling -> SENSOR ).
```

The corresponding LTS is as shown in Figure 2.5.

2.2.5 Role of Tagged Processes

Individual processes can be tagged, thereby providing a mechanism for more than one copy of a process to be used and uniquely identified in a system model. The notation is as follows: a:P prefixes each action label in the alphabet of process P with the label `a`.

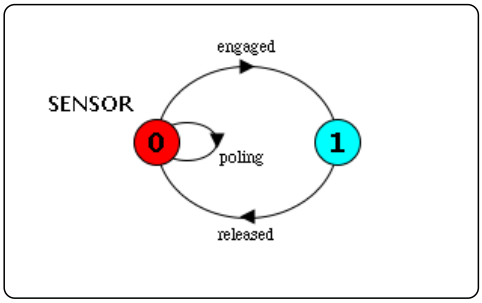

Figure 2.5: LTS for sensor behavior.

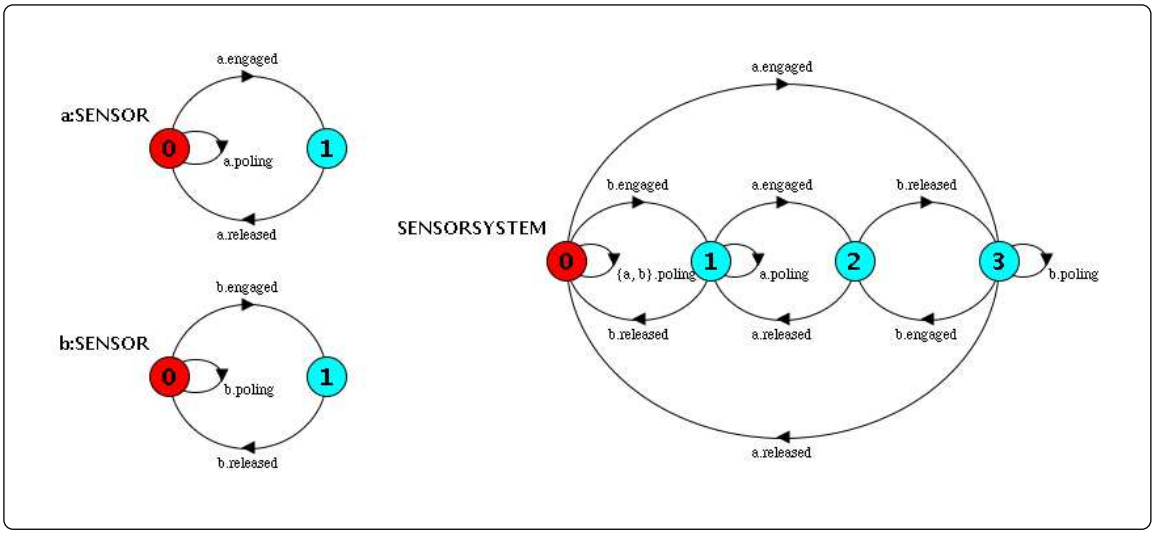

Figure 2.6: Processes for individual sensors and a two-sensor system.

Example 4. Process Behavior in a Two-Sensor System. The fragment of code:

```
  SENSOR      = (   engaged -> released -> SENSOR
                |   poling -> SENSOR ).
||SENSORSYSTEM = ( a:SENSOR || b:SENSOR ).
```

takes the sensor process (SENSOR) defined in Example 3, and creates a composite process (SENSORSYSTEM) from two individual sensor behaviors (i.e., `a:SENSOR` and b:SENSOR). As illustrated in Figure 2.6, the result is a system having four states. Their interpretation is as follows:

```
-------------------------------------------------------------------------
    Sensor System     a:SENSOR   b:SENSOR                    Interpretation
=========================================================================
Composite state 0      state 0    state 0                 a and b are poling
Composite state 1      state 0    state 1          a is poling, b is engaged
```

```
Composite state 2      state 1    state 1           a and b are both engaged
Composite state 3      state 1    state 0          a is engaged, b is poling
=============================================================================
```

The second and third columns of this table show how permutations of the individual sensor states relate to the composite system state (column 1). Also notice that because their are no shared actions between processes, `a:SENSOR` and `b:SENSOR`, the interleaved processes may advance stepwise in any order.

2.2.6 Role of Guarded Actions

An action that is conditional on a particular condition being true is terms a guarded action. The FSP syntax for guarded actions is `( when B x->P | y->Q )`. When guard `B` is true, actions `x` and `y` are both eligible to be chosen. Otherwise, only action `y` can be chosen.

Example 5. The script of code:

```
const N  =   3  // Number of ships passing through lock ...
const M  =   4  // Number of states in holding pattern queue...
range IQ =1..M  // Queue count

// Define input/output queues

   QUEUEIN = QUEUEIN[1],
   QUEUEIN[i:IQ]  = ( when (i<M) [i].arrive -> QUEUEIN[i%N+1] ).

   QUEUEOUT = QUEUEOUT[1],
   QUEUEOUT[i:IQ] = ( when (i<M) [i].depart -> QUEUEOUT[i%N+1] ).

// Model spaces in lock ....

   LOCK = SPACES[1],
   SPACES[i:0..1] = ( when(i>0) [j:1..N].arrive  -> SPACES[i-1]
                    | when(i<1) [j:1..N].depart  -> SPACES[i+1] ).

// Create model of controled lock system behavior ...

   ||CONTROL = ( LOCK || QUEUEIN || QUEUEOUT ).
```

simulates a convoy of three ships passing through a single lock. From a process modeling perspective, the lock is a bounded buffer of maximum capacity one. The structure diagram for the QUEUE and LOCK processes is as follows:

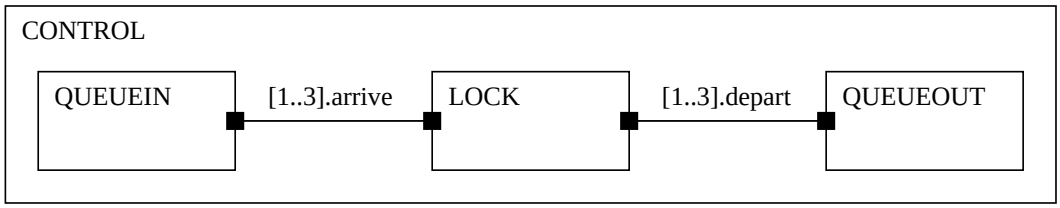

Figure 2.7: Structure diagram for control of three ships traversing a single lock.

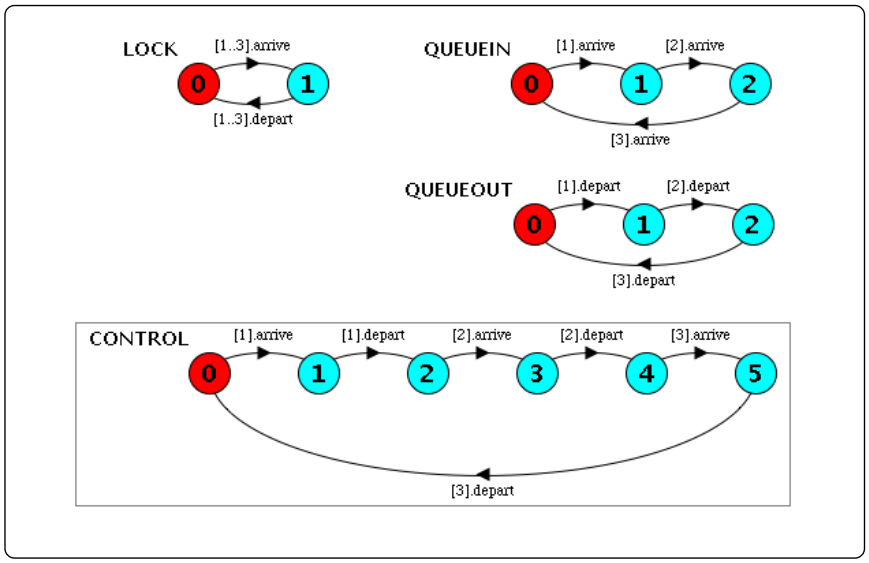

Figure 2.8: Use of guarded actions for queue and canal behavior models.

The LTS diagrams for each of the processes is given in Figure 2.8. Lock behavior is defined by the arrival of a ship (i.e., `[1..3].arrival`) followed by the departure of a ship (i.e., `[1..3].depart`). Upon arrival, the ships are required to join a queue (QUEUEIN). Upon departure, the ships join a second queue (QUEUEOUT). The number of ships is modeled by constant `N = 3`. Constant `M = 4` captures the potential length of arrival and departure queue processes.

Required transformations among process states are efficiently defined through the use of guarded actions. Queue processes, in particular, place constraints on the order in which ships can arrive and subsequently depart. For example, the fragment of code:

```
QUEUEIN[i:IQ]  = ( when (i<M) [i].arrive -> QUEUEIN[i%N+1] ).
```

defines the action `[0].arrive` between states 0 and 1, `[1].arrive` between states 1 and 2, and `[2].arrive` between states 2 and 0. LTSA requires that processes be continuous; hence, we simply wrap the loop of arrival/departure actions through use of modulo arithmetic (i.e., `i%N+1`). Similarly,

the script:

```
LOCK = SPACES[1],
SPACES[i:0..1] = ( when(i>0) [j:1..N].arrive  -> SPACES[i-1]
                 | when(i<1) [j:1..N].depart  -> SPACES[i+1] ).
```

sets up a process modeled as a resource defined in terms of spaces. The notation `[i:0..1]` indicates that we are only using one space (but the code can be easily generalized to a lock having multiple spaces). When a ship arrives, the number of available spaces in decremented by one – in other words, the lock space is occupied. That space becomes available again when a ship departs. It follows that the lock system has only two states, as indicated in the top left-hand corner of Figure 2.8. Notice that from a lock perspective, ships can arrive and depart in any order (i.e., `[1..3].arrive`, `[1..3].depart`).

2.2.7 Modeling of Shared Resources

In the previous example, orderly use of the lock system was enforced through the use of queue processes. A second possible approach is to define independent ship processes, and then simply state that the lock system process is a shared resource that can only be used by one ship process at a time. With FSP, this is achieved with the double-colon syntax `::`, as in $\{a_1, a_2, .. a_y\}$`::P`. The latter replaces every label n in the alphabet of P with labels a_1`.n`, a_2`.n` through a_y`.n`.

Example 6. The following script of code defines basic processes for a lock scheduler and a ship passing through a lock system.

```
// Define ship and scheduler processes.

   SHIP      = ( approach -> lock.acquire -> lock.transit -> lock.release ->
                 depart -> SHIP ).
   SCHEDULER = ( acquire -> transit -> release -> SCHEDULER ).

// Compose behavior model for scheduler and three ships....

 ||LOCK = ( a:SHIP || b:SHIP || c:SHIP || {a,b,c}::lock:SCHEDULER ) \
          {a.approach,a.depart,b.approach,b.depart,c.approach,c.depart}.
```

A model of lock-system behavior is obtained through the parallel composition of three ship behaviors with an instance of the scheduler implemented as a resource constraint. Points to note are as follows:

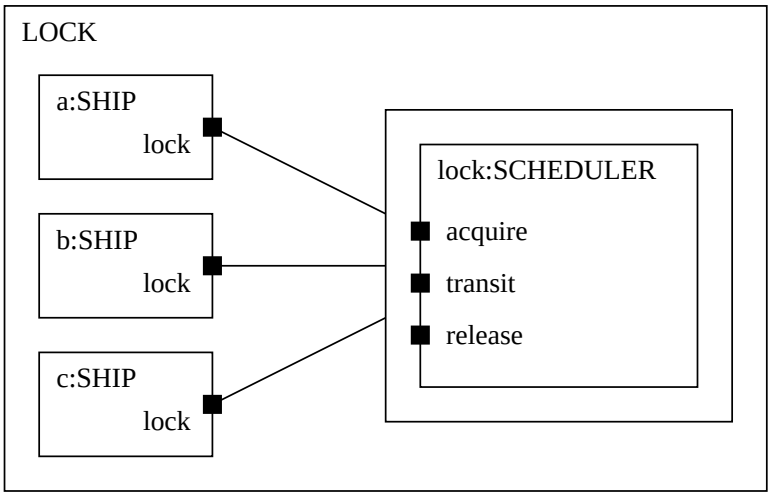

Figure 2.9: Structure diagram for lock system operating as a shared resource process.

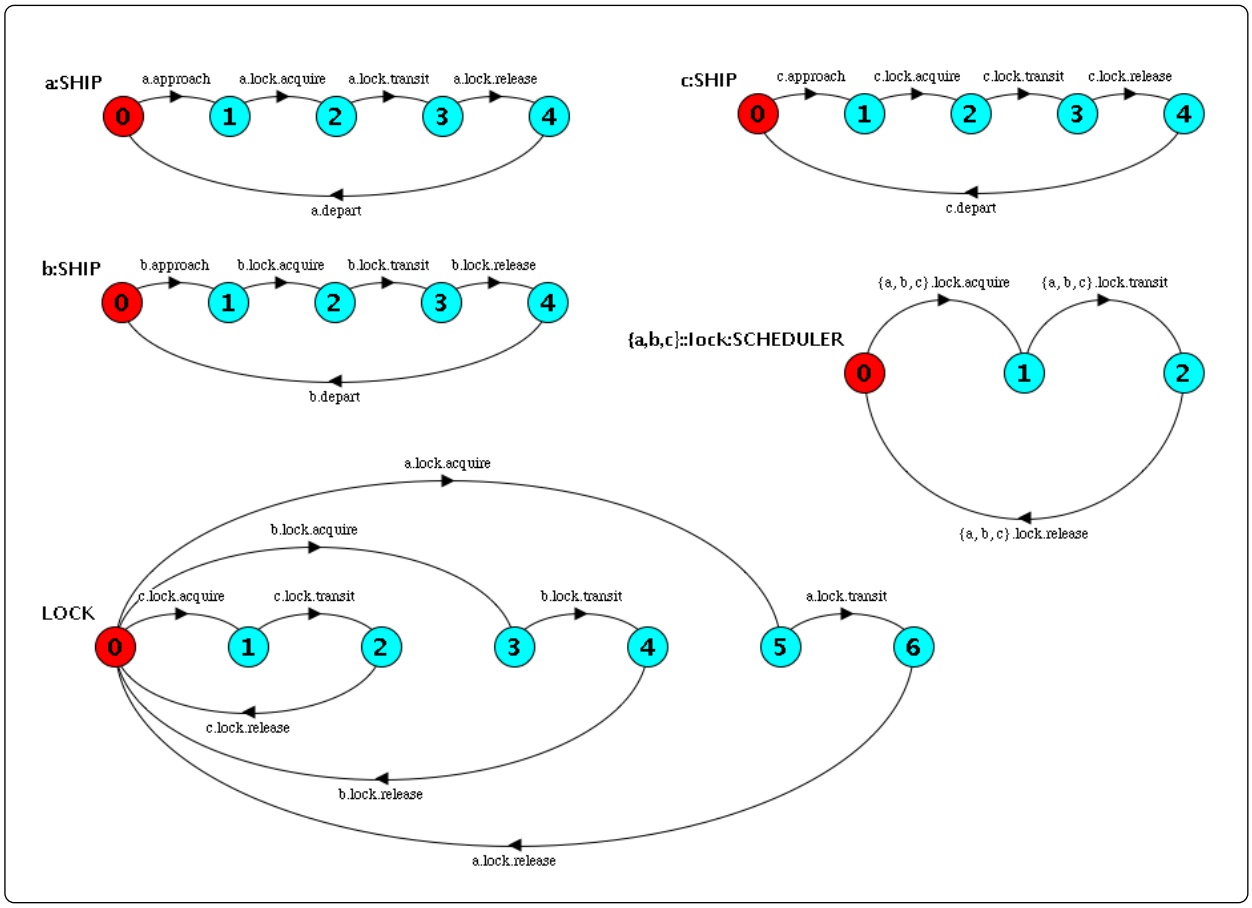

Figure 2.10: Behavior modeling of a lock as a shared resource.

1. Figures 2.10 and 2.9 show the basic component processes, the composed lock-level process, and a structure diagram for the lock system operating as a shared resource process. First, three component-level ship processes are created (i.e., `a:SHIP`, `b:SHIP` and `c:SHIP`). A lock scheduler walks ships through three actions: `acquire`, `transit` and `release`. Synchronization of shop and scheduling activities is achieved through the action labels `lock.acquire`, `lock.transit` and `lock.release`. Implementation of the shared resource is achieved with

```
{a,b,c}::lock:SCHEDULER )
```

2. The composed lock behavior (LOCK) has 15 states. A common strategy for simplifying interpretation is to remove or hide states not relevant to an immediate decision. Hence, for our purposes, the appendage,

```
{a.approach,a.depart,b.approach,b.depart,c.approach,c.depart}.
```

is a list of actions that are hidden in the state diagram. An equivalent notation is:

```
{{a,b,c}.approach,{a,b,c}.depart}.
```

Each of these actions is simply replaced by a `tau` resulting in a system-level model containing 81 states. The lower schematic in Figure 2.10 (i.e., process LOCK) is the result of the original lock process being minimized after the selected action have been hidden and then removed. The minimized model has 7 states. Interpretation is rather straight forward – starting at State 0, ships a, b, or c, may acquire, transit, and depart the lock, treating the latter as a resource that is shared.

3. There are two ways to obtain a reduced (or minimized) model. The most straightforward procedure is to select the option `Minimize` from within the `Build` pulldown menu in LTSA. This option is appropriate when models are small, perhaps resulting from a small number of process compositions. The principal shortcoming of this approach is that actions irrelevant to a particular model or viewpoint of a design are carried along in all process compositions. To combat the likelihood of exponential explosion in the number of states that need to be considered, intermediate processes can be minimized by using the `minimal` command within the FSP specification itself. For example,

```
minimal ||LOCK = ( {a,b,c}:SHIP || {a,b,c}::lock:SCHEDULER ) \
                 {{a,b,c}.approach,{a,b,c}.depart}.
```

2.3 Model Checking in LTSA

To keep system developments on course and to prevent serious design flaws, today we seek validation (i.e., are we building the right product?) and verification (i.e., are we building the product right?) procedures that support pre-deployment reasoning about system requirements and design [5]. The key problem with semi-formal validation procedures (e.g., manual inspection of UML diagrams) is that they lack the precise interpretation of scenarios needed for rigorous analysis and formal verification of system compliance. As documented by Baier and Katoen [5] and references therein, use of informal verification procedures can lead to system failure rates that are unacceptably high.

The goals of model checking procedures are to avoid these limitations through their use of formal descriptions of system behavior and properties expressed as logic formulae. Given a finite-state model of a system and a formal property, model checking procedures systematically check whether the property holds for that model.

2.3.1 Model Checking Procedure

Model checking begins with two activities that can occur concurrently. As shown along the left-hand side of Figure 2.11, informal requirements are transformed into formal specifications describing properties that any acceptable system implementation will satisfy. A property is an attribute of a process that is true for every possible execution of that process. Typical properties are of a qualitative nature (e.g., will the system ever reach a situation for which there is no pathway forward?). Some model checking procedures even allow for evaluation of timing properties to be checked (e.g., can a deadlock occur within the first 10 seconds of system operation?).

Then as illustrated on the right-hand side of Figure 2.11, a process model for behavior of the engineering system is assembled. System models are mostly expressed using finite-state automata (consisting of finite sets of states and transitions). To answer the above-mentioned questions in a precise and unambiguous manner, the system behavior model must be sufficiently detailed, but not too complex.

Model checking procedures examine the property specifications with respect to process models. Three outcome are possible:

1. The property specification is satisfied,

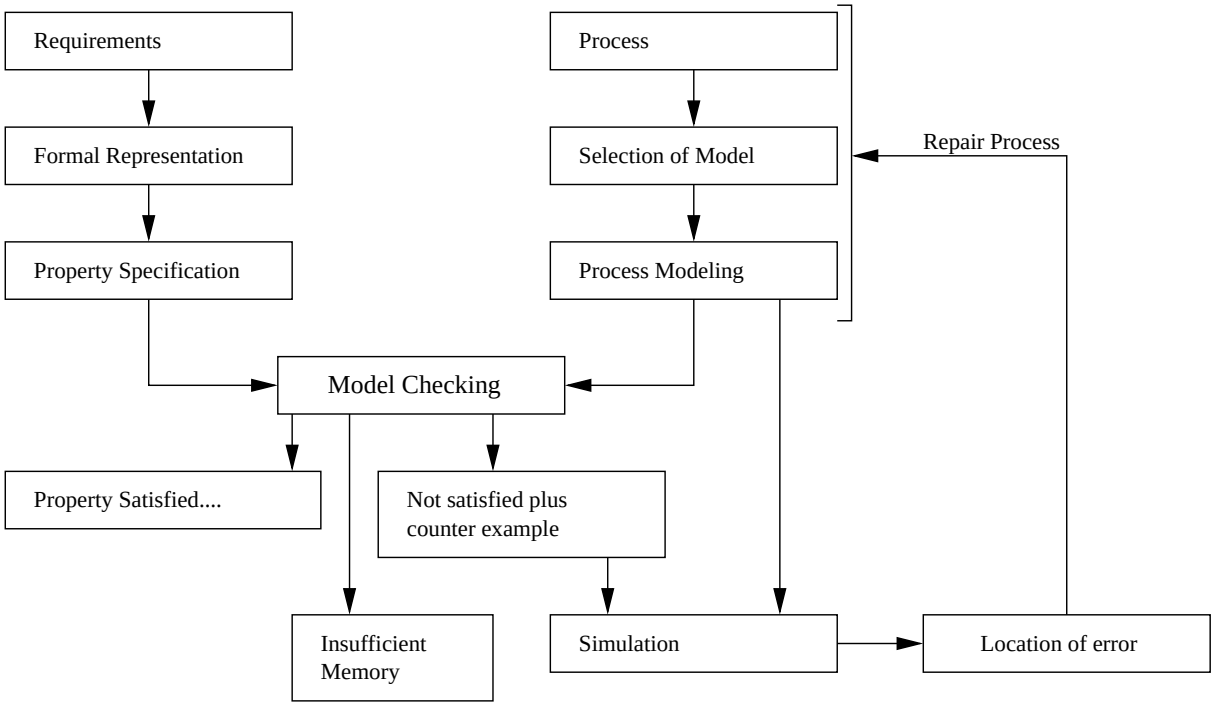

Figure 2.11: Model checking procedure and outcomes.

2. The property specification fails,

3. The model checking procedure fails because of insufficient computer memory.

When modeling checking procedures determine that property specification has failed, the result is accompanied by a counter example. In most cases, the underlying cause of a property specification failure can be identified through detailed analysis (i.e., simulation) of actions in the counter example. Generally, this will lead to refinement in one or more of the model, the design or the property. A third possibility is "insufficient memory." The only practical way of dealing with this situation is to reduce the size of the model and try again. Iterations of model checking continue until all of the property specification violations have been repaired.

2.3.2 Desirable Properties of System Behavior

Our objective is to design behavior models having properties that are guaranteed to be satisfied, including:

1. Safety. A safety property asserts that nothing bad happens,

2. **Liveliness.** A liveliness property asserts that "eventually" something good happens.

A complete treatment of liveliness involves reasoning with temporal logic, a topic beyond the scope of this study. We will instead employ a restricted form of liveliness called progress:

2. **Progress.** A progress property asserts that it is always the case that an action is eventually executed.

A good system design exhibits safety and liveliness/progress. Safety violations in behavior modeling correspond to undesirable sequences of actions. For example, two systems should not simultaneously attempt to acquire a shared resource. A safety violation will also occur if a state becomes blocked and cannot make further progress (i.e., it becomes deadlocked). Liveliness concerns include the ability of a process to eventually terminate and/or reach a critical state/action in its execution.

Mathematical Definition of Safety. A safety property prescribes a set of permissible traces which, in turn, can be modeled as state machines. A deterministic property automaton is defined $T = (S, A, \triangle, q)$. Property automaton are required to be free of trapped error states (i.e., π) and τ transitions. A property violation occurs when it is possible for a distributed system to perform a trace not acceptable to the property automaton. The computational procedure is as follows:

1. Create a image automaton that captures the prescribed property automaton and adds to it possible violations leading to the π state.
2. Compose the image automaton with the system description to which it applies.

The image automaton is defined as $T^{'} = \left(S \cup \{\pi\}, A, \triangle^{'}, q\right)$. It is derived from $T = (S, A, \triangle, q)$ by defining $\triangle^{'}$ to be $\triangle \cup \left\{(s, a, \pi) | (s, a) \in S \times A \quad \wedge \nexists s^{'} \in S : (s, a, s^{'}) \in \triangle\right\}$ The first part of the image construction process ensures that $\triangle$ is a subset of $\triangle^{'}$. The latter part of the construction ensures that for any transition $\left(s, a, s^{'}\right)$ belongs to $\triangle^{'} - \triangle$ and $s^{'}$ equals π. It follows that:

1. Automata T and $T^{'}$ will have the same set of non-trapping traces (i.e., $tr(T)$) and
2. for any process P, $P \| T^{'}$ will not contain traces to π, if and only if $tr(P \uparrow \alpha T) \subseteq tr(T)$.

Proofs can be found in Cheung et al. [8]. The second condition allows for the detection of safety violations in a system by checking for the existence of trapping states the composite process formed

by the system process and the image automata. If the composed process contains error states, then the safety property is violated.

Progress Properties and Analysis. As already noted, a progress property asserts that it is always the case that an action is eventually executed. Progress analysis begins with a search for sets of terminal states; that is, sets of states where every state is reachable from every other state in the set via one or more transitions, and, there are no transitions from within the set to any state outside the set. Given fair choice, each terminal set of states represents an execution where each transition is executed infinitely often. With this framework in place, checking that a progress property holds reduces to the problem of checking that the progress actions are part of each terminal set [24].

2.3.3 Model Checking in LTSA

LTSA mechanically checks that the specification of a concurrent system satisfies the properties required of its behavior. Safety properties are specified in FSP by property processes (a.k.a., deterministic finite-state machines called property automata). Each property automaton specifies the set of feasible execution sequences over the actions (transitions) that correspond to a safety property of interest.

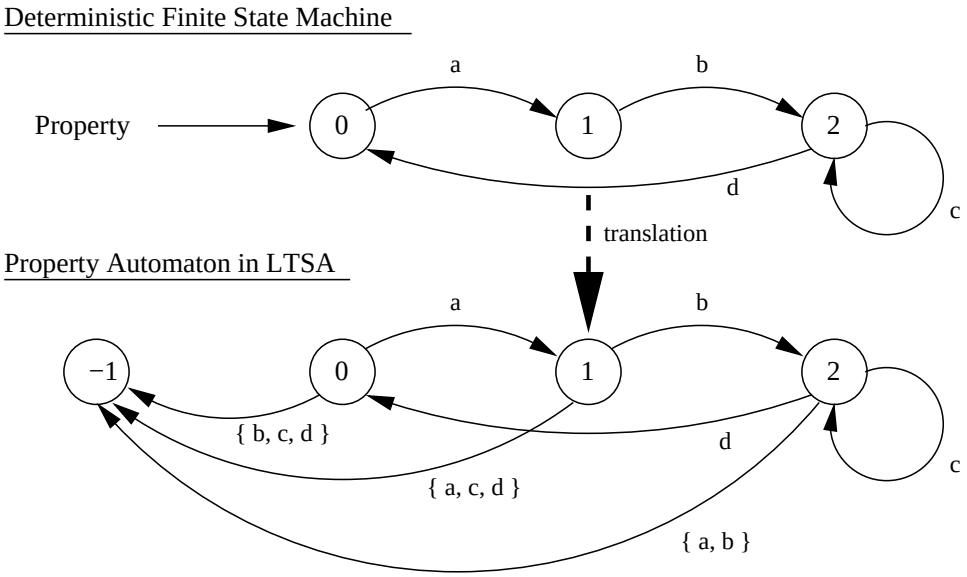

Figure 2.12: Procedure for definition of property automata in LTSA

The upper diagram in Figure 2.12 shows, for example, a property expressed as a deterministic

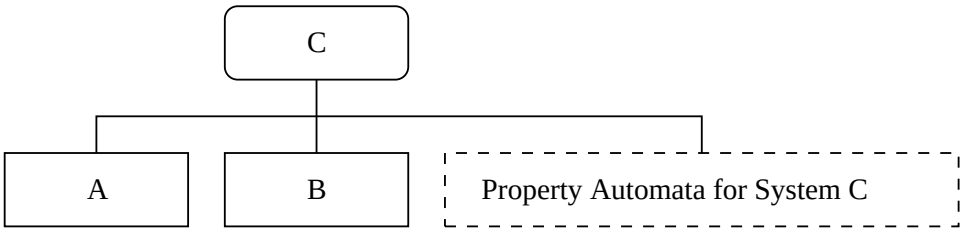

Figure 2.13: Visual representation of system C composed from processes A and B, and, validation of system C via composition with property automata.

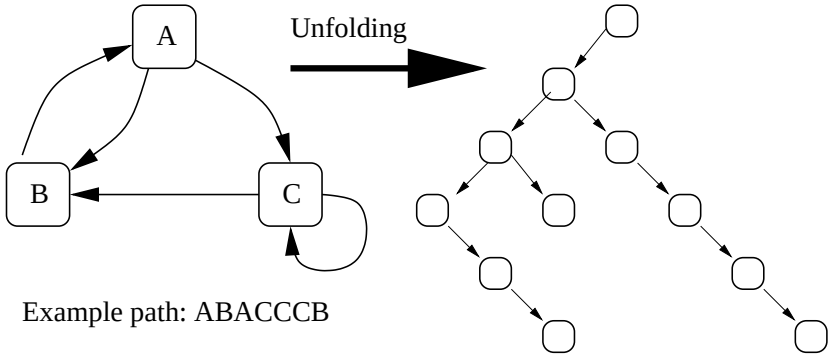

Figure 2.14: Evaluation of system properties through fsm unfolding and exhaustive search

.

finite state machine. The property states that action `a` must be followed by action `b` which, in turn, can be followed by a sequence of action `c` or action `d`. Conversely, action `b` must be preceded by action `a`, and so forth. LTSA creates an image automaton that captures the prescribed property automaton and adds to it possible violations leading to an error state. Suppose, for example, that action `a` has just completed. If the behavior model allows for any action other than `b` (i.e., the set of actions { `a, c, d`}), then the property automaton will transition to the error state.

Safety properties are evaluated in a procedure that is remarkably straightforward. All that we have to do is compose the property process with the system description process and look for the existence of π (an error state) in the global LTS. See, for example, Figure 2.13. This procedure works only because the (error) π state is preserved by both the restriction and composition operators – the appropriate details can be found in the upper and lower sections of Table 2.1.

Progress properties have the syntax, `progress P = { a1, a2, ... aN }`. They assert that in an infinite execution of the behavior model, eventually at least one of the actions `a1`, `a2` ... `aN` will be executed infinitely often.

From a user point of view, compositional approaches to behavior modeling and property evaluation rely on two things: (1) construction of an equivalent state machine of the system architecture against which properties can be checked, and (2) an explicit enumeration of states (also known as reachability analysis) against which property satisfaction can be determined. Steps 1 and 2 are linked via a process where state machines are unfolded into large trees of equivalent states. See Figure 2.14. Then, the tree of equivalent states is searched exhaustively to see if the required properties are actually satisfied.

Example 7. The script of code:

```
// ------------------------------------------------------------------------------
// Jack and Diane have conversation over coffee ....
// ------------------------------------------------------------------------------

// Create a person who: (1) talks and drinks coffee, or
//                      (2) just waits and then drinks coffee ....

   PERSON = (   talk -> drink -> PERSON
             |  wait -> drink -> PERSON ).

// Jack and Diane meet ....

   minimal ||JACK_AND_DIANE_MEET = ( jack:PERSON || diane:PERSON ).

// To learn, conversation needs to be two way ....

   TWO_WAY = ( jack.talk -> diane.talk -> TWO_WAY ).

// Define a property for polite conversation ...

   property POLITE = ( jack.talk -> diane.talk -> POLITE ).

// Check that the conversation model is in fact polite ...

   minimal ||JACK_AND_DIANE_LEARN = ( JACK_AND_DIANE_MEET || TWO_WAY || POLITE ) / {
                                     jack.talk/diane.wait, diane.talk/jack.wait }.

// Check progress properties

   progress DIANE_TALKS = { diane.talk }
   progress JACK_TALKS  = {  jack.talk }

// ------------------------------------------------------------------------------
// End!
```

systematically assembles a behavior model for two people, Jack and Diane, in polite conversation. The process hierarchy and sample LTSs assembled by this script are shown in Figures 2.15 and 2.16.

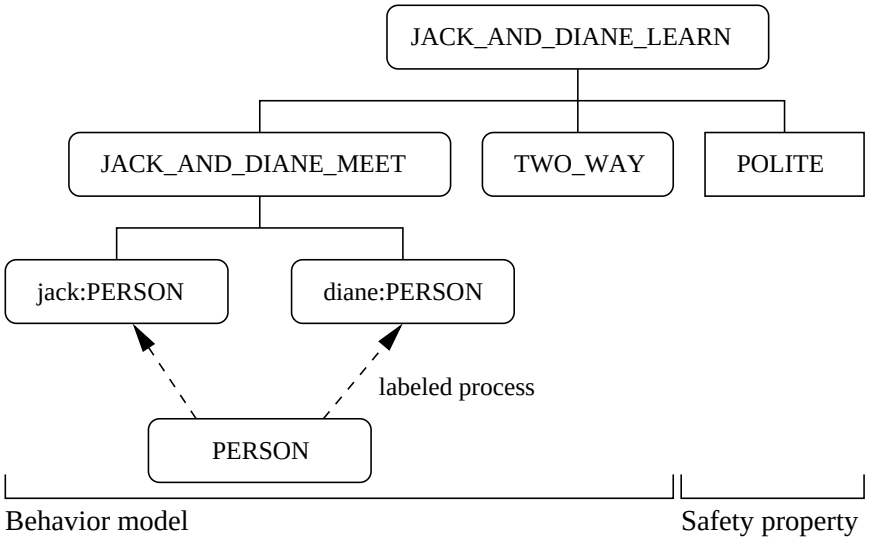

Figure 2.15: Process hierarchy for behavior model and validation of polite conversation.

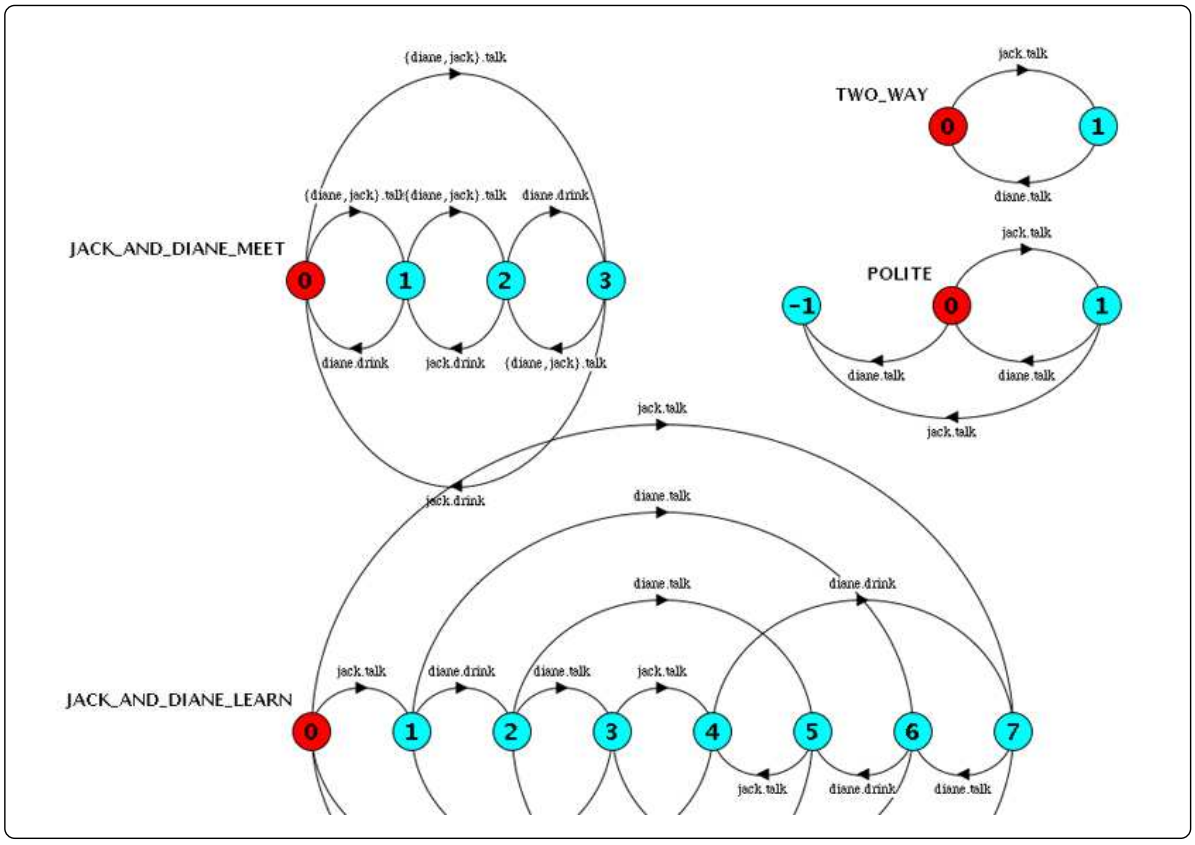

Figure 2.16: LTSs for processes in a behavior model of polite conversation.

The behavior modeling process begins with the definition of a generic PERSON, who can talk and drink, or simply wait and then drink. Jack and Diane are simply labeled instances of the process PERSON. The composed process JACK_AND_DIANE_MEET captures all of the possible sequences of actions that can occur. This model allows, for example, for one person to talk and talk and talk, with the other person not getting a word in edgewise. To improve the meeting, the TWO_WAY process places a constraint on the conversation, in particular, that Jack and Diane need to engage in alternate talking. The revised meeting model is,

```
||JACK_AND_DIANE_LEARN = ( JACK_AND_DIANE_MEET || TWO_WAY ).
```

But how do we know that this actually worked? To check that the composed model is in fact what we want, we can define the property POLITE, i.e.,

```
property POLITE = ( jack.talk -> diane.talk -> POLITE ).
```

and then compose POLITE with JACK_AND_DIANE_LEARN. Figure 2.16 shows the LTSs for each of the constituent processes including POLITE. Notice that POLITE will transition to an error state if Diane talks more than once or, alternatively, Jack talks more than once. If the composed process

```
( JACK_AND_DIANE_MEET || TWO_WAY || POLITE )
```

contains any of these sequences, then it too will also have an error state indicating that our model of behavior is not polite. As it turns out, the composed process (see Figure 2.16) is free of error states and the POLITE property is satisfied. The progress checks generate

```
Progress Check...
-- States: 8 Transitions: 16 Memory used: 1951K
No progress violations detected.
Progress Check in: 40ms
```

2.4 Systematic Organization of Processes into Layers

The well known difficulty in using exhaustive search techniques for validation of systems having concurrent behaviors is that size of the state space expands exponentially with increasing numbers of underlying processes (and actions therein). This is the state explosion problem. For families of processes that are primarily autonomous, only minor reductions in the size of composed processes will occur through constraints associated with synchronized actions. Techniques for further reducing problem complexity can be classified into two broad categories [8, 10, 27]:

1. **Reduction by Partial Ordering.** Reduction is achieved by avoiding generation of all paths formed by the same set of transitions.
2. **Reduction by Compositional Minimization.** Reduction is achieved through intermediate minimization of subsystems.

Partial ordering methods rely on a restricted number of process interleavings to obtain a simplified explorations of the state space. The key challenge is how to find the former while ensuring observable behavior with respect to a property (previously required by a design specification) is not affected. Researchers have determined a number of (rather complicated) solution strategies including use of modified semantics and/or identification of dependencies among transitions [27, 32]. In each case, the design property should view behavior of the simplified and full process models as being indistinguishable [5, 12, 15, 21, 22]. Two processes are said to be in "weak equivalence" when their observable behaviors are indistinguishable. Conversely, processes are said to be in "strong equivalence" when their observable behaviors are indistinguishable, while also taking into account unobservable behaviors. A mathematical treatment of weak/strong behavioral equivalence can be found in Cheung and Kramer [8].

The method of reduction by compositional minimization aims for simplified models through the removal of irrelevant detail. This method works well for systems that are naturally hierarchical and/or evolve over time. Each composite process is formed via the composition of lower-level composite processes and primitive processes (modeled as finite-state deterministic machines). The first opportunity for simplification stems from the tendency of safety properties to be locally checkable – validation can proceed without the need to assemble the global state space. And second, progress properties rely on a set of actions being activated – most often they can be evaluated with respect to actions at a particular level of detail, with lower level actions being removed from consideration.

Now let us assume that a violation has been detected and that a designer wishes know the cause of the violation. This task is facilitated if debugging traces show as much detail as possible.

These dual criteria point to a natural tension in the methodology. We wish, on one hand, to simplify models through removal of details. Yet at the same time, we need to maximize the availability of information for debugging purposes.

Simple Network of Two-Level Process Models. In a study focusing on algorithms for reduction by composition, Cheung and Kramer [8] deal with these concerns by assuming that each subsystem hides at its boundary only those internal actions that: (1) do not participate in interprocess communications, and (2) are not globally observable. In other words, actions that do not participate in interprocess communication and are not globally observable may be abstracted from consideration in system analysis. Figure 2.17 shows, for example, a simple network of two-level process models.

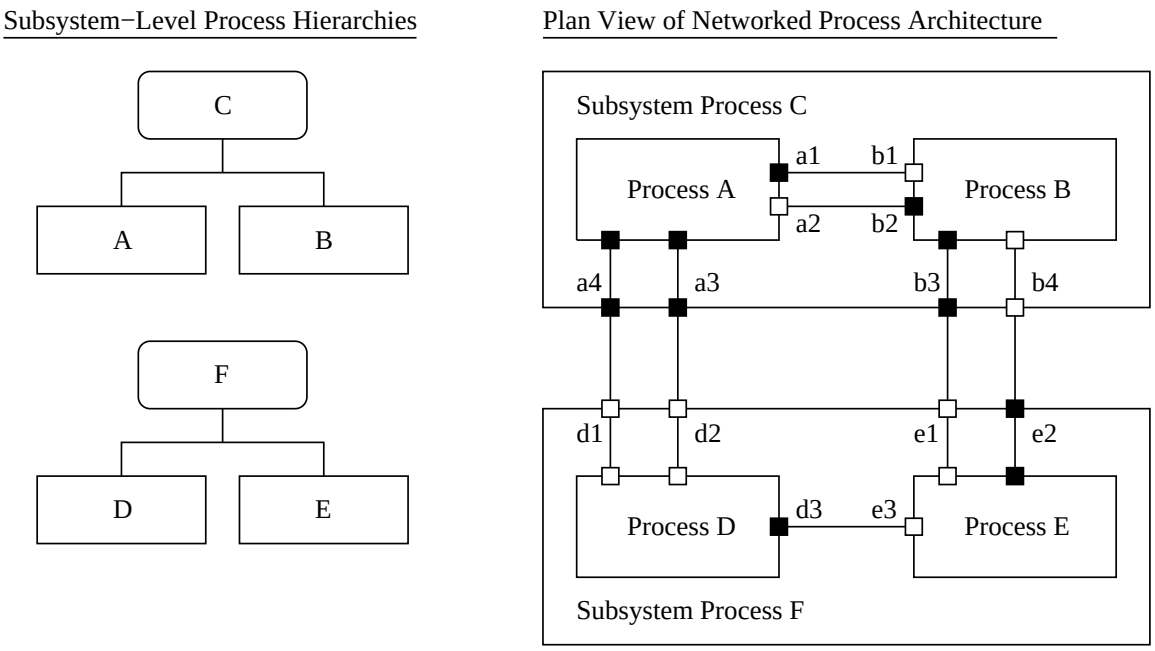

Figure 2.17: Visual model of a simple network of two-level process models. Actions internal to processes A, B, C and D are hidden. White dots represent requirements. Black dots represent provisions.

As illustrated in Figure 2.18, systematic assembly of the behavior model occurs in three steps. First, behavior models for subsystems C and F are created by computing `(A||B)` and `(D||E)` respectively. Then, the system-level behavior model is simply given by the composition `(C||F)`. Process interactions are achieved through the matching of bindings on requirements and provisions (see notation in black and white dots, respectively).

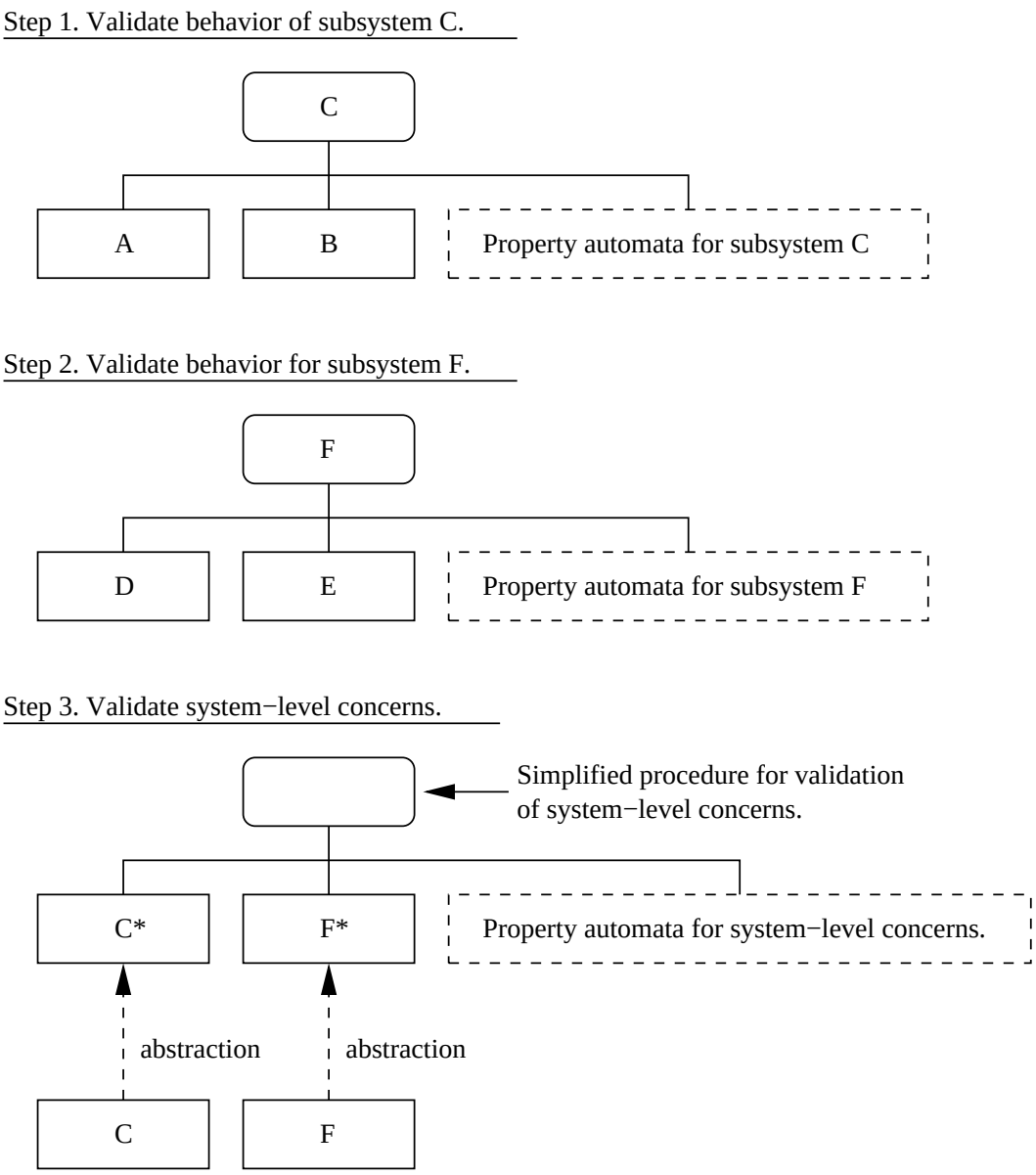

Figure 2.18: Systematic assembly of a two-level behavior model with property automata for validation of subsystem and system concerns.

When all of the actions in the low-level processes (i.e., A, B and D, E) are globally accessible, then model checking follows the step-by-step procedure described in the previous section. But now let us suppose that at system level (i.e., see Step 3 in Figure 2.18), such an approach is computationally intractable. Moving forward requires that process model size be controlled through the systematic removal of actions internal to processes C and F. The three-step procedure is as follows:

1. Validation of safety and progress properties associated with subsystem C.

```
||C = (A || B)/{a1/b1,a2/b3}.
```

2. Validation of safety and progress properties associated with subsystem D.

```
||F = (D || E)/{d3/e3}.
```

3. Validation of safety and progress properties associated with system-level concerns.

```
||C_star = ( C ) @ { a4, a3, b3, b4 },
||F_star = ( F ) @ { d1, d2, e1, e2 },
||System_Model = ( C_star || F_star ).
```

The system-level model is now described entirely in terms of the set of actions { a4, a3, b3, b4, d1, d2, e1, e2 }. Since all other actions are removed from consideration in the design, it is assumed that properties associated with system-level concerns will be described only in terms of these actions. In practice, however, no matter how well a system is organized into modules, so-called cross-cutting concerns will still reach across hierarchies [2].

2.5 Viewpoint-Action-Process Traceability

In the work of Cheung and Kramer [8], compositional reduction analysis procedures assume that a system can be organized into a modular architecture. Simplification of behavior models occurs through the hiding of as many internal actions as possible in each subsystem. As a consequence, the properties that are then available for reasoning in the model checking analysis are constrained by the remaining set of observable actions.

This study also assumes that validation procedures can be simplified through the organization of processes into modules. But in a departure from past work, modules will have boundaries (i.e., notions of inside and outside) that depend of a design viewpoint (or concern). We propose the

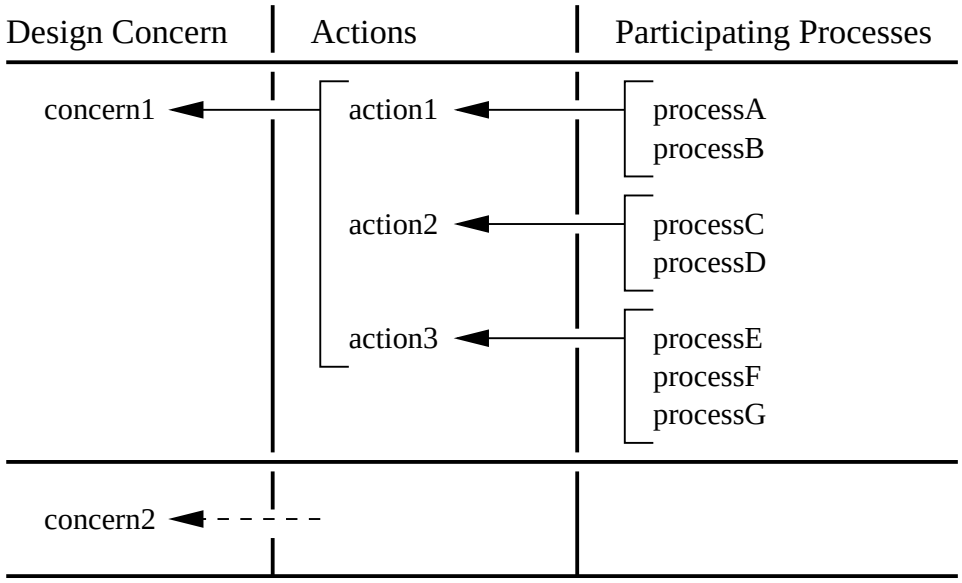

Figure 2.19: Schematic for tabular display of design viewpoint-action-process dependencies.

concept of viewpoint-action-process traceability as a means of representing connectivity between the formal description of a design concern (viewpoint), sets of actions, and their participating processes. See, for example, the linkages in Figure 2.19. We also immediately minimize the size of all intermediate results. The result is a computational procedure where process models achieve their required functionality, but may have a size that is orders of magnitude smaller than in the all-in-one approach to composition.

Chapter 3

The Panama Canal

3.1 Background and Capability

The Panama Canal is an 80 kilometer passageway that joins the Atlantic and Pacific Oceans at one of the narrowest saddles of the isthmus [33]. Canal operations date back to August 15, 1914, when, after more than a decade of construction, the S.S. Ancon was the first ship to transit the canal. Soon thereafter (1920) the Chagres River was dammed to create Lake Gatun. This enhancement increased both the transit capacity and size of ships that could pass through the canal. At the time, the canal capacity could easily handle state-of-the art cargo ships.

For the past 88 years the canal has continued to operate with pretty much the same sets of buttons, levers, and visual monitors, requiring only routine maintenance. During this same period, however, the emergence of global business operations has led to a four-to-five-fold increase in traffic demand. Nowadays, with round-the-clock operations (90% of total capacity), 13,000-14,000 ships transport a total of 280 million tons of cargo through the canal annually, primarily on trade routes between Asia and the Central and Eastern US. This represents approximately 5% of world trade. Annual revenues account for 14 percent of Panama's national GDP [17], as of 2006. These statistics make the Panama Canal one of the world's most important waterways [6, 17].

3.2 Ship Transit

Under ideal conditions a ship can transit the canal in approximately ten hours. As illustrated in Figure 3.1, a ship passing through the canal will ascend through a set of locks, traverse Lake Gatun, and then descend through the lock system on the other side. The locks function

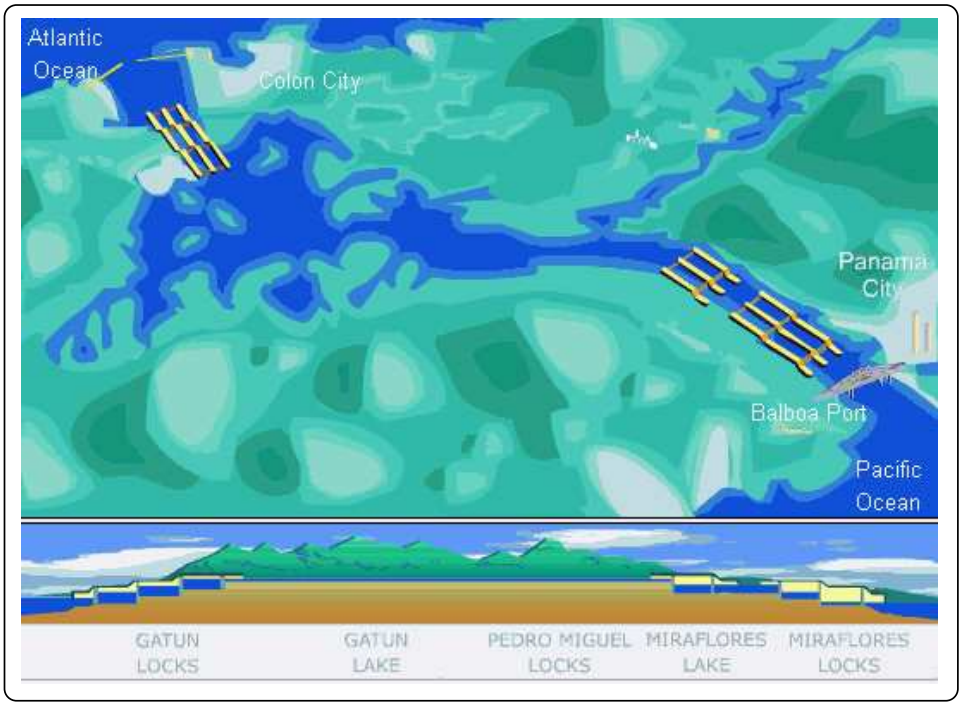

Figure 3.1: Plan and elevation views of Panama Canal.

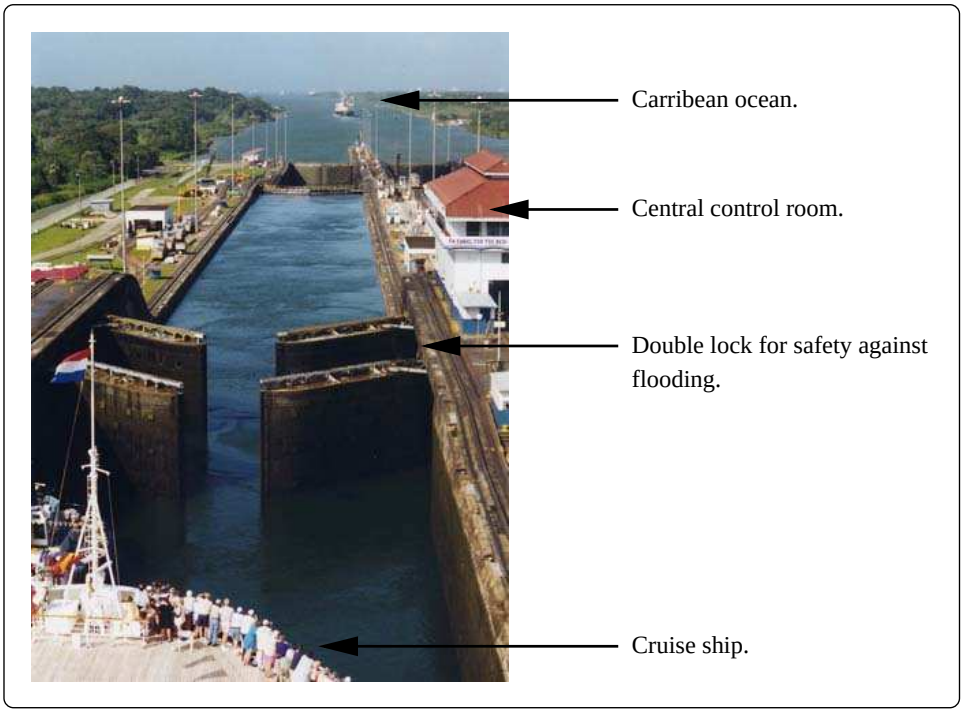

Figure 3.2: Gates of the Gatun locks open for a cruise ship making its way down to the Caribbean end of the canal. The gates at both ends of the upper chamber are doubled for safety. (Adapted from reference [34].)

as water lifts, raising ships from sea level to the elevation of Lake Gatun (26 meters above sea level). The lock chambers are 33.53 meters wide, 304.8 meters long and 12m deep. The maximum dimensions of a ship – so-called Panamax vessels – that can transit the canal are: 32.3 meters in the beam, 12 meters draft, and 294 meters long (depending on the type of ship). Individual locks are separated by gates. As a ship approaches the first chamber, valves below the compartment are released and the water level adjusts to that 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 open for a second time, allowing the ship to move into the second chamber. This repetitive process continues until the ship reaches the Lake Gatun. The ship traverses the lake and then descends through the exit lock system. The water used to raise and lower the vessels in each set of locks comes from Lake Gatun by gravity, and is delivered to the locks through a system of main culverts that extend under the lock chambers from the side walls and the center walls [33].

Ships are pulled through the lock systems by large locomotives on both sides of the canal. This process is illustrated along the right-hand side of Figure 1.1. To prevent ships from bumping into the canal side, care is needed to make sure tow wires on either side of the ship remain tight. A second area of concern is the potential for downstream flooding of lands caused by a runaway ship hitting a gate. Extra safety against this is provided by doubling the gates at both ends of the upper chamber in each flight of locks. See, for example, the double gate setup in Figure 3.2.

3.3 Control of Ship Movement

Since all the equipment of the locks is operated electrically, the whole process of locking a ship up or down can be controlled from a central control room, which is located on the center wall of the upper flight of locks. The controls were designed from the outset to minimize the chances of operator error, and include a complete model of the locks, with moving components which mirror the states of the real lock gates and valves. In this way, the operator can see exactly what state the locks and water valves are in. Mechanical interlocks are built into the controls to make sure that no component can be moved while another is in an incorrect state; for example, opening the drain and fill valves of a lock chamber simultaneously [34].

3.4 Limitations of Present-Day Canal Operations

The most important present-day problem dates back to the early 1990s, when the shipping industry decided to build post-Panamax (or super cargo) ships capable of carrying two-to-three times the cargo of present-day Panamax ships. Post-Panamax vessels currently account for 27% of the world's capacity of containerized shipping. However, this is expected to increase to 50 percent by 2010-2011. Pending obsolescence has put the Canal Authorities and, indeed, the whole of Panama at a cross roads [6].

A second problem with present-day operations is severe congestion and costly delays caused by accidents and maintenance operations. 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. The left-hand graphic in Figure 1.1 shows, for example, crowding of transiting ships in the Miraflores locks and lake. 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. The standard way of compensating for these difficulties is to introduce factors of safety into operational practices, and in the case of canals, to rely on human involvement for visual reference and line-of-sight validation. The downside to this approach is reductions in transit throughput during periods of low visibility. To date, these inefficiencies have been tolerable only because canal operations are less than peak capacity.

3.5 Panama Canal Renovation

To ensure that the Panama Canal remains important to the container shipping industry into the foreseeable future, the canal is now in the midst of a US $5.25 billion expansion [7, 26].

The new lock system will consist of three new locks on the Atlantic side and three new locks on the Pacific side. Each lock chamber will be 427m long, 55m wide and 18.3m deep. The dimensions of each lock chamber will be sufficient to support Post-Pamamax ships (366m long, 49m wide and a maximum draft of 15m). Lake Gatun will be raised 1.5m. The new lock system will use tug boats to position vessels, water from saving basins (for details, see Figure 3.3), and make increased use of automation to support day-to-day operations (e.g., sensors and lasers to determine the position of ships in the lock system) and to predict and plan for maintenance [17, 40].

The pathway from sensors to improved decision making is as follows: sensors gather data which is fed to computers for advanced processing. The enriched information leads to better decision

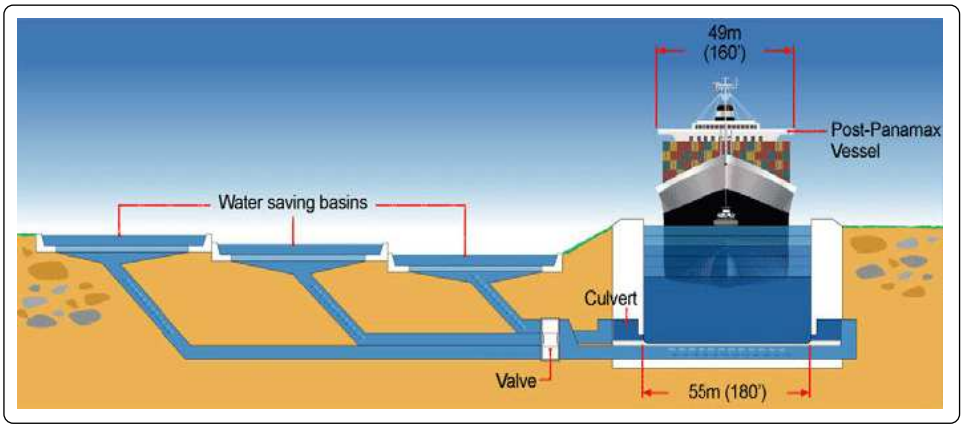

Figure 3.3: Cross section of a laterally filled lock chamber [35].

making which in turn is fed back to automated control systems. Throughout this process, the aforementioned barriers are overcome, thereby providing the desired results. Similar advances can be found in Turkey (Bosporus Straight), where large investments for traffic management systems have led to decision making procedures guided by GIS and data collected by land (sensors) centers [14, 28].

Chapter 4

Behavior Modeling and Validation of a Two-Stage Lockset Operation

In this chapter we formulate behavior models and validation procedures for ships transiting the two-stage lockset shown in Figure 4.1.

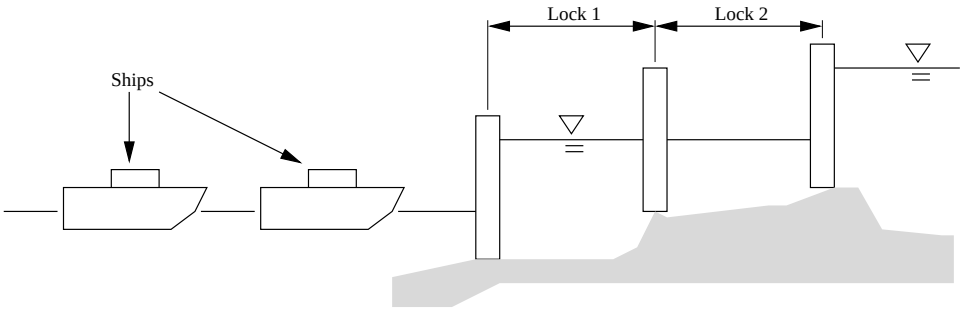

Figure 4.1: Elevation view of a two-stage lockset.

Queues of east-bound ships will ascend the lockset. West-bound ships will descend the lockset. Ships traveling in both directions will arrive at the lockset and request permission to transit the lockset. If the lockset system is empty and no other ships are waiting, then permission to acquire the lockset system's resources will be immediately granted. Otherwise, ships will be directed to join a queue and wait until the lockset scheduler authorizes permission to access the lockset.

Figure 4.2 shows the essential details of a process modeling framework for the planning and scheduling transit operations within a lockset, and later, the canal system. Planning objectives feed into a task planner which, in turn, pass plans for ship transit to a scheduler. A good plan will ensure that task operations are safe, and fair in the assignment of east- and west-bound transit

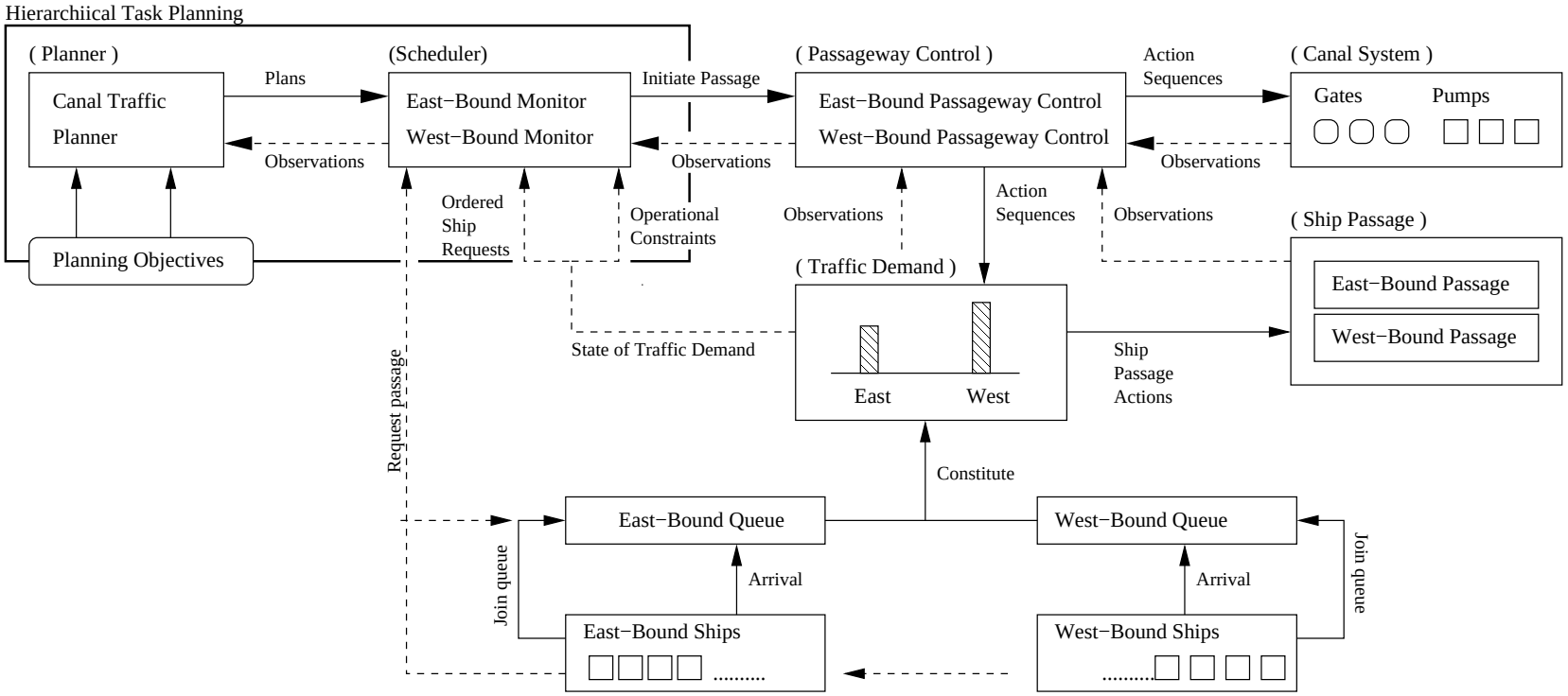

Figure 4.2: Schematic for scheduling and passageway control of ships transiting the canal system.

directions. The scheduler will receive east- and west-bound transit requests and schedule transit operations in a manner consistent with the task planner. The scheduler will also communicate permission for a particular ship to access the canal system to the passageway controllers. These lower level processes will be responsible for synchronizing low level activities, such as incremental ship movements with pump and gate operations.

Two models of lockset behavior will be formulated in this chapter. The first behavior model will demonstrate that: (1) the composition and validation processes work, (2) behavior model size will grow exponentially as a function of the number of ships transiting the canal system, and (3) the details of process design matter. As outlined in Chapter 1, the behavior model will be created through composition of processes on the lockset side (i.e., gates, pumps, ship and passageway controls, scheduler) and the traffic demand side (e.g., queues of east- and west-bound ships). The ship control and scheduler processes will be refined through two iterations of development. The purpose of these iterations will be to show how tradeoffs exist in the assignment of responsibility to processes versus their ability to multi-task. No effort will be made to remove actions not immediately relevant to decision making.

The second model will demonstrate that the systematic and repeated application of viewpoint-directed abstraction can keep the rate-of-growth of composed models in check. The second model will be assembled from the same set of constituent processes as in the first model, but the process composition will be organized into a multi-layer hierarchy. All intermediate process models will be minimized. Also, all actions internal to the lockset operations will be systematically abstracted from consideration in assessment of the ship behavior. This approach to abstraction stems from the premise that if behavior of the detailed lockset model can be validated, then actions not immediately coupled to ship transit actions (e.g., request, arrive, depart) can be removed from consideration.

4.1 Model 1. Detailed Behavior Modeling of a Two-Stage Lockset

In this section we create a detailed behavior model for operations of a two-stage lockset subject to east- and west-bound transit requests. A complete listing of the LTSA source code can be found in Appendix 1.

Figures 4.3 and 4.4 show the process architecture for the behavior models. In keeping with the principle of simplification through separation of concerns (see Figure 1.5), one process hierarchy will be assembled for the lockset system (it will be the composition of scheduler, passageway control, ship control, gate, pump and lock processes) and second process hierarchy will be assembled from

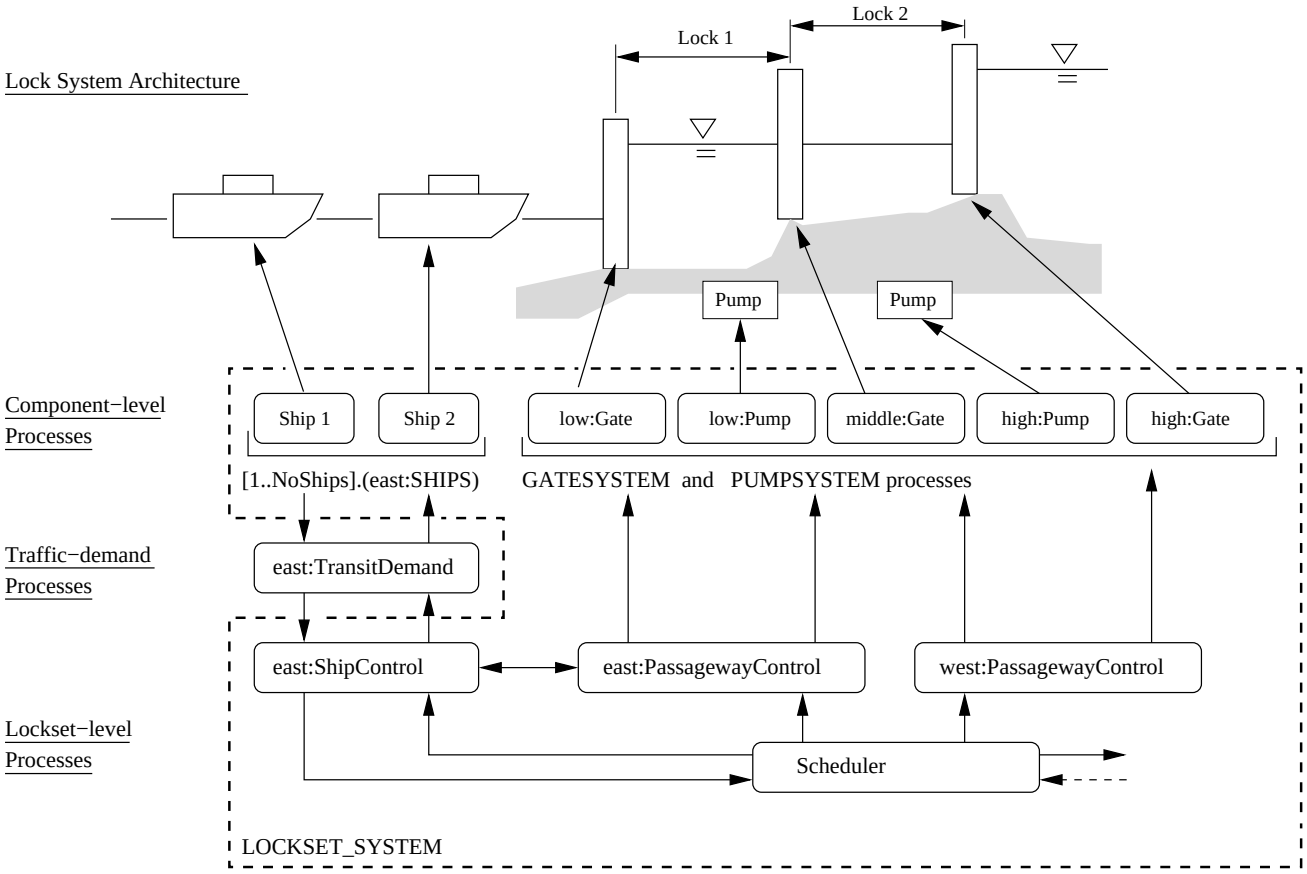

Figure 4.3: Elevation view of lockset, and component- and lockset-level process architecture.

the model of east- and west-bound traffic demand. The complete model of lockset behavior will simply correspond the composition of lockset and traffic demand process models.

Most of the relationships in Figures 4.3-4.4 can be classified as "observation," "action" or "sequences of actions." When the interaction between processes involves multiple actions, then notation can be simplified by defining sets of actions and then simply referring to that set by name. For example, the command

```
set TrafficControl = {  request, acquire, depart }
```

establishes the set TrafficControl to represent actions associated with ship interactions with the lockset system.

4.1.1 Basic Ship Behavior

In the simplest terms possible, a ship will transit a two-stage lockset by working through the following step-by-step procedure:

```
Step 1. Request permission to pass through the lockset.
Step 2. Wait in the arrival area.
Step 3. Acquire permission to enter the lockset.
Step 4. Enter lock 1.
Step 5. Enter lock 2.
Step 6. Depart.
```

The execution of steps 1-6 requires a specific sequencing of messages among ship, ship control, passageway control and scheduler processes, namely:

1. The `Ship` sends a message to the ship control `requesting` permission to transit the lockset. It then joins a queue of waiting ships in the arrival area.
2. At some point later in time, the scheduler will send a message to the `Ship` and `Ship Control` processes that it may `acquire` the resources of the lockset.
3. The `passageway controller` commands the ship to enter lock 1 (i.e., `enterlock1`).
4. The `passageway controller` commands the ship to enter lock 2 (i.e., `enterlock2`).
5. The `passageway controller` commands the ship to exit lock 2 and depart (i.e., `depart`).

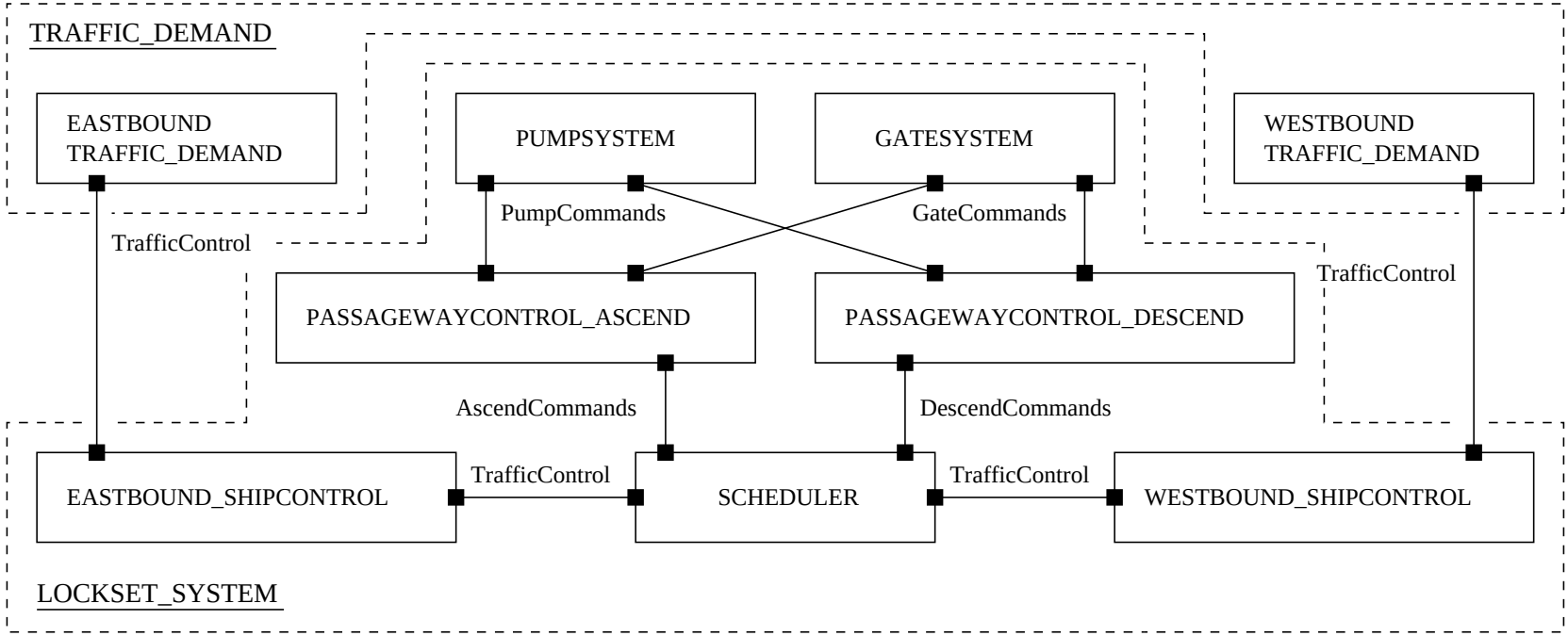

Figure 4.4: Process architecture for two-stage lockset.

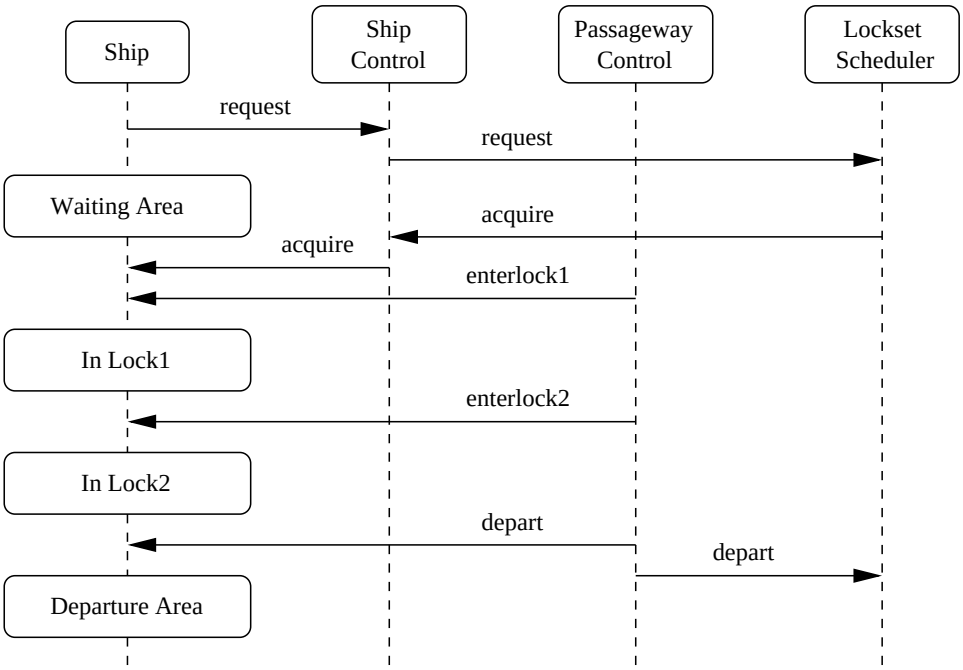

Figure 4.5: Simplified communication among ship, ship control, passageway control and scheduler processes for a ship transiting the lockset.

6. The `passageway controller` informs the lockset scheduler that the ship has departed lock 2.

Figure 4.5 provides a graphical summary of this sequence of message passing. The `Ship` time line shows that as a vessel moves through the lockset system it will actually progress through four spatial states; an arrival area, area lock1,, area lock2, and a departure area. For a ship ascending the lockset, lock1 will be the lower lock. For a ship descending the lockset, lock1 will be the upper lock. This subtle difference in context, coupled with the details of raising/lowering water levels and opening/closing gates requires the implementation of two passageway controllers – one for ascending the lockset and a second for descending the lockset.

Also notice that this model of ship behavior is deterministic – that is, the model assumes that sequences of actions will actually occur as planned. This leaves the task of formulating behavior models to the definition and composition of traffic demand and scheduler/ship control processes. A more sophisticated (non-deterministic) model of ship behavior would include provision for normal operations, plus breakdowns and accidents.

LTSA code. LTSA requires that processes operate continuously. Accordingly, in our preliminary implementation basic ship behavior is defined by the circular process:

```
SHIP = ( request -> acquire -> enterlock1 -> enterlock2 -> depart -> SHIP ).
```

Shipping personnel are certainly involved in the execution of the actions `request`, `acquire` and `depart`. The actions `enterlock1` and `enterlock2` are internal to the lockset itself. Hence, a much better solution is to simply define the ship behavior as

```
SHIP = ( request -> acquire -> depart -> SHIP ).
```

and then design the scheduler and passageway control processes to properly sequence `enterlock1` and `enterlock2` among other low-level actions for pump and gateway control.

4.1.2 Traffic Demand Processes

Travel demand processes are defined for convoys of east- and west-bound ships. The fragment of code:

```
const NoShips = 2
range S = 1..NoShips  // ship identities

SHIP = ( request -> acquire -> depart -> SHIP ).

||EASTBOUND_SHIPS = ( [i:S]:(east:SHIP) ).
||WESTBOUND_SHIPS = ( [i:S]:(west:SHIP) ).

// Create circular queue of east- and west-bound transit requests.

EASTBOUND_REQUESTS = QUEUE1 [1],
QUEUE1[i:S] = ( [i].east.request -> QUEUE1 [ i%NoShips + 1] ).
WESTBOUND_REQUESTS = QUEUE2 [1],
QUEUE2[i:S] = ( [i].west.request -> QUEUE2 [ i%NoShips + 1] ).

||EASTBOUND_TRAFFIC = ( EASTBOUND_SHIPS || EASTBOUND_REQUESTS ).
||WESTBOUND_TRAFFIC = ( WESTBOUND_SHIPS || WESTBOUND_REQUESTS ).
```

---

defines arrays of processes for ships traveling in the east- and west-bound directions. The corresponding actions are,

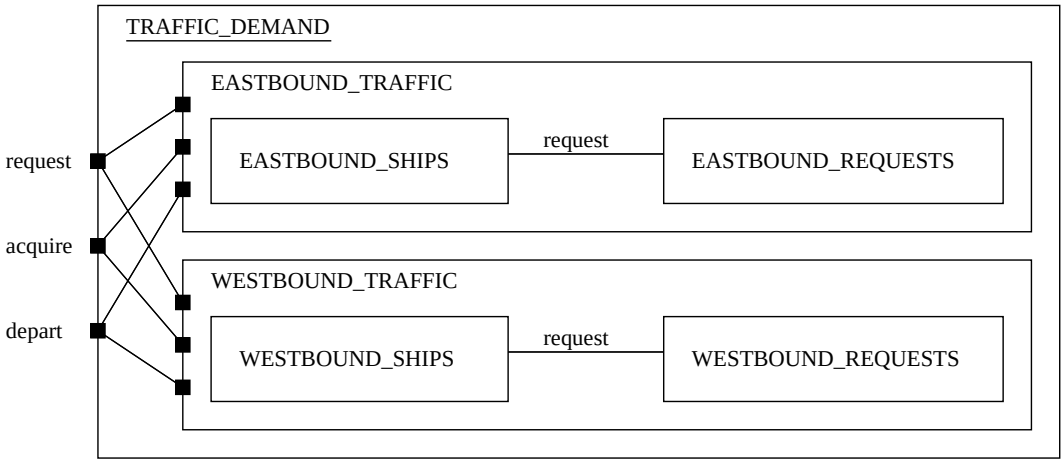

Figure 4.6: Process architecture for model of bi-directional (east- and west-bound) traffic demand.

```
[1].east.request    [2].east.request    [1].west.request    [2].west.request
[1].east.acquire    [2].east.acquire    [1].west.acquire    [2].west.acquire
[1].east.depart     [2].east.depart     [1].west.depart     [2].west.depart
```

and so forth. By themselves, the composed processes `EASTBOUND_SHIPS` and `WESTBOUND_SHIPS` do not imply an ordering of ships. This is handled by the circular queue of processes `EASTBOUND_REQUESTS` and `WESTBOUND_REQUESTS` which ensure requests are handled in the same order in which they are made. The overall traffic model is simply the composition of east- and west-bound traffic demands, i.e.,

```
||TRAFFIC_DEMAND = ( EASTBOUND_TRAFFIC || WESTBOUND_TRAFFIC ).
```

as illustrated in Figure 4.6.

4.1.3 Lockset System Processes

The physical lockset system corresponds to an assembly of scheduler, passageway control, ship control, and pump and gate processes. The lockset itself can be thought of a space process that can handle, at most, one ship at a time.

Pump and Gate Processes. The lockset contains three gates (tagged, low, middle, and high) and two pumps (tagged low and high). Gate and pump processes open and close gates and raise and lower water levels. Unconstrained behavior of the lock system components in achieve through composition of the pump and gate processes, e.g.,

```
||GATESYSTEM = ( low:GATE || middle:GATE || high:GATE ).
||PUMPSYSTEM = ( low:PUMP || high:PUMP ).
```

The GATESYSTEM and PUMPSYSTEM processes have 8 and 4 states, respectively.

It is important to note that in the present-day Panama Canal, water levels are lowered via gravity with water being drained from lake Gatun. So use of the term "pumps" is a little misleading. However, the renovated canal will cycle water from nearby ponds, and therefore, will use pumps to raise/lower water levels.

Ship Controller Processes. Ship controller processes are shared resources responsible for maintaining order between incoming requests and access to the lockset. This is achieved with the fragment of code:

```
// Create east- and west-bound ship control processes

   SHIPCONTROL = ( request -> acquire -> SHIPCONTROL).

 ||EASTBOUND_SHIPCONTROL = ( [i:S]::east:SHIPCONTROL ).
 ||WESTBOUND_SHIPCONTROL = ( [i:S]::west:SHIPCONTROL ).
```

At a glance (details to follow) it would seem that the ship control does the same thing as the scheduler and, thus, is not really needed. The important distinction between these processes lies in their view of ships – the ship controllers track the progress of specific ships by name. The lockset scheduler's relationship with ships is more abstract. It simply keeps track of incoming requests and provides permission for east- and west-bound ships to acquire the locksets resources.

Passageway Controller Processes. Passageway control is responsible for sequencing the ship and lockset actions (e.g., coordination of gate and pump operations) while a ship is inside the lockset. Accordingly, the fragment of code:

```
// Lockset-Level Passage Control. Sequences of actions for "ascend" operation ...
```

```
   PASSAGECONTROL_ASCEND = ( ascend   -> ASCEND
                           | resetlow -> low.pumpdown -> high.pumpdown -> ASCEND ),

   ASCEND = (  low.opengate -> enterlock1 ->    low.closegate ->  low.pumpup ->
            middle.opengate -> enterlock2 -> middle.closegate -> high.pumpup ->
              high.opengate ->  exitlock2 ->   high.closegate ->
                     depart -> PASSAGECONTROL_ASCEND ).

 ||EASTBOUND_PASSAGECONTROL = ( [i:S]::(east:PASSAGECONTROL_ASCEND) )/{
                forall [i:S] {     ascend/[i].east.ascend },
                forall [i:S] {   resetlow/[i].east.resetlow }}.

// Lockset-Level Passage Control. Sequences of actions for "descend" operation ...

   PASSAGECONTROL_DESCEND = ( descend   -> DESCEND
                            | resethigh -> low.pumpup -> high.pumpup -> DESCEND ),

   DESCEND = ( high.opengate -> enterlock1 ->   high.closegate -> high.pumpdown ->
             middle.opengate -> enterlock2 -> middle.closegate ->  low.pumpdown ->
                low.opengate ->  exitlock2 ->    low.closegate ->
                      depart -> PASSAGECONTROL_DESCEND ).

 ||WESTBOUND_PASSAGECONTROL = ( [i:S]::(west:PASSAGECONTROL_DESCEND) )/{
                forall [i:S] {   descend/[i].west.descend },
                forall [i:S] { resethigh/[i].west.resethigh }}.
```

---

defines required sequences of gate, pump and ship movement actions for ships ascending and descending the lockset system. As already mentioned at the top of this chapter, here we assume that east- and west-bound ships will ascend and descend the lockset respectively. It is a trivial matter to create a mirror passageway process with these dependencies swapped. Separate pathways are provided for the case where water levels need to be adjusted (i.e., resetlow or resethigh). The scheduler process will be designed to keep track of the chamber water levels and make appropriate adjustments.

Also notice that for both east- and west-bound pathways, ships will be required to ascend some locksets and descend others. To handle this requirement at the canal level, mirror copies of the lockset system will be created with the east/west parameters swapped. Then when the locksets are coupled with details of the traffic demand model, the appropriate sequences of high- and low-level operations will be composed.

Lockset Scheduler Process. The lockset scheduler process is the "brains" of the lockset operation and, as such, its design is the most challenging part of the behavior model formulation. The central design problem boils down to one of balancing services in multi-tasking environment. In

addition to being available to handle incoming transit requests, the scheduler needs to ensure that operations are efficient and fair, and will work with other processes (e.g., ship control) to ensure that physical (e.g., occupancy of lockset chamber) and operational constraints are respected. The concurrent nature of these activities leads to conflicts in the operational flexibility of the scheduler. For example, if the scheduler is busy monitoring the arrival/departure of a ship passing through the lockset, then it may not be immediately available to handle incoming transit requests.

Preliminary Implementation. Figure 4.7 shows elements of the scheduler operation in our preliminary implementation. The lockset scheduler will receive transit requests from east- and west-bound ships and at some later point issues permission to the ship controllers to begin the transit of a particular ship. The scheduler process needs to:

1. Take into account the number of ships waiting to transit the canal in either direction, and implement an appropriate policy of fairness, and
2. Take the correct action without the forced imposition of physical constraints (e.g., a constraint that says, at most, only one ship can occupy the lockset).

The fragment of code:

```
   const NoShips = 2
   range S = 1..NoShips  // ship identities

// Setup constants that will used by the scheduler.

   const East  = 0
   const West  = 1
   range TrafficDirection = East..West  // traffic direction for next ship.
   const Low   = 0
   const High  = 1
   range WaterLevel = Low..High         // waterlevel in locks...

// ======================================================================
// Lockset Scheduler that accounts for lock occupancy and number
// of east- and west-bound ships waiting to pass through the lock.
// ----------------------------------------------------------------------
// Variables:
//
//    we = number of east-bound ships waiting (0..NoShips).
//    ww = number of west-bound ships waiting (0..NoShips).
//    td = traffic direction for next ship (East or West).
//    wl = water level (Low or High).
// ======================================================================
```

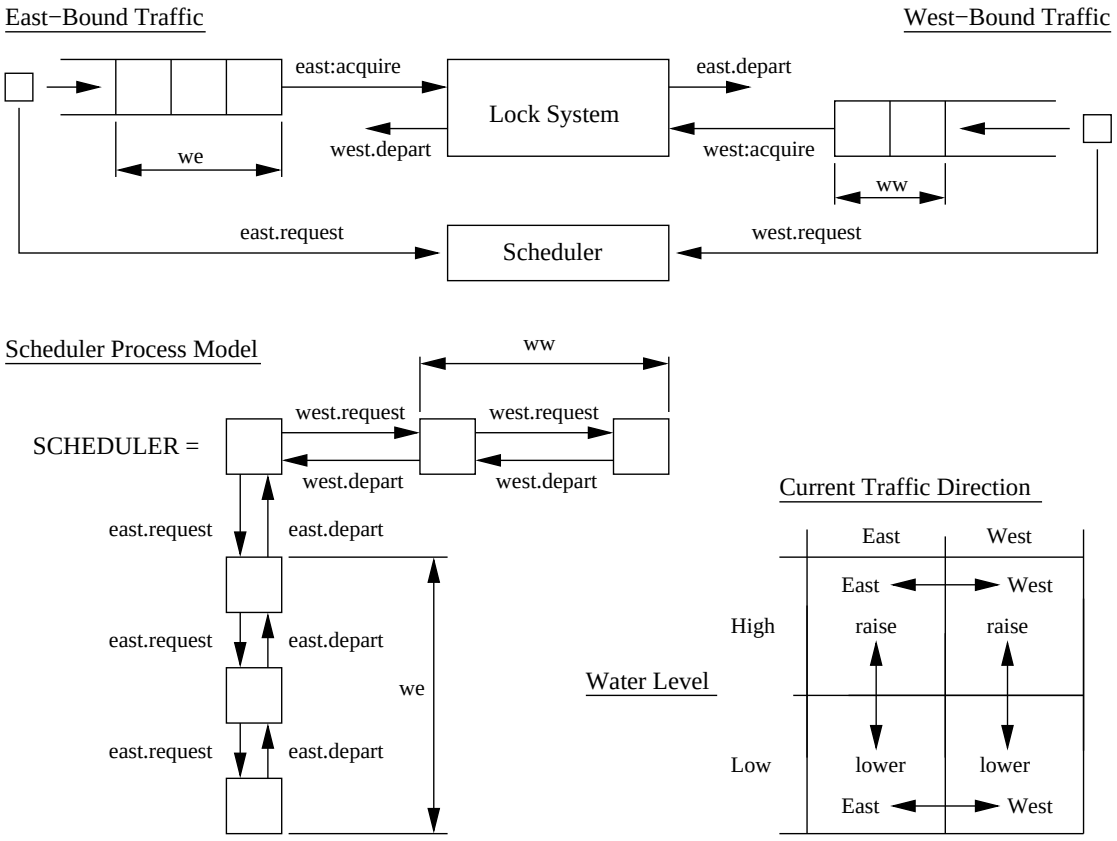

Figure 4.7: Schematic for process design of the traffic scheduler.

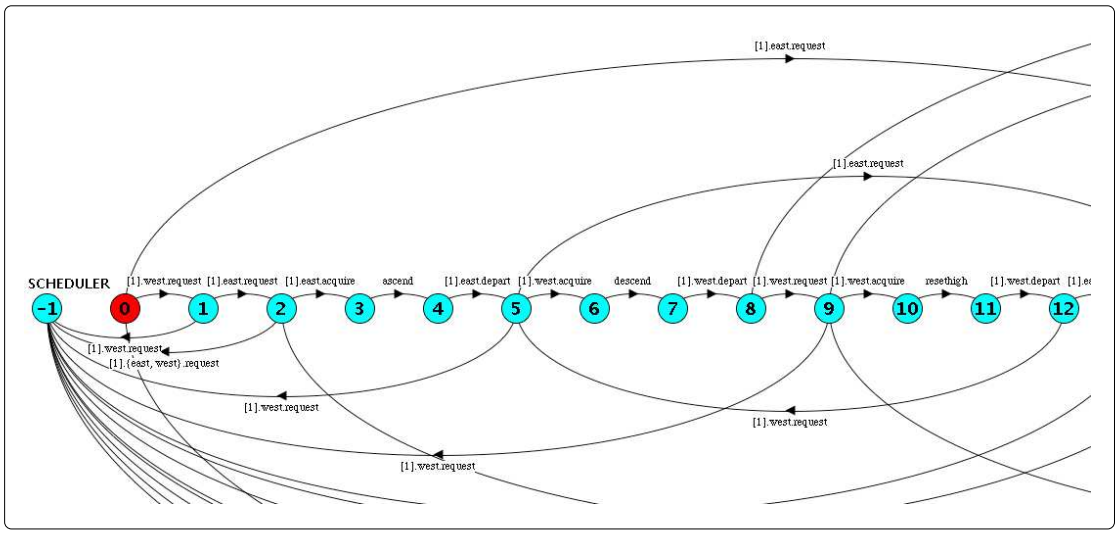

Figure 4.8: Scheduler processes and actions when NoShips = 1.

```
SCHEDULER = SCHEDULER[0][0][East][Low],
SCHEDULER[we:0..NoShips][ww:0..NoShips][td:TrafficDirection][wl:WaterLevel] = (

  // Register requests to transit lock in east- and west-bound directions.

      when ( we <= NoShips ) [S].east.request -> SCHEDULER[we+1][  ww][td][wl]
    | when ( ww <= NoShips ) [S].west.request -> SCHEDULER[  we][ww+1][td][wl]

  // East-bound assignments to ascend the lock system.

    | when ( ww >= 1 && we >= 1 && td == East && wl == Low ) [i:S].east.acquire ->
             ascend   -> [i].east.depart -> SCHEDULER[we-1][ ww][West][High]
    | when ( ww == 0 && we >= 1 && td == East && wl == Low ) [i:S].east.acquire ->
             ascend   -> [i].east.depart -> SCHEDULER[we-1][ ww][East][High]
    | when ( ww >= 1 && we >= 1 && td == East && wl == High ) [i:S].east.acquire ->
             resetlow -> [i].east.depart -> SCHEDULER[we-1][ ww][West][Low]
    | when ( ww == 0 && we >= 1 && td == East && wl == High ) [i:S].east.acquire ->
             resetlow -> [i].east.depart -> SCHEDULER[we-1][ ww][East][Low]

  // West-bound assignments to descend the lock system.

    | when ( ww >= 1 && we >= 1 && td == West && wl == High ) [i:S].west.acquire ->
             descend   -> [i].west.depart -> SCHEDULER[  we][ww-1][East][Low]
    | when ( ww >= 1 && we == 0 && td == West && wl == High ) [i:S].west.acquire ->
             descend   -> [i].west.depart -> SCHEDULER[  we][ww-1][West][Low]
    | when ( ww >= 1 && we >= 1 && td == West && wl == Low ) [i:S].west.acquire ->
             resethigh -> [i].west.depart -> SCHEDULER[  we][ww-1][East][High]
    | when ( ww >= 1 && we == 0 && td == West && wl == Low ) [i:S].west.acquire ->
             resethigh -> [i].west.depart -> SCHEDULER[  we][ww-1][West][High] ).
```

---

shows the scheduler implemented as a four-dimensional array of processes. The first and second dimensions keep track of the number of ships waiting for transit in the east- and west-bound directions (maximum value is `NoShips`). Variable we equals the number of east-bound ships waiting (0..NoShips). And variable ww equals the number of west-bound ships waiting (0..NoShips). Dimensions three and four keep track of the current traffic direction (td = East or West) and water level (wl = Low or High).

There are ten scheduler actions in our preliminary implementation:

Actions 1-2. Add requests to queues in the east and west-bound directions respectively. These actions leave the current traffic direction (td) and water level (wl) unchanged.

Actions 3-6. Deal with actions to ascend the lockset system. An ascend operation decreases the number of waiting east-bound ships by one (i.e., `we-1`).

Actions 7-10. Deal with actions to descend the lockset system. Each descend operation decreases the number of waiting west-bound ships by one (i.e., `ww-1`).

Actions 3-10 make adjustments to the traffic direction to take care of: (1) issues of fairness, and (2) the special case where a queue of waiting ships is empty and traffic demand simplifies to a uni-directional flow. Fairness is implemented through a very simply policy – if ships are waiting to traverse the canal in both the east- and west-bound directions (i.e. `ww >= 1` and `we >= 1`), then the traffic direction simply alternates between east- and west-bound directions. To get the process started, east-bound ships go first. A more realistic policy would take into account the size of ships, and possibly bundle them together as platoons. Also notice that size of the `SCHEDULER` process increases with the value of `NoShips`. Figure 4.8 shows, for example, a partial view of the states and actions in the scheduler process when `NoShips = 1`. Finally, the actions `ascend`, `descend`, `resetlow`, and `resethigh` only synchronize with the appropriate passageway controllers. They are neither part of the traffic demand nor ship behavior models.

A Second Implementation. The preliminary scheduler is limited in its ability to multi-task. Transit requests can be handled in any order. However, while a ship is traversing the canal chamber (i.e., between the actions acquire and depart), the scheduler process is unavailable to handle incoming requests. We can overcome this problem through a slight re-definition of the ship control processes, i.e.,

```
  SHIPCONTROL = ( request -> acquire -> depart -> SHIPCONTROL).

||EASTBOUND_SHIPCONTROL = ( [i:S]::east:SHIPCONTROL ).
||WESTBOUND_SHIPCONTROL = ( [i:S]::west:SHIPCONTROL ).
```

Now ship control is responsible for ensuring that `request`, `acquire`, `depart` action sequences are respected. This constraint specifically prohibits a ship from making a request to transit the canal before it has departed. It also gives the scheduler more freedom to multi-task. The revised code is as follows:

```
SCHEDULER = SCHEDULER[0][0][East][Low],
SCHEDULER[we:0..NoShips][ww:0..NoShips][td:TrafficDirection][wl:WaterLevel] = (

  // Register requests to transit lock in east- and west-bound directions.

      when ( we <= NoShips ) [S].east.request -> SCHEDULER[we+1][  ww][td][wl]
    | when ( ww <= NoShips ) [S].west.request -> SCHEDULER[  we][ww+1][td][wl]
```

```
// East-bound assignments to ascend the lock system.

  | when ( ww >= 0 && we >= 1 && td == East && wl == Low )
           [S].east.acquire ->   ascend -> SCHEDULER[we-1][ ww][East][High]
  | when ( ww >= 0 && we >= 1 && td == East && wl == High )
           [S].east.acquire -> resetlow -> SCHEDULER[we-1][ ww][East][High]

// East-bound departures from the lock system.

  | when ( ww >= 1 && td == East && wl == High )
           [S].east.depart -> SCHEDULER[we][ww][West][High]
  | when ( ww == 0 && td == East && wl == High )
           [S].east.depart -> SCHEDULER[we][ww][East][High]

// West-bound assignments to descend the lock system.

  | when ( ww >= 1 && we >= 0 && td == West && wl == High )
           [S].west.acquire ->   descend -> SCHEDULER[we][ww-1][West][Low]
  | when ( ww >= 1 && we >= 0 && td == West && wl == Low )
           [S].west.acquire -> resethigh -> SCHEDULER[we][ww-1][West][Low]

// West-bound departures from the lock system.

  | when ( we >= 1 && td == West && wl == Low )
           [S].west.depart -> SCHEDULER[we][ww][East][Low]
  | when ( we == 0 && td == West && wl == Low )
           [S].west.depart -> SCHEDULER[we][ww][West][Low] ).
```

Now we see that the scheduler can handle incoming transit requests while the lockset chamber is occupied.

Remark. The first and second implementations of the lockset scheduler are defined for bi-directional transit operations without break. If there are no ships waiting to transit the canal in a particular direction, then the scheduler operations default to the management of uni-directional traffic. In Chapter 5, the scheduler will be extended to account for the proper handling of traffic during maintenance and emergency events.

Interaction of Ship, Passageway and Scheduler Processes. When a SHIP process performs the action `request` it is shared by the scheduler and, in fact, is a communication to the scheduler that then uses that information to create a queue for the ships on that side of the lock. The acquiring of the use of the lock, performed by the action `acquire` must be able to be performed by the controller and the scheduler; meaning the scheduler must give the ship permission to pass

through, and the controller must be ready to send the ship through as well. For example, the controller can make a ship wait by not sharing the `acquire` action until appropriate adjustments to the water level have been made.

4.1.4 Lockset-Level Behavior

Lockset-level behavior model corresponds to a parallel composition of the lockset system with the process model for traffic demand, i.e.,

```
 ||LOCKSET_SYSTEM = ( SCHEDULER ||
                      WESTBOUND_SHIPCONTROL ||
                      EASTBOUND_SHIPCONTROL ||
                      WESTBOUND_PASSAGECONTROL ||
                      EASTBOUND_PASSAGECONTROL ||
                      PUMPSYSTEM || GATESYSTEM ).

// Compose model of lockset system behavior ...

 ||LOCKSET_BEHAVIOR = ( LOCKSET_SYSTEM || TRAFFIC_DEMAND ).
```

The upper- and lower- blocks of Table 4.1 show process sizes for preliminary and second implementations of the scheduler process, and NoShips equal to 1, 2 and 3. In both cases, the fully composed and minimized process models of lockset behavior have exactly the same number of states[1]. The computational procedure works for NoShips equal to 1 and 2, but fails thereafter (due to a stack overflow).

4.1.5 Lockset-Level Safety and Liveliness

As already noted in Chapter 2, a safety property asserts that nothing bad happens during the canal operation. At the lockset level we need to ensure that:

1. The canal scheduler will not assign more than one ship to a lock,
2. Floods will be prevented by ensuring that a gate will not open before water levels on both side of the gate are equalized.

Safety checks are compositional in the sense that if there is no violation at a subsystem level, then there cannot be a violation when that subsystem is composed with other subsystems.

[1] All computations were conducted on an Apple Powerbook G4, with 1 GB of memory.

Part 1. Preliminary Implementation of the Scheduler.

No Ships	E-W Ship Model	TRAFFIC_DEMAND States	SCHEDULER States	LOCKSET_SYSTEM States	LOCKSET_BEHAVIOR States	LOCKSET_BEHAVIOR Minimized
1	[1..1]	9	33	4,096	4,096	4,096
2	[1..2]	324	132	7,680	8,192	8,192
3	[1..3]	6,561	352	11,264	36,864	...failed!!

Part 2. Second Implementation of the Scheduler.

No Ships	E-W Ship Model	TRAFFIC_DEMAND States	SCHEDULER States	LOCKSET_SYSTEM States	LOCKSET_BEHAVIOR States	LOCKSET_BEHAVIOR Minimized
1	[1..1]	9	23	2,496	2,496	2,496
2	[1..2]	324	81	2,752	9,792	9,792
3	[1..3]	6,561	204	3,008	21,952	...failed!!

Table 4.1: No of states in the lockset behavior model for NoShips = 1, 2 and 3.

Validate Safety Against Flooding

Safety properties are specified in FSP by property processes. As noted in Chapter 2, LTSA requires that the property process specifications be deterministic. This constraint complicates the verification of system safety against flooding because: (1) the traffic demand model is non-deterministic (i.e., we have no control over the ordering of east- and west-bound transit requests), and (2) water levels in the canal depend of the sequencing of east- and west-bound transits. The abbreviated fragment of code:

```
// ================================================================================
// Validate that a flood will not occur in the lower locks.
// ================================================================================

// Check pump operation for a stream of east-bound ships ....

   property LOWER_PUMPS = ( [j:1..NoShips].east.acquire ->
                            ascend   -> [j].east.depart -> [j].east.acquire ->
                            resetlow -> [j].east.depart -> LOWER_PUMPS ).

// Check pump operation for a stream of west-bound ships ....

   property RAISE_PUMPS = ( [j:1..NoShips].west.acquire ->
                            descend   -> [j].west.depart -> [j].west.acquire ->
                            resethigh -> [j].west.depart -> RAISE_PUMPS ).

  ||SYSTEM_FLOOD_CHECK1 = ( LOCKSET_SYSTEM || LOWER_PUMPS ).
  ||SYSTEM_FLOOD_CHECK2 = ( LOCKSET_SYSTEM || RAISE_PUMPS ).
```

establishes properties for LOWER_PUMPS and RAISE_PUMPS associated with sequences of ascend and descend operations respectively. The purpose of these property checks is to ensure that the double locksets will never open the gate when there is an imbalance in water levels on either side of the first gate in transit. More precisely, if the canal handles a series of east-bound transit requests then an ascend operation should always be followed by a resetlow action, the latter being responsible for lowering water levels in the pumps. A detailed trace of actions generated from the safety check for LOWER_PUMPS in LTSA is as follows,

```
Trace to property violation in LOWER_PUMPS:
1.east.request
1.west.request
1.east.acquire
ascend
1.east.low.opengate
1.east.enterlock1
1.east.low.closegate
1.east.low.pumpup
1.east.middle.opengate
1.east.enterlock2
1.east.middle.closegate
1.east.high.pumpup
1.east.high.opengate
1.east.exitlock2
1.east.high.closegate
1.east.depart
1.east.request
1.west.acquire
descend
1.west.high.opengate
1.west.enterlock1
1.west.high.closegate
1.west.high.pumpdown
1.west.middle.opengate
1.west.enterlock2
1.west.middle.closegate
1.west.low.pumpdown
1.west.low.opengate
1.west.exitlock2
1.west.low.closegate
1.west.depart
1.east.acquire
ascend
```

Here we see that two ascend operations can occur, but only if they are separated by a descend operation (i.e., a west-bound ship transits the system). In this particular case the safety violation is okay because the west-bound ship will lower the water level, thereby allowing the second ascend operation to proceed safely.

Validate Single Lock Usage

The fragment of code:

```
property LOCK_OCCUPANCY =
         ( [j:1..NoShips].east.acquire -> [j].east.depart -> LOCK_OCCUPANCY
         | [i:1..NoShips].west.acquire -> [i].west.depart -> LOCK_OCCUPANCY ).
```

establishes a validation test for lock occupancy. The test basically says: if a specific ship acquires resources of the lockset, then it needs to depart before another ship acquires the lockset. Any other sequence of actions will result in an error.

Validation tests can also be established by defining an array of process states and letting LTSA automatically generate transitions to an error state for actions not consistent with the process definition. With the latter approach in mind, the fragment of code:

```
LOCK_OCCUPANCY = SPACES[1],
SPACES[i:0..1] = ( when(i>0)
                       [j:1..NoShips].east.acquire -> SPACES[i-1]
                     | [j:1..NoShips].west.acquire -> SPACES[i-1]
                   | when(i<1)
                       [j:1..NoShips].east.depart  -> SPACES[i+1]
                     | [j:1..NoShips].west.depart  -> SPACES[i+1] ).
```

creates a process for the lockset system modeled as resource that provides physical space for at most one ship at a time. Physical considerations dictate that once a ship has occupied the space in the lockset, it must depart before another ship can enter the lockset.

A basic implementation of this requirement is illustrated in Figure 4.9. Correct enforcement of the physical space constraint corresponds to the lock process behavior alternating between states 0 and 1. Notice that the model simply keeps track of acquire and depart operations, and does not include details of specific ship arrivals and departures.

The behavior model will contain errors if more than one ship acquires the space, or alternatively, when the lock system is occupied, more than one ship tries to depart the lock system. LTSA accounts for these possibilities by automatically inserting actions that will lead to an error state. Execution of the lock occupancy test is defined by fragment of code:

```
||LOCK_OCCUPANCY_CHECK1 = ( LOCKSET_SYSTEM || LOCK_OCCUPANCY ).
```

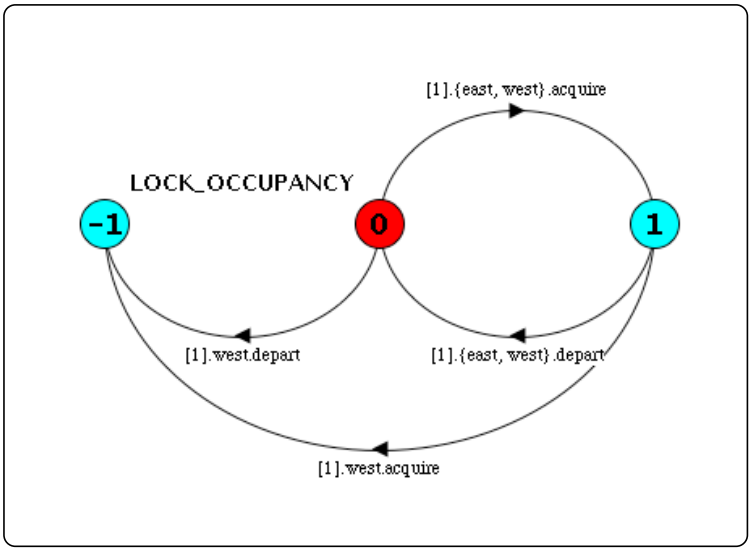

Figure 4.9: LTS diagram for the lockset system modeled as a single use space resource.

If LOCK_OCCUPANCY_CHECK1 contains an error state, then a pathway in the lockset system exists where an acquire operation is followed by a second acquire and/or a depart action is followed by a second depart operation. In either case, LOCK_OCCUPANCY_CHECK1 fails. For our model, however, the composed model is free of an error state and the test passes.

Validate Progress of Ship Transit

Liveliness/progress properties assert that something eventually good happens. At the lockset level, this translates to ensuring that if at ship makes a request to transit the lockset, eventually it will transit and depart the system. The fragment of code:

```
progress EASTPASS = { [S].east.depart }
progress WESTPASS = { [S].west.depart }
```

simply checks that at least one of the ships (i.e., 1..NoShips) will depart in each of the east- and west-bound directions,

4.2 Model 2. Simplified Models of Lockset-Level Behavior

From Section 4.1 it is evident that while the all-in-one approach to composition of lockset-level process models, i.e.,

```
||LOCKSET_SYSTEM = ( SCHEDULER ||
                     WESTBOUND_SHIPCONTROL ||
                     EASTBOUND_SHIPCONTROL ||
                     WESTBOUND_PASSAGECONTROL ||
                     EASTBOUND_PASSAGECONTROL ||
                     PUMPSYSTEM ||
                     GATESYSTEM ).

||LOCKSET_BEHAVIOR = ( LOCKSET_SYSTEM || TRAFFIC_DEMAND ).
```

it technically permissible, in practice the approach is fundamentally flawed because for all but the smallest problems, behavior models and their associated model checking procedures quickly become computationally intractable. Indeed, with only one east-bound ship and one west-bound ship, LOCKSET_SYSTEM composes into a model with 4,096 states. See Table 4.1. When the number of east- and west-bound ships is increased to three, the process model grows to more than 21,000 states.

The hypothesis of our investigation is that these difficulties can be mitigated by exploiting the natural structure of canal system behavior models. In related work, Cheung and Kramer [8] point out that when systems have a structure that is naturally hierarchical, safety properties tend to be associated with a particular subsystem. Therefore, in their evaluation there is no need to consider actions belonging to other subsystems. Moreover, progress properties rely on a set of actions being activated; most often they can be evaluated with respect to actions at a particular level of detail, with lower level actions being removed from consideration.

To address these concerns, in this section we derive a systematic procedure for composing models for specific viewpoints. Viewpoints serve two purposes. First, they motivate the generation of abstractions relative to a design concern (e.g., safety and progress checks). Their second purpose is one of simplification – by systematically removing actions that are not related to a design decision, models can be trimmed to minimize their complexity (e.g., generation of simplified abstractions for "higher-level" modeling). In the terminology of Cheung and Kramer [8] this is called partial ordering.

We propose a simple method for partial ordering achieved through the assembly action-process relationships and the identification of families of processes common to actions and groups

of actions. We also immediately minimize the size of all intermediate results. The result is a computational procedure where process models achieve their required functionality, but may have a size that is orders of magnitude smaller than in the all-in-one approach to composition. Details of the appropriate source code are located in Appendix 2.

4.2.1 Architecture of Lockset-Level Behavior Model

Figures 4.10 and 4.11 show abstract and detailed process hierarchy views of: (1) the traffic demands process interacting with the lockset system process, (2) the lockset system process organized into lockset control and passageway system process hierarchies, and (3) processes for compositional validation of lockset behavior properties. We employ white and black dots to show dependencies between the process hierarchies.

The traffic demand process model consists of families of east- and west-bound ships; it will stay exactly as formulated in Section 4.1.

The design challenge lies in the formulation of simplified versions of the `LOCKSET_SYSTEM` process model. In the initial model formulation, models of lockset system behavior were defined through "all-in-one composition" of seven basic processes (i.e., `SCHEDULER`, `WESTBOUND_SHIPCONTROL` and so forth). Now, assembly of the model occurs over three layers:

1. `LOCKSET_CONTROL` is the composition of scheduler and east- and west-bound ship controls.
2. `PASSAGEWAY_SYSTEM` is the composition of ascend and descend processes, plus processes for the pump and gate systems.
3. Gate and pump systems are defined through the composition of individual pump and gate processes. These details are not shown in Figure 4.11.

The `LOCKSET_SYSTEM` and `TRAFFIC_DEMAND` processes are connected by request, acquire, and depart actions. White and black box notation shows dependencies of the actions. For example, a ship will request transit of the lock system – the request is the requirement, the lockset system provides for processing of the request.

With this process hierarchy in place, the fundamental question is: can it lead to process models that are smaller (i.e., abstract) and/or capable of validating properties? We explore the potential benefits of the opportunity by defining alphabets for each process and then building tables

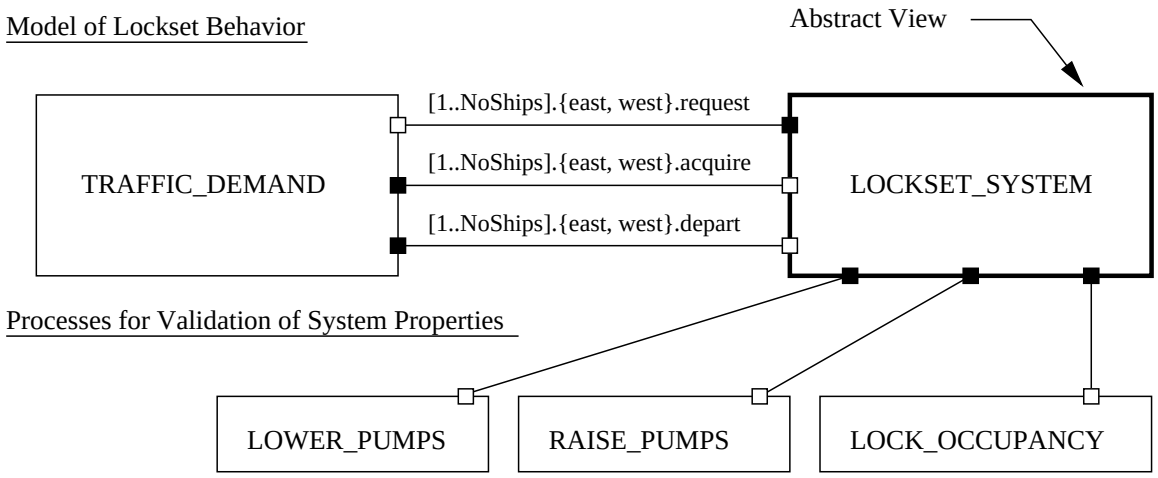

Figure 4.10: Schematic for development of simplified models of lockset system behavior. White dots represent requirements. Black dots represent provisions.

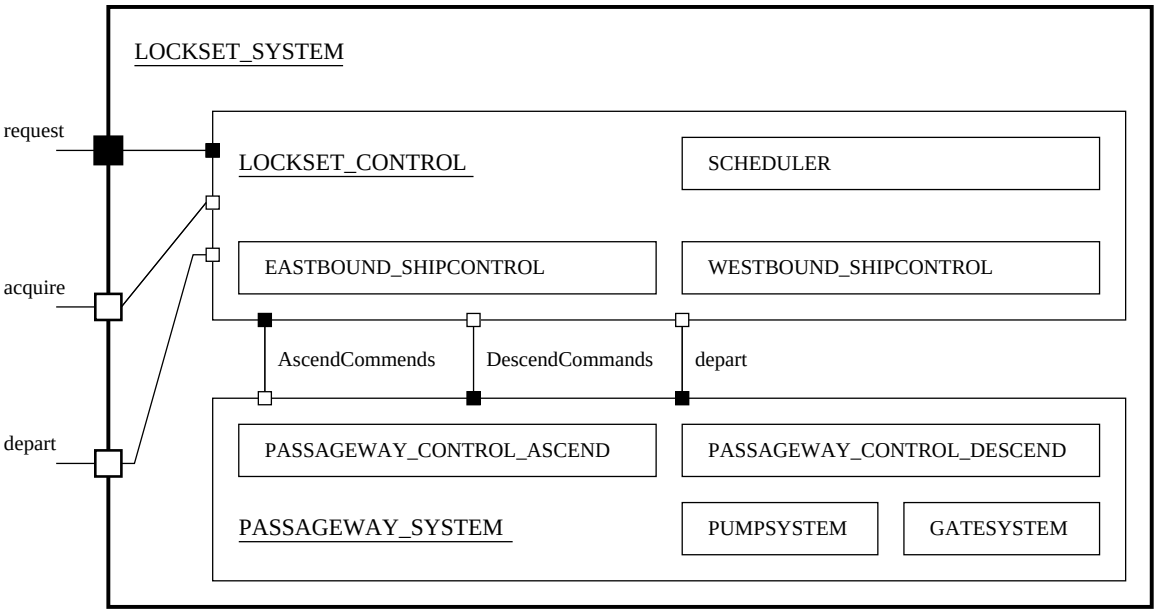

Figure 4.11: Schematic for development of simplified models of lockset system behavior. White dots represent requirements. Black dots represent provisions.

of action-process dependencies. The latter, in turn, allow for assembly of viewpoint-action-process dependencies.

4.2.2 Alphabets for Models of Traffic Demand and Lockset Behavior

By default, processes in LTSA are defined through the specification of sequences of actions. These actions, in turn, define an alphabet from which processes are assembled.

When models of lockset-level behavior are constructed with one east-bound ship and one west-bound ship, the traffic demand and the lockset system alphabets have 6 and 28 actions respectively. See Tables 4.2 and 4.3. The model of traffic demand is composed from 4 processes. The lockset system model is composed from 11 processes.

4.2.3 Viewpoint-Action-Process Traceability

To systematically determine which parts of the behavior model can be omitted without affecting a pre-defined viewpoint, we propose that the process-action dependency be reversed. That is, for each action (or, when appropriate, group of actions) a list of dependent processes is assembled. Then, if a viewpoint is defined in terms of critical actions, traceability links can be established from a viewpoint to dependent actions to dependent processes. Table 4.5 shows traceability dependencies in the pathway from viewpoints, to actions, to dependent processes.

Viewpoint 1. Simplified model of ship movement (i.e., a minimal version `LOCK_SYSTEM`),

Viewpoint 2. Passageway Safety against Flooding (i.e., `LOWER_PUMPS` and `RAISE_PUMPS`), and

Viewpoint 3. Passageway Occupancy (i.e., `LOCK_OCCUPANCY`).

All three viewpoints can be evaluated through the composition of `SCHEDULER`, `SHIPCONTROL` and `PASSAGECONTROL` processes (and variations thereof) in a manner consist with the process hierarchy defined in Figure 4.11, i.e.,

```
||LOCKSET_CONTROL   = ( SCHEDULER || WESTBOUND_SHIPCONTROL || EASTBOUND_SHIPCONTROL ).
||PASSAGEWAY_SYSTEM = ( WESTBOUND_PASSAGECONTROL || EASTBOUND_PASSAGECONTROL ).
||LOCKSET_SYSTEM    = ( LOCKSET_CONTROL || PASSAGEWAY_SYSTEM ).
```

The underlying assumption in this model is that the processes `LOCKSET_CONTROL` and `PASSAGEWAY_SYSTEM` will be autonomous and only synchronize through shared actions. As implemented, however,

Process	Alphabet
EASTBOUND_TRAFFIC.EASTBOUND_SHIPS.1:east:SHIP	[1].east.{acquire, depart, request}
EASTBOUND_TRAFFIC.EASTBOUND_REQUESTS	[1].east.request
WESTBOUND_TRAFFIC.WESTBOUND_SHIPS.1:west:SHIP	[1].west.{acquire, depart, request}
WESTBOUND_TRAFFIC.WESTBOUND_REQUESTS	[1].west.request

Table 4.2: Alphabet for model of traffic demand.

Process	Alphabet
SCHEDULER	{[1].{east, west}.{acquire, depart, request}, {ascend, descend, resethigh, resetlow}}
WESTBOUND_SHIPCONTROL.{1}::west:SHIPCONTROL	[1].west.{acquire, request, depart}
EASTBOUND_SHIPCONTROL.{1}::east:SHIPCONTROL	[1].east.{acquire, request, depart}
WESTBOUND_PASSAGECONTROL. {1}::west:PASSAGECONTROL_DESCEND	{[1].west.{{depart, enterlock1, enterlock2, exitlock2}, {high, low}.{closegate, opengate, pumpdown, pumpup}, middle.{closegate, opengate}}, {descend, resethigh}}
EASTBOUND_PASSAGECONTROL. {1}::east:PASSAGECONTROL_ASCEND	{[1].east.{{depart, enterlock1, enterlock2, exitlock2}, {high, low}.{closegate, opengate, pumpdown, pumpup}, middle.{closegate, opengate}}, {ascend, resetlow}}
PUMPSYSTEM.low:PUMP	low.{pumpdown, pumpup}
PUMPSYSTEM.high:PUMP	high.{pumpdown, pumpup}
GATESYSTEM.low:GATE	low.{closegate, opengate}
GATESYSTEM.middle:GATE	middle.{closegate, opengate}
GATESYSTEM.high:GATE	high.{closegate, opengate}
LOCK_OCCUPANCY	[1].{east, west}.{acquire, depart}

Table 4.3: Alphabet for lockset system model and LOCK_OCCUPANCY process needed for validation of lock occupancy.

Sets of Actions	Dependent Processes
ascend	SCHEDULER
resetlow	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
descend	SCHEDULER
resethigh	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
[1].west.{acquire, request}	SCHEDULER
	WESTBOUND_SHIPCONTROL.{1}::west:SHIPCONTROL
[1].east.{acquire, request}	SCHEDULER
	EASTBOUND_SHIPCONTROL.{1}::east:SHIPCONTROL
[1].east.depart	SCHEDULER
	EASTBOUND_SHIPCONTROL.{1}::east:SHIPCONTROL
	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
[1].west.depart	SCHEDULER
	WESTBOUND_SHIPCONTROL.{1}::west:SHIPCONTROL
	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
low.{pumpdown, pumpup}	PUMPSYSTEM.low:PUMP
	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
high.{pumpdown, pumpup}	PUMPSYSTEM.high:PUMP
	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
low.{closegate, opengate}	GATESYSTEM.low:GATE
	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
middle.{closegate, opengate}	GATESYSTEM.middle:GATE
	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
high.{closegate, opengate}	GATESYSTEM.high:GATE
	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
[1].west.{ enterlock1, enterlock2, exitlock2 }	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
[1].east.{ enterlock1, enterlock2, exitlock2 }	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND

Table 4.4: Action-process relation ships in lockset system model.

Viewpoint 1: Simplified model of ship movement (i.e., processes LOCK_SYSTEM)	
Defining Actions	Dependent Processes
[j:1..NoShips].east.request	SCHEDULER
[j:1..NoShips].east.acquire	WESTBOUND_SHIPCONTROL.{1}::west:SHIPCONTROL
[j:1..NoShips].east.depart	EASTBOUND_SHIPCONTROL.{1}::east:SHIPCONTROL
[j:1..NoShips].west.request	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
[j:1..NoShips].west.acquire	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
[j:1..NoShips].west.depart	
	EASTBOUND_TRAFFIC.EASTBOUND_SHIPS.1:east:SHIP
	EASTBOUND_TRAFFIC.EASTBOUND_REQUESTS
	WESTBOUND_TRAFFIC.WESTBOUND_SHIPS.1:west:SHIP
	WESTBOUND_TRAFFIC.WESTBOUND_REQUESTS
Viewpoint 2: Passageway Safety against Flooding (i.e., processes LOWER_PUMPS and RAISE_PUMPS)	
Defining Actions	Dependent Processes
ascend,	SCHEDULER
descend,	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
resetlow,	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
resethigh,	WESTBOUND_SHIPCONTROL.{1}::west:SHIPCONTROL
[j:1..NoShips].east.acquire	EASTBOUND_SHIPCONTROL.{1}::east:SHIPCONTROL
[j:1..NoShips].east.depart	
[j:1..NoShips].west.acquire	EASTBOUND_TRAFFIC.EASTBOUND_SHIPS.1:east:SHIP
[j:1..NoShips].west.depart	EASTBOUND_TRAFFIC.EASTBOUND_REQUESTS
	WESTBOUND_TRAFFIC.WESTBOUND_SHIPS.1:west:SHIP
	WESTBOUND_TRAFFIC.WESTBOUND_REQUESTS
Viewpoint 3: Passageway Occupancy (i.e., process LOCK_OCCUPANCY)	
Defining Actions	Dependent Processes
[j:1..NoShips].east.{acquire}	SCHEDULER
[j:1..NoShips].east.{depart}	EASTBOUND_PASSAGECONTROL.{1}::east:PASSAGECONTROL_ASCEND
[j:1..NoShips].west.{acquire}	WESTBOUND_PASSAGECONTROL.{1}::west:PASSAGECONTROL_DESCEND
[j:1..NoShips].west.{depart}	WESTBOUND_SHIPCONTROL.{1}::west:SHIPCONTROL
	EASTBOUND_SHIPCONTROL.{1}::east:SHIPCONTROL
	EASTBOUND_TRAFFIC.EASTBOUND_SHIPS.1:east:SHIP
	EASTBOUND_TRAFFIC.EASTBOUND_REQUESTS
	WESTBOUND_TRAFFIC.WESTBOUND_SHIPS.1:west:SHIP
	WESTBOUND_TRAFFIC.WESTBOUND_REQUESTS

Table 4.5: Schematic for viewpoint-action-process traceability in lockset model.

PASSAGEWAY_SYSTEM is not fully autonomous and, in fact, only responds to actions instigated by the scheduler (i.e., a master-slave relationship among processes). Moreover, although the east- and west-bound passageway processes share common actions (e.g., enterlock1, enterlock2) and theoretically synchronize on those actions, in practice, the scheduler process ensures that this never happens – the canal system is handling either an east-bound ship or a west-bound ship, but never east- and west-bound ships concurrently. Together these observations suggest that reliable evaluation of the viewpoints may be possible by using even fewer processes than as indicated in Table 4.5.

4.2.4 Viewpoint 1: Composition of Behavior for Ship Movement

This viewpoint is motivated by the need for a simplified model of ship movement (i.e., a minimal version LOCK_SYSTEM), which downstream, will be suitable for inclusion in a canal-level model of behavior. With the above-mentioned observations in place, the fragment of FSP code:

```
   minimal ||LOCKSET_CONTROL = ( SCHEDULER || WESTBOUND_SHIPCONTROL || EASTBOUND_SHIPCONTROL ).

   minimal ||PASSAGEWAY_SYSTEM = ( WESTBOUND_PASSAGECONTROL || EASTBOUND_PASSAGECONTROL ) @ {
                                   resethigh, resetlow, ascend, descend,
                                   [S].east.depart, [S].west.depart }.

// Lockset system that interacts with the traffic demand model...

   minimal ||LOCKSET_SYSTEM1 = ( LOCKSET_CONTROL ).

// Lockset system that omits details of the pump and gate operations ....

   minimal ||LOCKSET_SYSTEM2 = ( LOCKSET_CONTROL || PASSAGEWAY_SYSTEM ).

// ================================================================================
// Viewpoint 1. Composition of Behavior for Ship Movement.
// ================================================================================

   minimal ||LOCKSET_BEHAVIOR1 = ( LOCKSET_SYSTEM1 || TRAFFIC_DEMAND ) @ {
                                  [S].{east,west}.request,
                                  [S].{east,west}.acquire,
                                  [S].{east,west}.depart }.

   minimal ||LOCKSET_BEHAVIOR2 = ( LOCKSET_SYSTEM2 || TRAFFIC_DEMAND ) @ {
                                  [S].{east,west}.request,
                                  [S].{east,west}.acquire,
                                  [S].{east,west}.depart }.
```

implements and minimizes two versions of LOCKSET_SYSTEM, one that includes LOCKSET_CONTROL and PASSAGEWAY_SYSTEM, and a second model LOCKSET_SYSTEM based on LOCKSET_CONTROL alone. Notice that for the PASSAGEWAY_SYSTEM process we only carry forward actions that are critical to communication between LOCKSET_CONTROL and PASSAGEWAY_SYSTEM (i.e., resethigh, resetlow, ascend, descend), and/or the ship movement model (i.e., [S].east.depart, [S].west.depart). Finally, two versions of lockset behavior are composed.

Figures 4.12 and 4.13 show representative lockset and lockset-system behaviors when NoShips = 1. The important point to note is that east- and west-bound requests for transit can arrive in any order. However, once a request is made, it cannot be made again until transit of the locksystem is complete. Moreover, in the case where one east-bound request and one west-bound request have been made, the east-bound request will acquire access to the lockset system first. This strategy is encoded within the scheduler.

Scalability of the Lockset Behavior Model. Table 4.6 shows the size of the constituent processes as a function of NoShips.

No Ships	E-W Ship Model	TRAFFIC_DEMAND States	SCHEDULER States	LOCKSET_CONTROL Minimized States	PASSAGEWAY_SYSTEM Minimized States
1	[1..1]	9	23	26	4
2	[1..2]	324	81	26	4
3	[1..3]	6,561	204	26	4

No Ships	E-W Ship Model	LOCKSET_SYSTEM1 Minimized States	LOCKSET_SYSTEM2 Minimized States	LOCKSET_BEHAVIOR1 Minimized States	LOCKSET_BEHAVIOR2 Minimized States
1	[1..1]	26	26	12	12
2	[1..2]	26	26	48	48
3	[1..3]	26	26	108	108

Table 4.6: No of states in the lockset behavior model for NoShips = 1, 2 and 3.

Not only are the process sizes several orders of magnitude smaller than in the initial formulation (for details, see Table 4.1), but the computational procedure remains computationally tractable. The dual strategy of only including processes related to a specific decision, and incrementally assembling minimized processes has a huge impact on the computational feasibility of the analysis. As a case

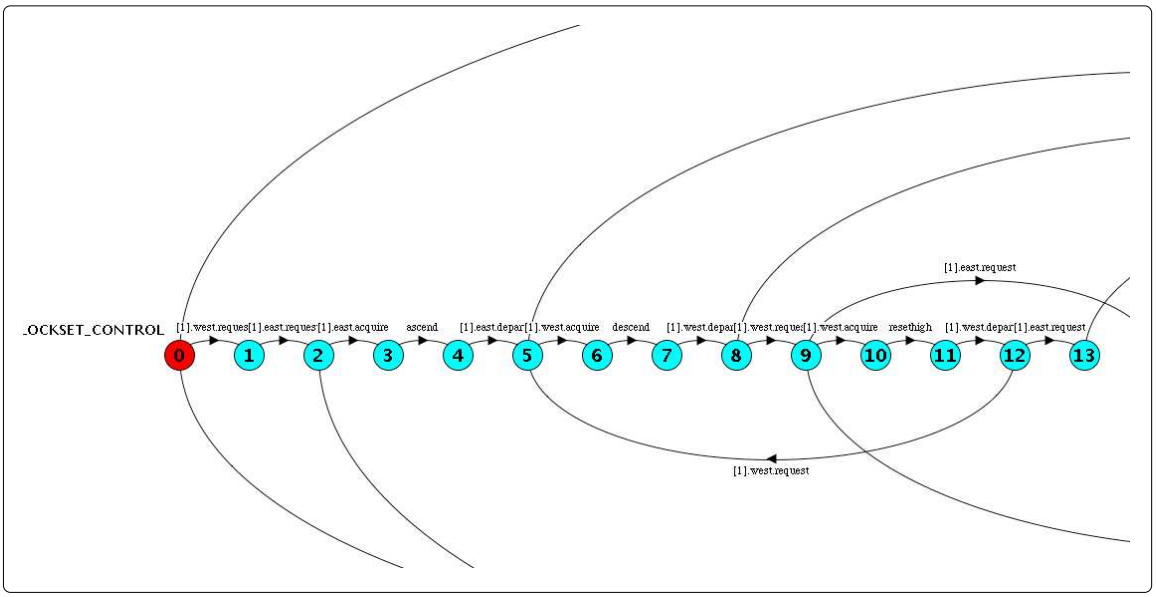

Figure 4.12: Lockset control for NoShips = 1.

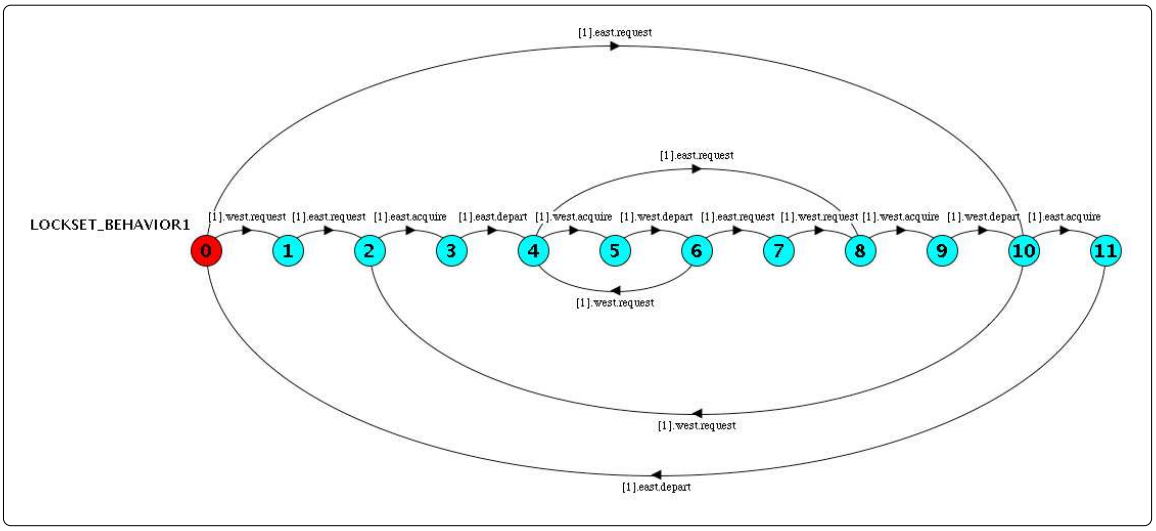

Figure 4.13: Lockset behavior for NoShips = 1.

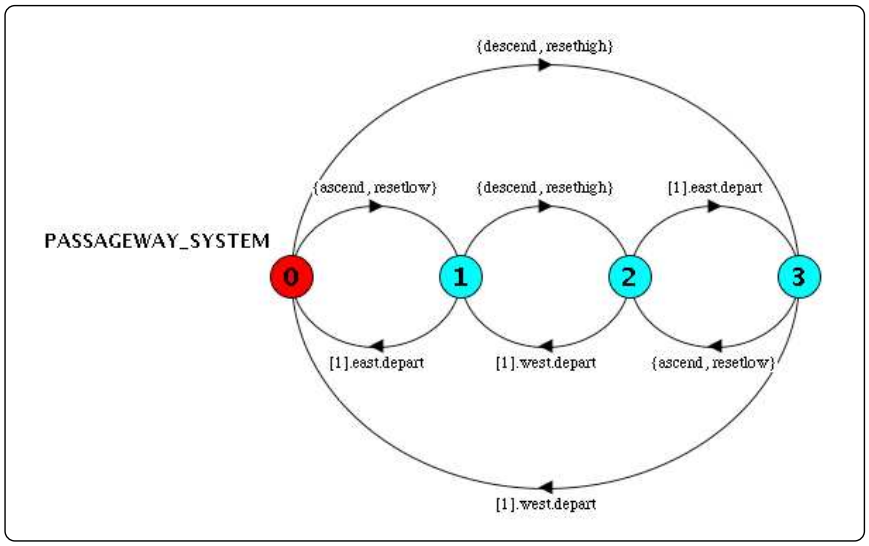

Figure 4.14: Passageway system for NoShips = 1.

in point, consider the PASSAGEWAY_SYSTEM model. Pump and gate processes each have two states. The decision to exclude three pump and two gate processes from the process model automatically reduces the size of the process model by a factor of 2*2*2*2*2 = 32. Still, when NoShips = 3, the unminimized PASSAGEWAY_SYSTEM model has 15*15 = 225 states. The minimized model has only 4 states and, in fact, this doesn't change with increasing numbers of ships.

4.2.5 Viewpoint 2: Composition for Verification of Safety Against Flooding

Viewpoint 2 focuses on a minimal process composition for verification of safety against flooding. Flooding will occur if a gate is opened when there is an imbalance in water levels on either side of the gate. As already explained, safety properties can be defined through the FSP code:

```
property LOWER_PUMPS = ( [j:1..NoShips].east.acquire ->
                         ascend   -> [j].east.depart -> [j].east.acquire ->
                         resetlow -> [j].east.depart -> LOWER_PUMPS ).

property RAISE_PUMPS = ( [j:1..NoShips].west.acquire ->
                         descend   -> [j].west.depart -> [j].west.acquire ->
                         resethigh -> [j].west.depart -> RAISE_PUMPS ).
```

The first property states, for example, that if the canal handles a series of east-bound transit requests then an ascend operation should always be followed by a resetlow action, the latter being

responsible for lowering water levels in the pumps. Now, instead of verifying these properties against the complete process model, we simply check safety against LOCKSET_SYSTEM1, i.e.,

```
||SYSTEM_FLOOD_CHECK1 = ( LOCKSET_SYSTEM1 || LOWER_PUMPS ).
||SYSTEM_FLOOD_CHECK2 = ( LOCKSET_SYSTEM1 || RAISE_PUMPS ).
```

Traces to property violations in the lower and upper pumps are as follows:

```
Trace to property violation in LOWER_PUMPS:
1.east.request
1.west.request
ascend
1.east.depart
1.east.request
1.west.acquire
descend
1.west.depart
1.east.acquire
ascend
```

```
Trace to property violation in RAISE_PUMPS:
1.east.request
1.east.acquire
ascend
1.east.depart
1.east.request
1.west.request
1.east.acquire
resetlow
1.east.depart
1.west.acquire
resethigh
```

Here we see that two ascend operations can occur, but only if they are separated by a descend operation (i.e., a west-bound ship transits the system). In this particular case the safety violation is okay because the west-bound ship will lower the water level, thereby allowing the second ascend operation to proceed safely.

4.2.6 Viewpoint 3: Composition for Verification of Passageway Occupancy

This viewpoint model is designed for verification that the lockset control will not act in a manner inconsistent with physical constraints on the lockset chamber occupancy. Again, the lock occupancy constraint can be represented:

```
LOCK_OCCUPANCY = SPACES[1],
SPACES[i:0..1] = ( when(i>0)
                       [j:1..NoShips].east.acquire -> SPACES[i-1]
                     | [j:1..NoShips].west.acquire -> SPACES[i-1]
                   | when(i<1)
                       [j:1..NoShips].east.depart  -> SPACES[i+1]
                     | [j:1..NoShips].west.depart  -> SPACES[i+1] ).
```

Validation of the lock occupancy property is achieved through its composition with LOCKSET_SYSTEM1, i.e.,

```
||LOCK_OCCUPANCY_CHECK1 = ( LOCKSET_SYSTEM1 || LOCK_OCCUPANCY ).
```

Chapter 5

Behavior Modeling and Validation of Canal-Level Operations

Now that simplified models of lockset behavior are in place, we can move onto the specification, composition, and validation of behavior models for the complete canal operation. This step forward requires several important extensions to the model. First, the traffic demand model will be expanded to include ascend/descend operations through the three locksets; in turn, this change will trigger minor adjustments to lockset-level schedulers. Second, functionality of the scheduler will be extended so that it can properly respond to maintenance/emergency events. Overall control and scheduling of traffic activities in response to an emergency/maintenance will be handled by a canal-level monitor process.

We already know from Chapter 4 that the lockset-level model of behavior is safe and that it satisfies liveliness/progress checks. Safety checks are compositional meaning that if a safety property is satisfied at the lockset level of concern, then it will also be satisfied at the canal level. But progress properties are not compositional. We will see that very subtle features in the lockset-level scheduler (not important to safety or progress) suddenly become critically important at the canal level. Thus, in order to guarantee progress properties, this chapter also contains a fourth revision of the scheduler process. Finally, we will see that because the Pacific, Middle and Atlantic locksets have behavior models that operate independently (i.e., no coupling of actions between models), a key challenge is state explosion. To overcome this problem (or, at least, keep state explosion at bay), strategies of targeted abstraction developed in Chapter 4 will also be applied to the full canal model.

Appendix 3 contains the LTSA source code for the full canal model. Appendix 4 contains extensions of the full canal model to include provision for maintenance and emergency concerns.

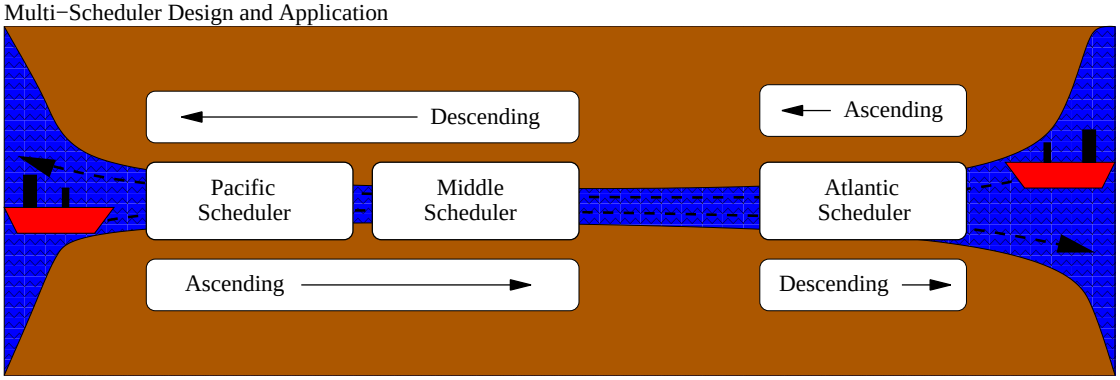

Figure 5.1: System-level view of scheduling/control of ship behaviors. See Appendix 3.

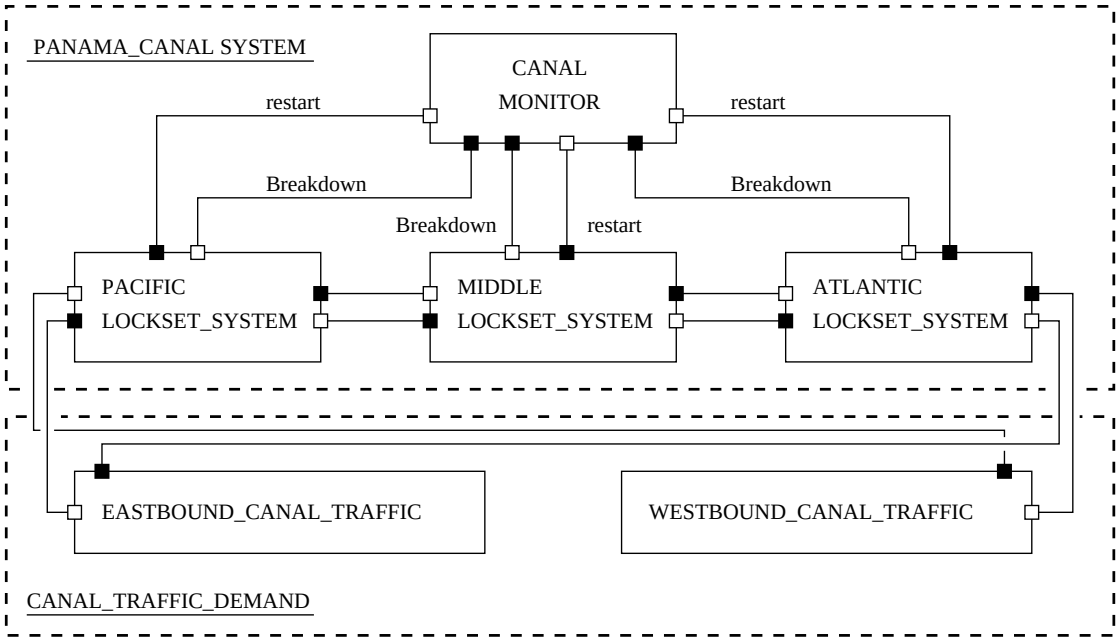

Figure 5.2: Process architecture for full canal model. White dots represent requirements. Black dots represent provisions.

5.1 Process Architecture for the Full Canal Model

Figures 5.1 and 5.2 contain birds-eye views of canal-level transit operations and schematics of the process architecture for the full canal model (white dots represent requirements; black dots represent provisions). The canal traffic demand model will be a composition of `EASTBOUND` and `WESTBOUND` traffic. East-bound ships will ascend the Pacific and Middle locksets, cross lake Gatun (details not shown), and then descend the Atlantic lockset. West-bound ships will ascend and descend the Atlantic and Pacific and Middle locksets respectively.

The Panama Canal System corresponds to a parallel composition of three lockset-level processes (i.e., the Pacific, Middle and Atlantic lockset systems). As already noted, east-bound ships will ascend through the Pacific and Middle locksets. West-bound ships will ascend through the Atlantic lockset. To accommodate the latter, a mirror version of the lockset scheduler will be implemented (i.e., where west-bound ships ascend the lockset).

A canal monitor process will will be responsible for coordinating transit activities during maintenance and emergency events, and for restoring normal operations after the event has past. The canal monitor will communicate to lockset level scheduler processes which, in turn, will synchronize with passageway controller processes. We employ the action set:

```
set Breakdown = { emergency, maintenance }
```

to simplify the description of communication between the lockset and monitor processes.

The two principle design concerns at the canal level are: (1) ensuring maintenance and emergency events are properly handled, and (2) ensuring progress checks at the lockset level propagate up to the canal level.

5.2 Traffic Demand Model

The full canal traffic demand model is a direct extension of lockset traffic demand model. For east-bound traffic, transit requests will be handled at the Pacific lockset in the same order in which they are made. Similarly, for west-bound traffic, transit requests will be handled at the Atlantic lockset in the same order in which they are made. For both east- and west-bound traffic, transit is simply defined as a chain of three lockset traversals.

The LTSA source code is as follows:

```
// ================================================================================
// Create system-level model of east- and west-bound traffic demand.
// ================================================================================

// Create circular queue of east- and west-bound transit requests.

   EASTBOUND_REQUESTS = QUEUE1 [1],
   QUEUE1[i:S] = ( [i].east.request -> QUEUE1 [ i%NoShips + 1] ).
   WESTBOUND_REQUESTS = QUEUE2 [1],
   QUEUE2[i:S] = ( [i].west.request -> QUEUE2 [ i%NoShips + 1] ).

// Create circular queue of east-bound ship traffic ....

   EASTBOUND_CANAL_SHIPS = ( pac.[i:S].east.request ->
                             pac.[i].east.acquire ->
                             pac.[i].east.depart ->
                             mid.[i].east.request ->
                             mid.[i].east.acquire ->
                             mid.[i].east.depart ->
                             atl.[i].east.request ->
                             atl.[i].east.acquire ->
                             atl.[i].east.depart -> EASTBOUND_CANAL_SHIPS ).

 ||EASTBOUND_CANAL_REQUESTS = ( pac:EASTBOUND_REQUESTS ||
                                mid:EASTBOUND_REQUESTS ||
                                atl:EASTBOUND_REQUESTS ).

 ||EASTBOUND_CANAL_TRAFFIC  = ( EASTBOUND_CANAL_SHIPS ||
                                EASTBOUND_CANAL_REQUESTS ).

// Create circular queue of west-bound traffic ....

   WESTBOUND_CANAL_SHIPS = ( atl.[i:S].west.request ->
                             atl.[i].west.acquire ->
                             atl.[i].west.depart ->
                             mid.[i].west.request ->
                             mid.[i].west.acquire ->
                             mid.[i].west.depart ->
                             pac.[i].west.request ->
                             pac.[i].west.acquire ->
                             pac.[i].west.depart  -> WESTBOUND_CANAL_SHIPS ).

 ||WESTBOUND_CANAL_REQUESTS = ( atl:WESTBOUND_REQUESTS ||
                                mid:WESTBOUND_REQUESTS ||
                                pac:WESTBOUND_REQUESTS ).

 ||WESTBOUND_CANAL_TRAFFIC  = ( WESTBOUND_CANAL_SHIPS ||
                                WESTBOUND_CANAL_REQUESTS ).

// Compose models of canal traffic demand and lockset actions.
```

```
minimal ||CANAL_TRAFFIC_DEMAND = ( WESTBOUND_CANAL_TRAFFIC ||
                                   EASTBOUND_CANAL_TRAFFIC ).
```

Points to note:

1. The full canal system will be modeled as a composition of three lockset systems (i.e., `pac:SCHEDULER`, `mid:SCHEDULER` and `atl:SCHEDULER`) plus the appropriate ship control processes. The traffic demand model will be a composition of east- and west-bound traffic demand processes. The Panama Canal system model will be a composition of three lockset-level processes (i.e., `pac:SCHEDULER`, `mid:SCHEDULER` and `atl:SCHEDULER`) plus a canal monitor process. Key steps in the traffic demand model will synchronize with scheduler actions – for example, `pac.[i].east.acquire` indicates that the i-th east-bound ship acquires permission to enter the Pacific lockset.

2. Actions of the form `atl:WESTBOUND_REQUESTS` ensure that requests are handled in the same order that they are made. Notice that this occurs at each of the three lock sets.

3. The lockset-level behavior model assumed continuous traffic demand. In the full canal model, transit operations in one or more directions may be halted in response to lockset maintenance and/or emergency events.

5.3 Composition of Full Canal Model

5.3.1 Preliminary Composition

East and west-bound ascend operations are handled by the scheduler processes `SCHEDULER_EBA` and `SCHEDULER_WBA`. Composition of the canal-level behavior model involves carrying forward only those actions associated with ship transit operations and/or handling of emergency and maintenance operations.

```
// ========================================================================
// Compose and minimize lockset system models. Focus only on ship actions ..
// ========================================================================
```

```
minimal ||PACIFIC_LOCKSET_SYSTEM = (  pac:SCHEDULER_EBA ||
             pac:WESTBOUND_SHIPCONTROL || pac:EASTBOUND_SHIPCONTROL ) @ {
             pac.[S].{east,west}.{request, acquire, depart }}.

minimal ||MIDDLE_LOCKSET_SYSTEM = (  mid:SCHEDULER_EBA ||
             mid:WESTBOUND_SHIPCONTROL || mid:EASTBOUND_SHIPCONTROL ) @ {
             mid.[S].{east,west}.{request, acquire, depart }}.

minimal ||ATLANTIC_LOCKSET_SYSTEM = ( atl:SCHEDULER_WBA ||
             atl:WESTBOUND_SHIPCONTROL || atl:EASTBOUND_SHIPCONTROL ) @ {
             atl.[S].{east,west}.{request, acquire, depart }}.
```

The canal-level lockset system is given by the parallel composition of Pacific, Middle and Atlantic lock sets, i.e.,

```
minimal ||LOCKSET_SYSTEM = (  PACIFIC_LOCKSET_SYSTEM ||
                              MIDDLE_LOCKSET_SYSTEM ||
                              ATLANTIC_LOCKSET_SYSTEM ).
```

The complete model of canal behavior is given by:

```
minimal ||CANAL_BEHAVIOR = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ).
```

5.3.2 Progress Checks

From the previous chapter we already know that once ship makes a request to transit a lockset it will eventually transit the system. This property is guaranteed through design of the lockset-level scheduler. But progress properties are not compositional – just because a sub-system works doesn't mean that progress at the system level will also occur.

The uni-directional traffic model compiles and runs without a problem. However, when the full canal model is composed with a bi-directional traffic model, the result in a progress error. The detailed trace of actions to deadlock is as follows:

```
Bi-Directional Traffic                        Interpretation
--------------------------------------------------------------------------------
Progress Check...
-- States: 93 Transitions: 149 Memory used: 3036K
```

```
Finding trace to cycle...
Finding trace in cycle...
Progress violation for actions:
{atl, mid, pac}[1].{east, west}.{acquire, depart, request}
Trace to terminal set of states:

atl.1.west.request                          -- West-bound ship transits
atl.1.west.acquire                             Atlantic lockset.
atl.1.west.depart

pac.1.east.request                          -- East-bound ship transits
pac.1.east.acquire                             Pacific lockset.
pac.1.east.depart

mid.1.east.request                          -- East-bound ship transits
mid.1.east.acquire                             Middle lockset.
mid.1.east.depart

mid.1.west.request -- West-bound ship requests transit of Middle lockset.
atl.1.east.request -- East-bound ship requests transit of Atlantic lockset.

Cycle in terminal set:
Actions in terminal set:
{}
Progress Check in: 32ms
-------------------------------------------------------------------------------
```

As illustrated in Figure 5.3, this scenario corresponds to the situation where two ships successfully traverse a small number of locksets and then simply deadlock.

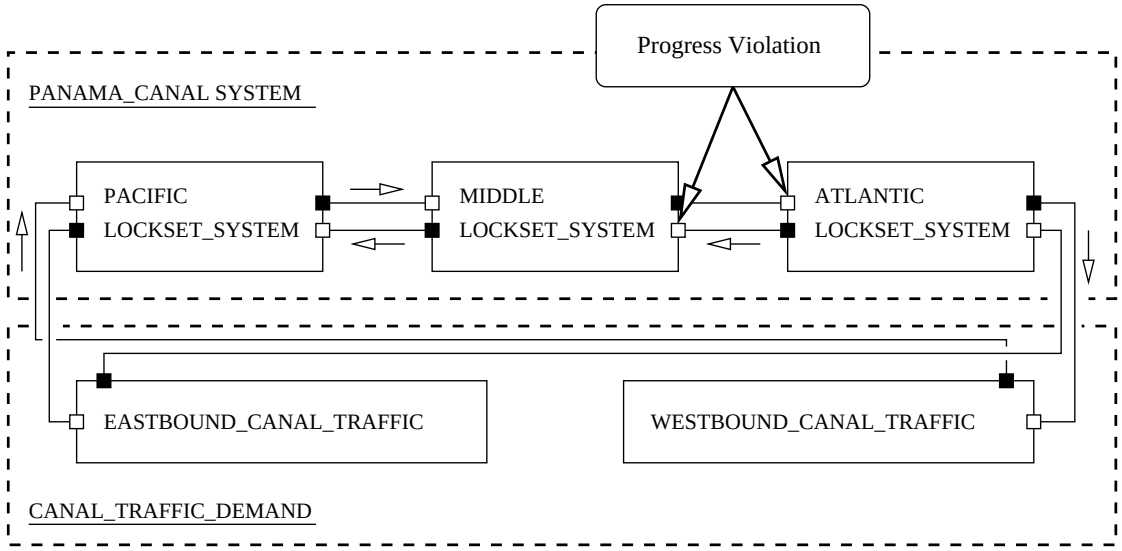

Figure 5.3: Schematic of progress violations in full canal model. White dots represent requirements. Black dots represent provisions.

A detailed examination of the trace reveals the following points:

1. A west-bound ship traverses the Atlantic lockset. Since their no east-bound ships waiting transit the lockset, the current traffic direction is set to `West`.

2. Then, an east-bound ship traverses the Pacific lockset. Again, since their no west-bound ships waiting to immediately transit the lockset, the current traffic direction is set to `East`.

3. Step 2 is repeated for the Middle lockset. The ship departs the lockset with the current traffic direction set to `East`.

4. The deadlock occurs when an east-bound ship attempts to enter the Atlantic lockset and a west-bound ship attempts to enter the Middle lockset.

At the point of deadlock both ships are attempting to acquire the resources of a lockset where there is no oncoming traffic, and, where the current traffic direction (i.e., `td`) opposes the direction of ship traffic. The first obvious question is: *why wasn't this picked up at lockset level?* And second, *if the lockset level model is free of deadlocks, then in what fundamental ways do the lockset- and canal-level models differ?*

The lockset level model is free of deadlocks – that takes care of the first question. So this moves us onto consideration of the second issue. It is important to note that design of our second iteration scheduler is based (implicitly) on continuous streams of traffic. When a ship departs the lockset, the next event is request access to transit the lockset again. However, when a ship departs a lockset at the canal level, it either proceeds to another lockset or loops around to transit the canal system again. From an implementation standpoint, this means that a lockset may need to admit a ship even if it is traveling in a direction that opposes the current traffic direction.

5.3.3 Third Iteration of Lockset-Level Scheduler

To overcome these problems, the third iteration of implementation for the lockset-level scheduler contains additional tests for when east- and west-bound ships may acquire the lockset. The implementation also keeps track of the chamber usage.

The details of source code are as follows:

```
   const Empty    = 0
   const Occupied = 1
   range ChamberUsage = Empty..Occupied // chamber usage ....

   .... code removed ....

// Version 1: East-bound ships ascend through lockset (i.e., pacific/middle locksets).

   SCHEDULER_EBA = SCHEDULER_EBA[0][0][Empty][East][Low],
   SCHEDULER_EBA[we:0..NoShips][ww:0..NoShips]
                [cu:ChamberUsage][td:TrafficDirection][wl:WaterLevel] = (

     // Register requests to transit lock in east- and west-bound directions.

       when ( we <= NoShips ) [S].east.request -> SCHEDULER_EBA[we+1][  ww][cu][td][wl]
     | when ( ww <= NoShips ) [S].west.request -> SCHEDULER_EBA[  we][ww+1][cu][td][wl]

     // East-bound assignments to ascend the lock system.

     | when ( ww == 0 && we >= 1 && cu == Empty && wl == Low )
          [S].east.acquire ->   ascend -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]
     | when ( ww == 0 && we >= 1 && cu == Empty && wl == High )
          [S].east.acquire -> resetlow -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == Low )
          [S].east.acquire ->   ascend -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == High )
          [S].east.acquire -> resetlow -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]

     // East-bound departures from the lock system.

     | when ( ww >= 1 && cu == Occupied && td == East && wl == High )
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][West][High]
     | when ( ww == 0 && cu == Occupied && td == East && wl == High )
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][East][High]

     // West-bound assignments to descend the lock system.

     | when ( ww >= 1 && we == 0 && cu == Empty && wl == High )
          [S].west.acquire ->   descend -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]
     | when ( ww >= 1 && we == 0 && cu == Empty && wl == Low )
          [S].west.acquire -> resethigh -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == High )
          [S].west.acquire ->   descend -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == Low )
          [S].west.acquire -> resethigh -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]

     // West-bound departures from the lock system.

     | when ( we >= 1 && cu == Occupied && td == West && wl == Low )
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][East][Low]
     | when ( we == 0 && cu == Occupied && td == West && wl == Low )
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][West][Low]
).
```

In the previous iteration, permission to acquire the lockset was based exclusively on queues of waiting ships (i.e. `we` and `ww`) and the current traffic direction (i.e., `td` equals either `East` or `West`). The latter requirement is now dropped. In particular, when the west-bound queue is empty an east-bound ship can acquire the lockset resources, without regard to the current traffic direction. The same condition holds for west-bound ships. When ships are waiting to transit the lockset in both the east- and west-bound directions, then the current traffic direction alternates between `East` and `West` (i.e., e-w-e-w...). This is the implementation of fairness.

5.4 Viewpoint-Specific Behavior

The third iteration scheduler has a slightly larger number of states than its second iteration counterpart (e.g., when `NoShips = 1` the second and third generation schedulers have 23 and 33 states respectively). Furthermore, when `NoShips = 1`, the minimized individual locksets and canal behavior processes have 12 and 371 states respectively. `LOCKSET_SYSTEM` has $12^3 = 1{,}728$ states. Increasing `NoShips` to 2 results in individual lockset process models having 64 states. The composed `LOCKSET_SYSTEM` has $64^3 = 262{,}144$ states. A run-time error occurs during the minimization of 2^{18} states in the canal behavior process.

In all of these cases, graphical representations of the state models are too large to be useful. Hence, we simplify our examination of overall canal-level behavior by creating viewpoint-specific behavioral abstractions. Two cases are considered here.

Lockset-Level Behavior. The fragment of code:

```
minimal ||CANAL_VIEWPOINT1 = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ) @ {
                             pac.[S].{east,west}.request, pac.[S].{east,west}.acquire,
                             pac.[S].{east,west}.depart }.
```

creates a minimal full canal model with all actions hidden except `pac.[S].east,west.request`, `pac.[S].east,west.acquire` and `pac.[S].east,west.depart`. In other words, this viewpoint focuses exclusively on canal-level behavior at the Pacific lockset. See Figure 5.4.

Behavior of West-Bound Ships. The fragment of code:

```
minimal ||CANAL_VIEWPOINT3 = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ) @ {
```

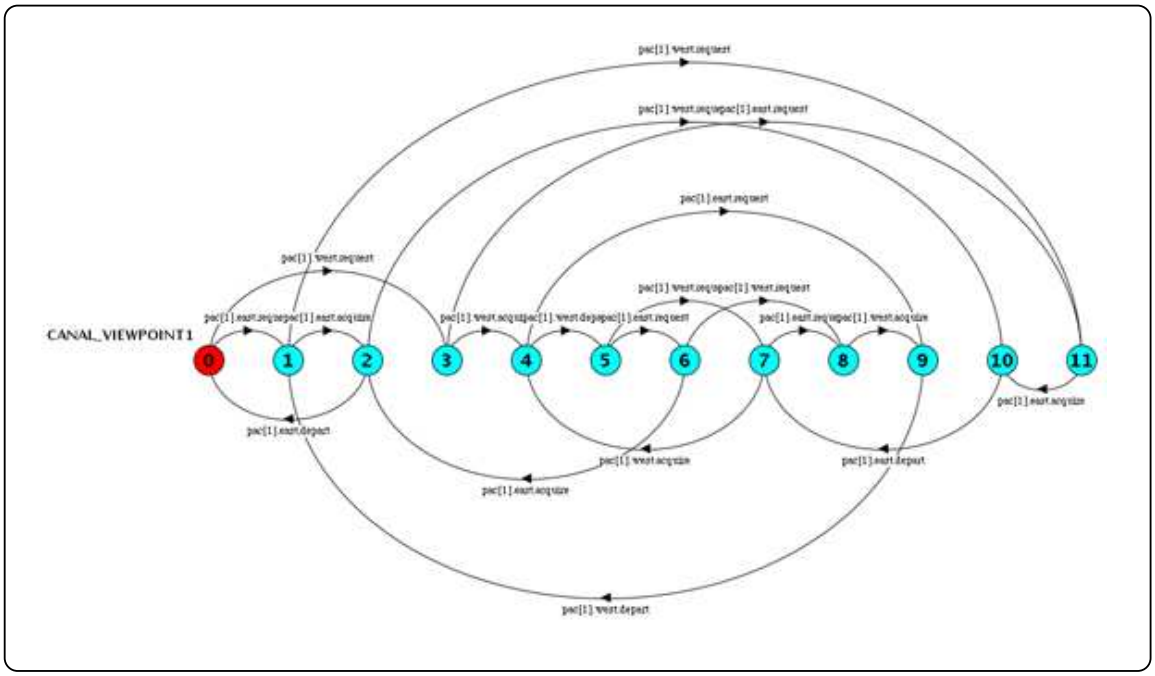

Figure 5.4: Behavior of Pacific Lockset when NoShips = 1.

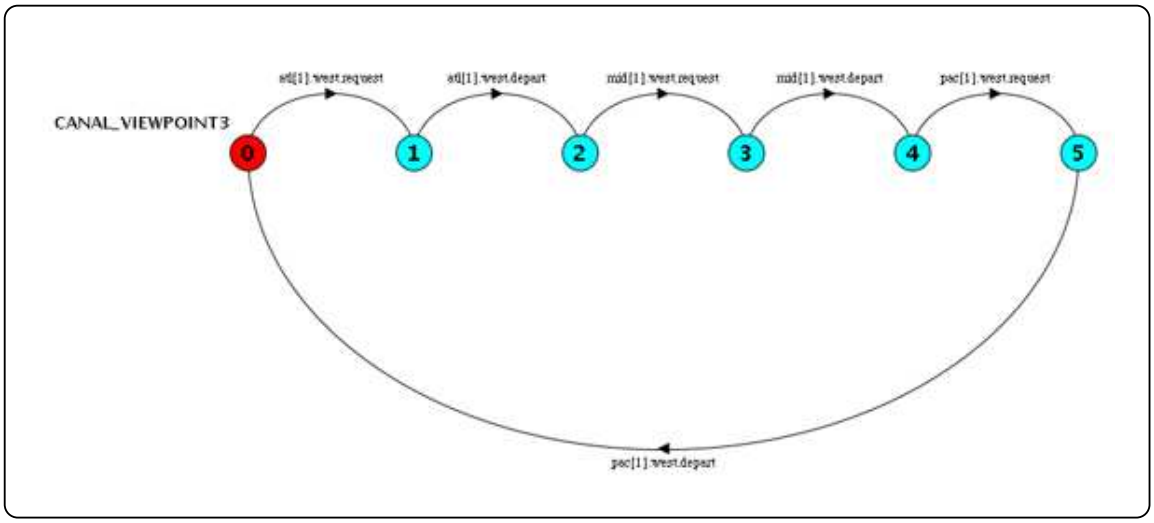

Figure 5.5: Behavior of west-bound ships when NoShips = 1.

```
{pac,mid,atl}.[S].west.request, {pac,mid,atl}.[S].west.depart }.
```

creates a minimal full canal model with all actions hidden except those associated with west-bound traffic. See Figure 5.5.

5.5 Maintenance, Accident and Emergency Concerns

Continual maintenance work on the Panama Canal and its associated facilities is needed to keep it in good working operation. This is typical of many large-scale infrastructure systems. Maintenance activities include dredging channels, scheduling overhauls of locks, and repairing and replacing machinery [1]. As previously mentioned, some maintenance activities on locksets are only possible when it is out-of-operation. Accidents are primarily restricted to collisions in fog and between ships sharing the same lockset (not modeled in this study). Fortunately, the number of accidents is small (e.g., only 10 reported in 2006). The Panama Canal Authority has an emergency plan in place to deal with oil pollution [1, 34].

Modeling Emergency and Maintenance Concerns. We assume that emergency and maintenance events will both originate at the lockset-level (e.g., due to a collision). While emergency events can occur any time, maintenance can be scheduled to occur only when the lock is vacant. In either case, the lockset scheduler will inform the canal monitor of an event. The canal monitor will then mandate appropriate restrictions to transit operations in the east- and west- directions.

Our preliminary implementation assumes that all incoming traffic will be immediately halted. All outgoing traffic will be allowed to continue onwards and clear the system. This simple policy leads to Figure 5.6, a schematic of system-level response to maintenance/emergency events in the Pacific, Middle and Atlantic locksets. Schematics for the Pacific, Middle and Atlantic locksets are shown in columns 1 through 3 respectively. A shaded box indicates that the canal system will be shut down. An empty box indicates that the canal system can continue operating. Now suppose that an accident occurs in the Pacific Lockset, for example. It makes sense to let all outgoing traffic continue their transit to the Atlantic and to halt all incoming traffic. Eventually queues of ships in the permissible directions of operation will clear and the system will wait until the maintenance/emergency is cleared and operation restarts.

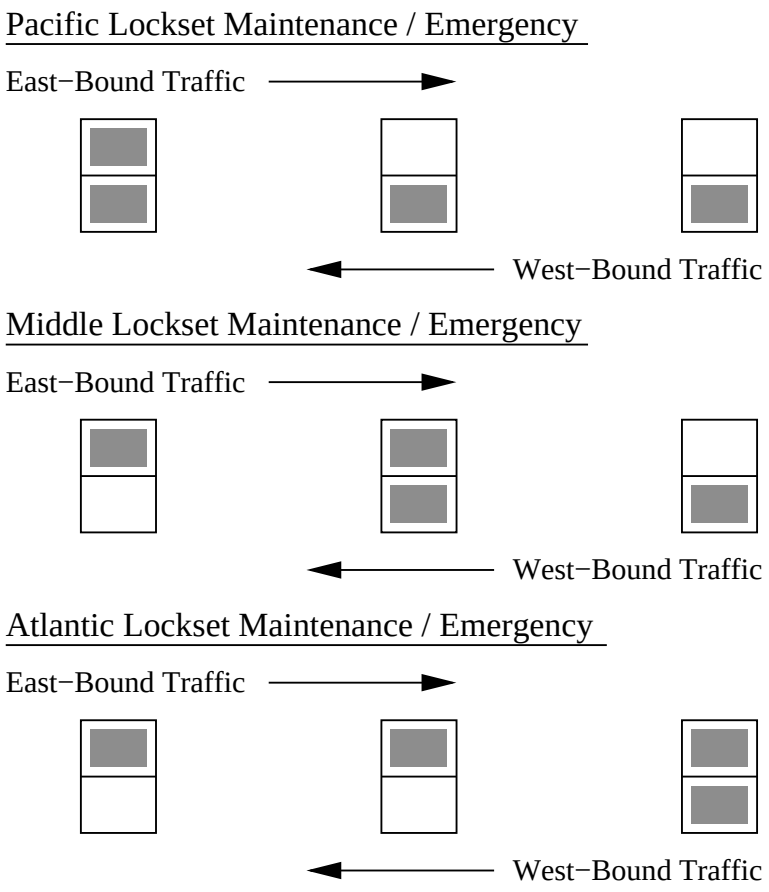

Figure 5.6: Schematic of system-level response to maintenance/emergency events in the Pacific, Middle and Atlantic locksets.

5.5.1 Fourth Iteration of Lockset-Level Scheduler

In the first three iterations of development of the scheduler process, transit operations are assumed to operate continuously (and without interruption). Now, in response to a maintenance/emergency event, the traffic flow will be constrained in directions consistent with the policies of Figure 5.6. The set of constants

```
const Free      = 0
const ForceEast = 1
const ForceWest = 2
range ForcedDir = Free..ForceWest
```

represent the three modes of directional transit operation. To handle these events, a sixth dimension – "Forced Direction" – is added to the scheduler implementation. Its variables (i.e., fd) are activated only after the departure of a ship and while another lockset is halted in either an emergency or maintenance task.

The abbreviated details of FSP code are as follows:

```
// ================================================================================
// Lockset schedulers for east- and west-bound traffic.
// --------------------------------------------------------------------------------
// Variables:
//
// we = number of east-bound ships waiting (0..NoShips).
// ww = number of west-bound ships waiting (0..NoShips).
// td = traffic direction for next ship (East or West).
// cu = chamber usage (Empty, Occupied).
// wl = water level (Low or High).
// fd = forced traffic direction (Free, ForcedEast, ForcedWest ).
// ================================================================================

// Version 1: East-bound ships ascend through lockset (i.e., pacific/middle locksets).

   SCHEDULER_EBA = SCHEDULER_EBA[0][0][Empty][0][Low][Free],
   SCHEDULER_EBA[we:0..NoShips][ww:0..NoShips]
                [cu:ChamberUsage][td:TrafficDirection][wl:WaterLevel][fd:ForcedDir] = (

     // Register requests to transit lock in east- and west-bound directions.

       when ( we <= NoShips ) [S].east.request -> SCHEDULER_EBA[we+1][  ww][cu][td][wl][fd]
     | when ( ww <= NoShips ) [S].west.request -> SCHEDULER_EBA[  we][ww+1][cu][td][wl][fd]

     // East-bound assignments to ascend the lock system.
```

```
     .... details removed ....

     // East-bound departures from the lock system.

     .... details removed ....

     // Traffic forced in East and West directions ....

     | when ( ww >= 1 && cu == Occupied && td == East && wl == High && fd == ForceEast)
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][East][High][fd]
     | when ( ww == 0 && cu == Occupied && td == East && wl == High && fd == ForceWest)
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][West][High][fd]

     // West-bound assignments to descend the lock system.

     .... details removed ....

     // West-bound departures from the lock system.

     .... details removed ....

     // Traffic forced in East and West directions ....

     | when ( we >= 1 && cu == Occupied && td == West && wl == Low && fd == ForceWest)
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][West][Low][fd]
     | when ( we == 0 && cu == Occupied && td == West && wl == Low && fd == ForceEast)
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][East][Low][fd]

     // Initiate single directional traffic

     | when ( cu == Empty && fd == Free)
          forceWest -> SCHEDULER_EBA[we][ww][cu][West][wl][ForceWest]
     | when ( cu == Empty && fd == Free)
          forceEast -> SCHEDULER_EBA[we][ww][cu][East][wl][ForceEast]
     | when ( cu == Empty )
          freeflow -> SCHEDULER_EBA[we][ww][cu][td][wl][Free]

     // Emergency and Maintenance Functions

     | emergency -> emergency_resolved -> restart -> SCHEDULER_EBA[we][ww][cu][td][wl][fd]
     | when ( cu == Empty )
         maintenance_req -> maintenance_complete -> restart ->
         SCHEDULER_EBA[we][ww][Empty][td][wl][fd]
).
```

Points to note:

1. The scheduler implementation allows for an emergency event that can occur anytime. Maintenance requests will be accepted only when the lockset is empty (i.e., `cu == Empty`).

2. By default, permissible directions of traffic flow will be "Free" from constraint. Strategies for ensuring bi-directional traffic flow are embedded into the logic of the scheduler. Moreover, in the absence of maintenance/emergency events, the fourth implementation of the scheduler operates in a manner identical to version 3.

3. Once a maintenance/emergency event occurs, permissible traffic directions at the lockset level will be constrained to be east-bound (i.e., ForceEast) and west-bound (i.e., ForceWest). Issues of fairness no longer apply. Instead, the transit goal is to clear the system of traffic.

4. When NoShips = 1, the minimized lockset system model:

```
minimal ||ATLANTIC_LOCKSET_SYSTEM = ( atl:SCHEDULER_WBA ||
                                      atl:WESTBOUND_SHIPCONTROL ||
                                      atl:EASTBOUND_SHIPCONTROL ) @ {
                                      atl.[S].{east,west}.request,
                                      atl.[S].{east,west}.acquire,
                                      atl.[S].{east,west}.depart,
                                      atl.forceWest,       atl.forceEast, atl.freeflow,
                                      atl.maintenance_req, atl.restart,   atl.emergency }.
```

contains 72 states (up from 33 states in the previous version). Notice, however, that the sets of actions associated with transit operations (e.g., `request`, `acquire` and `depart`) are decoupled from those for maintenance, emergency and forced flow actions. Removing actions associated with the handling of emergency/maintenance events, i.e.,

```
minimal ||ATLANTIC_LOCKSET_SYSTEM = ( atl:SCHEDULER_WBA ||
                                      atl:WESTBOUND_SHIPCONTROL ||
                                      atl:EASTBOUND_SHIPCONTROL ) @ {
                                      atl.[S].{east,west}.request,
                                      atl.[S].{east,west}.acquire,
                                      atl.[S].{east,west}.depart }.
```

reduces the model size to 8 states. Conversely, removing actions associated with the ship transit,

```
minimal ||ATLANTIC_LOCKSET_SYSTEM = ( atl:SCHEDULER_WBA ||
                                      atl:WESTBOUND_SHIPCONTROL ||
                                      atl:EASTBOUND_SHIPCONTROL ) @ {
                                      atl.forceWest,       atl.forceEast, atl.freeflow,
                                      atl.maintenance_req, atl.restart,   atl.emergency }.
```

reduces the model size to only 4 states.

5. Notice that no provision is made for the handling of cascading failures.

5.5.2 Canal-Level Monitor Processes

The canal-level monitor process is responsible for restricting transit operations during and in response to maintenance and emergency events. While the scheduler process has separate provisions for handling maintenance and emergency events, these differences are not discernible to the monitor since in both cases, transit operations in the affected lockset are simply halted.

The essential details of FSP code are as follows:

```
// ======================================================================================
// Canal Monitor Process interacts with locksets and controls traffic only when
// there is an emergency or maintenance underway.
//
// Activities..
//
//   1. Most of the time the Canal Monitor is just waiting (monitoring)
//   2. Notes an emergency has occur in the atl. and issues shutdown
//   3. Restart occurs when the emergency has been cleared (not right away)
//   4. Issues command for maintenance
//
// Variables
//
// oa = operational mode for Atlantic lockset.
// om = operational mode for Middle lockset.
// op = operational mode for Pacific lockset.
//
// Activities are repeated for each lockset monitor...
// ======================================================================================

CANAL_MONITOR = CANAL_MONITOR[0][0][0],
CANAL_MONITOR [oa:LockOperation][om:LockOperation][op:LockOperation] = (

   // Waiting for a maintenance or emergency event ....

   {atl,mid,pac}.[S].{east,west}.acquire -> wait-> CANAL_MONITOR [oa][om][op]

   // Emergency/maintenance event in Atlantic lockset ....

   | when ( oa == BiDirectional )
      { atl.emergency, atl.maintenance_req } ->
        mid.forceWest -> pac.forceWest -> CANAL_MONITOR [Halted][Descend][Descend]
   | when ( oa == Halted )
        atl.restart -> mid.freeflow -> pac.freeflow -> CANAL_MONITOR [0][0][0]

   // Emergency/maintenance event in Middle lockset ....

   | when ( om == BiDirectional )
      { mid.emergency, mid.maintenance_req } ->
```

```
        pac.forceWest  -> atl.forceEast -> CANAL_MONITOR [Descend][Halted][Descend]
   | when ( om == Halted )
        mid.restart -> pac.freeflow -> atl.freeflow -> CANAL_MONITOR [0][0][0]

   // Emergency/maintenance event in Pacific lockset ....

   | when ( op == BiDirectional )
      { pac.emergency, pac.maintenance_req } ->
        mid.forceEast -> atl.forceEast -> CANAL_MONITOR [Descend][Ascend][Halted]
   | when ( op == Halted )
        pac.restart   -> mid.freeflow -> atl.freeflow -> CANAL_MONITOR [0][0][0]

   // Ensure that traffic can never be in only one direction through the lock...

   | pac.forceEast -> pac.freeflow -> CANAL_MONITOR [oa][om][op]
   | atl.forceWest -> atl.freeflow -> CANAL_MONITOR [oa][om][op]
).

// ================================================================
// Control policy for canal monitor forced flow of traffic direction.
// ================================================================

   CANAL_MONITOR_TRAFFIC_CONTROL = (
                forceEast -> freeflow -> CANAL_MONITOR_TRAFFIC_CONTROL
              | forceWest -> freeflow -> CANAL_MONITOR_TRAFFIC_CONTROL ).
```

Points to note:

1. The set of variables

   ```
   const BiDirectional = 0
   const Halted        = 1
   range LockOperation = BiDirectional..Halted // mode of transit operation.
   ```

   keeps track of the lockset transit operations.

2. The `CANAL_MONITOR_TRAFFIC_CONTROL` process implements a control policy for monitor mandated traffic flow that basically says `forceEast` or `forceWest` actions must be followed by a `freeflow` action. Implementation of this policy ensures that the monitor cannot over-ride strategies of fairness built into the scheduler logic.

3. The `CANAL_MONITOR` and `CANAL_MONITOR_TRAFFIC_CONTROL` processes have 54 and 3 states respectively.

5.6 Composition of Full Canal Behavior Model

The full canal behavior behavior corresponds to a canal-level traffic demand model composed with the ensemble of lockset-level processes. As indicated by the fragment of code:

```
// ================================================================
// Complete model of canal behavior
// ================================================================

minimal ||CANAL_MONITOR_SYSTEM   = ( CANAL_MONITOR ||
                                     pac:CANAL_MONITOR_TRAFFIC_CONTROL ||
                                     mid:CANAL_MONITOR_TRAFFIC_CONTROL ||
                                     atl:CANAL_MONITOR_TRAFFIC_CONTROL ).

minimal ||FULL_CANAL_SYSTEM   = ( LOCKSET_SYSTEM || CANAL_MONITOR_SYSTEM ).

minimal ||FULL_CANAL_BEHAVIOR = ( CANAL_TRAFFIC_DEMAND || FULL_CANAL_SYSTEM ).
```

the procedure for process composition is remarkably straight forward. However, due to the overall size and number of processes, the principal challenge is one of avoiding state explosion. Consider, for example, the simplest case: NoShips = 1. The canal-level traffic demand model has 81 states. The minimized canal monitor system (i.e., `CANAL_MONITOR_SYSTEM`) has 21 states. Each of the minimized lockset system models will have 72 states. Given that the lockset systems are completely uncoupled processes, and that the monitor is only loosely coupled to the locksets, the fully composed model will be approximately $72^3 \times 21 = 7{,}838{,}208$ states. Not surprisingly, minimization of the `FULL_CANAL_SYSTEM` process fails due to a stack overflow!

5.6.1 Viewpoint 1: Focus on Transit Operations

As previously demonstrated, problems of state explosion can be avoided through the systematic removal of detail not relevant to a particular design concern or viewpoint.

As a case in point, the fragment of code:

```
minimal ||PACIFIC_LOCKSET_SYSTEM1 = (  pac:SCHEDULER_EBA ||
                            pac:WESTBOUND_SHIPCONTROL || pac:EASTBOUND_SHIPCONTROL ) @{
                            pac.[S].{east,west}.request, pac.[S].{east,west}.acquire,
                            pac.[S].{east,west}.depart }.

minimal ||MIDDLE_LOCKSET_SYSTEM1 = (  mid:SCHEDULER_EBA ||
                            mid:WESTBOUND_SHIPCONTROL || mid:EASTBOUND_SHIPCONTROL ) @ {
                            mid.[S].{east,west}.request, mid.[S].{east,west}.acquire,
                            mid.[S].{east,west}.depart }.

minimal ||ATLANTIC_LOCKSET_SYSTEM1 = ( atl:SCHEDULER_WBA ||
                            atl:WESTBOUND_SHIPCONTROL || atl:EASTBOUND_SHIPCONTROL ) @ {
                            atl.[S].{east,west}.request, atl.[S].{east,west}.acquire,
                            atl.[S].{east,west}.depart }.

minimal ||LOCKSET_SYSTEM1 = ( PACIFIC_LOCKSET_SYSTEM1 || MIDDLE_LOCKSET_SYSTEM1 ||
                             ATLANTIC_LOCKSET_SYSTEM1 ).

minimal ||FULL_CANAL_SYSTEM1    = ( LOCKSET_SYSTEM1 || CANAL_MONITOR_SYSTEM ).
minimal ||FULL_CANAL_VIEWPOINT1 = ( CANAL_TRAFFIC_DEMAND || FULL_CANAL_SYSTEM1 ) @ {
                            pac.[S].{east,west}.request, pac.[S].{east,west}.acquire,
                            pac.[S].{east,west}.depart }.
```

---

systematically assembles a full canal behavior model where details not relevant to normal transit operations are removed. In the `LOCKSET_SYSTEM` processes, for example, only the `request`, `acquire` and `depart` actions are retained. Actions associated with maintenance and emergency events are removed from further consideration. As a result, the scheduler process size is reduced to only 8 states (down from 72 states).

For the purposes of illustration, the `FULL_CANAL_VIEWPOINT1` process model focuses on transit operations at the Pacific lockset alone. The fully composed model contains $81 \times 10752 =$ 870,912 states. The minimized model contains only 8 states. See Figure 5.7.

5.6.2 Viewpoint 2: Focus on Emergency/Maintenance Operations

This viewpoint assumes that the traffic operations function correctly and focuses, instead, on sequences of actions associated with maintenance and emergency events. The fragment of code:

```
minimal ||PACIFIC_LOCKSET_SYSTEM2 = (  pac:SCHEDULER_EBA ||
```

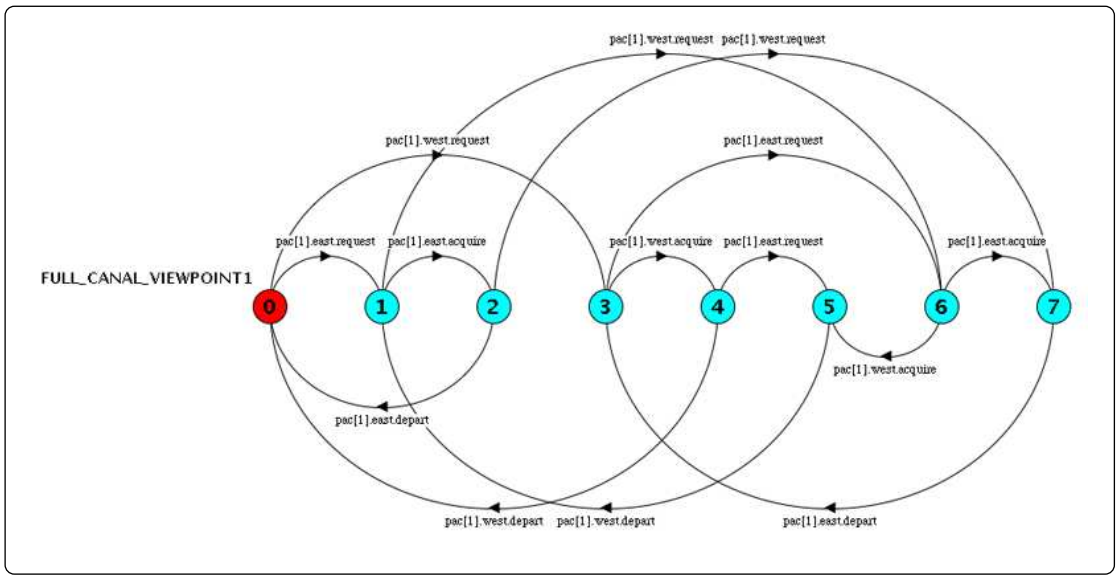

Figure 5.7: Behavior of ships transiting the Pacific lockset.

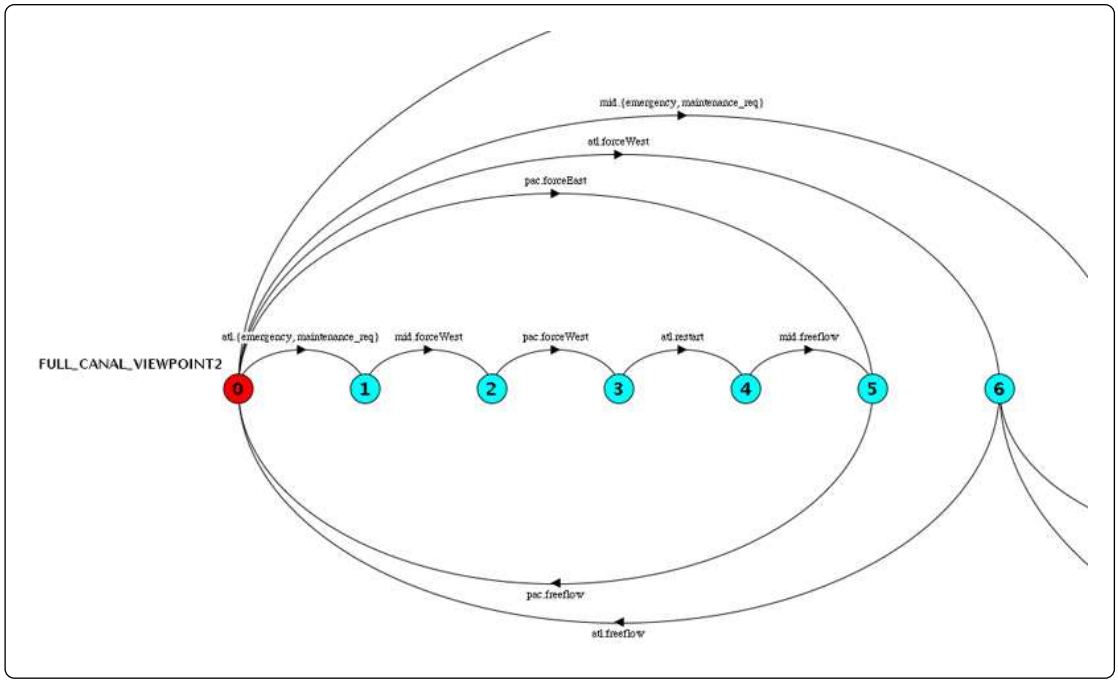

Figure 5.8: Abbreviated view of behavior associated with maintenance/emergency events.

```
                              pac:WESTBOUND_SHIPCONTROL ||
                              pac:EASTBOUND_SHIPCONTROL ) @{
                              pac.forceWest, pac.forceEast, pac.freeflow,
                              pac.maintenance_req, pac.restart, pac.emergency}.

minimal ||MIDDLE_LOCKSET_SYSTEM2 = (  mid:SCHEDULER_EBA ||
                              mid:WESTBOUND_SHIPCONTROL ||
                              mid:EASTBOUND_SHIPCONTROL ) @ {
                              mid.forceWest, mid.forceEast, mid.freeflow,
                              mid.maintenance_req, mid.restart, mid.emergency}.

minimal ||ATLANTIC_LOCKSET_SYSTEM2 = ( atl:SCHEDULER_WBA ||
                              atl:WESTBOUND_SHIPCONTROL ||
                              atl:EASTBOUND_SHIPCONTROL ) @ {
                              atl.forceWest, atl.forceEast, atl.freeflow,
                              atl.maintenance_req, atl.restart, atl.emergency}.

minimal ||LOCKSET_SYSTEM2 = ( PACIFIC_LOCKSET_SYSTEM2 ||
                                 MIDDLE_LOCKSET_SYSTEM2 ||
                               ATLANTIC_LOCKSET_SYSTEM2 ).

minimal ||FULL_CANAL_SYSTEM2    = ( LOCKSET_SYSTEM2 || CANAL_MONITOR_SYSTEM ).
minimal ||FULL_CANAL_VIEWPOINT2 = ( CANAL_TRAFFIC_DEMAND || FULL_CANAL_SYSTEM2 ) @ {
                              {pac,mid,atl}.forceWest,
                              {pac,mid,atl}.forceEast,
                              {pac,mid,atl}.freeflow,
                              {pac,mid,atl}.maintenance_req,
                              {pac,mid,atl}.restart,
                              {pac,mid,atl}.emergency}.
```

---

generates LOCKSET_SYSTEM processes that contain only the actions associated with emergency, maintenance and directed traffic operations. The fully composed model contains $81 \times 19 = 1{,}539$ states. Minimization reduces the number of states to 15. Figure 5.8 contains a partial view of behavior associated with maintenance/emergency operations – it shows, in particular, that a maintenance/emergency event at the Atlantic lockset will be followed by actions directing the Middle and Pacific locksets to process only west-bound traffic.

Chapter 6

Conclusions and Future Work

The conclusions of this work are as follows:

1. The Panama Canal is typical of many large-scale infrastructure systems (e.g., buildings; water, rail, and power networks) in sense that working lifecycles last decades, incremental development occurs in a piecemeal fashion, and development procedures are often less than formal. When engineers do not have models to formally understand behavior, decision making and design tradeoffs occur in the face of incomplete information and uncertainty. This approach to design and operation of the Panama Canal has worked for more than nine decades.

2. This project has been motivated by the observation that as modern canal management systems move toward increased use of automation in their day-to-day operations, a looming problem is the need for formal approaches to behavior modeling and validation of correctness of system functionality. Solutions to this problem are complicated by the large number of concurrent processes defining component- and system-level behavior. Part of this problem can be solved through hierarchal decomposition of processes, with each level in the hierarchy dealing with a specific set of design concerns. Decomposition does not solve the problem completely, however, because naive approaches to the parallel composition of processes quickly become computationally intractable. To overcome the latter problem, we have proposed a mechanism for process abstraction via viewpoint-action-process traceability. The repeated application of abstraction and process minimization leads to behavior models having size that remains almost constant with respect to problem size (e.g., number of ships traversing the canal system).

3. This project has been enabled by finite state process formalisms and the labeled transition analyzer (LTSA). LTSA has its roots in the analysis of computer operating system processes; yet, as we have shown in this report, fundamental properties for behavior and design can be

placed in a canal design setting. When this project began we expected that the definition, assembly and validation and system processes would be straightforward. We certainly did not appreciate the critical role that repeated applications of targeted process abstraction would play in keeping models computationally tractable. And we certainly didn't expect that we'd be coding four versions of the lockset level scheduler!

4. Future work will focus on several important extensions. First, in this study we have investigated the correctness of system functionality with respect to the sequencing of actions. The time needed to complete these actions has been abstracted from consideration. We are currently working on a new behavior model of the canal system represented by networks of timed automata. This extension offers the possibility of formally examining the correctness of canal management operations in terms of delays.

Although we have talked about the need for sensor-enabled control, the lockset model does not explicitly contain sensor processes. A second generation of process models would place sensors at the center of monitoring activities – to detect the arrival of ships, monitor water levels, and ensure locks are restricted to single use operations. An important benefit of this approach is that ships do not need to be modeled as processes. Instead, they are simply viewed as objects that are directed to pass through the canal system. An example of this approach to modeling can be found in Jeff Magee and Jeff Kramer's book [24], in the form of a parcel router (page 295), where packages are sent through a series of splitting chutes and sensors read their destinations and flip doors open and shut to send the parcels to their corresponding bins at the end of the chutes.

Acknowledgment. This work reported in this paper was supported, in part, by a summer grant to the first author from the NSF Research Experiences for Undergraduates (REU) Program.

Bibliography

[1] 2008. Panama Canal Maintenance, Britannical Online Encyclopedia, See http://www.britannica.com/EBchecked/topic/440784/Panama-Canal/40008/Maintenance, Accessed September 2008.

[2] Arnautovic E., and Kaindl H. Aspects for Crosscutting Concerns in System Architectures. In *CSER: Conference on Systems Engineering Research*, University of Southern California, Los Angeles, April 2003.

[3] Austin M.A. *Information-Centric Systems Enginering.* Lecture Notes for ENPM 643/ENSE 623, Institute for Systems Research, University of Maryland, College Park, MD 20742., September 2008.

[4] Austin M.A. (with help from Baras J.). *An Introduction to Information-Centric Systems Engineering.* Tutorial F06, INCOSE, Toulouse, France, June 2004.

[5] Baier C. and Katoen J.P. *Principles of Model Checking.* MIT Press, Cambridge, MA 02142, 2008.

[6] Balaguer A. Weighing the future, a lock at a time: six years after taking control of the Panama Canal, Panamanians face new challenges to keep this vital waterway economically viable and to protect their livelihood. *Americas*, 59, January-February 2007.

[7] Bidding: Firms Eye Billions in Expansion Work At Panama Canal. *ENR: Engineering News-Record*, 258(12), 2007.

[8] Cheung S.C., and Kramer J. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1), January 1999.

[9] Dai, M. D. and Schonfeld, P. Metamodels for Estimating Waterway Delays Through a Series of Queues. *Transportation Research*, 32(1):1–19, 1998.

[10] Dams D., Gerth R., Knaack B., and Kuiper R. Partial-Order Reduction Techniques for Real-Time Model Checking. *Formal Aspects of Computing*, 10(5-6):469–482, 1998.

[11] DeSalvo, J. S. and Lave. L. B.. An Analysis of Towboat Delays. *J. Ttranspn Econ. Policy.*, pages 232–241, 1968.

[12] Devadas S. and Keutzer K. An Automata-Theoretic Approach to Behavioral Equivalence. *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on Computer Aided Design*, pages 30–33, November 1990.

[13] Famous Software Bugs (Accessed May 2007), 2007. See http://infotech.fanshawec.on.ca/ gsantor/Computing/FamousBugs.htm.

[14] Gribar J.C., and Bocanegro J.A. Passage to 2000 (Moderization of the Panama Canal). *Civil Engineering Magazine*, December 1999.

[15] Goguen J. and Grigore Rosu G. Hiding more of hidden algebra. In *FM'99 – Formal Methods*, pages 1704–1719. Springer, 1999.

[16] Jackson D. Dependable Software by Design. *Scientific American*, 294(6), June 2006.

[17] James T. All Locked Up. *Computing and Control Engineering*, 17:16–21, 2006.

[18] Kaisar E., and Austin M.A. Synthesis and Validation of High-Level Behavior Models for Narrow Waterway Management Systems. *Journal of Computing in Civil Engineering, ASCE*, 21(5):373–378, September 2007.

[19] Kaisar E., Austin M.A., and Papadimitriou S. Formal Development and Evaluation of Narrow Passageway System Operations. *European Transport/Transporti Europei*, 34:88–104, December 2006.

[20] Lee E.A. Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report. Technical report, University of California, Berkeley, CA 94720, 2007.

[21] Lowry M.R. The Abstraction/Implementation Model of Problem Reformulation. In *IJCAI 87*, Milan, Italy, August 1987.

[22] Lowry M. and Subramaniam M. Abstraction for analytic verification of concurrent software systems, 1998.

[23] Magee J.L., and Kramer J. *Concurrency: State Models and Java Programs*. John Wiley and Sons, New York, 1999.

[24] Magee J.L., and Kramer J. *Concurrency: State Models and Java Programs (2nd Edition)*. John Wiley and Sons, New York, 2006.

[25] Magee J.L. Kramer J., Uchitel S. Labeled Transition System Analyzer (LTSA) Home Page. See:http://www.doc.ic.ac.uk/ jnm/book/ltsa/LTSA.html. 2004.

[26] Maxing Out: Container Ships. *Economist*, 382, March 3 2007.

[27] Minea M. *Partial Order Reduction for Verification of Timed Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1999.

[28] Moore M. The Bosporus: A Clogged Artery. *The Washington Post*, 16th November 2000.

[29] Nierstrasz O. and Tsichritzis D., editors. *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[30] Olsen R. *Design Patterns in Ruby*. Pearson Education, Boston, MA 02116, 2008.

[31] Osterhout J.K. Scripting: Higher-Level Programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[32] Pagani M. Partial Orders and Verification of Real-Time Systems. In Jonnson B. and Parrow J., editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 327–346, Uppsala, Sweden, 1996.

[33] Panama Canal Web Site: http://www.pancanal.com/eng/index.html. Accessed July 2007.

[34] 2008. Panama Canal Locks, Wikipedia, See http://en.wikipedia.org/wiki/Panama_Canal_Locks, Accessed September 2008.

[35] Reid R.L. Panama Plans to Expand Canal with Larger Locks. *Civil Engineering*, 77(1), Janary 2007.

[36] Sangiovanni-Vincentelli A. Automotive Electronics: Trends and Challenges. In *Presented at Convergence 2000*, Detroit, MI, October 2000.

[37] Sangiovanni-Vincentelli A., McGeer P.C., Saldanha A. Verification of Electronic Systems : A Tutorial. In *Proceedings of the 33rd Desgin Automation Conference*, Las Vegas, 1996.

[38] Sidorova N. Lecture Notes in Process Modeling. 2007. Department of Mathematics and Computer Science, Eindhoven University, Netherlands.

[39] Sutter H. and Larus J. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, 1997.

[40] The Panama Canal's Ultimate Upgrade. *Popular Mechanics*, 184:56–61, March 2007.

[41] Ting, C.J. and Schonfeld, P. Integrated Control For Series of Waterway Locks. *ASCE Journal of Waterway, Port, Coastal, and Ocean Engineering*, 124(4):199–206, July/August 1998.

[42] Uchitel S. *Incremental Elaboration of Scenario-Based Specifications and Behavior Models using Implied Scenarios*. PhD thesis, Imperial College, London, England, 2003.

[43] Uchitel S., Kramer J., and Magee J. Incremental Elaboration of Scenario-Based Specifications and Behavior using Implied Scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, January 2004.

[44] Zhu, l., Schonfeld. P., Kim. Y., Flood, I., and Ting, C.J.,. Queuing Network Analysis for Waterways with Artificial Neural Networks. *Artificial intelligence for Engineering Design, Analysis and Manufacturing*, B:365–275, 1998.

Appendix 1. Detailed Behavior for a Two-Stage Lockset Module

Here is the source code:

```
// ================================================================================
// Appendix1.lts: Behavior model for a two-stage lockset module.
// ---------------------------------------------------------------
//
// -- East- and west-bound traffic demand is modeled arrays of ships organized
//    into a circular queue.
// -- The lock is modeled as a space that can be occupied by at most one ship.
// -- Linkage between east/west traffic direction and ascend/descend
//    actions is established within the passageway control process.
// -- Gate and pump operations are all controled by the ascend and
//    descend passage control processes.
//
// Here we assemble the full model without removal of actions from processes.
//
// Written by: Mark Austin and John Johnson                          May/August 2008
// ================================================================================

   const NoShips = 1
   range S = 1..NoShips  // ship identities

// Simplify notation through sets of actions.

   set    PumpCommands = {   pumpup,  pumpdown }
   set    GateCommands = { opengate, closegate }
   set    ShipCommands = {  acquire, enterlock1, enterlock2, depart }
   set  Ascendcommands = {   ascend, resetlow }
   set Descendcommands = {  descend, resethigh }

// Setup constants that will used by the scheduler.

   const East  = 0
   const West  = 1
   range TrafficDirection = East..West  // traffic direction for next ship.
   const Low   = 0
   const High  = 1
   range WaterLevel = Low..High         // waterlevel in locks...

// ================================================================
// Create model of east- and west-bound traffic demand.
// Note. Ships don't care about internal details of the lock system.
// ================================================================

   SHIP = ( request -> acquire -> depart -> SHIP ).

 ||EASTBOUND_SHIPS = ( [i:S]:(east:SHIP) ).
 ||WESTBOUND_SHIPS = ( [i:S]:(west:SHIP) ).

// Create circular queue of east- and west-bound transit requests.
```

```
   EASTBOUND_REQUESTS = QUEUE1 [1],
   QUEUE1[i:S] = ( [i].east.request -> QUEUE1 [ i%NoShips + 1] ).
   WESTBOUND_REQUESTS = QUEUE2 [1],
   QUEUE2[i:S] = ( [i].west.request -> QUEUE2 [ i%NoShips + 1] ).

 ||EASTBOUND_TRAFFIC = ( EASTBOUND_SHIPS || EASTBOUND_REQUESTS ).
 ||WESTBOUND_TRAFFIC = ( WESTBOUND_SHIPS || WESTBOUND_REQUESTS ).

// Compose model of traffic demand ....

 ||TRAFFIC_DEMAND = ( EASTBOUND_TRAFFIC || WESTBOUND_TRAFFIC ).

// =======================================================
// Simple models for gate and pump processes.
// =======================================================

   GATE = ( opengate -> closegate -> GATE).
   PUMP = (   pumpup ->  pumpdown -> PUMP).

 ||PUMPSYSTEM = (low:PUMP || high:PUMP ).
 ||GATESYSTEM = (low:GATE || middle:GATE || high:GATE).

// ================================================================
// Create east- and west-bound ship control processes
// ================================================================

   SHIPCONTROL = ( request -> acquire -> SHIPCONTROL).

 ||EASTBOUND_SHIPCONTROL = ( [i:S]::east:SHIPCONTROL ).
 ||WESTBOUND_SHIPCONTROL = ( [i:S]::west:SHIPCONTROL ).

// ================================================================
// Lockset-Level Passage Control.
//
//  -- Specify sequences of actions for ships ascending and/or
//     descending through the lockset system.
//  -- Associate ascend/descend operations with east- and
//     west-bound traffic. Here we assume that east-bound traffic
//     ascends the lockset. West-bound traffic decends the system.
//
// ================================================================

// Sequence of actions for "ascend" operation ...
// Relabel lock operations to match east-bound traffic actions.

   PASSAGECONTROL_ASCEND = ( ascend   -> ASCEND
                           | resetlow -> low.pumpdown -> high.pumpdown -> ASCEND ),

   ASCEND = (  low.opengate -> enterlock1 ->    low.closegate ->  low.pumpup ->
            middle.opengate -> enterlock2 -> middle.closegate -> high.pumpup ->
              high.opengate ->  exitlock2 ->   high.closegate ->
                     depart -> PASSAGECONTROL_ASCEND ).

 ||EASTBOUND_PASSAGECONTROL = ( [i:S]::(east:PASSAGECONTROL_ASCEND) )/{
```

```
                forall [i:S] {     ascend/[i].east.ascend },
                forall [i:S] {   resetlow/[i].east.resetlow }}.

// Sequence of actions for "descend" operation ...
// Relabel lock operations to match west-bound traffic actions.

   PASSAGECONTROL_DESCEND = ( descend   -> DESCEND
                            | resethigh -> low.pumpup -> high.pumpup -> DESCEND ),

   DESCEND = ( high.opengate -> enterlock1 ->   high.closegate -> high.pumpdown ->
             middle.opengate -> enterlock2 -> middle.closegate ->  low.pumpdown ->
                low.opengate ->  exitlock2 ->    low.closegate ->
                      depart -> PASSAGECONTROL_DESCEND ).

 ||WESTBOUND_PASSAGECONTROL = ( [i:S]::(west:PASSAGECONTROL_DESCEND) )/{
                forall [i:S] {   descend/[i].west.descend },
                forall [i:S] { resethigh/[i].west.resethigh }}.

// ====================================================================
// Lockset Scheduler that accounts for lock occupancy and number
// of east- and west-bound ships waiting to pass through the lock.
// --------------------------------------------------------------------
// Variables:
//
//    we = number of east-bound ships waiting (0..NoShips).
//    ww = number of west-bound ships waiting (0..NoShips).
//    td = traffic direction for next ship (East or West).
//    wl = water level (Low or High).
// ====================================================================

   SCHEDULER = SCHEDULER[0][0][East][Low],
   SCHEDULER[we:0..NoShips][ww:0..NoShips][td:TrafficDirection][wl:WaterLevel] = (

     // Register requests to transit lock in east- and west-bound directions.

         when ( we <= NoShips ) [S].east.request -> SCHEDULER[we+1][  ww][td][wl]
       | when ( ww <= NoShips ) [S].west.request -> SCHEDULER[  we][ww+1][td][wl]

     // East-bound assignments to ascend the lock system.

       | when ( ww >= 1 && we >= 1 && td == East && wl == Low ) [i:S].east.acquire ->
                ascend   -> [i].east.depart -> SCHEDULER[we-1][ ww][West][High]
       | when ( ww == 0 && we >= 1 && td == East && wl == Low ) [i:S].east.acquire ->
                ascend   -> [i].east.depart -> SCHEDULER[we-1][ ww][East][High]
       | when ( ww >= 1 && we >= 1 && td == East && wl == High ) [i:S].east.acquire ->
                resetlow -> [i].east.depart -> SCHEDULER[we-1][ ww][West][Low]
       | when ( ww == 0 && we >= 1 && td == East && wl == High ) [i:S].east.acquire ->
                resetlow -> [i].east.depart -> SCHEDULER[we-1][ ww][East][Low]

     // West-bound assignments to descend the lock system.

       | when ( ww >= 1 && we >= 1 && td == West && wl == High ) [i:S].west.acquire ->
                descend   -> [i].west.depart -> SCHEDULER[  we][ww-1][East][Low]
       | when ( ww >= 1 && we == 0 && td == West && wl == High ) [i:S].west.acquire ->
                descend   -> [i].west.depart -> SCHEDULER[  we][ww-1][West][Low]
```

```
      | when ( ww >= 1 && we >= 1 && td == West && wl == Low ) [i:S].west.acquire ->
               resethigh -> [i].west.depart -> SCHEDULER[  we][ww-1][East][High]
      | when ( ww >= 1 && we == 0 && td == West && wl == Low ) [i:S].west.acquire ->
               resethigh -> [i].west.depart -> SCHEDULER[  we][ww-1][West][High] ).

// ================================================================
// Compose models for lockset-level behavior...
// ================================================================

// Compose lockset system process...

 ||LOCKSET_SYSTEM = (              SCHEDULER ||
                       WESTBOUND_SHIPCONTROL ||  EASTBOUND_SHIPCONTROL ||
                    WESTBOUND_PASSAGECONTROL || EASTBOUND_PASSAGECONTROL ||
                                 PUMPSYSTEM || GATESYSTEM ).

// Compose model of lockset system behavior ...

 ||LOCKSET_BEHAVIOR = ( LOCKSET_SYSTEM || TRAFFIC_DEMAND ).

// ================================================================================
// Validate that a flood will not occur in the lower locks.
// ================================================================================

// Check pump operation for a stream of east-bound ships ....

   property LOWER_PUMPS = ( [j:1..NoShips].east.acquire ->
                            ascend   -> [j].east.depart -> [j].east.acquire ->
                            resetlow -> [j].east.depart -> LOWER_PUMPS ).

// Check pump operation for a stream of west-bound ships ....

   property RAISE_PUMPS = ( [j:1..NoShips].west.acquire ->
                            descend   -> [j].west.depart -> [j].west.acquire ->
                            resethigh -> [j].west.depart -> RAISE_PUMPS ).

  ||SYSTEM_FLOOD_CHECK1 = ( LOCKSET_SYSTEM || LOWER_PUMPS ).
  ||SYSTEM_FLOOD_CHECK2 = ( LOCKSET_SYSTEM || RAISE_PUMPS ).

// ================================================================================
// Validate that at most only one ship can occupy the lock ....
// ================================================================================

// Model lock as a space that can be occupied by at most one ship.

   LOCK_OCCUPANCY = SPACES[1],
   SPACES[i:0..1] = ( when(i>0)
                          [j:1..NoShips].east.acquire -> SPACES[i-1]
                        | [j:1..NoShips].west.acquire -> SPACES[i-1]
                    | when(i<1)
                          [j:1..NoShips].east.depart  -> SPACES[i+1]
                        | [j:1..NoShips].west.depart  -> SPACES[i+1] ).

  ||LOCK_OCCUPANCY_CHECK1 = ( LOCKSET_SYSTEM || LOCK_OCCUPANCY ).
```

```
// ================================================================================
// Validate progress of ships
// ================================================================================

   progress EASTPASS = { [S].east.depart }
   progress WESTPASS = { [S].west.depart }

// ================================================================================
// End!
```

Appendix 2. Simplified Two-Stage Lockset Module

Here is an abbreviated source code listing:

```
// ================================================================================
// Appendix2.lts: Abstract behavior model for a two-stage lockset module.
//
// Written by: Mark Austin and John Johnson                          September 2008
// ================================================================================

   const NoShips = 1
   range S = 1..NoShips  // ship identities

// Simplify notation through sets of actions.

   set    PumpCommands = {   pumpup,  pumpdown }
   set    GateCommands = { opengate, closegate }
   set    ShipCommands = {  acquire, enterlock1, enterlock2, depart }
   set  Ascendcommands = {   ascend, resetlow }
   set Descendcommands = {  descend, resethigh }

// Setup constants that will used by the scheduler.

   const East  = 0
   const West  = 1
   range TrafficDirection = East..West  // traffic direction for next ship.
   const Low   = 0
   const High  = 1
   range WaterLevel = Low..High         // waterlevel in locks...

// ================================================================================
// Create model of east- and west-bound traffic demand.
// Note: Ships don't care about internal details of the lock system.
// ================================================================================

   .... Same code as in Appendix 1....

// Create circular queue of east- and west-bound transit requests.

   .... Same code as in Appendix 1....

// Compose model of traffic demand ....

 ||TRAFFIC_DEMAND = ( EASTBOUND_TRAFFIC || WESTBOUND_TRAFFIC ).

// ========================================================
// Passageway Control System
// --------------------------------------------------------
//
// Part 1. Simple models for gate and pump processes.
// ========================================================
```

```
   .... Same code as in Appendix 1....

// ========================================================
// Part 2. Lockset-Level Passage Control.
// ========================================================

// Sequence of actions for "ascend" operation ...
// Relabel lock operations to match east-bound traffic actions.

   PASSAGECONTROL_ASCEND = ( ascend   -> ASCEND
                           | resetlow -> low.pumpdown -> high.pumpdown -> ASCEND ),

   .... Same code as in Appendix 1....

 ||EASTBOUND_PASSAGECONTROL = ( [i:S]::(east:PASSAGECONTROL_ASCEND) )/{
               forall [i:S] {     ascend/[i].east.ascend },
               forall [i:S] {   resetlow/[i].east.resetlow }}.

// Sequence of actions for "descend" operation ...
// Relabel lock operations to match west-bound traffic actions.

   PASSAGECONTROL_DESCEND = ( descend   -> DESCEND
                            | resethigh -> low.pumpup -> high.pumpup -> DESCEND ),

   .... Same code as in Appendix 1....

 ||WESTBOUND_PASSAGECONTROL = ( [i:S]::(west:PASSAGECONTROL_DESCEND) )/{
               forall [i:S] {   descend/[i].west.descend },
               forall [i:S] { resethigh/[i].west.resethigh }}.

// ========================================================
// Lockset Control System
// --------------------------------------------------------
//
// Part 1. Create east- and west-bound ship control processes
// ========================================================

   SHIPCONTROL = ( request -> acquire -> depart -> SHIPCONTROL).

 ||EASTBOUND_SHIPCONTROL = ( [i:S]::east:SHIPCONTROL ).
 ||WESTBOUND_SHIPCONTROL = ( [i:S]::west:SHIPCONTROL ).

// ========================================================================
// Part 2. Scheduler for east- and west-bound ships.
// ========================================================================

   SCHEDULER = SCHEDULER[0][0][East][Low],
   SCHEDULER[we:0..NoShips][ww:0..NoShips][td:TrafficDirection][wl:WaterLevel] = (

     // Register requests to transit lock in east- and west-bound directions.

         when ( we <= NoShips ) [S].east.request -> SCHEDULER[we+1][  ww][td][wl]
       | when ( ww <= NoShips ) [S].west.request -> SCHEDULER[  we][ww+1][td][wl]
```

```
      // East-bound assignments to ascend the lock system.

        | when ( ww >= 0 && we >= 1 && td == East && wl == Low )
                 [S].east.acquire ->   ascend -> SCHEDULER[we-1][ ww][East][High]
        | when ( ww >= 0 && we >= 1 && td == East && wl == High )
                 [S].east.acquire -> resetlow -> SCHEDULER[we-1][ ww][East][High]

      // East-bound departures from the lock system.

        | when ( ww >= 1 && td == East && wl == High )
                 [S].east.depart -> SCHEDULER[we][ww][West][High]
        | when ( ww == 0 && td == East && wl == High )
                 [S].east.depart -> SCHEDULER[we][ww][East][High]

      // West-bound assignments to descend the lock system.

        | when ( ww >= 1 && we >= 0 && td == West && wl == High )
                 [S].west.acquire ->   descend -> SCHEDULER[we][ww][West][Low]
        | when ( ww >= 1 && we >= 0 && td == West && wl == Low )
                 [S].west.acquire -> resethigh -> SCHEDULER[we][ww][West][Low]

      // West-bound departures from the lock system.

        | when ( we >= 1 && td == West && wl == Low )
                 [S].west.depart -> SCHEDULER[we][ww-1][East][Low]
        | when ( we == 0 && td == West && wl == Low )
                 [S].west.depart -> SCHEDULER[we][ww-1][West][Low] ).

// ============================================================================
// Compose and abstract views of the lockset system
// ============================================================================

   minimal ||LOCKSET_CONTROL = ( SCHEDULER ||
                                 WESTBOUND_SHIPCONTROL ||
                                 EASTBOUND_SHIPCONTROL ).

   minimal ||PASSAGEWAY_SYSTEM = ( WESTBOUND_PASSAGECONTROL ||
                                   EASTBOUND_PASSAGECONTROL ) @ {
                                   resethigh, resetlow, ascend, descend,
                                   [S].east.depart, [S].west.depart }.

// Lockset system that interacts with the traffic demand model...

   minimal ||LOCKSET_SYSTEM1 = ( LOCKSET_CONTROL ).

// Lockset system that omits details of the pump and gate operations ....

   minimal ||LOCKSET_SYSTEM2 = ( LOCKSET_CONTROL || PASSAGEWAY_SYSTEM ).

// ============================================================================
// Viewpoint 1. Composition of Behavior for Ship Movement.
// ============================================================================

   minimal ||LOCKSET_BEHAVIOR1 = ( LOCKSET_SYSTEM1 || TRAFFIC_DEMAND ) @ {
                                 [S].{east,west}.request,
```

```
                                [S].{east,west}.acquire,
                                [S].{east,west}.depart }.

   minimal ||LOCKSET_BEHAVIOR2 = ( LOCKSET_SYSTEM2 || TRAFFIC_DEMAND ) @ {
                                [S].{east,west}.request,
                                [S].{east,west}.acquire,
                                [S].{east,west}.depart }.

// ================================================================================
// Viewpoint 2: Verification of Safety Against Flooding
// ================================================================================

   property LOWER_PUMPS = ( [j:1..NoShips].east.acquire ->
                            ascend   -> [j].east.depart -> [j].east.acquire ->
                            resetlow -> [j].east.depart -> LOWER_PUMPS ).

// Check pump operation for a stream of west-bound ships ....

   property RAISE_PUMPS = ( [j:1..NoShips].west.acquire ->
                            descend   -> [j].west.depart -> [j].west.acquire ->
                            resethigh -> [j].west.depart -> RAISE_PUMPS ).

  ||SYSTEM_FLOOD_CHECK1 = ( LOCKSET_SYSTEM2 || LOWER_PUMPS ).
  ||SYSTEM_FLOOD_CHECK2 = ( LOCKSET_SYSTEM2 || RAISE_PUMPS ).

// ================================================================================
// Viewpoint 3: Verification of Passageway Occupancy
// ================================================================================

   LOCK_OCCUPANCY = SPACES[1],
   SPACES[i:0..1] = ( when(i>0)
                        [j:1..NoShips].east.acquire -> SPACES[i-1]
                      | [j:1..NoShips].west.acquire -> SPACES[i-1]
                    | when(i<1)
                        [j:1..NoShips].east.depart  -> SPACES[i+1]
                      | [j:1..NoShips].west.depart  -> SPACES[i+1] ).

  ||LOCK_OCCUPANCY_CHECK1 = ( LOCKSET_SYSTEM1 || LOCK_OCCUPANCY ).

// ================================================================================
// Viewpoint 4: Check for Ship Progress
// ================================================================================

   progress EASTPASS = { [S].east.depart }
   progress WESTPASS = { [S].west.depart }

// ================================================================================
// End!
```

Appendix 3. Full Canal Design

Here is the source code:

```
// ============================================================================
// Appendix3.lts: Model full canal as a sequence of locksets.
// ----------------------------------------------------------------------------
//
// From a "full canal" viewpoint:
//
// -- East- and west-bound traffic demand is modeled arrays of ships organized
//    into a circular queue.
// -- We assume that ships pass through locksets on the pacific and atlantic
//    sides of the canal. Ascend/descend operations are as follows:
//
//    East-bound ships: pac.ascend -> mid.ascend  -> atl.descend
//    West-bound ships: atl.ascend -> mid.descend -> pac.descend
//
// -- Lockset models are abstracted to only include request, access and depart
//    actions. All actions internal to the lockset are removed from consideration.
// -- Lockset and traffic demand models are developed separately.
//
// Written by: Mark Austin and John Johnson                  August-September 2008
// ============================================================================

   const NoShips = 1
   range S = 1..NoShips  // ship identities

// Simplify notation through sets of actions.

   set    PumpCommands = {   pumpup,  pumpdown }
   set    GateCommands = { opengate, closegate }
   set    ShipCommands = {  acquire, enterlock1, enterlock2, depart }
   set  Ascendcommands = {   ascend, resetlow }
   set Descendcommands = {  descend, resethigh }

// Setup constants that will used by the schedulers.

   const Empty    = 0
   const Occupied = 1
   range ChamberUsage = Empty..Occupied // chamber usage ....
   const East  = 0
   const West  = 1
   range TrafficDirection = East..West  // traffic direction for next ship.
   const Low   = 0
   const High  = 1
   range WaterLevel = Low..High         // waterlevel in locks...

// ============================================================
// Create east- and west-bound ship control processes
// ============================================================
```

```
   SHIPCONTROL = ( request -> acquire -> depart -> SHIPCONTROL).

 ||EASTBOUND_SHIPCONTROL = ( [i:S]:(east:SHIPCONTROL) ).
 ||WESTBOUND_SHIPCONTROL = ( [i:S]:(west:SHIPCONTROL) ).

// ================================================================================
// Lockset schedulers for east- and west-bound traffic.
// --------------------------------------------------------------------------------
// Variables:
//
//    we = number of east-bound ships waiting (0..NoShips).
//    ww = number of west-bound ships waiting (0..NoShips).
//    td = traffic direction for next ship (East or West).
//    wl = water level (Low or High).
// ================================================================================

// Version 1: East-bound ships ascend through lockset (i.e., pacific/middle locksets).

   SCHEDULER_EBA = SCHEDULER_EBA[0][0][Empty][East][Low],
   SCHEDULER_EBA[we:0..NoShips][ww:0..NoShips]
                [cu:ChamberUsage][td:TrafficDirection][wl:WaterLevel] = (

     // Register requests to transit lock in east- and west-bound directions.

       when ( we <= NoShips ) [S].east.request -> SCHEDULER_EBA[we+1][  ww][cu][td][wl]
     | when ( ww <= NoShips ) [S].west.request -> SCHEDULER_EBA[  we][ww+1][cu][td][wl]

     // East-bound assignments to ascend the lock system.

     | when ( ww == 0 && we >= 1 && cu == Empty && wl == Low )
          [S].east.acquire ->   ascend -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]
     | when ( ww == 0 && we >= 1 && cu == Empty && wl == High )
          [S].east.acquire -> resetlow -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == Low )
          [S].east.acquire ->   ascend -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == High )
          [S].east.acquire -> resetlow -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High]

     // East-bound departures from the lock system.

     | when ( ww >= 1 && cu == Occupied && td == East && wl == High )
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][West][High]
     | when ( ww == 0 && cu == Occupied && td == East && wl == High )
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][East][High]

     // West-bound assignments to descend the lock system.

     | when ( ww >= 1 && we == 0 && cu == Empty && wl == High )
          [S].west.acquire ->   descend -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]
     | when ( ww >= 1 && we == 0 && cu == Empty && wl == Low )
          [S].west.acquire -> resethigh -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == High )
          [S].west.acquire ->   descend -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == Low )
```

```
          [S].west.acquire -> resethigh -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low]

     // West-bound departures from the lock system.

     | when ( we >= 1 && cu == Occupied && td == West && wl == Low )
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][East][Low]
     | when ( we == 0 && cu == Occupied && td == West && wl == Low )
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][West][Low]
).

// Version 2: West-bound ships ascend through lockset (i.e., atlantic lockset).

   SCHEDULER_WBA = SCHEDULER_WBA[0][0][Empty][West][Low],
   SCHEDULER_WBA[we:0..NoShips][ww:0..NoShips]
                [cu:ChamberUsage][td:TrafficDirection][wl:WaterLevel] = (

     // Register requests to transit lock in east- and west-bound directions.

       when ( we <= NoShips ) [S].east.request -> SCHEDULER_WBA[we+1][  ww][cu][td][wl]
     | when ( ww <= NoShips ) [S].west.request -> SCHEDULER_WBA[  we][ww+1][cu][td][wl]

     // West-bound assignments ascend the lock system.

     | when ( ww >= 1 && we == 0 && cu == Empty && wl == Low )
          [S].west.acquire ->   ascend -> SCHEDULER_WBA[we][ww-1][Occupied][West][High]
     | when ( ww >= 1 && we == 0 && cu == Empty && wl == High )
          [S].west.acquire -> resetlow -> SCHEDULER_WBA[we][ww-1][Occupied][West][High]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == Low )
          [S].west.acquire ->   ascend -> SCHEDULER_WBA[we][ww-1][Occupied][West][High]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == High )
          [S].west.acquire -> resetlow -> SCHEDULER_WBA[we][ww-1][Occupied][West][High]

     // West-bound departures from the lock system.

     | when ( we == 0 && cu == Occupied && wl == High && td == West )
          [S].west.depart -> SCHEDULER_WBA[we][ww][Empty][West][High]
     | when ( we >= 1 && cu == Occupied && wl == High && td == West )
          [S].west.depart -> SCHEDULER_WBA[we][ww][Empty][East][High]

     // East-bound assignments descend the lock system.

     | when ( ww == 0 && we >= 1 && cu == Empty && wl == High )
          [S].east.acquire ->   descend -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low]
     | when ( ww == 0 && we >= 1 && cu == Empty && wl == Low )
          [S].east.acquire -> resethigh -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == High )
          [S].east.acquire ->   descend -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == Low )
          [S].east.acquire -> resethigh -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low]

     // East-bound departures from the lock system.

     | when ( ww >= 1 && cu == Occupied && wl == Low && td == East )
            [S].east.depart -> SCHEDULER_WBA[we][ww][Empty][West][Low]
     | when ( ww == 0 && cu == Occupied && wl == Low && td == East )
```

```
            [S].east.depart -> SCHEDULER_WBA[we][ww][Empty][East][Low]
).

// ================================================================================
// Compose and minimize lockset system models. Focus only on ship actions ..
// ================================================================================

   minimal ||PACIFIC_LOCKSET_SYSTEM = (  pac:SCHEDULER_EBA ||
                                pac:WESTBOUND_SHIPCONTROL ||
                                pac:EASTBOUND_SHIPCONTROL ) @ {
                                pac.[S].{east,west}.request,
                                pac.[S].{east,west}.acquire,
                                pac.[S].{east,west}.depart }.

   minimal ||MIDDLE_LOCKSET_SYSTEM = (  mid:SCHEDULER_EBA ||
                                mid:WESTBOUND_SHIPCONTROL ||
                                mid:EASTBOUND_SHIPCONTROL ) @ {
                                mid.[S].{east,west}.request,
                                mid.[S].{east,west}.acquire,
                                mid.[S].{east,west}.depart }.

   minimal ||ATLANTIC_LOCKSET_SYSTEM = ( atl:SCHEDULER_WBA ||
                                atl:WESTBOUND_SHIPCONTROL ||
                                atl:EASTBOUND_SHIPCONTROL ) @ {
                                atl.[S].{east,west}.request,
                                atl.[S].{east,west}.acquire,
                                atl.[S].{east,west}.depart }.

// ================================================================================
// Create system-level model of east- and west-bound traffic demand.
// ================================================================================

// Create circular queue of east- and west-bound transit requests.

   EASTBOUND_REQUESTS = QUEUE1 [1],
   QUEUE1[i:S] = ( [i].east.request -> QUEUE1 [ i%NoShips + 1] ).
   WESTBOUND_REQUESTS = QUEUE2 [1],
   QUEUE2[i:S] = ( [i].west.request -> QUEUE2 [ i%NoShips + 1] ).

// Create circular queue of east-bound ship traffic ....

   EASTBOUND_CANAL_SHIPS = ( pac.[i:S].east.request ->
                             pac.[i].east.acquire ->
                             pac.[i].east.depart ->
                             mid.[i].east.request ->
                             mid.[i].east.acquire ->
                             mid.[i].east.depart ->
                             atl.[i].east.request ->
                             atl.[i].east.acquire ->
                             atl.[i].east.depart -> EASTBOUND_CANAL_SHIPS ).

 ||EASTBOUND_CANAL_REQUESTS = ( pac:EASTBOUND_REQUESTS ||
                                mid:EASTBOUND_REQUESTS ||
                                atl:EASTBOUND_REQUESTS ).
```

```
 ||EASTBOUND_CANAL_TRAFFIC  = ( EASTBOUND_CANAL_SHIPS ||
                                EASTBOUND_CANAL_REQUESTS ).

// Create circular queue of west-bound traffic ....

   WESTBOUND_CANAL_SHIPS = ( atl.[i:S].west.request ->
                             atl.[i].west.acquire ->
                             atl.[i].west.depart ->
                             mid.[i].west.request ->
                             mid.[i].west.acquire ->
                             mid.[i].west.depart ->
                             pac.[i].west.request ->
                             pac.[i].west.acquire ->
                             pac.[i].west.depart  -> WESTBOUND_CANAL_SHIPS ).

 ||WESTBOUND_CANAL_REQUESTS = ( atl:WESTBOUND_REQUESTS ||
                                mid:WESTBOUND_REQUESTS ||
                                pac:WESTBOUND_REQUESTS ).
 ||WESTBOUND_CANAL_TRAFFIC  = ( WESTBOUND_CANAL_SHIPS ||
                                WESTBOUND_CANAL_REQUESTS ).

// Compose models of canal traffic demand and lockset actions.

minimal ||CANAL_TRAFFIC_DEMAND = ( WESTBOUND_CANAL_TRAFFIC || EASTBOUND_CANAL_TRAFFIC ).

minimal ||LOCKSET_SYSTEM = (  PACIFIC_LOCKSET_SYSTEM ||
                              MIDDLE_LOCKSET_SYSTEM ||
                            ATLANTIC_LOCKSET_SYSTEM ).

// ================================================================
// Complete model of canal behavior
// ================================================================

minimal ||CANAL_BEHAVIOR = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ).

// ================================================================
// Canal behavior. Viewpoint 1: Focus on actions at pacific lockset.
// ================================================================

minimal ||CANAL_VIEWPOINT1 = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ) @ {
                            pac.[S].{east,west}.request,
                            pac.[S].{east,west}.acquire,
                            pac.[S].{east,west}.depart }.

// ================================================================
// Canal behavior. Viewpoint 2: Focus on actions at atlantic lockset.
// ================================================================

minimal ||CANAL_VIEWPOINT2 = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ) @ {
                            atl.[S].{east,west}.request,
                            atl.[S].{east,west}.acquire,
                            atl.[S].{east,west}.depart }.

// ================================================================
// Canal behavior. Viewpoint 3: Focus on west-bound traffic alone.
```

```
// ================================================================

minimal ||CANAL_VIEWPOINT3 = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ) @ {
                             {pac,mid,atl}.[S].west.request,
                             {pac,mid,atl}.[S].west.depart }.

// ================================================================
// Canal behavior. Viewpoint 4: Focus on east-bound traffic alone.
// ================================================================

minimal ||CANAL_VIEWPOINT4 = ( CANAL_TRAFFIC_DEMAND || LOCKSET_SYSTEM ) @ {
                             {pac,mid,atl}.[S].east.request,
                             {pac,mid,atl}.[S].east.depart }.

// ================================================================================
// End!
```

Appendix 4. Provision for Emergencies and Maintenance

Provision for emergencies and maintenance requires addition of a CANAL_MONITOR process and minor extensions to the scheduler processes. The abbreviated source code is as follows:

```
// ============================================================================
// Appendix4.lts: In this example we had a system-level canal monitor process for
//                handling of emergency and maintenance operations.
//
// This requires:
//
// 1. Extensions to the lockset-level schedulers.
// 2. Addition of a CONTROL_MONITOR process for system-wide handling of
//    maintenance and emergency events.
//
// Written by: Mark Austin and John Johnson                          October 2008
// ============================================================================

   const NoShips = 1
   range S = 1..NoShips  // ship identities

// Simplify notation through sets of actions.

   set    PumpCommands = {   pumpup,  pumpdown }
   set    GateCommands = { opengate, closegate }
   set    ShipCommands = {  acquire, enterlock1, enterlock2, depart }
   set  Ascendcommands = {   ascend, resetlow }
   set Descendcommands = {  descend, resethigh }

// Setup constants that will used by the schedulers.

   const Free  = 0
   const ForceEast = 1
   const ForceWest = 2
   range ForcedDir = Free..ForceWest
   const East  = 0
   const West  = 1
   range TrafficDirection = East..West  // traffic direction for next ship.
   const Low   = 0
   const High  = 1
   range WaterLevel = Low..High         // waterlevel in locks...
   const BiDirectional = 0
   const Ascend  = 1
   const Descend = 2
   const Halted  = 3
   range LockOperation = BiDirectional..Halted // mode of transit operation.
   const Empty    = 0
   const Occupied = 1
   range ChamberUsage = Empty..Occupied // chamber usage ....

// ============================================================
// Create east- and west-bound ship control processes
```

```
// ================================================================

   SHIPCONTROL = ( request -> acquire -> depart -> SHIPCONTROL).

 ||EASTBOUND_SHIPCONTROL = ([i:S]:(east:SHIPCONTROL)).
 ||WESTBOUND_SHIPCONTROL = ([i:S]:(west:SHIPCONTROL)).

// ================================================================================
// Lockset schedulers for east- and west-bound traffic.
// --------------------------------------------------------------------------------
// Variables:
//
// we = number of east-bound ships waiting (0..NoShips).
// ww = number of west-bound ships waiting (0..NoShips).
// td = traffic direction for next ship (East or West).
// cu = chamber usage (Empty, Occupied).
// wl = water level (Low or High).
// fd = forced traffic direction (Free, ForcedEast, ForcedWest ).
// ================================================================================

// Version 1: East-bound ships ascend through lockset (i.e., pacific/middle locksets).

   SCHEDULER_EBA = SCHEDULER_EBA[0][0][Empty][0][Low][Free],
   SCHEDULER_EBA[we:0..NoShips][ww:0..NoShips]
                [cu:ChamberUsage][td:TrafficDirection][wl:WaterLevel][fd:ForcedDir] = (

     // Register requests to transit lock in east- and west-bound directions.

       when ( we <= NoShips ) [S].east.request -> SCHEDULER_EBA[we+1][  ww][cu][td][wl][fd]
     | when ( ww <= NoShips ) [S].west.request -> SCHEDULER_EBA[  we][ww+1][cu][td][wl][fd]

     // East-bound assignments to ascend the lock system.

     | when ( ww == 0 && we >= 1 && cu == Empty && wl == Low )
          [S].east.acquire ->   ascend -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High][fd]
     | when ( ww == 0 && we >= 1 && cu == Empty && wl == High )
          [S].east.acquire -> resetlow -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High][fd]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == Low )
          [S].east.acquire ->   ascend -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High][fd]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == High )
          [S].east.acquire -> resetlow -> SCHEDULER_EBA[we-1][ ww][Occupied][East][High][fd]

     // East-bound departures from the lock system.

     | when ( ww >= 1 && cu == Occupied && td == East && wl == High )
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][West][High][fd]
     | when ( ww == 0 && cu == Occupied && td == East && wl == High )
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][East][High][fd]

     // Traffic forced in East and West directions ....

     | when ( ww >= 1 && cu == Occupied && td == East && wl == High && fd == ForceEast)
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][East][High][fd]
     | when ( ww == 0 && cu == Occupied && td == East && wl == High && fd == ForceWest)
          [S].east.depart -> SCHEDULER_EBA[we][ww][Empty][West][High][fd]
```

```
     // West-bound assignments to descend the lock system.

     | when ( ww >= 1 && we == 0 && cu == Empty && wl == High )
          [S].west.acquire ->   descend -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low][fd]
     | when ( ww >= 1 && we == 0 && cu == Empty && wl == Low )
          [S].west.acquire -> resethigh -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low][fd]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == High )
          [S].west.acquire ->   descend -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low][fd]
     | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == Low )
          [S].west.acquire -> resethigh -> SCHEDULER_EBA[we][ww-1][Occupied][West][Low][fd]

     // West-bound departures from the lock system.

     | when ( we >= 1 && cu == Occupied && td == West && wl == Low )
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][East][Low][fd]
     | when ( we == 0 && cu == Occupied && td == West && wl == Low )
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][West][Low][fd]

     // Traffic forced in East and West directions ....

     | when ( we >= 1 && cu == Occupied && td == West && wl == Low && fd == ForceWest)
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][West][Low][fd]
     | when ( we == 0 && cu == Occupied && td == West && wl == Low && fd == ForceEast)
          [S].west.depart -> SCHEDULER_EBA[we][ww][Empty][East][Low][fd]

     // Initiate single directional traffic

     | when ( cu == Empty && fd == Free)
  forceWest -> SCHEDULER_EBA[we][ww][cu][West][wl][ForceWest]
     | when ( cu == Empty && fd == Free)
  forceEast -> SCHEDULER_EBA[we][ww][cu][East][wl][ForceEast]
     | when ( cu == Empty )
  freeflow -> SCHEDULER_EBA[we][ww][cu][td][wl][Free]

     // Emergency and Maintenance Functions

     | emergency -> emergency_resolved -> restart -> SCHEDULER_EBA[we][ww][cu][td][wl][fd]
     | when ( cu == Empty )
         maintenance_req -> maintenance_complete -> restart ->
         SCHEDULER_EBA[we][ww][Empty][td][wl][fd]
).

// Version 2: West-bound ships ascend through lockset (i.e., atlantic lockset).

   SCHEDULER_WBA = SCHEDULER_WBA[0][0][Empty][West][Low][Free],
   SCHEDULER_WBA[we:0..NoShips][ww:0..NoShips]
                [cu:ChamberUsage][td:TrafficDirection][wl:WaterLevel][fd:ForcedDir] = (

     // Register requests to transit lock in east- and west-bound directions.

       when ( we <= NoShips ) [S].east.request -> SCHEDULER_WBA[we+1][  ww][cu][td][wl][fd]
     | when ( ww <= NoShips ) [S].west.request -> SCHEDULER_WBA[  we][ww+1][cu][td][wl][fd]

     // West-bound assignments ascend the lock system.
```

```
   | when ( ww >= 1 && we == 0 && cu == Empty && wl == Low )
        [S].west.acquire ->   ascend -> SCHEDULER_WBA[we][ww-1][Occupied][West][High][fd]
   | when ( ww >= 1 && we == 0 && cu == Empty && wl == High )
        [S].west.acquire -> resetlow -> SCHEDULER_WBA[we][ww-1][Occupied][West][High][fd]
   | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == Low )
        [S].west.acquire ->   ascend -> SCHEDULER_WBA[we][ww-1][Occupied][West][High][fd]
   | when ( ww >= 1 && we >= 1 && cu == Empty && td == West && wl == High )
        [S].west.acquire -> resetlow -> SCHEDULER_WBA[we][ww-1][Occupied][West][High][fd]

   // West-bound departures from the lock system.

   | when ( we == 0 && cu == Occupied && wl == High && td == West )
        [S].west.depart -> SCHEDULER_WBA[we][ww][Empty][West][High][fd]
   | when ( we >= 1 && cu == Occupied && wl == High && td == West )
        [S].west.depart -> SCHEDULER_WBA[we][ww][Empty][East][High][fd]

   // Traffic forced in East and West directions ....

   | when ( we >= 1 && cu == Occupied && wl == High && td == West && fd == ForceWest)
        [S].west.depart -> SCHEDULER_WBA[we][ww][Empty][West][High][fd]
   | when ( we == 0 && cu == Occupied && wl == High && td == West && fd == ForceEast)
        [S].west.depart -> SCHEDULER_WBA[we][ww][Empty][East][High][fd]

   // East-bound assignments descend the lock system.

   | when ( ww == 0 && we >= 1 && cu == Empty && wl == High )
        [S].east.acquire ->   descend -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low][fd]
   | when ( ww == 0 && we >= 1 && cu == Empty && wl == Low )
        [S].east.acquire -> resethigh -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low][fd]
   | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == High )
        [S].east.acquire ->   descend -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low][fd]
   | when ( ww >= 1 && we >= 1 && cu == Empty && td == East && wl == Low )
        [S].east.acquire -> resethigh -> SCHEDULER_WBA[we-1][ww][Occupied][East][Low][fd]

   // East-bound departures from the lock system.

   | when ( ww >= 1 && cu == Occupied && wl == Low && td == East )
        [S].east.depart -> SCHEDULER_WBA[we][ww][Empty][West][Low][fd]
   | when ( ww == 0 && cu == Occupied && wl == Low && td == East )
        [S].east.depart -> SCHEDULER_WBA[we][ww][Empty][East][Low][fd]

   // Traffic forced in East and West directions ....

   | when ( ww >= 1 && cu == Occupied && wl == Low && td == East && fd == ForceEast)
        [S].east.depart -> SCHEDULER_WBA[we][ww][Empty][East][Low][fd]
   | when ( ww == 0 && cu == Occupied && wl == Low && td == East && fd == ForceWest)
        [S].east.depart -> SCHEDULER_WBA[we][ww][Empty][West][Low][fd]

   // Initiate single directional traffic

   | when ( cu == Empty && fd == Free)
forceWest -> SCHEDULER_WBA[we][ww][cu][West][wl][ForceWest]
   | when ( cu == Empty && fd == Free)
forceEast -> SCHEDULER_WBA[we][ww][cu][East][wl][ForceEast]
```

```
     | when ( cu == Empty )
  freeflow -> SCHEDULER_WBA[we][ww][cu][td][wl][Free]

     // Emergency and Maintenance Functions

     | emergency -> emergency_resolved -> restart -> SCHEDULER_WBA[we][ww][cu][td][wl][fd]
     | when ( cu == Empty )
         maintenance_req -> maintenance_complete -> restart ->
         SCHEDULER_WBA[we][ww][Empty][td][wl][fd]
).

// ================================================================================
// Canal Monitor Process interacts with locksets and controls traffic only when
// there is an emergency or maintenance underway.
//
// Activities..
//
//   1. Most of the time the Canal Monitor is just waiting (monitoring)
//   2. Notes an emergency has occur in the atl. and issues shutdown
//   3. Restart occurs when the emergency has been cleared (not right away)
//   4. Issues command for maintenance
//
// Variables
//
// oa = operational mode for Atlantic lockset.
// om = operational mode for Middle lockset.
// op = operational mode for Pacific lockset.
//
// Activities are repeated for each lockset monitor...
// ================================================================================

CANAL_MONITOR = CANAL_MONITOR[0][0][0],
CANAL_MONITOR [oa:LockOperation][om:LockOperation][op:LockOperation] = (

   // Waiting for a maintenance or emergency event ....

   {atl,mid,pac}.[S].{east,west}.acquire -> wait-> CANAL_MONITOR [oa][om][op]

   // Emergency/maintenance event in Atlantic lockset ....

   | when ( oa == BiDirectional )
      { atl.emergency, atl.maintenance_req } ->
        mid.forceWest -> pac.forceWest -> CANAL_MONITOR [Halted][Descend][Descend]
   | when ( oa == Halted )
        atl.restart -> mid.freeflow -> pac.freeflow -> CANAL_MONITOR [0][0][0]

   // Emergency/maintenance event in Middle lockset ....

   | when ( om == BiDirectional )
      { mid.emergency, mid.maintenance_req } ->
        pac.forceWest  -> atl.forceEast -> CANAL_MONITOR [Descend][Halted][Descend]
   | when ( om == Halted )
        mid.restart -> pac.freeflow -> atl.freeflow -> CANAL_MONITOR [0][0][0]

   // Emergency/maintenance event in Pacific lockset ....
```

```
   | when ( op == BiDirectional )
      { pac.emergency, pac.maintenance_req } ->
        mid.forceEast -> atl.forceEast -> CANAL_MONITOR [Descend][Ascend][Halted]
   | when ( op == Halted )
        pac.restart   -> mid.freeflow -> atl.freeflow -> CANAL_MONITOR [0][0][0]

   // Ensure that traffic can never be in only one direction through the lock...

   | pac.forceEast -> pac.freeflow -> CANAL_MONITOR [oa][om][op]
   | atl.forceWest -> atl.freeflow -> CANAL_MONITOR [oa][om][op]
).

// ===================================================================
// Control policy for canal monitor forced flow of traffic direction.
// ===================================================================

   CANAL_MONITOR_TRAFFIC_CONTROL = (
                forceEast -> freeflow -> CANAL_MONITOR_TRAFFIC_CONTROL
              | forceWest -> freeflow -> CANAL_MONITOR_TRAFFIC_CONTROL ).

// ============================================================================
// Create system-level model of east- and west-bound traffic demand.
// ============================================================================

// Create circular queue of east- and west-bound transit requests.

   EASTBOUND_REQUESTS = QUEUE1 [1],
   QUEUE1[i:S] = ( [i].east.request -> QUEUE1 [ i%NoShips + 1] ).
   WESTBOUND_REQUESTS = QUEUE2 [1],
   QUEUE2[i:S] = ( [i].west.request -> QUEUE2 [ i%NoShips + 1] ).

// Create circular queue of east-bound ship traffic ....

   EASTBOUND_CANAL_SHIPS = ( pac.[S].east.request ->
                             pac.[S].east.acquire ->
                             pac.[S].east.depart ->
                             mid.[S].east.request ->
                             mid.[S].east.acquire ->
                             mid.[S].east.depart ->
                             atl.[S].east.request ->
                             atl.[S].east.acquire ->
                             atl.[S].east.depart -> EASTBOUND_CANAL_SHIPS ).

 ||EASTBOUND_CANAL_REQUESTS = ( pac:EASTBOUND_REQUESTS ).
 ||EASTBOUND_CANAL_TRAFFIC  = ( EASTBOUND_CANAL_SHIPS ||
                                EASTBOUND_CANAL_REQUESTS ).

// Create circular queue of west-bound traffic ....

   WESTBOUND_CANAL_SHIPS = ( atl.[S].west.request ->
                             atl.[S].west.acquire ->
                             atl.[S].west.depart ->
                             mid.[S].west.request ->
                             mid.[S].west.acquire ->
```

```
                              mid.[S].west.depart ->
                              pac.[S].west.request ->
                              pac.[S].west.acquire ->
                              pac.[S].west.depart  -> WESTBOUND_CANAL_SHIPS ).

 ||WESTBOUND_CANAL_REQUESTS = ( atl:WESTBOUND_REQUESTS ).
 ||WESTBOUND_CANAL_TRAFFIC  = ( WESTBOUND_CANAL_SHIPS ||
                                WESTBOUND_CANAL_REQUESTS ).

// ================================================================
// Compose models of canal traffic demand and lockset actions.
// ================================================================

 ||CANAL_TRAFFIC_DEMAND = ( WESTBOUND_CANAL_TRAFFIC ||
                            EASTBOUND_CANAL_TRAFFIC ).

// ================================================================
// Compose full model of canal behavior
// ================================================================

   minimal ||PACIFIC_LOCKSET_SYSTEM = (  pac:SCHEDULER_EBA ||
                                pac:WESTBOUND_SHIPCONTROL ||
                                pac:EASTBOUND_SHIPCONTROL ) @{
                                pac.[S].{east,west}.request,
                                pac.[S].{east,west}.acquire,
                                pac.[S].{east,west}.depart,
                                pac.forceWest,
                                pac.forceEast,
                                pac.freeflow,
                                pac.maintenance_req,
                                pac.restart,
                                pac.emergency}.

   minimal ||MIDDLE_LOCKSET_SYSTEM = (  mid:SCHEDULER_EBA ||
                                mid:WESTBOUND_SHIPCONTROL ||
                                mid:EASTBOUND_SHIPCONTROL ) @ {
                                mid.[S].{east,west}.request,
                                mid.[S].{east,west}.acquire,
                                mid.[S].{east,west}.depart,
                                mid.forceWest,
                                mid.forceEast,
                                mid.freeflow,
                                mid.maintenance_req,
                                mid.restart,
                                mid.emergency}.

   minimal ||ATLANTIC_LOCKSET_SYSTEM = ( atl:SCHEDULER_WBA ||
                                atl:WESTBOUND_SHIPCONTROL ||
                                atl:EASTBOUND_SHIPCONTROL ) @ {
                                atl.[S].{east,west}.request,
                                atl.[S].{east,west}.acquire,
                                atl.[S].{east,west}.depart,
                                atl.forceWest,
                                atl.forceEast,
                                atl.freeflow,
```

```
                              atl.maintenance_req,
                              atl.restart,
                              atl.emergency}.

   minimal ||LOCKSET_SYSTEM = ( PACIFIC_LOCKSET_SYSTEM ||
                                MIDDLE_LOCKSET_SYSTEM ||
                              ATLANTIC_LOCKSET_SYSTEM ).

   minimal ||CANAL_MONITOR_SYSTEM   = ( CANAL_MONITOR ||
                                        pac:CANAL_MONITOR_TRAFFIC_CONTROL ||
                                        mid:CANAL_MONITOR_TRAFFIC_CONTROL ||
                                        atl:CANAL_MONITOR_TRAFFIC_CONTROL ).

   minimal ||FULL_CANAL_SYSTEM   = ( LOCKSET_SYSTEM || CANAL_MONITOR_SYSTEM ).
   minimal ||FULL_CANAL_BEHAVIOR = ( CANAL_TRAFFIC_DEMAND || FULL_CANAL_SYSTEM ).

// ================================================================
// Canal behavior. Viewpoint 1: Focus on actions at pacific lockset.
// ================================================================

   minimal ||PACIFIC_LOCKSET_SYSTEM1 = (  pac:SCHEDULER_EBA ||
                              pac:WESTBOUND_SHIPCONTROL ||
                              pac:EASTBOUND_SHIPCONTROL ) @{
                              pac.[S].{east,west}.request,
                              pac.[S].{east,west}.acquire,
                              pac.[S].{east,west}.depart }.

   minimal ||MIDDLE_LOCKSET_SYSTEM1 = (  mid:SCHEDULER_EBA ||
                              mid:WESTBOUND_SHIPCONTROL ||
                              mid:EASTBOUND_SHIPCONTROL ) @ {
                              mid.[S].{east,west}.request,
                              mid.[S].{east,west}.acquire,
                              mid.[S].{east,west}.depart }.

   minimal ||ATLANTIC_LOCKSET_SYSTEM1 = ( atl:SCHEDULER_WBA ||
                              atl:WESTBOUND_SHIPCONTROL ||
                              atl:EASTBOUND_SHIPCONTROL ) @ {
                              atl.[S].{east,west}.request,
                              atl.[S].{east,west}.acquire,
                              atl.[S].{east,west}.depart }.

   minimal ||LOCKSET_SYSTEM1 = ( PACIFIC_LOCKSET_SYSTEM1 ||
                                 MIDDLE_LOCKSET_SYSTEM1 ||
                               ATLANTIC_LOCKSET_SYSTEM1 ).

   minimal ||FULL_CANAL_SYSTEM1    = ( LOCKSET_SYSTEM1 || CANAL_MONITOR_SYSTEM ).
   minimal ||FULL_CANAL_VIEWPOINT1 = ( CANAL_TRAFFIC_DEMAND || FULL_CANAL_SYSTEM1 ) @ {
                              pac.[S].{east,west}.request,
                              pac.[S].{east,west}.acquire,
                              pac.[S].{east,west}.depart }.

// ================================================================
// Canal behavior. Viewpoint 2: Focus on emergency/maintenance events.
// ================================================================
```

```
   minimal ||PACIFIC_LOCKSET_SYSTEM2 = (  pac:SCHEDULER_EBA ||
                                pac:WESTBOUND_SHIPCONTROL ||
                                pac:EASTBOUND_SHIPCONTROL ) @{
                                pac.forceWest, pac.forceEast, pac.freeflow,
                                pac.maintenance_req, pac.restart, pac.emergency}.

   minimal ||MIDDLE_LOCKSET_SYSTEM2 = (  mid:SCHEDULER_EBA ||
                                mid:WESTBOUND_SHIPCONTROL ||
                                mid:EASTBOUND_SHIPCONTROL ) @ {
                                mid.forceWest, mid.forceEast, mid.freeflow,
                                mid.maintenance_req, mid.restart, mid.emergency}.

   minimal ||ATLANTIC_LOCKSET_SYSTEM2 = ( atl:SCHEDULER_WBA ||
                                atl:WESTBOUND_SHIPCONTROL ||
                                atl:EASTBOUND_SHIPCONTROL ) @ {
                                atl.forceWest, atl.forceEast, atl.freeflow,
                                atl.maintenance_req, atl.restart, atl.emergency}.

   minimal ||LOCKSET_SYSTEM2 = ( PACIFIC_LOCKSET_SYSTEM2 ||
                                   MIDDLE_LOCKSET_SYSTEM2 ||
                                  ATLANTIC_LOCKSET_SYSTEM2 ).

   minimal ||FULL_CANAL_SYSTEM2    = ( LOCKSET_SYSTEM2 || CANAL_MONITOR_SYSTEM ).
   minimal ||FULL_CANAL_VIEWPOINT2 = ( CANAL_TRAFFIC_DEMAND || FULL_CANAL_SYSTEM2 ) @ {
                                {pac,mid,atl}.forceWest,
                                {pac,mid,atl}.forceEast,
                                {pac,mid,atl}.freeflow,
                                {pac,mid,atl}.maintenance_req,
                                {pac,mid,atl}.restart,
                                {pac,mid,atl}.emergency}.

// ============================================================================
// End!
```

Optional Passageway Controller Processes. Given that emergency and maintenance events are assumed to originate at the lockset, then it makes sense that these events will be first registered by the passageway controller processes. Further, the timing of these events will be completely random. These criteria can be inserted into the passageway controller process as follows:

```
// Sequence of actions for "ascend" operation ...

   EMERGENCY = ( emergency -> shutdown -> restart -> EMERGENCY ).

   PASSAGECONTROL_ASCEND = ( ascend   -> ASCEND
                           | resetlow -> low.pumpdown -> CHECK_1
   | emergency -> restart -> PASSAGECONTROL_ASCEND ),
```

```
CHECK_1 = ( high.pumpdown -> ASCEND
          | emergency -> restart -> high.pumpdown -> ASCEND),

 ASCEND = ( low.opengate -> enterlock1 -> CHECK_2),

CHECK_2 = ( low.closegate -> STAGE_1
          | emergency -> restart -> low.closegate -> STAGE_1),

STAGE_1 = ( low.pumpup -> STAGE_1B
          | emergency  -> restart ->
            low.pumpup -> STAGE_1B),

STAGE_1B = ( middle.opengate -> enterlock2 -> STAGE_2
          | emergency -> restart ->
            middle.opengate -> enterlock2 -> STAGE_2),

STAGE_2 = ( middle.closegate -> STAGE_2B
          | emergency -> restart ->
            middle.closegate -> STAGE_2B),

STAGE_2B = ( high.pumpup -> STAGE_3
          | emergency   -> restart ->
            high.pumpup -> STAGE_3),

STAGE_3 = ( high.opengate -> exitlock2 -> STAGE_3B
          | emergency     -> restart ->
            high.opengate -> exitlock2 -> STAGE_3B),

STAGE_3B = ( high.closegate -> depart -> PASSAGECONTROL_ASCEND
          | emergency -> restart ->
            high.closegate -> depart -> PASSAGECONTROL_ASCEND).
```

The passageway controller will operate the ship movements as a shared resource through:

```
||PASSAGECONTROL_EBA = ( [i:S]::(east:PASSAGECONTROL_ASCEND || east:EMERGENCY ) )/{
          forall [i:S] {     ascend/[i].east.ascend },
          forall [i:S] {   resetlow/[i].east.resetlow },
          forall [i:S] {  emergency/[i].east.emergency},
          forall [i:S] {   shutdown/[i].east.shutdown },
          forall [i:S] {    restart/[i].east.restart  }}.
```

PASSAGECONTROL_DESCEND can be extended in a similar way.