systems

MDPI

Article

Compositional Approach to Distributed System Behavior Modeling and Formal Validation of Infrastructure Operations with Finite State Automata: Application to Viewpoint-Driven Verification of Functionality in Waterways

Mark A. Austin [1,*,†] and John Johnson [2,†]

[1] Department of Civil and Environmental Engineering, and Institute for Systems Research, University of Maryland, College Park, MD 20742, USA
[2] Department of Civil and Environmental Engineering, University of Maryland, College Park, MD 20742, USA; johnj620@gmail.com
* Correspondence: austin@isr.umd.edu; Tel./Fax: +1-301-405-6627
† These authors contributed equally to this work.

Received: 16 October 2017; Accepted: 9 January 2018; Published: 12 January 2018

Abstract: Now that modern infrastructure systems are moving toward an increased use of automation in their day-to-day operations, there is an emerging need for new approaches to the formal analysis and validation of system functionality with respect to correctness of operations. This paper describes a compositional approach to the multi-level behavior modeling and formal validation of large-scale distributed system operations with hierarchies and networks of finite state automata. To avoid the well-known state explosion problem, we develop a new procedure for viewpoint-action-process traceability, thereby allowing parts of a behavior model not relevant to a specific decision to be removed from consideration. Key features of the methodology are illustrated through the development of behavior models and validation procedures for polite conversation between two individuals, and lockset- and system-level concerns for ships traversing a large-scale waterway system.

Keywords: model-based systems engineering; infrastructure; distributed system; behavior model; formal verification; automation; modeling; waterways operations

1. Introduction

1.1. Problem Statement

One of the important outcomes of remarkable advances in computing and communications over the past few decades is the way in which they have opened doors to the replacement of aging infrastructure with new types of infrastructure comprising physical networks connected to cyber components (data, information, software) for decision making [1]. In order for these so-called "next-generation infrastructure systems" to serve as enablers of long-term economic growth, they will be required to provide superior levels of performance, new forms of functionality and agility, and good economics over long time horizons. While end-users applaud the benefits that these technological advances afford, systems engineers are faced with a multitude of new design challenges that can be traced to the presence of heterogeneous content (multiple disciplines, and types of data and information), network structures that are spatial, multi-layer, interwoven and dynamic, behaviors and

control that are distributed and concurrent, and interdependencies among coupled subsystems that are not always well understood.

In a decentralized system structure no decision maker knows all of the information known to all of the other decision makers, yet as a group, they must cooperate to achieve system-wide objectives. Communication and information exchange are important to the decision makers because communication establishes common knowledge among the decision makers which, in turn, enhances the ability of decision makers to make decisions appropriate to their understanding, or situational awareness, of the system state, and its goals and objectives. While each of the participating disciplines may have a preference toward operating their domain as independently as possible from the other disciplines, achieving target levels of performance and correctness of functionality nearly always requires that disciplines coordinate activities at key points in the system operation. And even if the resulting cross-domain relationships are only weakly linked, they are nonetheless, still linked. When part of a system fails, there exists a possibility that the failure will cascade across interdisciplinary boundaries, thus making connected systems more vulnerable to various kinds of disturbances than their independent counterparts [2]. To complicate matters, the introduction of automation into a system's operation expands the range of design concerns that need to be addressed. For example, a new fundamental question is: How do we know that an automated management system will always do the right thing? A second question is: If part of the system fails unexpectedly, what assurances do we have that the system will handle and recover from disruptions in a manner that is both sensible and timely?

Established approaches to infrastructure systems development are concerned with the design of large-scale physical networks to support flows of people, goods, energy and potable water. To keep the complexities of development in check, design procedures strive for independence in subsystem-level operations. Analysis procedures favor estimation of performance and cost over correctness of functionality. As a case in point, waterway systems provide cost effective transport of bulk and containerized goods to support global trade. Vexing problems that need to be managed include: (1) Traffic demands on canal systems that have far exceeded initial expectations; (2) Increases in the prevalence and severity of delays caused by aging infrastructure; (3) Locks that are too small for modern ships and barges; and (4) Competition from alternatives such as intermodal freight transportation [3,4]. Sophisticated approaches to performance analysis [5,6] are justified by the adverse economics of poor system throughput. Looking forward, it is now evident that in order for waterway systems to remain economically competitive well into the 21st Century, modernization efforts will need to improve traffic flows through replacement of aging infrastructure with systems that make increased use of automation to: (1) Expand the range of environmental conditions within which a system can safely operate; (2) Support and enhance human performance; and (3) Improve the ability of a system to quickly recover from unexpected disruptions [7,8]. These new requirements make waterway systems design a lot more difficult than it used to be.

1.2. Mechanisms for Systems Development

Experience [9–12] tells us solutions that overcome these barriers are likely to employ a combination of the following mechanisms:

1. Formal Models. To help prevent serious flaws in design and operation, design representations and validation and verification procedures need to be based on formal languages having precise semantics, logic and reasoning.
2. Abstraction. Abstraction mechanisms eliminate details that are of no importance when evaluating system functionality, system performance, and/or checking that a design satisfies a particular property.
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 subsystems and components. We seek, in particular, methods that allow for the systematic assembly of behavior models for complex systems from behavior models for simpler systems and components.

Simplified views of the traditional and desired way of dealing with systems design and management are shown in Figure 1. Both approaches to system development begin with an informal specification (i.e., what we expect the system to do). In the traditional pathway of development, we say the system is correct based upon sequences of actions and task completion. Engineering analysis and simulation play a central role in verifying system functionality and attainable levels of performance. By making logic and formal approaches to process modeling the main tool for system construction, we hope to be able to say that a system operation will be correct by construction (i.e., the system does exactly what it is supposed to do) based on sequences of truth during the system operation.

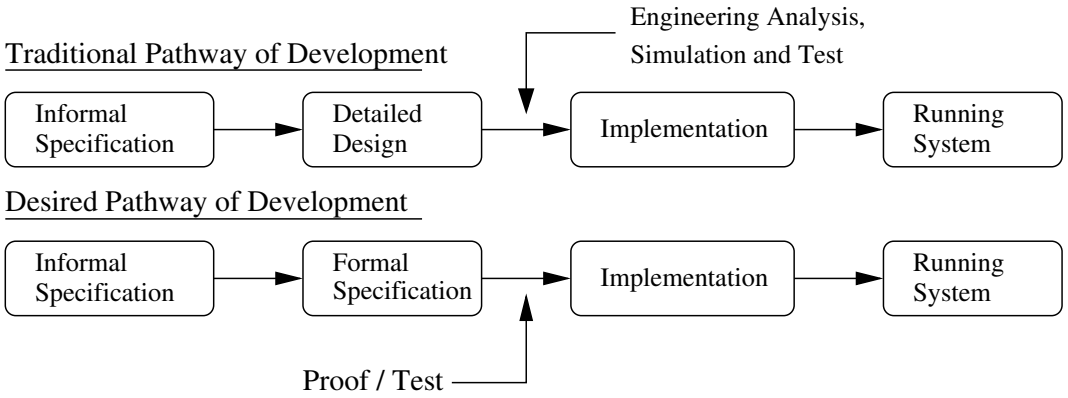

Figure 1. Traditional and desired pathways of system development.

1.3. Scope and Objectives

Our first steps [7,8] toward understanding these issues focused on the use of finite state processes for the top-down synthesis and analysis of behavior models for waterway management systems, and development of a methodology for the incremental transformation of informal operations concepts into lockset-level behavior models that are formal enough for automated validation. The contributions of this article are as follows:

1. We investigate the behavior modeling problem from top-down (decomposition) and bottom-up (composition) perspectives, and system operations that are of sufficient size that naive approaches to composition will fail because the composed models are beyond the capabilities of modern desktop systems.
2. We propose a new, but simple, framework for the formal evaluation of system functionality associated with a stakeholder viewpoint (or design concern). The framework is supported by a new procedure for systematic construction of minimum detail behavior models.
3. We explore the extent to which sequences of targeted abstraction (i.e., systematic removal of actions from behavior models) can be applied to behavior modeling and formal validation of behaviors likely to occur in automated infrastructure systems.

Over the past twenty five years the vast majority of research [13] in compositional behavior modeling and formal approaches to validation has been targeted to software and hardware systems. This article, in contrast, explores opportunities for adapting these techniques for the analysis and design of behaviors in large-scale infrastructure systems. While there are similarities in these domains (e.g., many concurrent behaviors) there are also important differences. For example, on the physical side of the problem, low-level concurrent behaviors in infrastructure systems only interact with other processes that are geographically co-located. This observation opens the door to the design of process

hierarchies for behavior modeling, and simplification of validation procedures through sequences of formal validation on behavior model modules.

The mathematical formalisms underlying our work are state machines known as labelled transition systems (LTS). Each component of a system specification is defined in terms of the set of states a system may be in, sets of action labels, and all of the transitions it may perform. We employ a process algebra notation (FSP) for concise description of component behavior in text. The computational implementation and visualization of labeled transition systems is handled by LTSA (Labelled Transition System Analyzer), a tool for validating communication and sequencing among entities in systems containing concurrent behaviors. LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties [14–16]. Finally, we exercise the proposed approach through development of behavior models and validation procedures for polite conversation, and lockset- and system-level concerns for ships traversing a stylized model of the Panama Canal System. To deal with the large number of concurrent behaviors and distributed system operations, the second application area is organized into a process modeling hierarchy, with each level having its own set of design concerns and viewpoints. The systematic application of viewpoint–action–process traceability results in process models that are orders of magnitude smaller than naive approaches to process composition. Finally, we note that state-of-the-art approaches to verification of behavior functionality rely on simulation and trial-and-error testing, both of which can be very time consuming and costly. Thus, a second key benefit of this work is progress toward a replacement procedure where mathematically rigorous approaches to verification prove functionality is correct.

2. Background and Related Theory

The driving tenet of our investigation is that specifications for functional system behavior can be assembled and formally analyzed in compositional (hierarchical) manner. Initially developed in the early 1990s, process algebras have emerged as an important mathematical basis for assembly and reasoning about system structure and behavior in a compositional way.

2.1. Background to Compositional Behavior Modeling

In each decade since the 1960s, remarkable advances in computer and networking technology have allowed for expanded expectations of computing which, in turn, have driven the need for new software development environments, tools and programming languages. One important aspect of the advances has focused on the modeling of systems having concurrent behaviors. State-of-the art design procedures correspond to a top-down specification of behaviors using visual modeling languages such as SysML [17], and rely on "threads of control" (e.g., lightweight processes) 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 [18,19]. 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 behavior through scripting and composition mechanisms [20–22]. Instead of starting with highly nondeterministic mechanisms (e.g., 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.

Figure 2 shows, for example, a process modeling framework for the systematic assembly of a transportation (canal) system and its interaction with the surrounding environment. A reasonable approach for representing canal system behaviors is to assume that each subsystem (or component)

has 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.

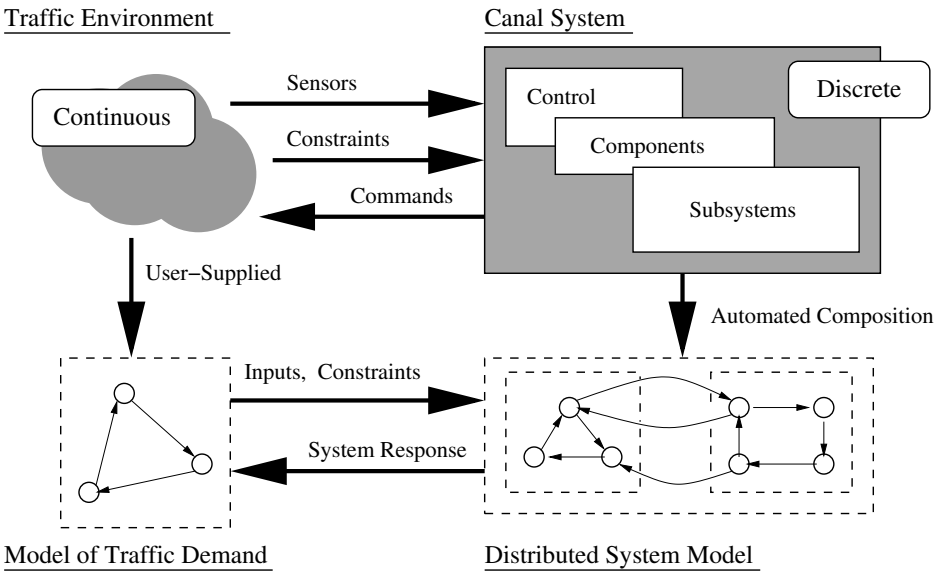

Figure 2. Automated assembly and behavior modeling of reactive processes in a canal system and traffic environment.

2.2. Finite State Automata

Automata are simple, but useful, models of computation where behavior includes the notions of state, transitions between states, and start and end states, but removes from consideration notions of memory, variables, commands and expressions. From a systems modeling point of view, the use of automata in behavior models is attractive because sophisticated models of behavior can be systematically assembled into networks and hierarchies of interacting processes, each defined by a simplified finite state machine representation (or automata). This strategy of model assembly works because automata constructions operate on automata yielding new automata and, in fact, provide closure under the operations of union, intersection, complement and concatenation [13]. For example, a product construction takes two deterministic finite state automata (DFAs) and generates a single DFA that conceptually runs its two component machines in parallel on the same input string.

2.3. Labeled Transition Systems

Definition 1. *A labeled transition system (LTS) process, P, 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,*

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.*

Process behavior is defined through sequences of actions (or transitions) a process may perform. The set of actions relevant to the 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 $(S, A, \triangle, q) \xrightarrow{a} \left(S, A, \triangle, q^{'}\right)$ if and only if $\left(q, \alpha, q^{'}\right) \in \triangle$ and $q^{'} \neq \pi$.

Composition of Labeled Transition Processes. Given two labeled transition systems 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:

$$\text{Behavior Model} = P_1 \| P_2 \| P_3 \| \cdots \| P_n \tag{1}$$

where P_i is the finite state model for the i-th component among n interacting components. 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.

Table 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. Equation (5) through (10) define the transitional semantics of the parallel composition operator. The composition operator is both commutative and associative, as indicated in Equations (9) and (10). 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.

Table 1. Translational semantics for restriction and composition operators. The notation $P \uparrow L.$ represents the process projected from P in which only the actions in the set L are observable.

Rules on Restriction		
$\frac{P \xrightarrow{a} P^{'}}{P \uparrow L \xrightarrow{a} P^{'} \uparrow L}$	$\left(a \in L, P^{'} \neq \pi\right)$	(1)
$\frac{P \xrightarrow{a} \pi}{P \uparrow L \xrightarrow{a} \pi}$	$(a \in L)$	(2)
$\frac{P \xrightarrow{a} P^{'}}{P \uparrow L \xrightarrow{\tau} P \uparrow L}$	$\left(a \notin L, P^{'} \neq \pi\right)$	(3)
$\frac{P \xrightarrow{a} \pi}{P \uparrow L \xrightarrow{\tau} \pi}$	$(a \notin L)$	(4)
Rules on Parallel Composition		
$\frac{P \xrightarrow{a} P^{'}}{P \| Q \xrightarrow{a} P^{'} \| Q}$	$\left(a \notin \alpha Q, P^{'} \neq \pi\right)$	(5)
$\frac{P \xrightarrow{a} \pi}{P \| Q \xrightarrow{a} \pi}$	$(a \notin \alpha Q)$	(6)
$\frac{Q \xrightarrow{a} Q^{'}}{P \| Q \xrightarrow{a} P \| Q^{'}}$	$\left(a \notin \alpha P, Q^{'} \neq \pi\right)$	(7)
$\frac{Q \xrightarrow{a} \pi}{P \| Q \xrightarrow{a} \pi}$	$(a \notin \alpha P)$	(8)
$\frac{P \xrightarrow{a} P^{'} \quad Q \xrightarrow{a} Q^{'}}{P \| Q \xrightarrow{a} P^{'} \| Q^{'}}$	$\left(a \in \alpha P \cap \alpha Q, P^{'} \neq \pi, Q^{'} \neq \pi\right)$	(9)
$\frac{P \xrightarrow{a} P^{'} \quad Q \xrightarrow{a} Q^{'}}{P \| Q \xrightarrow{a} \pi}$	$\left(a \in \alpha P \cap \alpha Q, P^{'} = \pi \vee Q^{'} = \pi\right)$	(10)

Systematic Removal of Detail from Behavior Models. From a modeling and design standpoint, the alphabet of a behavior model needs to achieve the dual objectives of supporting the required decision making, while also working to keep a model's size 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 in a process can be controlled by a restriction operator $\uparrow$. In particular, the notation $P \uparrow L.$ represents the process projected from P in which only the actions in the set L are observable. Equations (1) through (4) in Table 1 define the transitional semantics of the restriction operator. As a case in point, Equation (1) 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 $P^{'} \uparrow L$.

2.4. Model Checking and Property Validation

Model Checking. Given a finite-state model of a system and a formal property, model checking procedures [23] systematically check whether the property holds for that model. Model checking begins with two activities that can occur concurrently.

As shown along the left-hand side of Figure 3, 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?). Then as illustrated on the right-hand side of Figure 3, a finite state process model for behavior of the engineering system is assembled.

To answer the above-mentioned questions in a precise and unambiguous manner, the system behavior model must be sufficiently detailed, but for computational purposes, not too complex. Model checking procedures examine property specifications with respect to process models. Three outcome are possible:

1. The property specification is satisfied,
2. The property specification fails,
3. The model checking procedure fails because of insufficient computer memory.

When a property specification fails, the result is accompanied by a counter example. In most cases, the underlying cause 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. The only practical way of dealing with "insufficient memory" 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.

Framework for Compositional Validation of System Behavior. A good system design satisfies safety properties and exhibits liveliness (or 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/progress concerns include the ability of a process to eventually terminate and/or reach a critical state (or outcome) in its execution.

Safety properties are specified in FSP as deterministic property automata. Each property automaton specifies the set of feasible execution sequences over the actions (transitions) that correspond to a safety property of interest.

The upper diagram in Figure 4 shows, for example, a property where action `a` must be followed by action `b` which, in turn, can be followed by a sequence of action `c` or action `d`. LTSA creates an image automaton that captures the prescribed property automaton and adds to it possible violations leading to an error state. Now suppose 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. Next, we note from Table 1 that the (error) π state is preserved by both the restriction and composition operators. This property allows for a validation procedure that is remarkably straightforward. All that we have to do is compose the property automaton with the system description process—see, for example, the upper half of Figure 5—and look for the existence of π (an error state) in the global LTS. If the composed process contains an error state, then the safety property is violated. Otherwise, the safety property is satisfied. For a mathematical treatment of property automata and safety operations, we refer the interested reader to Austin and Johnson [24] and references therein.

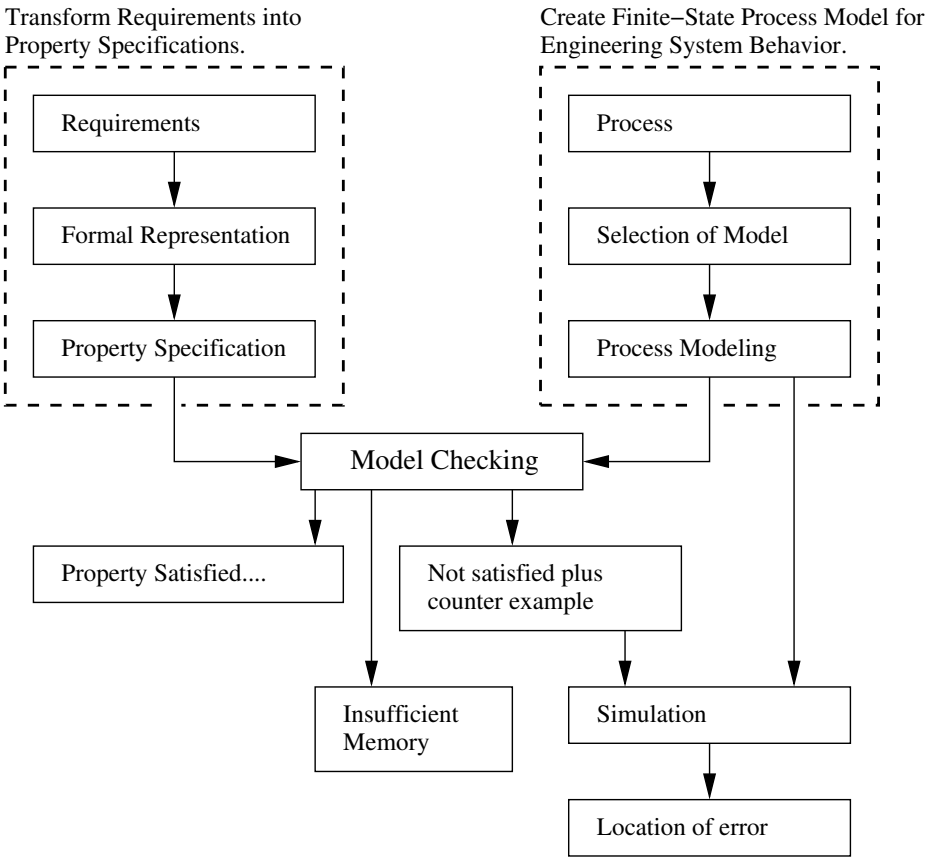

Figure 3. Model checking procedure and outcomes.

Progress Properties and Analysis. As already noted, a progress property asserts that it is always the case that an action—usually, a desirable 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 [25].

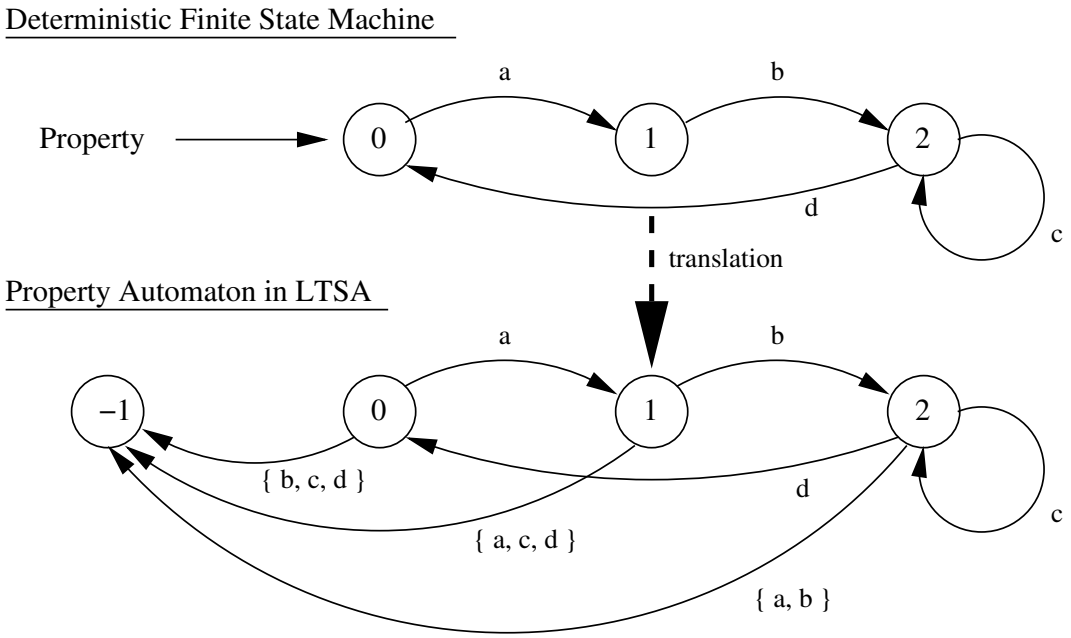

Figure 4. Procedure for definition of property automata in LTSA.

3. Targeted Abstraction and Viewpoint-Action-Process Traceability

The well known difficulty in using exhaustive search techniques for validation of concurrent system behaviors is that size of the state space expands exponentially with increasing numbers of underlying processes (and actions therein). This occurs because interleaving models for asynchronous system behavior allow concurrent events to be ordered arbitrarily and, as such, events are interleaved in all possible ways. For a model where n transitions can occur concurrently, and are primarily autonomous, there are n different orderings and 2^n different system states. Only minor reductions in the size of composed processes will occur through constraints associated with synchronized actions.

For the behavior modeling and verification for distributed systems operations, the central challenge is not composition of the behaviors, but design of strategies that:

1. Use decomposition to organize design concerns, viewpoints, and processes into a hierarchy, and
2. Employ abstraction to eliminate details of a behavior model that are of no importance when evaluating system functionality with respect to a specific set of design concerns.

To this end, our goal is to support decision making with process models and verification procedures of minimal size.

3.1. Related Work in State Graph Reduction

Techniques for reducing the size and complexity of processes can be classified into two broad categories:

1. Reduction by partial ordering, and
2. Reduction by compositional minimization [26–28].

Strategies for partial order reduction are based on the dependency relations that exist between the transitions of a system, and the observation that two interleavings can be regarded as equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (independent) execution steps. Algorithms for partial order reduction explore at least one trace from each (so-called Mazurkiewicz) equivalence class [29]. Thus, partial order reduction provides a full coverage of all behaviors that can occur in any interleaving, even though it explores only a subset of traces. Simplified exploration

of the state space can also occur through consideration of only those process interleavings that are relevant to a specific property being validated [23,30–33]. Solution strategies include use of bounded and context-dependent searches [34,35], modified semantics and/or identification of dependencies among transitions [28,36]. Recent advances include dynamic approaches to partial ordering [37,38] and use of annotations in the analysis of trace spaces. While the general state space exploration problem is NP complete, Chatterjee et al. [39] have shown for acyclic trace structures, it is possible to explore the state space with polynomial effort.

3.2. Compositional Reduction Analysis

Procedures for compositional reduction analysis are ideally suited to systems that can be naturally organized into a hierarchy of behavior modules. The simplification of behavior models occurs through the hiding of as many internal actions as possible in each subsystem, and through the tendency of safety properties to be locally checkable. The lower half of Figure 5 shows, for example, a scenario where behavior of process C corresponds to the composition of processes A and B (i.e., A $\|$ B). If process C is too large (i.e., requires resources beyond the capability of the environment within which it is executing), then the composition is simplified by removing actions internal to the process module (i.e., $A \rightarrow A^*$ and $B \rightarrow B^*$). Safety properties are then validated against the simplified composition ($A^* \parallel B^*$). Progress properties rely on a set of actions being activated at a particular level of detail; similar reductions in complexity can occur when lower level actions are removed from consideration.

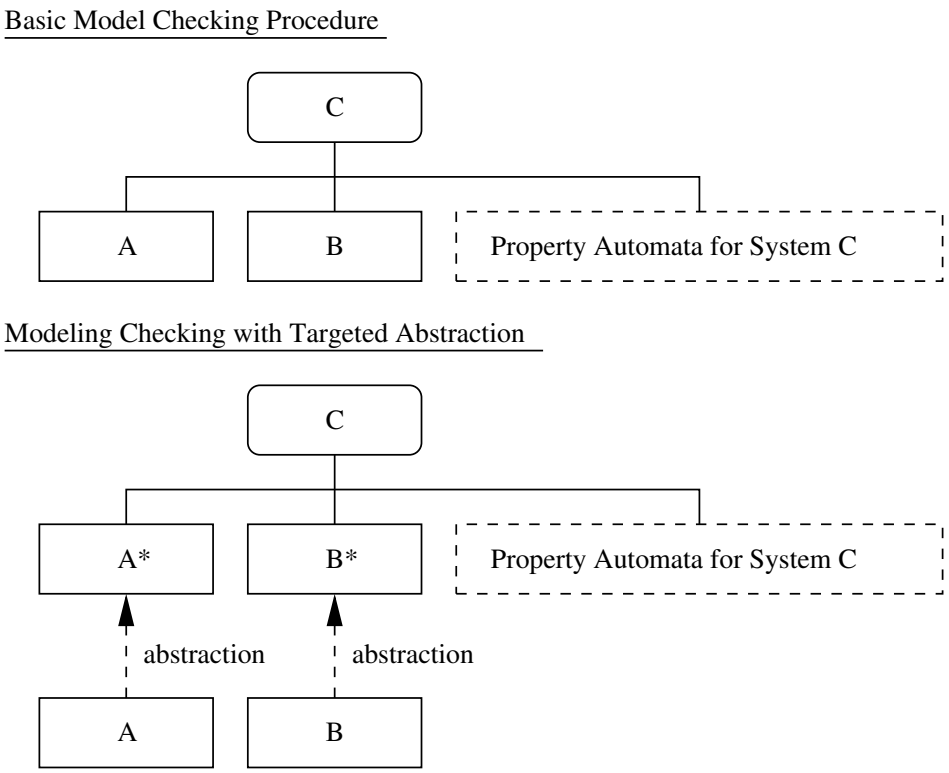

Figure 5. Visual representation of system C composed from processes A and B, and, validation of system C via composition with property automata.

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.

3.3. Viewpoint-Action-Process Traceability

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).

We propose that process validation procedures can be simplified through: (1) Systematic organization of design concerns, viewpoints and processes into hierarchies (system decomposition), and (2) Reduction by compositional minimization (abstraction). In a departure from past work, modules will have boundaries (i.e., notions of inside and outside) that depend on a design viewpoint (or family of design concerns). They will be organized into hierarchies; as such, low-level processes of minimal size can be composed into higher-level processes that will be validated against the concerns of a specific viewpoint.

To streamline this process we propose the 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. The step-by-step procedure is as follows:

1. For each design concern, identify the set of actions that are part of the associated behavior.
2. Identify the processes that participate in the execution of these actions.
3. Remove from the behavior, actions that are not associated with the design concerns or interactions among the participating processes.

This procedure lends itself to a tabular display of traceability linkages, as illustrated in Figure 6. After each application of viewpoint-action-process traceability, we also immediately minimize the size of all intermediate results. As we will see in the application below, 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.

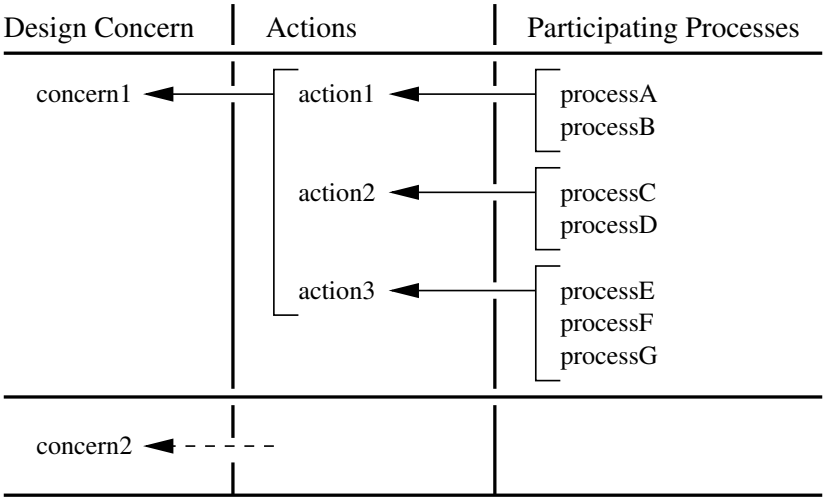

Figure 6. Schematic for tabular display of design viewpoint-action-process dependencies. A design concern depends on a set of actions, which, in turn, belong to a set of participating processes.

4. Case Study A: Formal Validation of a Behavior Model for Polite Conversation

To see how compositional approaches to behavior modeling work on a small-scale problem, let us consider a simple scenario where two people, Jack and Diane, meet for coffee and polite conversation. The goal is to devise a system-level behavior model where both Jack and Diane talk, but neither party dominates the conversation. The properties of this behavior model will be formally checked.

The script of code shown in Table 2 systematically assembles the hierarchy of processes shown in Figure 7. The key activities in development of the behavior model are as follows:

1. Identify main events, actions and interactions,
2. Identify main component-level processes,
3. Identify and define properties of interest, and
4. Compose component-level behaviors into a model of architectural-level behavior.

The behavior modeling process begins with the definition of a generic `PERSON`, who can talk and drink, or simply wait and then drink. LTSA requires that processes operate continuously. Thus, if `talk` and `drink` are actions, and `PERSON` is a process, then the fragment of FSP code `talk -> drink -> PERSON` describes a process that initially engages in the action `talk`, then the action `drink`, and then behaves exactly as prescribed by `PERSON`. In practical terms, an action might be a communication, a signal, or perhaps, traditional execution of a task. Jack and Diane are simply labeled instances of the process `PERSON` (i.e., with the notation `jack:PERSON` and `diane:PERSON`).

The composed process `JACK_AND_DIANE_MEET` captures all of the possible sequences of actions that can occur. This under-constrained model allows, for example, for one person to talk and talk and talk, with the other person not getting a word in edgewise. The keyword minimal minimizes the size of the composed process from nine states to four states.

Table 2. Systematic assembly of a behavior model for polite conversation.

```
// ================================================================================
// Behavior Model: Jack and Diane meet for coffee and conversation.
// ================================================================================

// 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!
```

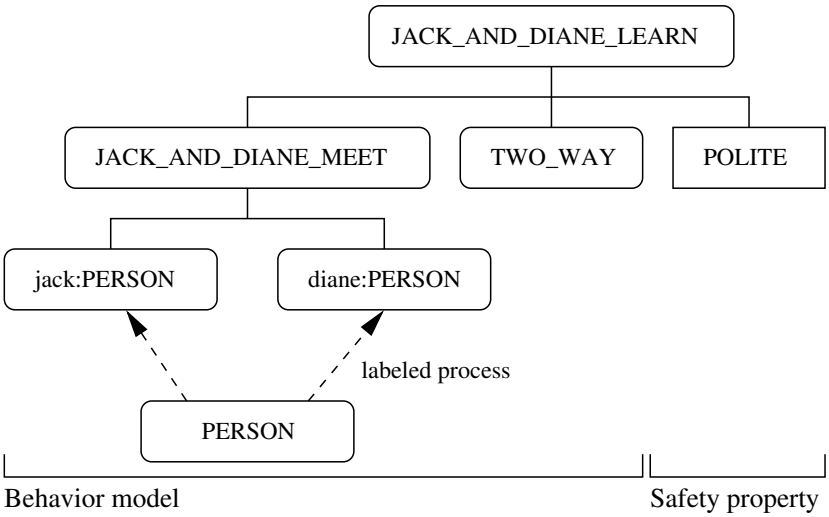

Figure 7. Process hierarchy for behavior model and validation of polite conversation.

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 ).
```

However, 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`. The FSP notation `jack.talk/diane.wait` and `diane.talk/jack.wait` changes (i.e., relabels) the action notation `diane.wait` to `jack.talk`, and `jackwait` to `diane.talk`, respectively, thereby simplifying complexity of the behavior model.

Figure 8 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 8) 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
```

indicating that hoth Jack and Diane engage in polite conversation.

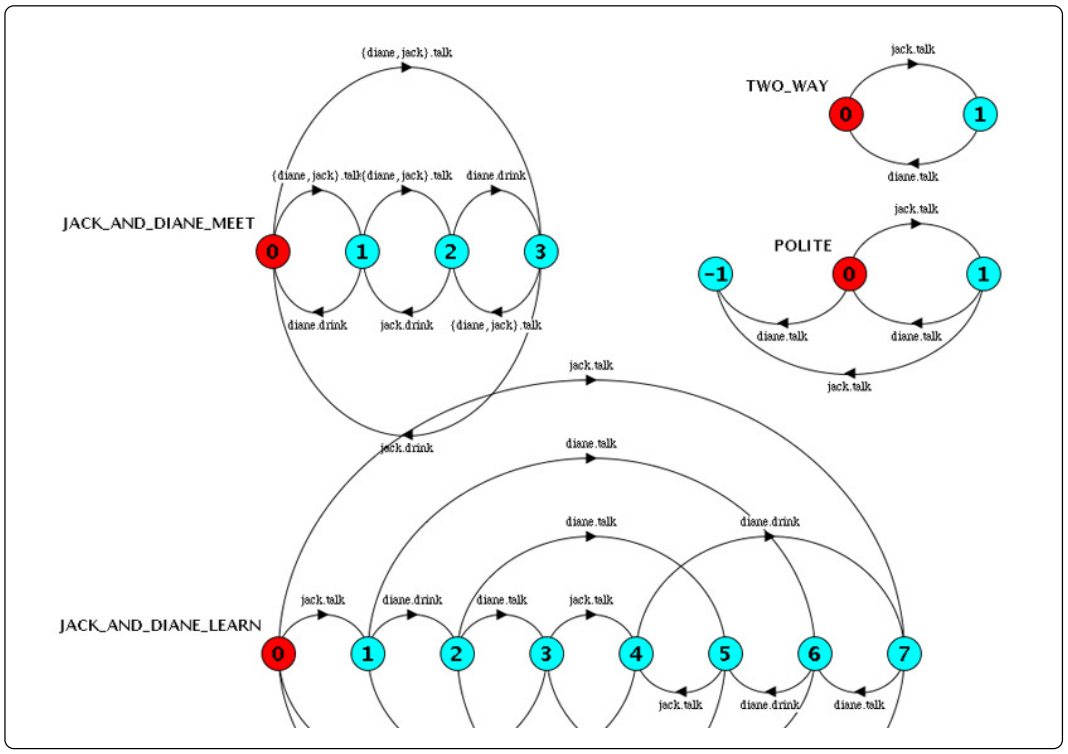

Figure 8. LTSs for processes in a behavior model of polite conversation.

5. Case Study B: Compositional Behavior Modeling and Validation of Waterway System Operations

We now exercise the proposed methodology by working step-by-step through the model-based design and formal validation of behaviors for collections of ships traversing a simplified model of the Panama Canal, an 80 km passageway that joins the Atlantic and Pacific Oceans at one of the narrowest saddles of the isthmus [40]. 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.

Solutions to this problem are complicated by the large number of concurrent processes (e.g., ship operations, pumps, gates, lockset scheduler) 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 these limitations, we apply viewpoint-action-process mechanisms, thereby abstracting from the composition, actions and groups of actions that are irrelevant to decisions associated with a particular viewpoint. Our goals is to devise strategies where:

1. Behavior models focus on the modeling of ship movements, and
2. The size of the canal system processes models remains constant, regardless of the number of ships waiting to traverse the canal system.

Constraints on behavior will ensure that when ships arrive at a lockset they are handled on a first-come, first-served basis. Complete details of the input files for these case studies can be found in the appendices of Austin and Johnson [24].

5.1. Framework for Multi-Layer Behavior Model Development

The step-by-step procedure for design, implementation and validation of behavior for Panama Canal operations is simplified through a top-down decomposition of concerns (i.e., requirements and design specifications) followed by a bottom-up assembly of process models and testing/validation.

Figures 9 and 10 show the two-level V-model of system development assumed for this study, and an elevation view of the lockset, and component- and lockset-level process architectures. In an extension of the procedure described in Case Study A, behavior models are organized into a two-level process hierarchy. For the lockset-level behavior, localized control will be employed for the safe, fair, and efficient scheduling of ship transit operations. A scheduler process 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. At the system level, the primary design concerns include provision for: (1) Adjustment of operations during emergency and/or /maintenance events and; (2) Cooperation of asynchronous lockset-level behaviors to ensure efficient transit of ships through the canal system. Because emergency and/or 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.

Lockset- and Canal-Level Requirements. The functional requirements associated with lockset-level concerns are as follows:

1. At any point in time, the scheduler must assign no more than one ship to a lockset;
2. All ships must request transit before they can acquire access to a lockset. They must acquire access to the lockset before they depart.
3. Flooding must be prevented. A gate must not open until water levels on both sides of the gate have been equalized; and
4. All ships that request passage through the lockset must eventually depart the lockset.

At the canal system level, functional requirements include:

1. When an emergency or maintenance occurs, all incoming canal traffic must be halted. Traffic can resume after the emergency (or maintenance) has been cleared (or repaired).
2. All east- and west-bound ships must be guaranteed to reach the Atlantic and Pacific Oceans respectively.

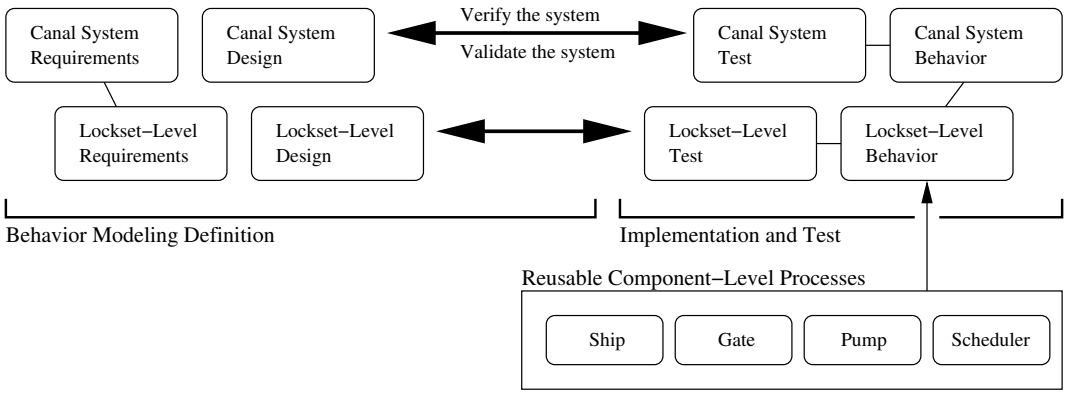

Figure 9. Step-by-step procedure for behavior model development, implementation, and testing/validation.

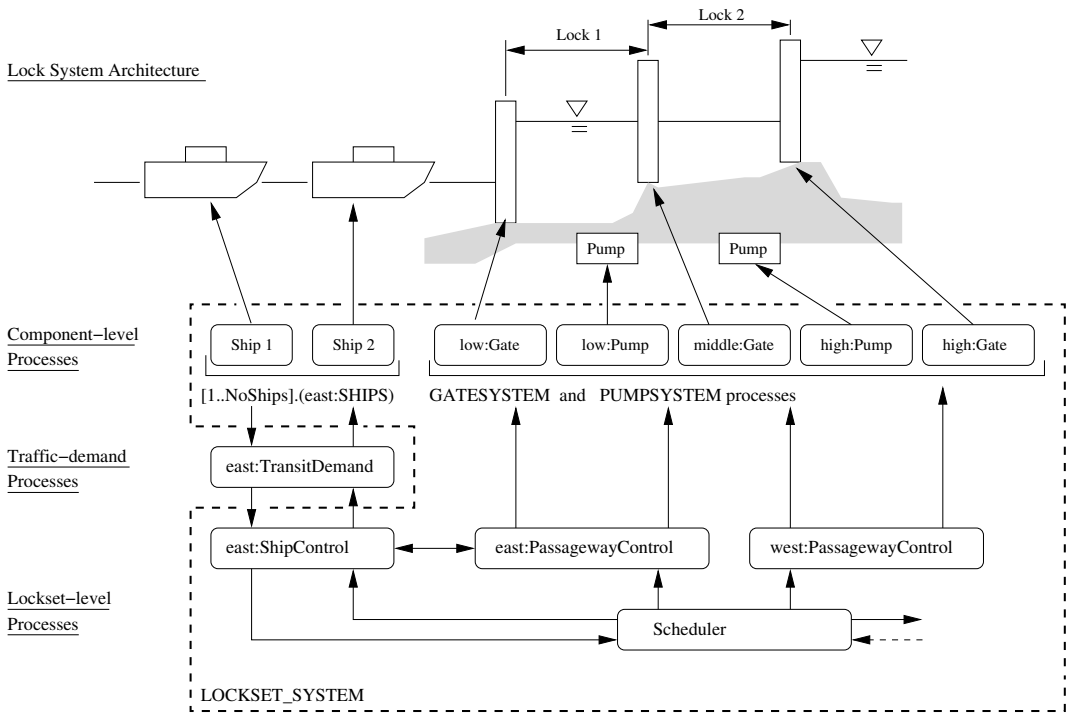

Figure 10. Elevation view of lockset, and component- and lockset-level process architecture.

Top-Down Specification of System Functionality. In the simplest terms possible, a ship will transit a two-stage lockset by working through the following step-by-step procedure:

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

The execution of Steps 1–6 implies a specific sequencing of messages among ship, ship control, passageway control and scheduler processes. The key communications among processes are as follows:

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

Figure 11 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.

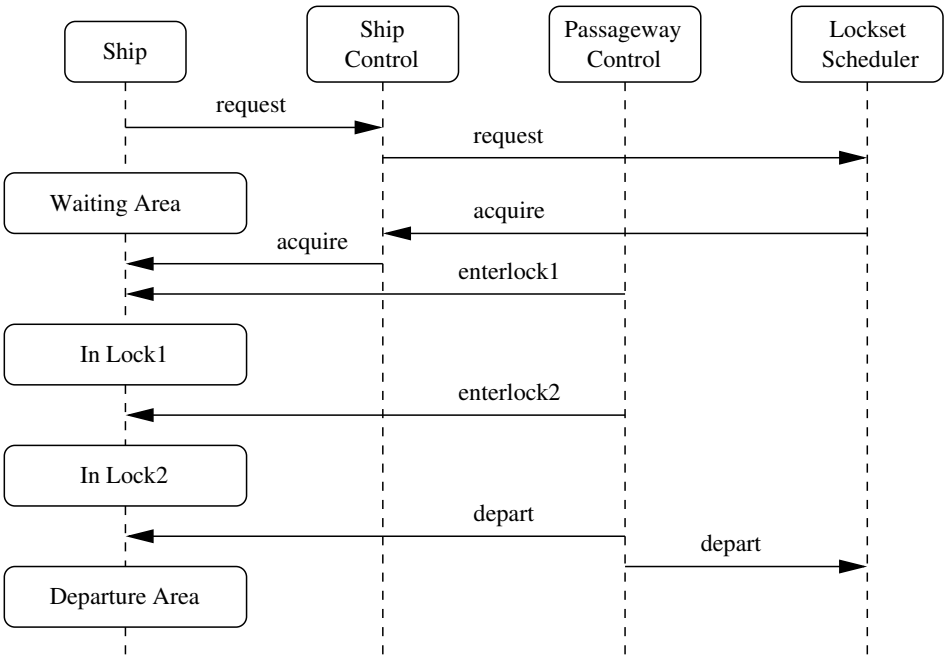

Figure 11. Simplified communication among ship, ship control, passageway control and scheduler processes for a ship transiting the lockset.

Simplified Model of Ship Behavior. The sequence of messages in Figure 11 suggests that basic ship behavior can be 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`. However, 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 (details to follow) to properly sequence `enterlock1` and `enterlock2` among other low-level actions for pump and gateway control.

5.2. Bottom-Up Composition of Travel Demand and Lockset-Level Behavior

Figure 10 shows the process architecture for the implementation of lockset-level behavior. One process hierarchy is assembled for the lockset system itself – it is the composition of scheduler, passageway control, ship control, gate, pump and lock processes. A second process hierarchy is assembled for the east- and west-bound traffic demand. The complete model of lockset behavior corresponds to the parallel composition of lockset and traffic demand process models, i.e.,

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

Model of Lockset-Level Traffic Demand. Travel demand processes are defined for convoys of east- and west-bound ships (i.e., processes `EASTBOUND_SHIPS` and `WESTBOUND_SHIPS`) and circular queuing processes to ensure that transit requests are handled in the same order in which they are made, i.e.,

```
// Simplified model of a single ship passing through 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 models of East- and West-bound traffic ....

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

The command `SHIP = ( request -> acquire -> depart -> SHIP )` defines a single process for a sequence of actions defining ship behavior. Next, the lines `EASTBOUND_SHIPS = ( [i:S]:(east:SHIP) )` and `WESTBOUND_SHIPS = ( [i:S]:(west:SHIP) )` create arrays of `S` ships traveling in West- and East-bound directions. It is important to notice that from a ship's point of view, acquiring access to the lock system is what matters – once that is achieved, the next significant action is departure from the lock system. Ships do not need to know (or care) about the internal details of the lock system operations. Finally, the overall traffic demand model is the composition of east- and west-bound traffic demands.

First-cut Composition of Lockset-System Behavior. In this first-cut implementation we simply compile the scheduler, west- and east-bound ship control, passage control, and gate and pump system processes together, then at the last point possible attempt to minimize the process size. As illustrated in Figure 10, 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. Ship controller processes are shared resources responsible for maintaining order between incoming requests and access to the lockset.

Design of the passageway control and lockset scheduler processes is the most interesting part of the lockset behavior model formulation. 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. The lockset scheduler process receives 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 take into account the number of ships waiting to transit the canal in either direction, and implement an appropriate policy of fairness. Moreover, the scheduler process should also 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).

As illustrated in Figure 12, the scheduler is 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 = number of east-bound ships waiting (0..NoShips). Variable ww = 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). At a glance it would seem that the ship control processes are not needed because their operations are completely covered by the scheduler process. The important distinction and, hence, their roles lie in their view of the ships. Ship controllers track the progress of specific ships by name (e.g., `[1].east`). 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.

5.3. Preliminary Results

Table 3 summarizes the number of states in the traffic demand and lockset system (and select lockset subsystem) processes as a function of the number of east- and west-bound ships. On an Apple Macbook Pro with 4GB of memory, the problem formulation becomes computationally intractable when three east-bound ships and three west-bound ships are traversing the canal.

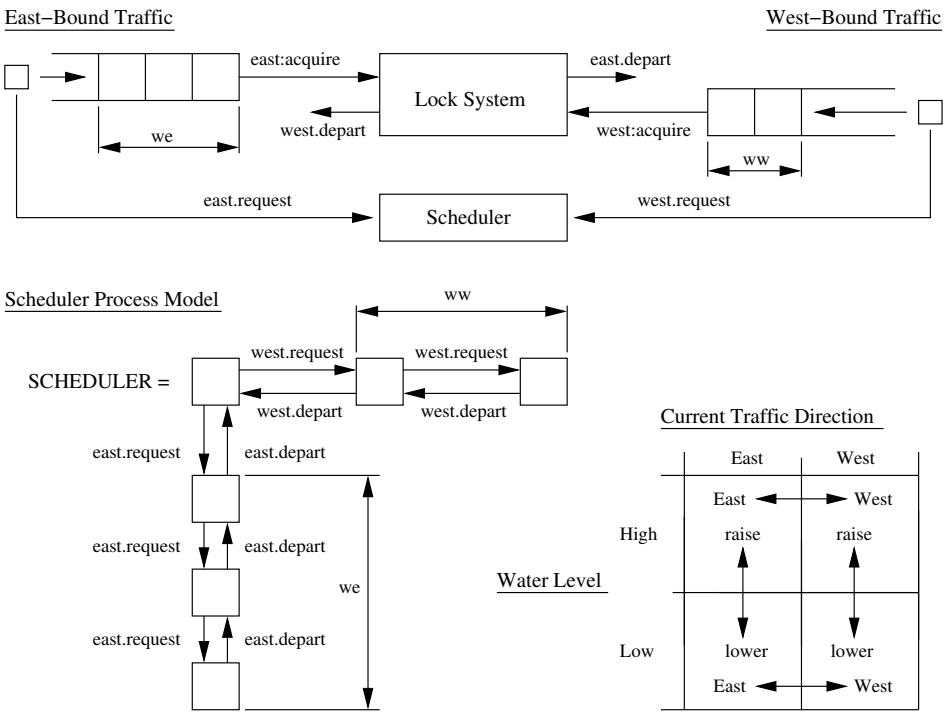

Figure 12. Schematic for process design of the traffic scheduler.

Table 3. No of states in the lockset behavior model for NoShips = 1, 2 and 3. Legend: TD = traffic demand, LS = lockset system; LB = lockset behavior.

```
Part 1: Naive composition of lockset-level processes
-------------------------------------------------------------------------
    No    E-W Ship      TD    SCHEDULER       LS       LB          LB
 Ships       Model  States      States   States   States   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!!
=========================================================================
```

To overcome this problem we need to be much smarter in controlling the size of participating processes by only including those actions (and critical dependencies) that are tied to a particular viewpoint of system functionality.

5.4. Composition of Simplified Lockset-Level Behavior

The composition of simplified lockset-level behavior models takes advantage of the natural hierarchy of canal system behavior models, and only considers actions that are directly related to a specific decision or behavior modeling viewpoint. Viewpoint driven models can be assembled for: (1) simplified modeling of ship movement (i.e., a minimal version of `LOCK_SYSTEM`); (2) examination of passageway safety against flooding (i.e., `LOWER_PUMPS` and `RAISE_PUMPS`); and (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.

Figures 13 and 14 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 `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. In the initial formulation, models of lockset system behavior were defined through an "all-in-one composition" of seven 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 passageway processes, plus processes for the pump and gate systems, and
3. Gate and pump systems are defined through the composition of individual pump and gate processes (these details are not shown in Figure 14).

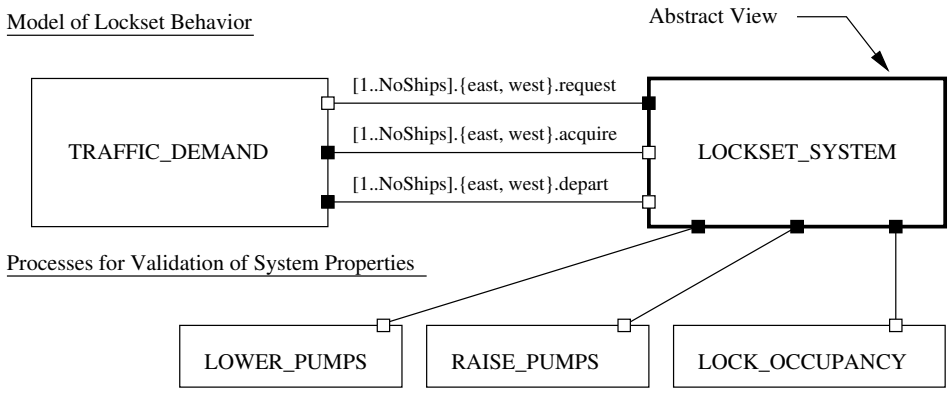

Figure 13. Schematic for development of simplified models of lockset system behavior. White dots represent requirements. Black dots represent provisions.

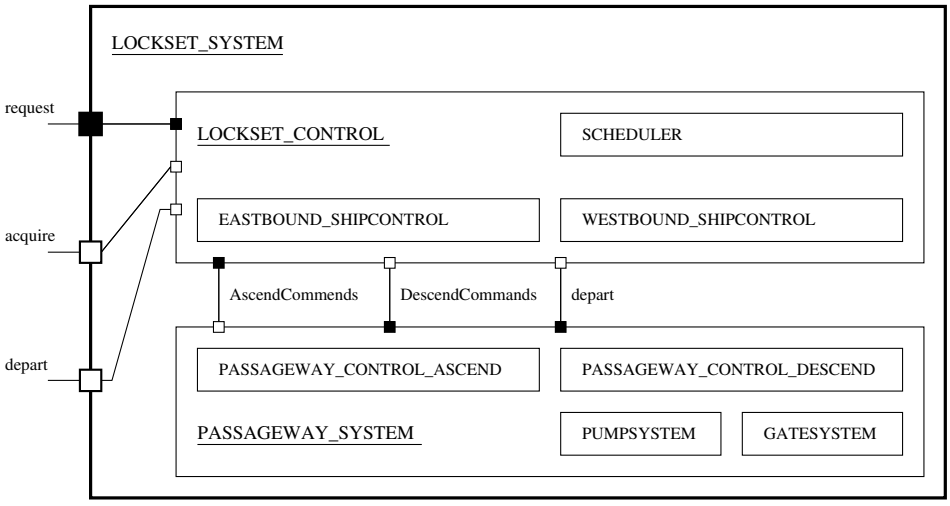

Figure 14. Schematic for development of simplified models of lockset system behavior. White dots represent requirements. Black dots represent provisions.

5.5. Application of Viewpoint-Action-Process Traceability

To systematically determine which parts of the behavior model can be omitted without affecting a pre-defined viewpoint, the process-action dependency is reversed. For each action (or, when appropriate, group of actions) a list of dependent processes is assembled; an abbreviated example is shown Table 4. 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. Abbreviated list of action-process relationships in lockset system model.

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
... Details of action-process relationships removed	
[1].east.{ enterlock1, EASTBOUND_PASSAGECONTROL.{1}::east: PASSAGECONTROL_ASCEND enterlock2, exitlock2 }	

As a starting point, the lockset system behavior model is organized into a three layer hierarchy:

1. The lockset control (i.e., ‖ LOCKSET_CONTROL) is composed from scheduler and west- and east-bound ship control processes.
2. The passageway system (i.e., ‖ PASSAGEWAY_SYSTEM) is composed from west- and east-bound passageway control processes.
3. Finally, lockset system behavior is assembled from the previously composed models for lockset control and the 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. However, 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.

Sample Viewpoint (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 in Table 5 implements and minimizes two versions of LOCKSET_SYSTEM, one that includes LOCKSET_CONTROL and PASSAGEWAY_SYSTEM, and a second model based on LOCKSET_CONTROL alone. The PASSAGEWAY_SYSTEM process only carries forward actions that are critical to communication (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.

Table 5. Sample composition and minimization of lockset behavior.

```
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 omits details of the pump and gate operations ....

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

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

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

Scalability of the Lockset Behavior Model. Table 6 shows the size of the constituent processes as a function of NoShips. Not only are the process sizes several orders of magnitude smaller than in the initial formulation, 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 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 32. Still, when NoShips = 3, the unminimized PASSAGEWAY_SYSTEM model has $15 \times 15 = 225$ states. In contrast the minimized model has only 4 states and, in fact, this does not change with increasing numbers of ships.

Table 6. No of states in the lockset behavior model for NoShips = 1, 2 and 3. Legend: TD = traffic demand, LSC = lockset control; PS = passageway system; Min States = minimized states.

Part 2: Processes after viewpoint-action-process abstraction

No Ships	E-W Ship Model	TD States	SCHEDULER States	LSC Min. States	PS Min. States
1	[1..1]	9	33	32	4
2	[1..2]	324	132	48	4
3	[1..3]	6,561	352	64	4

No Ships	E-W Ship Model	LOCKSET_SYSTEM1 Minimized States	LOCKSET_BEHAVIOR1 Minimized States
1	[1..1]	32	12
2	[1..2]	48	48
3	[1..3]	64	108

No Ships	E-W Ship Model	LOCKSET_SYSTEM2 Minimized States	LOCKSET_BEHAVIOR2 Minimized States
1	[1..1]	32	12
2	[1..2]	48	48
3	[1..3]	64	108

5.6. Formal Validation of Lockset-Level Safety Concerns

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

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

The short fragment of code in Table 7 establishes a validation test for lock occupancy that says:

> If a specific ship acquires resources of the lockset, then it needs to depart before another ship acquires the lockset.

Next, execution of the lock occupancy test is defined through composition of the lockset system with the lock occupancy property test. If the composed system (i.e., `LOCK_OCCUPANCY_CHECK1`) contains an error state (i.e., it fails), then there exists at least one pathway in the lockset system where an acquire operation is followed by a second acquire and/or a depart action is followed by a second depart operation. This interpretation of property satisfaction is allowed for via Equation (10) in Table 1. For our model, however, the composed model is free of an error state and the test passes.

Table 7. Composition validation of lock occupancy properties.

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

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

5.7. Composition and Validation of System-Level Behaviors

The Panama Canal System corresponds to a parallel composition of three lockset-level processes (i.e., the Pacific, Middle and Atlantic lockset systems) plus a control monitor process.

Figure 15 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. LTSA notation for the system-level traffic demand model is a straight forward extension of the lockset level behavior model. For example, the action `pac.[S].east.request` indicates a request action by an eastbound ship to pass the Pacific Lockset. Using the strategy described in the previous section, simplified models of canal system behavior can be composed for a variety of viewpoints (e.g., traffic flow in one direction; behavior of a lockset within the canal system). Figure 16 shows, for example, ship behavior at the Pacific Lockset when the number of east- and west-bound ships is one.

The two principle design concerns at the canal level are: (1) ensuring maintenance and emergency events are properly handled [41,42], and (2) ensuring progress checks at the lockset level propagate up to the canal level. 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.

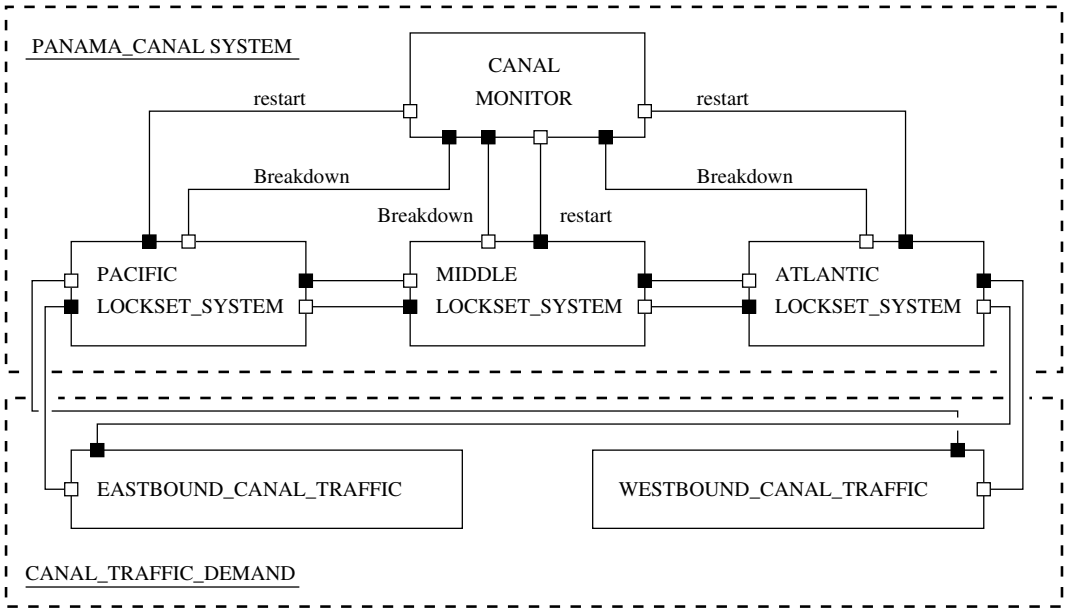

Figure 15. Process architecture for full canal model.

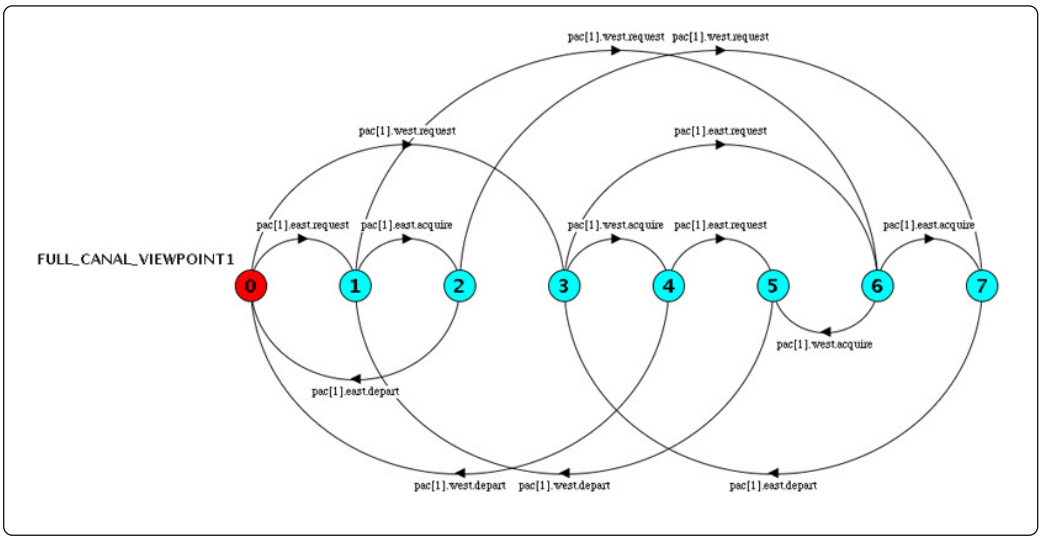

Figure 16. Behavior of ships transiting the Pacific lockset.

This policy leads to Figure 17, a schematic of system-level response to maintenance and 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 18 contains a partial view of behavior associated with maintenance/emergency operations and, in particular, shows 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.

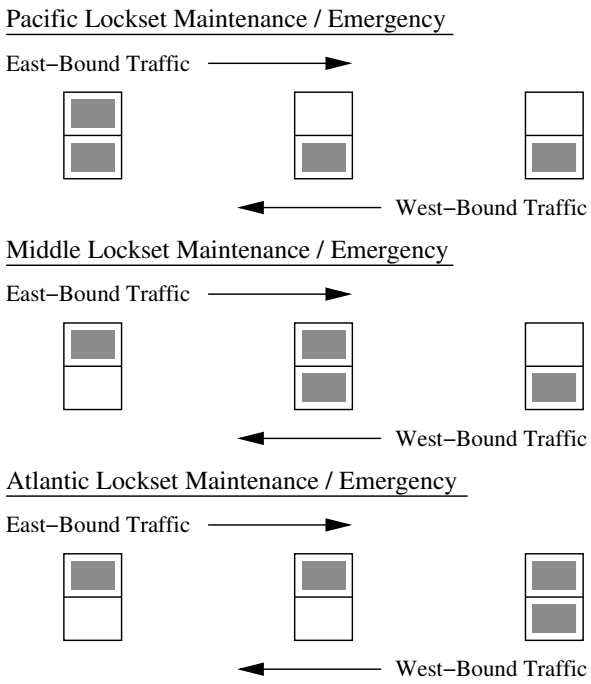

Figure 17. Schematic of system-level response to maintenance events and emergency events in the Pacific, Middle and Atlantic locksets.

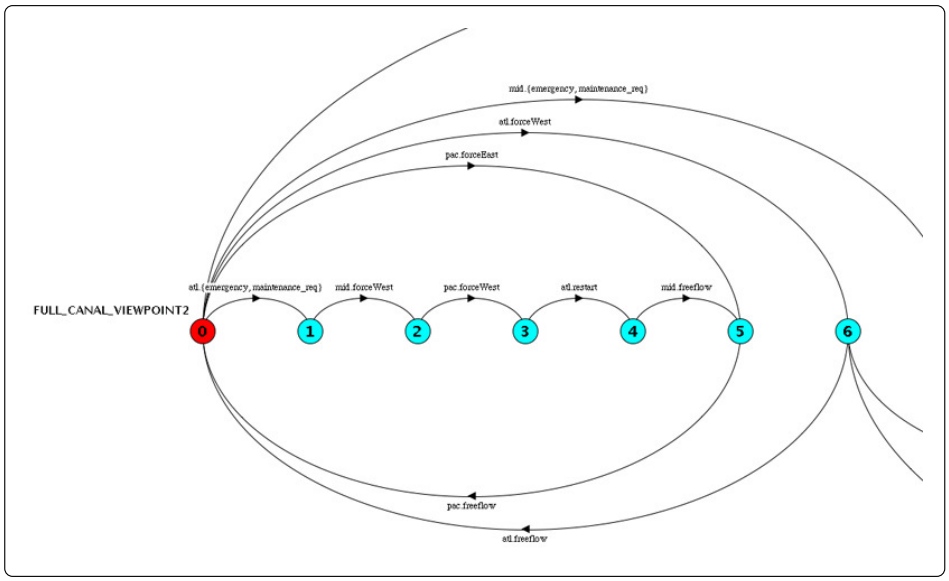

Figure 18. Abbreviated view of behavior associated with maintenance/emergency events.

6. Conclusions and Future Directions

Our work is motivated by the tenet that modern infrastructure systems are a necessary foundation for the transport of goods and services to support global trade and long-term economic growth. The unfortunate reality is that too often failures, delays, and accidents in aging infrastructure systems do little to attract the technical and economic assistance required for modernization. At the same time, remarkable advances in computing and communication technologies over the past few decades have opened doors to the development of physical infrastructure networks tightly connected to

cyber (e.g, data, information, and software) for decision making. We believe that modernization efforts will be defined by increased use of automation to: (1) Expand the range of environmental conditions within which systems can safely operate; (2) Support and enhance human performance; and (3) Improve the ability of a system to quickly recover from unexpected disruptions. From a design standpoint, increased use of automation in infrastructure operations introduces new challenges in terms of correctness (assurance and security) of system functionality. From an systems analysis standpoint, compositional approaches to behavior modeling are an essential element to achieving required levels of system agility.

This paper takes a first step toward the development of a compositional approach to distributed system behavior modeling and formal validation of infrastructure operations with finite state automata. Solutions to this problem are complicated by the large number of concurrent processes defining component- and system-level behavior. The difficulty of this problem can be mitigated 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).

There is need for further work in a number of important directions. 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 modeled in UPPAAL [43]. 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. Finally, we have been creating formalisms [2,11] for distributed system behavior modeling of urban systems with ontologies and rules, and exploring ways in which urban networks can interact via message passing mechanisms.

Author Contributions: The authors contributed equally to this work.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

π	Deadlock state
$\rightarrow$	Remove actions internal to a process
τ	An internal action
$\triangle$	Transition relation
$\uparrow$	Restriction operator on observability of actions
A	Communication alphabet
L	Set of observable actions
P	Process
Q	Process
q	Current state selected from S
S	Set of states
DFA	Deterministic Finite Automata

FSP	Finite State Process
LB	Lockset Behavior
LS	Lockset Systems
LSC	Lockset Control
LTS	Labeled Transition System
LTSA	Labeled Transition System Analyzer
MBSE	Model-based Systems Engineering
PS	Passageway System
SysML	Systems Modeling Language
TD	Traffic Demand
UML	Unified Modeling Language

References

1. Preuss, P. Smart Moves: California's Next-Gen Infrastructure. In *Berkeley Engineer*; Publication of UC Berkeley College of Engineering, University of California: Berkeley, CA, USA, 2017; Volume 11.
2. Coelho, M.; Austin, M.A.; Blackburn, M. Distributed System Behavior Modeling of Urban Systems with Ontologies, Rules and Many-to-Many Association Relationships. In Proceedings of the The Twelth International Conference on Systems (ICONS 2017), Venice, Italy, 23–27 April 2017; pp. 10–15.
3. Grier D.V. *The Declining Reliability of the U.S. Inland Waterway System*; U.S. Army Corps of Engineers, Institute for Water Resources: Alexandria, VA, USA, 2005.
4. Iowa Department of Transportation. *U.S. Inland Waterway Modernization: A Reconnaissance Study*; HDR Engineering, Inc.: Omaha, NE, USA, 2013.
5. Dai, M.D.; Schonfeld, P. Metamodels for Estimating Waterway Delays Through a Series of Queues. *Transp. Res.* **1998**, *32*, 1–19.
6. Ting, C.J.; Schonfeld, P. Integrated Control For Series of Waterway Locks. *ASCE J. Waterw. Port Coast. Ocean Eng.* **1998**, *124*, 199–206.
7. Kaisar, E.; Austin, M.A.; Papadimitriou, S. Formal Development and Evaluation of Narrow Passageway System Operations. *Eur. Transp. Transp. Eur.* **2006**, *34*, 88–104.
8. Kaisar, E.; Austin, M.A. Synthesis and Validation of High-Level Behavior Models for Narrow Waterway Management Systems. *J. Comput. Civ. Eng. ASCE* **2007**, *21*, 373–378.
9. Jackson, D. Dependable Software by Design. *Sci. Am.* **2006**, *294*, 68–75.
10. Austin, M.A.; Baras, J.S.; Kositsyna, N.I. Combined Research and Curriculum Development in Information-Centric Systems Engineering. In Proceedings of the Twelth Annual International Symposium of The International Council on Systems Engineering (INCOSE), Washington, DC, USA, 29 June–3 July 2003.
11. Austin, M.A.; Delgoshaei, P.; Nguyen, A. Distributed System Behavior Modeling with Ontologies, Rules, and Message Passing Mechanisms. *Procedia Comput. Sci.* **2015**, *44*, 373–382.
12. Sangiovanni-Vincentelli, A.; McGeer, P.C.; Saldanha, A. Verification of Electronic Systems: A Tutorial. In Proceedings of the 33rd Desgin Automation Conference, Las Vegas, NV, USA, 3–7 June 1996.
13. Hopcroft, J.E.; Motwani, R.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*, 3rd ed.; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2006.
14. Magee, J.L.; Kramer, J.; Uchitel, S. Labeled Transition System Analyzer (LTSA) Home Page. 2013. Available online: https://www.doc.ic.ac.uk/ltsa/ (accessed on 23 June 2017).
15. Uchitel, S. Incremental Elaboration of Scenario-Based Specifications and Behavior Models using Implied Scenarios. Ph.D. Thesis, Imperial College, London, UK, 2003.
16. Uchitel, S.; Kramer, J.; Magee, J. Incremental Elaboration of Scenario-Based Specifications and Behavior using Implied Scenarios. *ACM Trans. Softw. Eng. Methodol.* **2004**, *13*, 37–85.
17. Fridenthal, S.; Moore, A.; Steiner, R. *A Practical Guide to SysML*; MK-OMG: San Francisco, CA, USA, 2008.
18. Lee, E.A. *Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report*; Technical Report; University of California: Berkeley, CA, USA, 2007.
19. Sutter, H.; Larus, J. Software and the Concurrency Revolution. *ACM Queue* **1997**, *3*, 54–62.
20. Nierstrasz, O.; Tsichritzis, D. (Eds.) *Object-Oriented Software Composition*; T.J. Press (Padstow) Ltd: Padstow, Cornwall, UK, 1995.

21. Olsen, R. *Design Patterns in Ruby*; Pearson Education: Boston, MA, USA, 2008.
22. Osterhout, J.K. Scripting: Higher-Level Programming for the 21st century. *IEEE Comput.* **1998**, *31*, 23–30.
23. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT Press: Cambridge, MA, USA, 2008.
24. Austin, M.A.; Johnson, J. *Compositional Behavior Modeling and Formal Validation of Canal System Operations with Finite State Automata*; Technical Report ISR-TR2011-04; Institute for Systems Research, University of Maryland: College Park, MD, USA, 2011.
25. Magee, J.L.; Kramer, J. *Concurrency: State Models and Java Programs*, 2nd ed.; John Wiley and Sons: New York, NY, USA, 2006.
26. Cheung, S.C.; Kramer, J. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Trans. Softw. Eng. Methodol.* **1999**, *8*.
27. Dams, D.; Gerth, R.; Knaack, B.; Kuiper, R. Partial-Order Reduction Techniques for Real-Time Model Checking. *Formal Asp. Comput.* **1998**, *10*, 469–482.
28. Minea, M. Partial Order Reduction for Verification of Timed Systems. PhD Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1999.
29. Mazurkiewicz, A. *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*; Springer-Verlag: New York, NY, USA, 1987; pp. 279–324.
30. Devadas, S.; Keutzer, K. An Automata-Theoretic Approach to Behavioral Equivalence. In Proceedings of the IEEE International Conference on Computer Aided Design, Santa Clara, CA, USA, 11–15 November 1990; pp. 30–33.
31. Goguen, A.G.; Rosu, G. Hiding More of Hidden Algebra. In *FM'99—Formal Methods*; Wing, J.M., Woodcock, J., Davies, J., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; pp. 1704–1719.
32. Lowry, M.R. The Abstraction/Implementation Model of Problem Reformulation. In Proceedings of the 10th International Joint Conference on Artificial Intelligence, Milan, Italy, 23–18 August 1987.
33. Lowry, M.; Subramaniam, M. Abstraction for Analytic Verification of Concurrent Software Systems. In Proceedings of the Symposium on Abstraction, Reformulation and Approximation, Kananaskis, AL, Canada, 9–12 May 1998.
34. Lal, A.; Reps, T. Reducing Concurrent Analysis under a Context Bound to Sequential Analysis. *FMSD* **2009**, *35*, 73–97.
35. Musuvathi, M.; Qadeer, S. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. *SIGPLAN Not.* **2007**, *42*, 446–455.
36. Pagani, M. Partial Orders and Verification of Real-Time Systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*; Jonnson, B., Parrow, J., Eds.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 327–346.
37. Flanagan, C.; Godefroid, P. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, CA, USA, 12–14 January 2005.
38. Abdulla, P.; Aronis, S.; Jonsson, B.; Sagonas, K. Optimal Dynamic Partial Order Reduction. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, 22–24 January 2014.
39. Chatterjee, K.; Paglogiannis, A.; Sinha, N.; Vaidya, K. Data-centric Dynamic Partial Order Reduction. *CoRR* **2016**, arxiv:1610.01188.
40. Panama Canal Web Site. Available online: http://www.pancanal.com/eng/index.html (accessed on 11 January 2018).
41. Panama Canal Maintenance, Britannical Online Encyclopedia. 2008. Availabel online: http://www.britannica.com/EBchecked/topic/440784/Panama-Canal/40008/Maintenance (accessed on 10 September 2008).
42. Panama Canal Locks, Wikipedia. 2018. Available online: https://en.wikipedia.org/wiki/Panama_Canal_locks (accessed on 11 January 2018).
43. UPPAAL: Design Verification for Embedded Systems. 2018. Available online: http://www.uppaal.com/ (accessed on 11 January 2018).

© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).