ENSE 623 Systems Validation and Verification

Emerging Approaches to System Validation/Verification

Mark Austin

E-mail: austin@isr.umd.edu

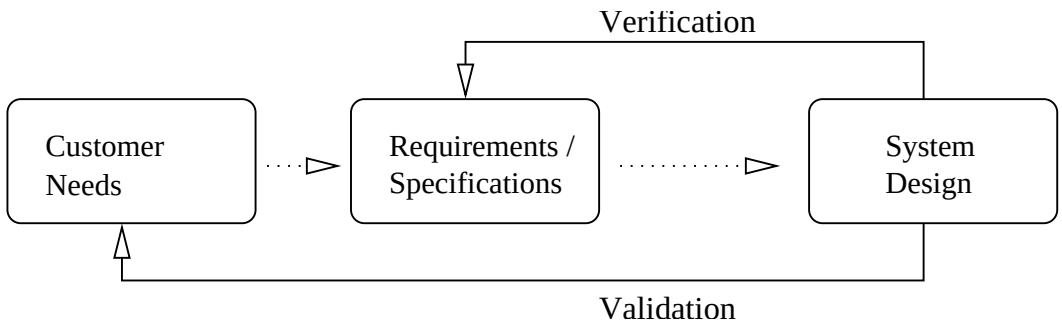
Institute for Systems Research, University of Maryland, College Park

Table of Contents

Limitations of Established Approaches to V&V

Definition and Complementary Roles

- Verification ⟶ “are we building the product right?”
- Validation ⟶ “are we building the right product?”



Limitations of Traditional Approaches

Inspection procedures and testing procedures both suffer from diminishing returns.

- The diminishing returns from testing stem from the fact that testing is good at finding some kinds of faults, and bad at finding other types of faults.
- The same can be said of inspection.

Limitations of Established Approaches to V&V

Limitations of Traditional Approaches

Fortunately, inspection and testing approaches are largely orthogonal – that is ....

> **... the effort needed to find a given fault with testing is largely unrelated to the effort needed to find the same fault with system inspection.**

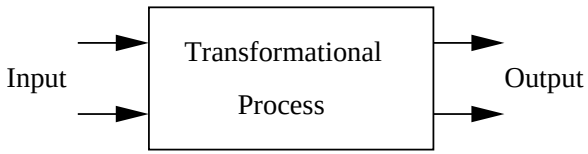
In particular:

- Faults that are hard to find with testing (e.g., faults linked to unusual situations) are sometimes much easier to find with inspection.
- Conversely, faults that are hard to find with system inspection (e.g., non-local faults) are sometimes much easier to find with testing.

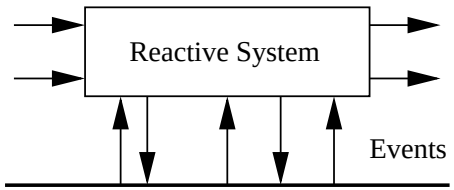
These observations suggest that system validation procedures should employ a combined strategy.

Growing Importance of Embedded Systems

Increasing Prevalence of Reactive Systems



Transformational System



Reactive (Event−based) System

Transformational Systems

- Transformational systems transfer inputs to outputs. Correctness of operation is assessed in terms of:

  .. relationships of input to output.

  Testing usually involves verification of behavior for some typical and some borderline cases. In this context, the purpose of analysis is to:

  ... prove something for all inputs.

Growing Importance of Embedded Systems

Reactive Systems

- Reactive systems run continously and respond to events in the surrounding environment.
- Correctness of operation depends not only on the logical ordering of events, but also on their timing.
- The desired run-time properties of transition systems can be represented by temporal logic.

Note ...

Traditional software development is simplified by ...

... the lack of physical constraints.

Embedded software must ...

... account for constraints of the physical domain.

Growing Importance of Embedded Systems

Research: Can we prove that a system is correct?

These difficulties have spurred research into methods that

> **...attempt to prove a system is correct, in the same sense that a mathematical theorem is proved correct.**

Ideally, an algorithm would ...

> **... analyze a system model and either conclude was correct or reveal a bug within a reasonable number of steps.**

Formal Approaches to Validation and Verification

Conceptual Mechanisms

1. **Formal Models**

   We need ways to capture the design representation and it specification in an unambiguous "formal language" that has precise semantics.

2. **Abstraction**

   Abstraction mechanism eliminate details that are of no importance when 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 sub-systems and components that can be designed and verified almost independently.

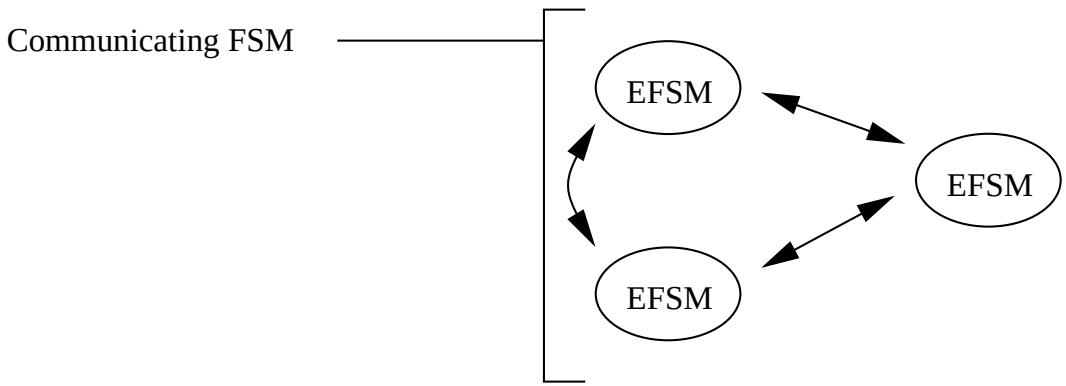
Formal Approaches to Validation and Verification

Big Idea ..

We need methods for the systematic assembly of complex systems from simpler systems and components.

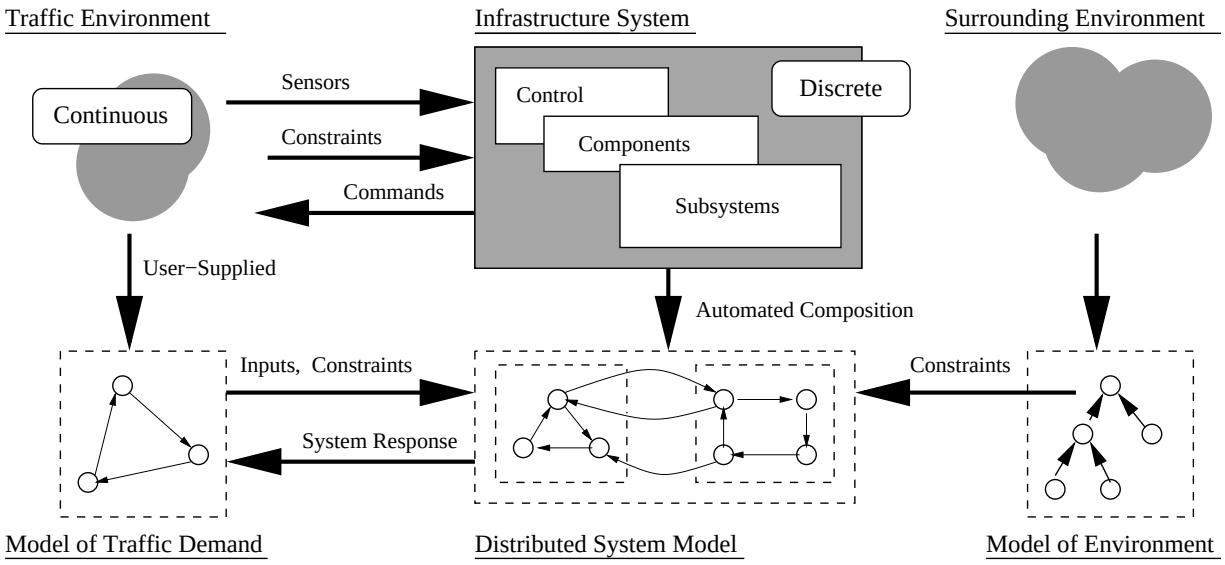
Solution Approach

Model embedded systems as networks of communicating finite state machines.

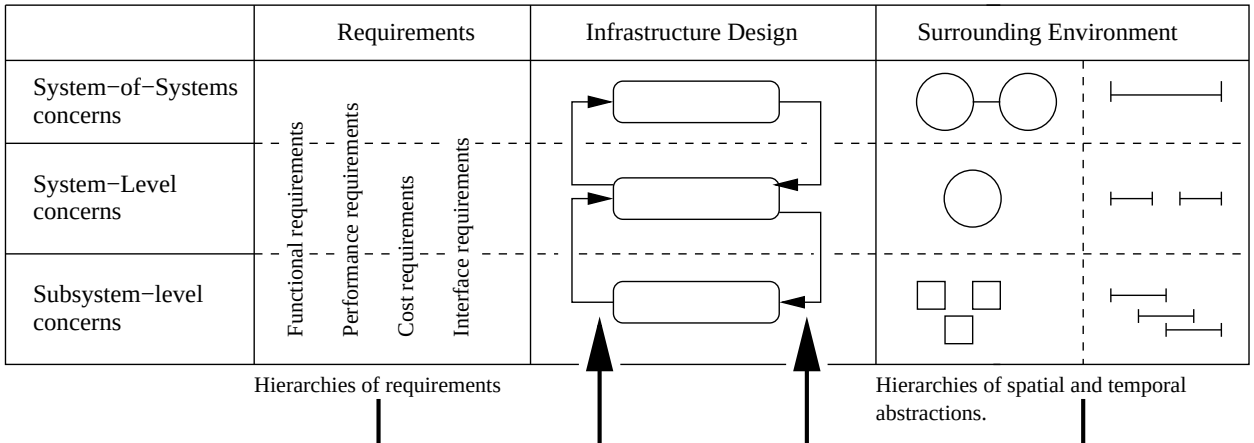


Formal Approaches to Validation and Verification

Framework for Model−Based Design



Framework for Plaform−Based Design



Formal Approaches to Validation and Verification

Representing Requirements as Properties

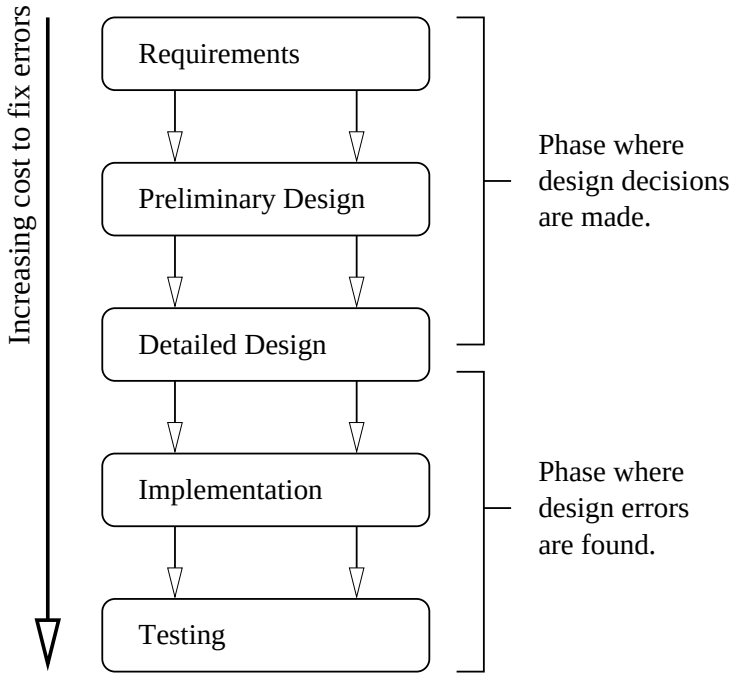
One idea is to:

> **...create formal representations of requirements in the form of "properties the system must satisfy."**

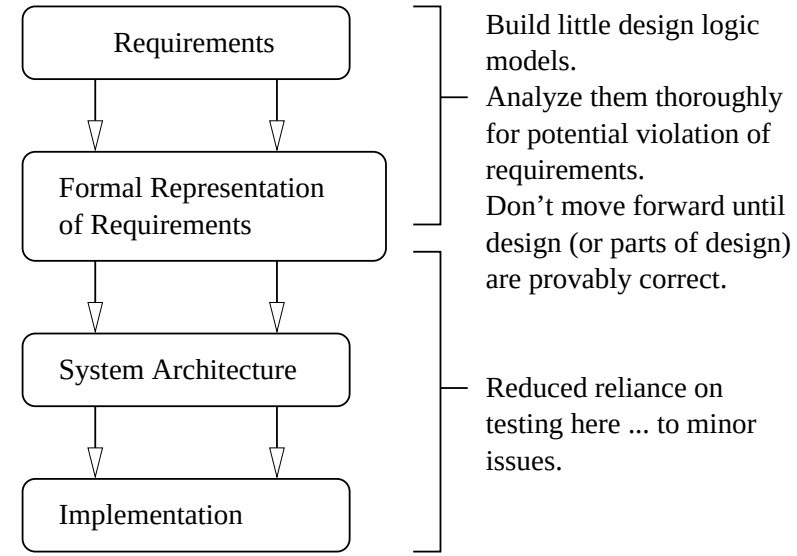
Then, models of system behavior are created in such a way that these properties are **guarenteed to hold**.

Formal Approaches to Validation and Verification

Early validation of system design (Adapted from the work of Natalia Sidorova, 2007).



Traditional Approach to Design and Test....



Early detection of errors and "system operation" that is "correct−by−construction"...

Transition Systems

Transition Systems

Transition systems (TSs) are basically directed graphs where the nodes represent states,and edges model transitions (i.e., state changes).

Mathematical Definition

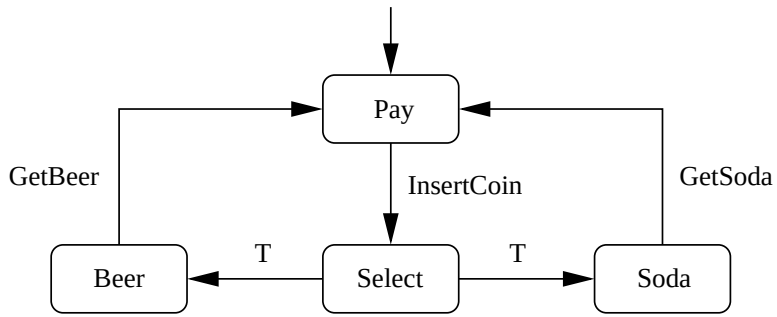
A transition system is a tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- S is a set of states.
- Act is a set of actions.
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation.
- I $\subseteq$ is a set of initial states.
- AP is a set of atomic propositions, and
- L:S $\rightarrow 2^{AP}$ is a labeling function.

TS is called finite if S, Act, and AP are finite. 2^{AP} is the powerset of AP, that is, the set of all subsets of AP.

Transition Systems

Example. Preliminary design of a simple beverage vending machine:



Points to note:

- The state space S = $\{Pay, Select, Soda, Beer\}$.
- The initial state I = $\{Pay\}$.
- The set of actions = $\{InsertCoin, GetSoda, GetBeer, \tau\}$.
- Sample transitions include:

$$Pay \xrightarrow{InsertCoin} Select \qquad Select \xrightarrow{\tau} Beer$$

Finite State Automata

Elements of Automata

Automata are ...

... simple, but useful, models of computation initially proposed as a simple model for the behavior of neurons.

Summary of Abstractions in Finite State Automata

We keep	We drop
• Some notion of state. • Stepping between states. • Start and end states.	• Notions of memory. • Variables, commands, expressions. • Syntax.

Finite State Automata

Deterministic Finite State Automata

A deterministic finite automaton (DFA) is ...

... a simple machine that reads an input string (one symbol at a time)...

and then after the input has been completely read ...

... decides whether to accept or reject the input.

Mathematical Definition (pg 38, Slind 2004).

A DFA is a five-tuple $(Q, \sum, \delta, q_o, F)$ where:

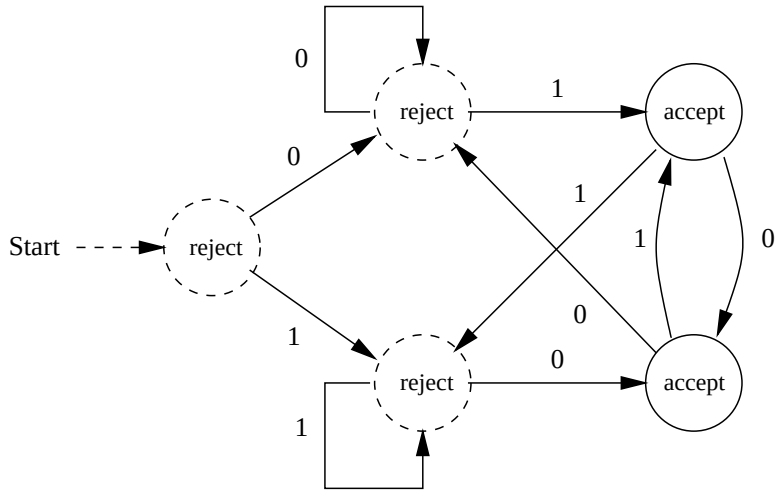
- Q is a set of states.
- $\sum$ is a finite alphabet.
- δ : Q $\times \sum \rightarrow$ Q is the transition function.
- $q_o \in$ Q is the start state.
- F $\subseteq$ Q is the set of accepting states.

The language recognized by a DFA M is denoted L(M).

Finite State Automata

Example of DFA (Source: http://www-bcl.cs.may.ie/dfa.html)

The following DFA accepts strings of 0's and 1's which end with either "01" or "10".



For instance, the strings "10", "001", "101", "110", "01010", "00110", "001001", and "0000101" are accepted,

The strings "" (the empty string), "0", "1", "00", and "0011" are rejected.

Finite State Automata

Example of DFA (Source: http://www-bcl.cs.may.ie/dfa.html)

The table of inputs, states, and transitions can be represented:

```
Start state: state 1

state 1: REJECT
   on 0 - goto state 2
   on 1 - goto state 3

state 2: REJECT
   on 0 - goto state 2
   on 1 - goto state 4

state 3: REJECT
   on 0 - goto state 5
   on 1 - goto state 3

state 4: ACCEPT
   on 0 - goto state 5
   on 1 - goto state 3

state 5: ACCEPT
   on 0 - goto state 4
   on 1 - goto state 2
```

and implemented using the State design pattern. A computer implementation will return either REJECT or ACCEPT.

Finite State Automata

Non-Deterministic Finite State Automata

Two extensions to DFA are allowed:

1. **Multiple Next States**

   Suppose that a system is in state "q" with symbol "a" – there could be multiple next states to go to. Formally, we write:

$$delta(q, a) = \{q_1, q_2\} \tag{1}$$

2. **Epsilon-Transitions**

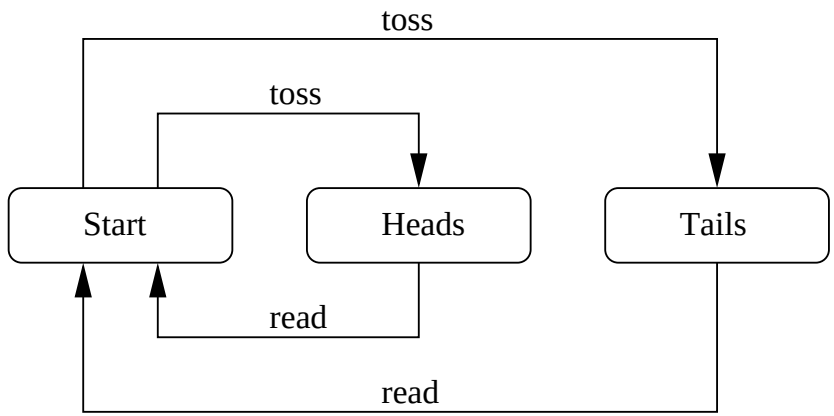
   Epsilon-transitions allow a machine to move to the next state without consuming any input. Formally, we write:

$$delta(q, \epsilon) = \{q_1, q_2\} \tag{2}$$

Finite State Automata

Example of Non-DFA: Tossing and Reading a Coin

Scenario ... Behavior Model



Set of states = $\{Start, Heads, Tails\}$; Set of actions = $\{toss, read\}$.

$$delta(Start, toss) = \{Heads, Tails\}$$

Finite State Automata

Automata Constructions

Automata constructions operate on automata yielding new automata.
Constructions are a way of building automata from components.

Types of Automata Construction

- **Product Construction**

  A product construction takes two DFAs and generates a single DFA that conceptually runs its two component machines in parallel on the same input string.

- **Concatenation Construction**

  A concatenation construction wires two DFA machines together in series.

- **Subset Construction**

  The subset construction can be used to map any NFA N to a DFA M, such that L(N) = L(M).

Automata Constructions

Properties of Automata Constructions

To be useful ...

... operations associated with automata constructions need to be precise.

General Characteristics

- Closure under union.
- Closure under intersection.
- Closure under complement.
- Closure under concatenation.

For details, see pg's 52-58 of Slind 2004.

Automata Constructions

Product Construction

A product construction takes two DFAs and generates a single DFA that conceptually runs its two component machines in parallel on the same input string.

Mathematical Definition

Let $M_1 = (Q_1, \sum, \delta, q_1, F_1)$ and $M_2 = (Q_2, \sum, \delta, q_2, F_2)$ be two DFAs.

The product of M_1 and M_2 is written $M_1 \times M_2 = (Q, \sum, \delta, q, F)$ where:

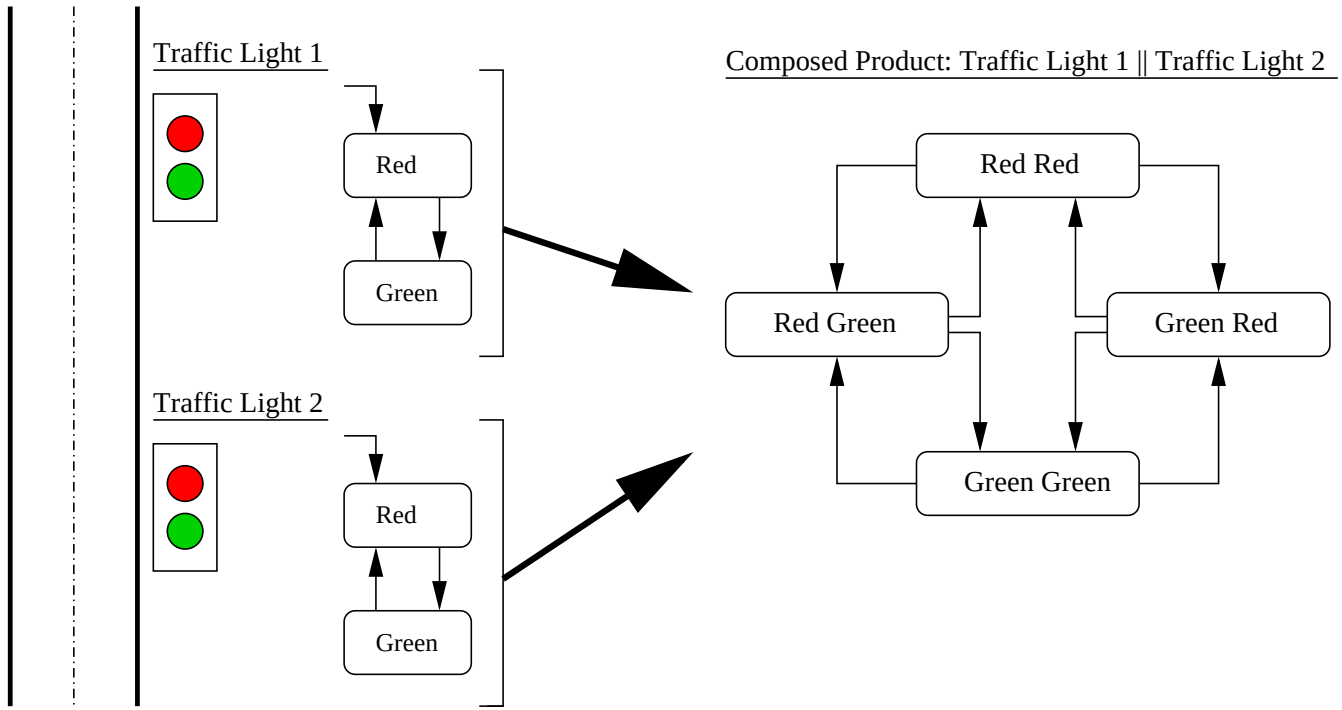
- Q = $Q_1 \times Q_2$.
- $\sum$ is unchanged.
- δ is defined by behavior on pairs of states.
- q = (q_1, q_2), the pair of initial states.
- The set of accepting states (F) is built either through a union construction or an intersection construction.

For complete details, see pg 52 of Slind 2004.

Automata Constructions

Simple Example of Product Construction

Combined behavior of two independent traffic lights ...



The parallel composition corresponds to an interleaving of the states and transitions for the individual traffic lights.

Model Checking

Mathematical Problem Formulation

Given a state-transition graph (M) and a formula (f), the model checking problem aims to decide ...

... whether the formula (f) is true for all possible runs.

In mathematical terms, model checking solves the problem

$$M, s| = f.$$

That is, ...

... find all states ”s” of ”M” such that formula ”f” holds.

If the property does not hold, then a counter example will be provided.

Model Checking

Simplified Problem Formulation

Model checking problems are composed of **variables** (V) and **transition relations** (R) as described below:

- **Variables**

  V is the set of state variables in the model. Each state variable v_i has an associated domain (D).

  A state is simply an assignment to the variables in the set V from their domain.

- **Transition Relations**

  The transition relation (R) captures all possible/valid transitions in the model.

  We say that states (s, s*) belong to R, meaning that from state "s" it is possible (or valid) to transition to state (s*).

The set of all possible states is called the **state space** of the model as is denoted "S".

Model Checking

Simplified Problem Formulation (Cont'd)

A path is a finite or infinite sequence of states

$$(s_o, s_1, s_2, s_3, ...)$$

such that states (s_i, s_{i+1}) is a valid transition.

We use **atomic properties** to describe **local properties** about states.

A basic atomic property has the form:

- **Atomic Property**

  v_i = x, or, $v_i > x$, or $v_i < x$. Here x is another variable from variables (V) or a constant from the domain (D) associated with the variable v_i.

More complicated atomic property are constructed from basic atomic properties via conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$).

Model Checking

Simple Example: Two-Variable Problem

Suppose that a simple system has two variables/domains.

```
Variable                    Domain
---------------------------------
       x                  {1,2,3}
       y                    {a,b}
---------------------------------
```

At some point in time, s_o = (x=2,y=b) could be the state of the system.

A valid transition might be written

```
( (x=2,y=b), (x=1,y=a) )
```

A valid path might look like ...

```
( (x=2,y=b), (x=1,y=a), (x=1,y=b), (x=2,y=a) )
```

Model Checking

Atomic Property for Two-Variable Problem

An atomic property might take the form:

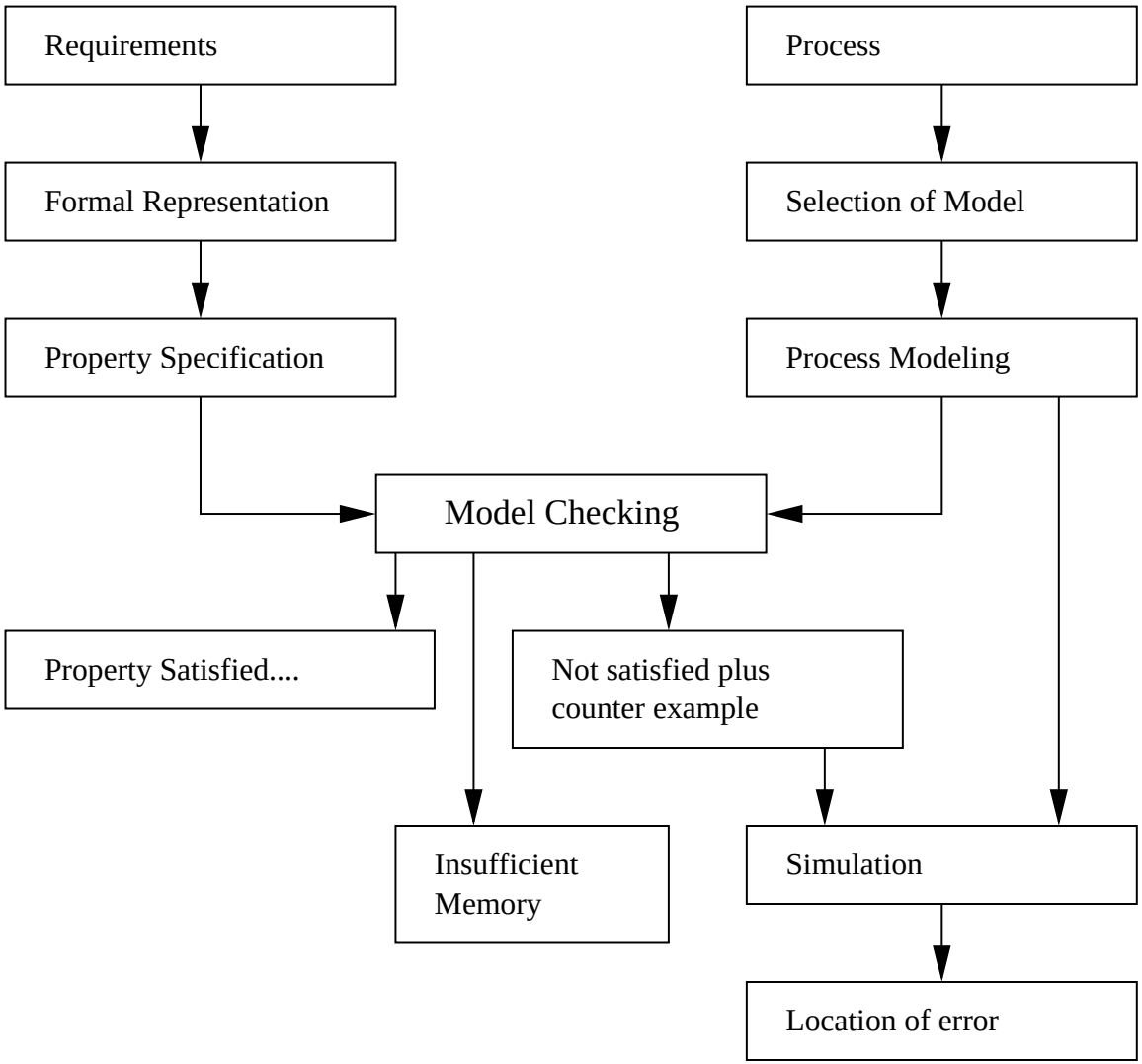
$$f = (x > 1) \wedge (y = b). \tag{3}$$

We can construct a table of states to see when property f is satisfied.

```
State      (x > 1)   (y = b)     (x > 1)^(y = b)
=============================================
(x=2,y=b)    true      true               true
(x=1,y=a)   false     false              false
(x=1,y=b)   false      true              false
(x=2,y=a)    true     false              false
---------------------------------------------
```

Clearly, only state (x=2,y=b) satisfies the property "f"...

Flowchart for Model Checking Procedures



Model Checking

Model Checking Outcomes

Three outcome are possible:

- The property specification is satisfied.
- The property specification fails.

  Modeling checking procedures will generate a counterexample, which can then be simulated to locate the source of the error.
- The model checking procedure fails because of insufficient computer memory.

Iterations of model checking continue until all of the property specification violations have been repaired.

Model Checking

Advantages of Model Checking

- No proofs!,
- Fast and automatic,
- Produces counterexamples, and
- No problem handling partial specifications.
- Can find subtle errors that might not be found by conventional testing and simulations.

Weaknesses of Model Checking

- Mainly appropriate for control-intensive applications; less suited to data-centric applications.
- Only verifies a model of the system, not the system itself.
- It only checks the stated requirements – there is no guarantee of completeness.

Model Checking

Key Technical Challenges

How to:

- Devise **algorithms and data structures** that allow us to handle large search spaces?
- Make ...

  ... design verification procedures an integral part of the design process,

  without also causing ...

  ... unacceptable delays in production or excessive cost?

Model Checking

Desirable Properties of System Behavior

We would like to design systems having properties that are guaranteed to be satisfied, including:

- **Safety**

  A safety property asserts that nothing bad happens.

- **Liveliness**

  A liveliness property asserts that **eventually** something good happens.

  A complete treatment of liveliness involves reasoning with temporal logic.

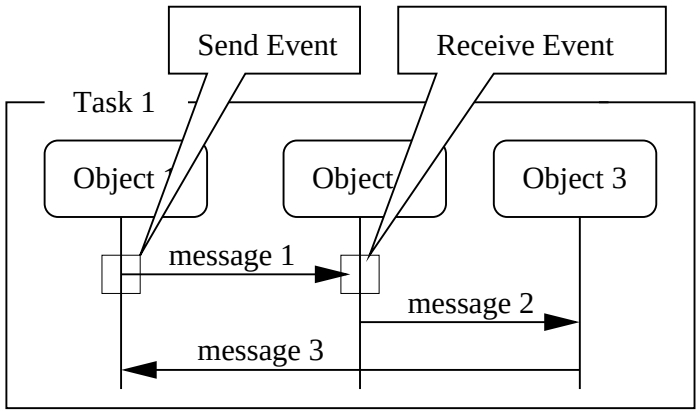
- **Progress**

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

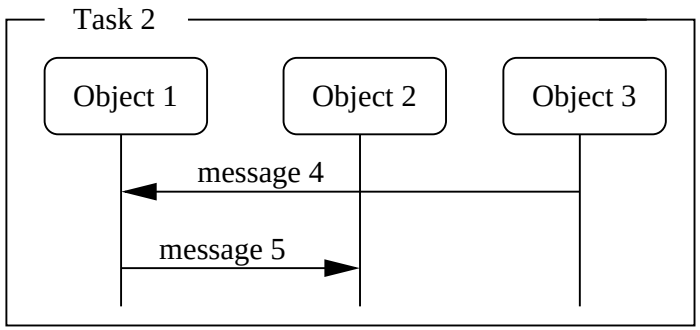
  Progress is the opposite of starvation, that is, an action is never executed. (It is also a restricted form of liveliness).

Synthesis of Automata from Sequence Diagrams

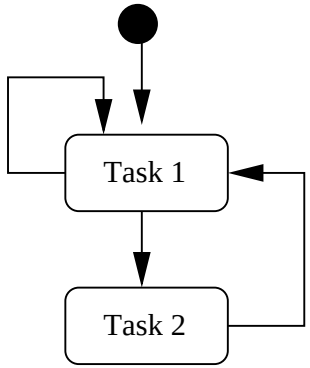
Definition of Basic- and High-Level Message Sequence Charts



Basic Message Sequence Chart for Task 1



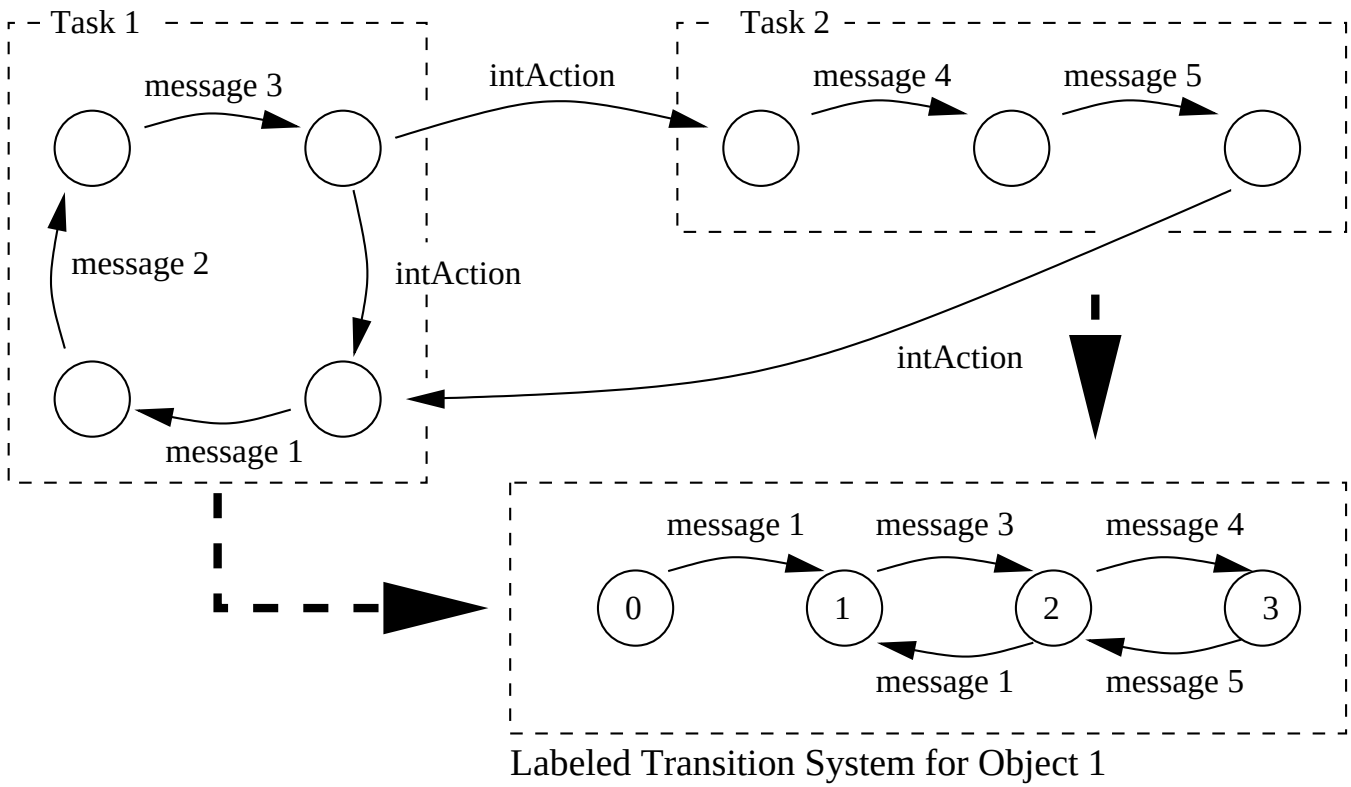
Basic Message Sequence Chart for Task 2



High−Level Message Sequence Chart

Synthesis of Automata from Sequence Diagrams

Processing Basic- and High-Level Message Sequence Charts

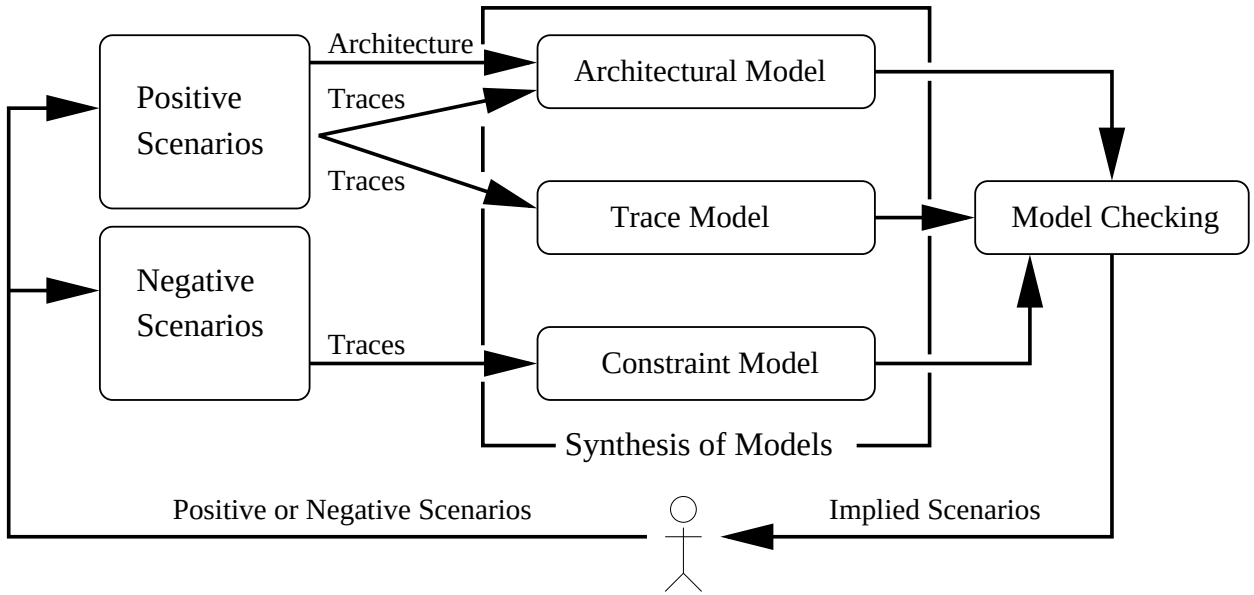


Labeled Transition System for Object 1

Incremental Synthesis of Automata/Architectures

Incremental Synthesis of Scenarios, Architectures and Constraint Models

Flowchart for incremental synthesis of positive and negative scenarios, architecture, trace and constraint models:



Source: S. Uchitel (2003).

Case Study: Management of Narrow Passageways

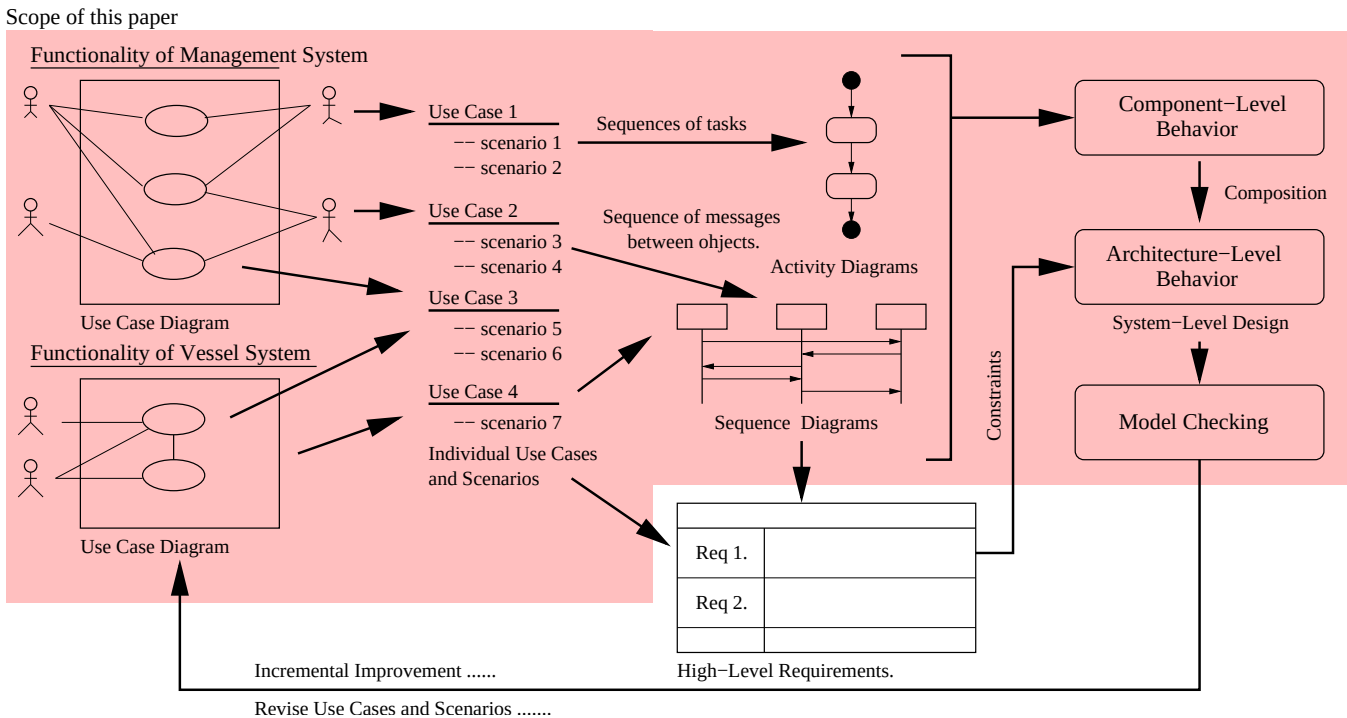
Problem Statement

Efficient management of the world's narrow passageways is needed to relieve congestion, and facilitate global transportion of goods, e.g., Panama Canal:

Case Study: Management of Narrow Passageways

Step-by-Step Procedure

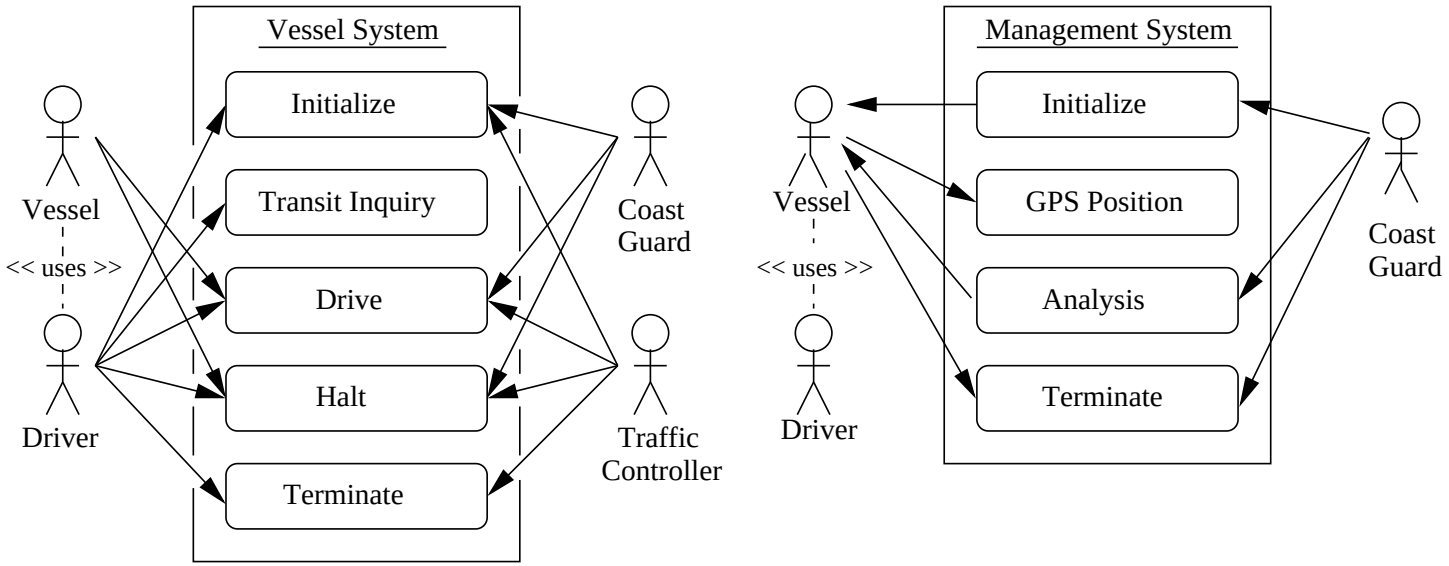
Step-by-step procedure for synthesis and validation of concurrent bbject-based models for management of narrow passageways.



Case Study: Management of Narrow Passageways

Case Study: Statement of Functionality

Usecase diagrams for the vessel and waterway management systems

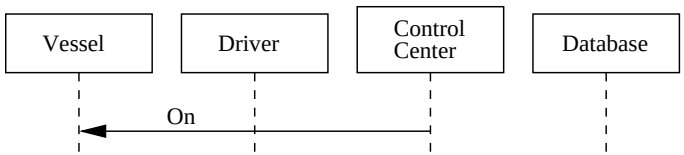


Case Study: Management of Narrow Passageways

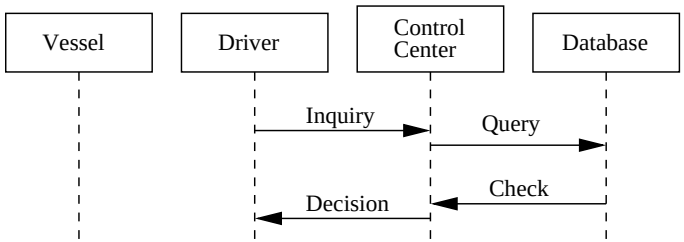
Basic- and high-level message sequence chart (MSC) specifications for waterway system functionality.

Basic Message Sequence Charts

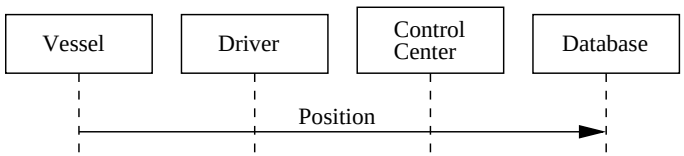
Initialize



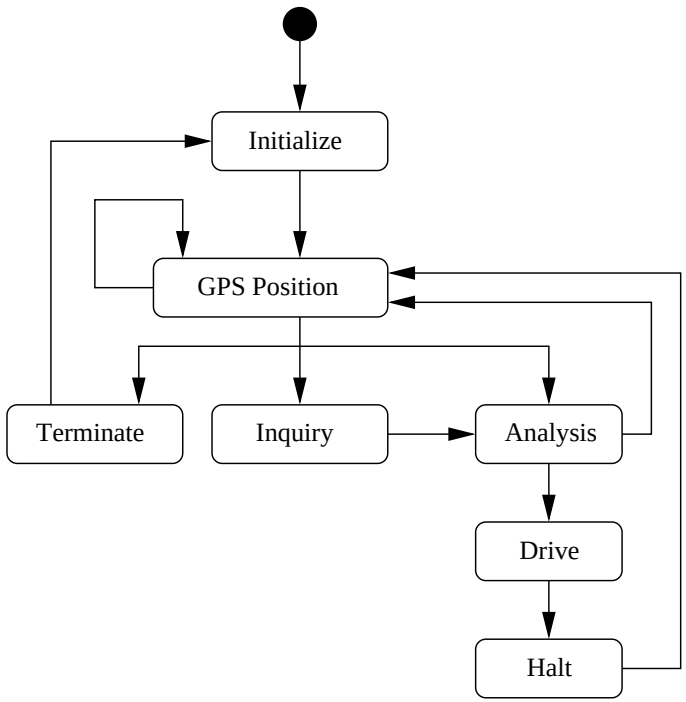
Transit Inquiry



GPS Position



High−Level Message Sequence Chart

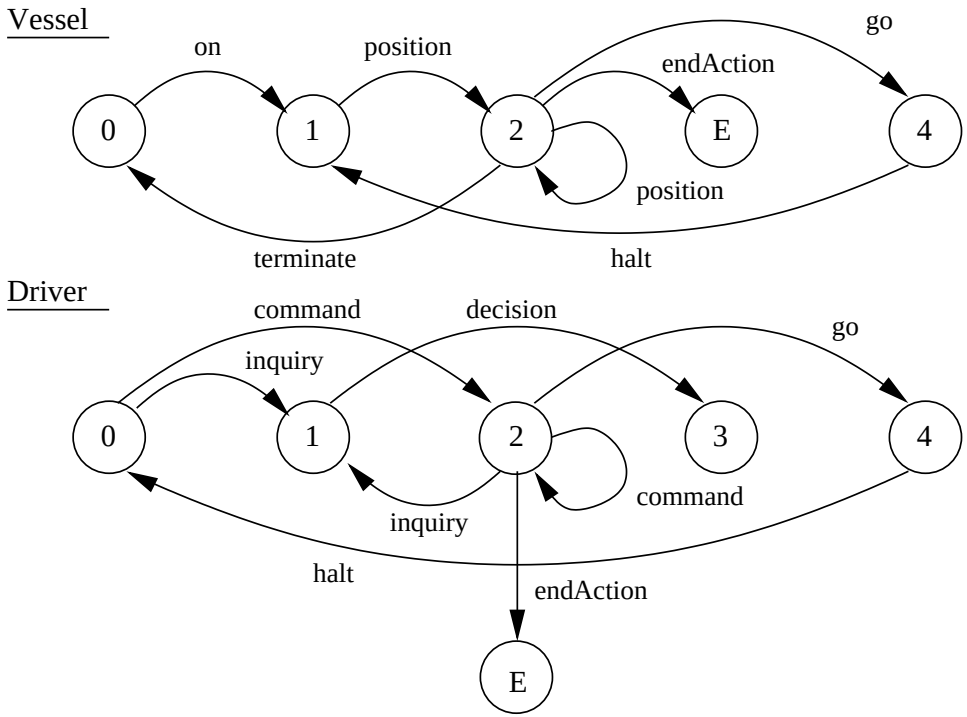


Note. Similar Sequence Message Charts can be drawn for use cases: Analysis, Drive, Halt and Terminate.

Case Study: Management of Narrow Passageways

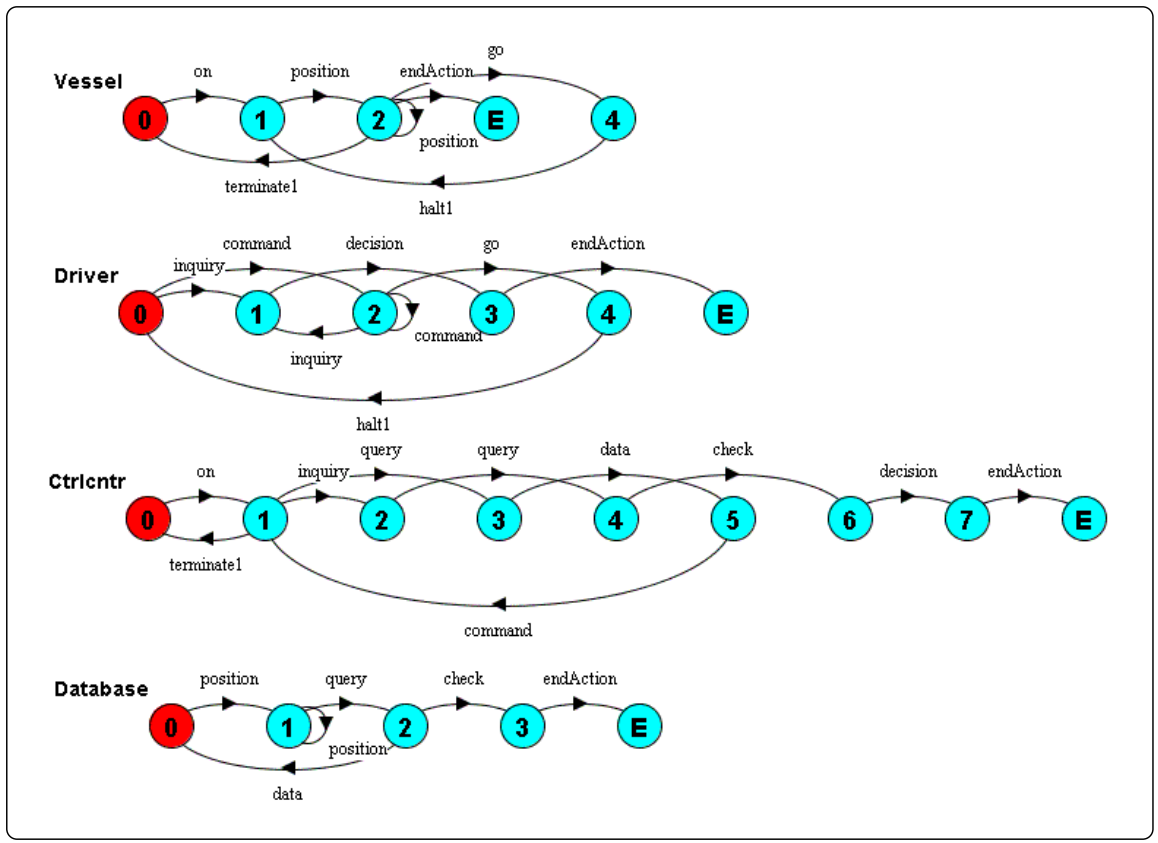
Case Study: Behavior Modeling for Management of Narrow Waterways

Component models for vessel and driver behavior ...



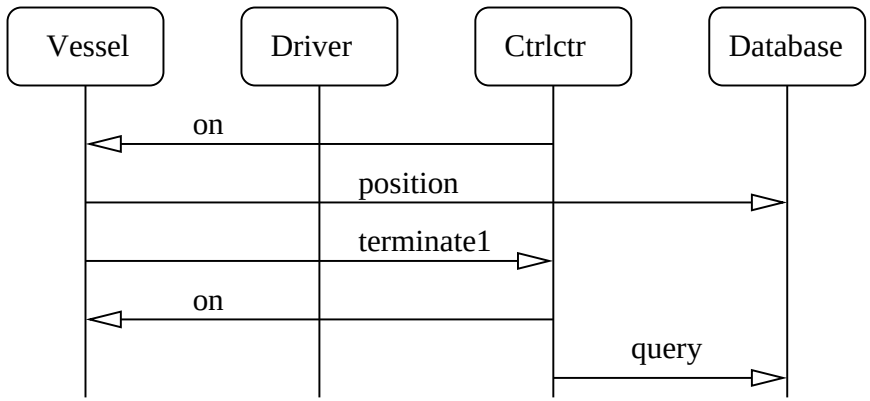
Case Study: Management of Narrow Passageways

Synthesis of Behavior Models in LTSA. Details to follow...

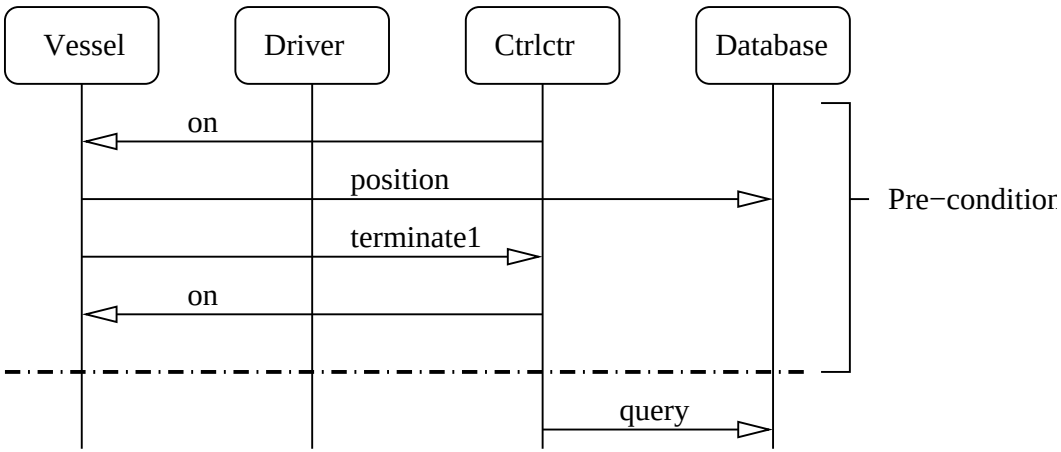


Case Study: Management of Narrow Passageways

Implied Scenario in Model of Waterway Behavior



Basic Negative Scenario

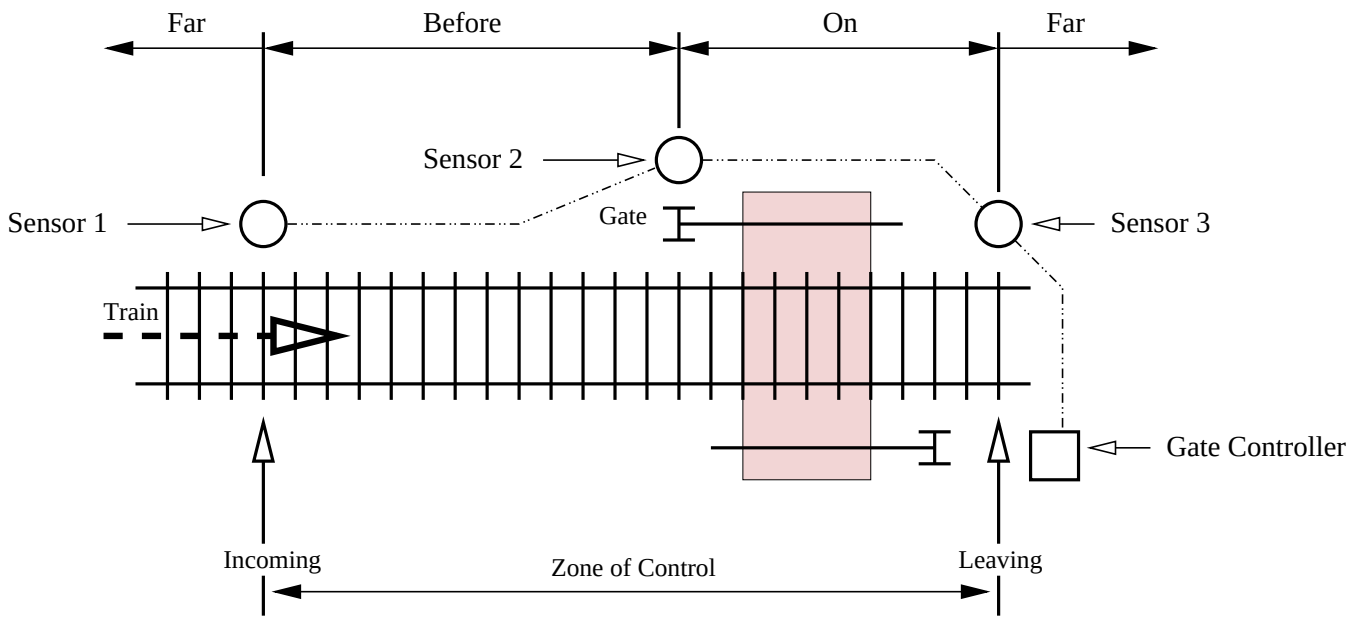


Generalized Railroad Crossing Problem

A set of trains travel through a region R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R.

Overly Simplified Problem Setup

Only one track; only one train; details of time and train velocity omitted.

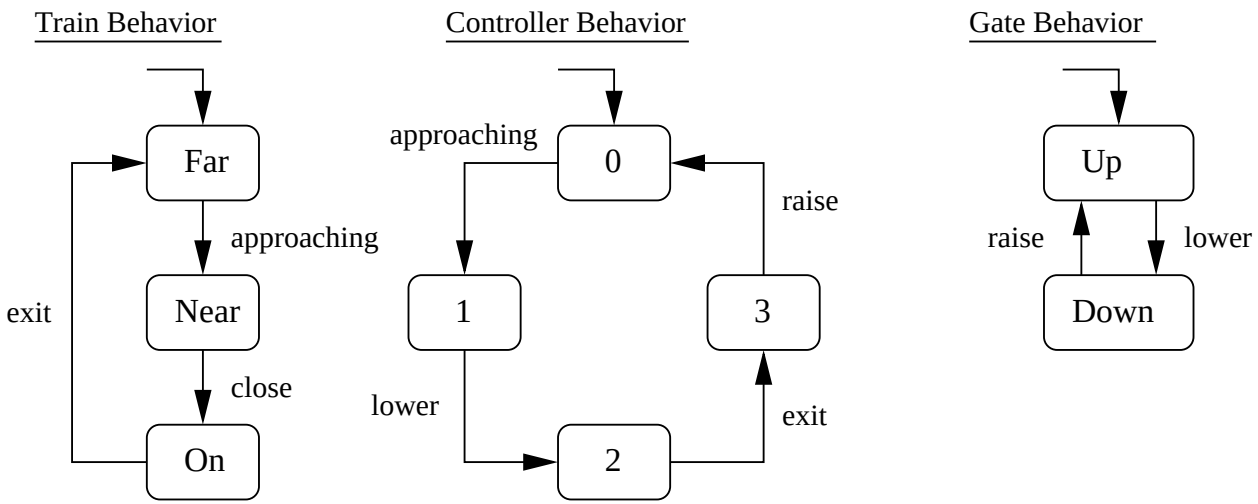


Generalized Railroad Crossing Problem

Events and Actions

- Sensor 1 registers a train → Generate event: approaching
- Sensor 2 registers a train → Generate event: close
- Sensor 3 registers a train → Generate event: exit

Automata for Train, Gate and Controller Behavior



Generalized Railroad Crossing Problem

Composition of System-Level Behavior

$$\text{System Behavior} = (Train||Gate||Controller)$$

Desirable Properties of System Behavior

Given two "positive tolerance" constants ξ_1 and ξ_2, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

1. **Safety Property**

   The gate is down during all occupancy intervals.

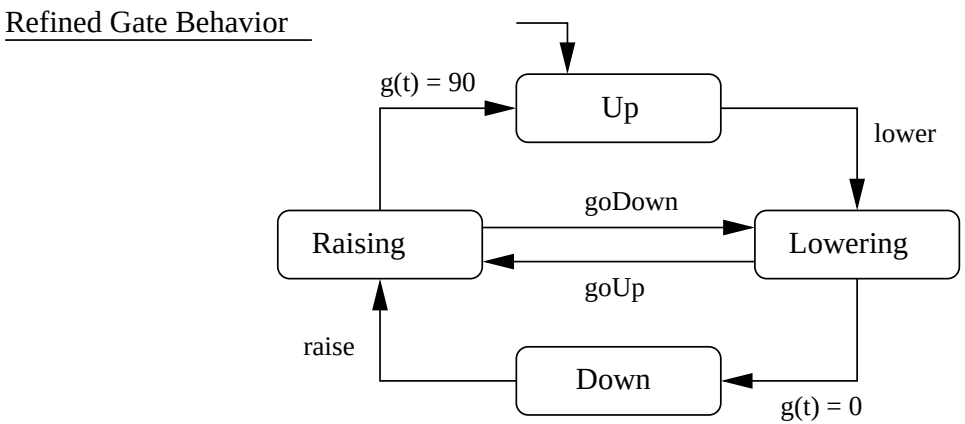
2. **Utility Property**

   If t is not in any occupancy interval, nor within ξ_1 prior to an occupancy interval, nor within ξ_2 after an occupancy interval, then the gate is up.

Note. The gate is a physical system that requires time to change states (e.g., Up to Down).

Generalized Railroad Crossing Problem

Refined Model for Gate Behavior



To describe the system formally, we define:

- A gate function g from real times to the interval [0,90], where:
  - g(t)=0 means the gate is down, and
  - g(t)=90 means the gate is up.
- A sequence of “occupancy intervals” where each occupancy interval is a maximal time interval during which one or more trains are in I.

Generalized Railroad Crossing Problem

Length of the Train

- Each sensor will measure the absense or lack thereof of a train, and will be activated by arrival (i.e. the front of the train), and deactivated by departure (i.e., the rear) of the train.
- Thus, the sequencing of sensor operations will be affected by the length of the train.
- A practical way of handling this is to validate the gate operations for three lengths of train:
  - A really short train,
  - A medium length train, and
  - A really long train.

Questions

- What would the refined controller look like?
- What happens if two trains come?

References

- Kaisar E., Austin M.A., and Papadimitriou S., Formal Development and Evaluation of Narrow Passageway System Operations, European Transport/Transporti Europei , Vol. 34, December 2006, pp. 88-104.
- 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, September 2007, pp. 373-378.
- Sangiovanni-Vincentelli A., McGeer P.C., Saldanha A., Verification of Electronic Systems : A Tutorial, Proceedings of the 33rd Desgin Automation Conference, Las Vegas, 1996.
- Sidorova N., Lecture Notes in Process Modeling, Department of Mathematics and Computer Science, Eindhoven University, Netherlands, 2007.
- Slind K., Class Notes on Theory of Computation, Department of Computer Science, University of Utah, October 2004.