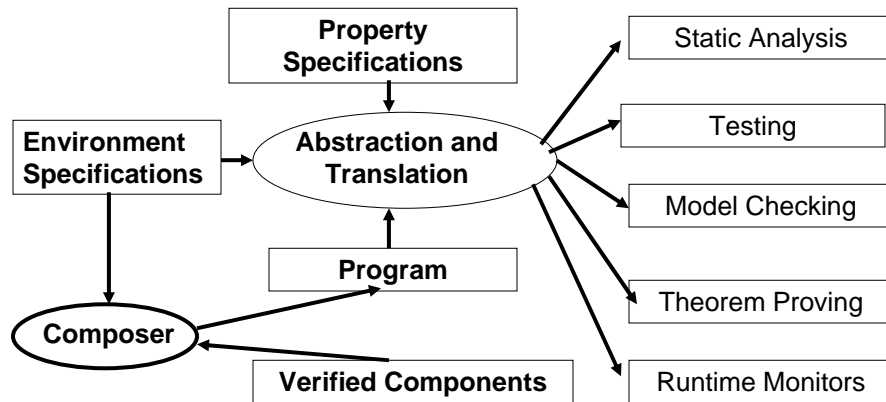


Approach



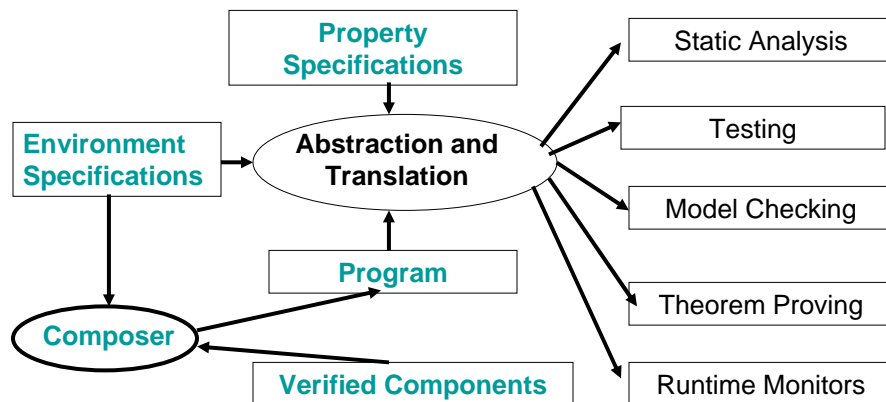
Abstraction and Translation is implemented through static analysis

9/10/2007

Design For Verification

1

Approach: Design for Verification

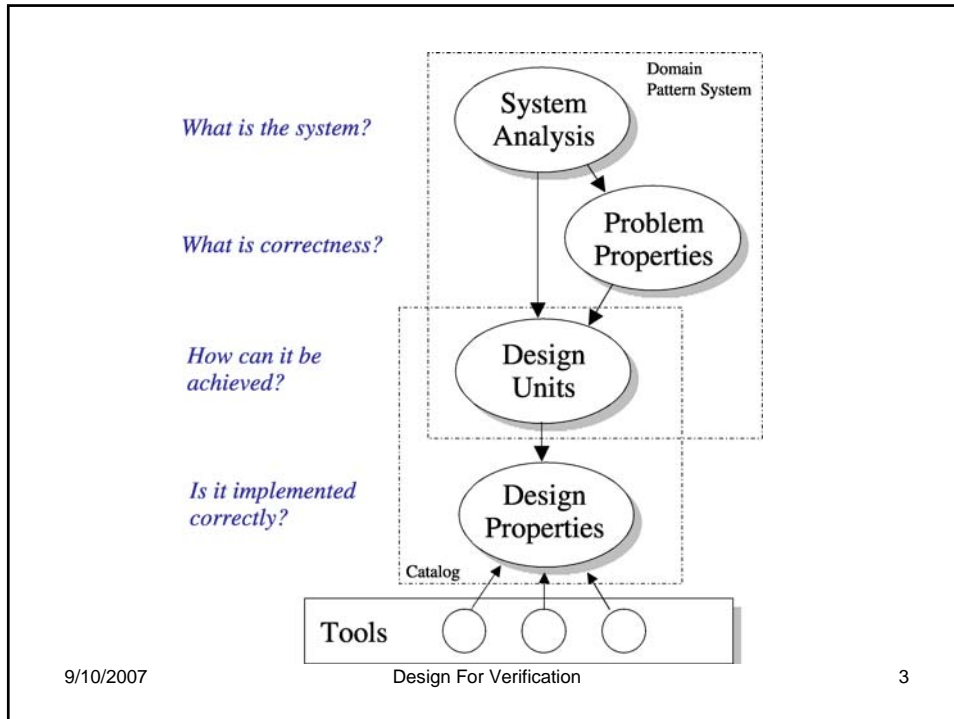


Abstraction and Translation is implemented through static analysis

9/10/2007

Design For Verification

2



Design For Verification

- Design Goals
 - Functionality
 - Robustness
 - Performance
 - Security
 - Verifiability
- Design for verifiability is required for the other goals to be meaningful
- Verification has no hope for success unless the program is designed with verification as a goal.
- We need a design process leading to verifiable systems.

What Makes a System Difficult to Verify?

- Complexity of structure
- Size of units to be verified
- Non-local state
- Obscure specification of state
- Distribution of functionality across multiple units.

9/10/2007

Design For Verification

5

What Design Characteristics Make a System Verifiable?

- Precisely Specified Structure
- Separation of functional concerns
- Localized state
- Verifiable components

9/10/2007

Design For Verification

6

Development for Verification

- Formal architectural specification for system
 - Components and relationships among components
- Formal specification of components
 - Unit of execution with specifiable properties.

9/10/2007

Design For Verification

7

Why Components?

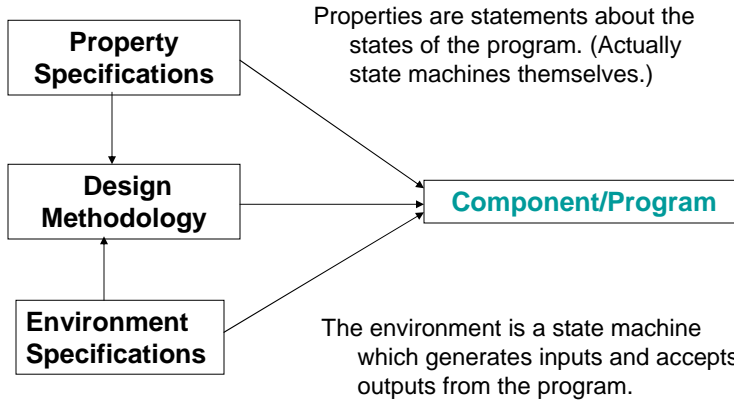
1. Components provide a semantic basis for definition of properties and assumptions.
2. Properties can be established on components under assumptions which model compositions and execution environments.
3. The components can then be replaced in verifications of compositions by an adequate set of established properties.
4. Exhaustive analysis and/or testing is sometimes possible on a component by component basis.
5. Components provide a basis for larger semantic units for monitoring and definition of redundancy.
6. Patterns of components enable definition of properties to be defined and assumptions for verification of properties

9/10/2007

Design For Verification

8

Design Methodology

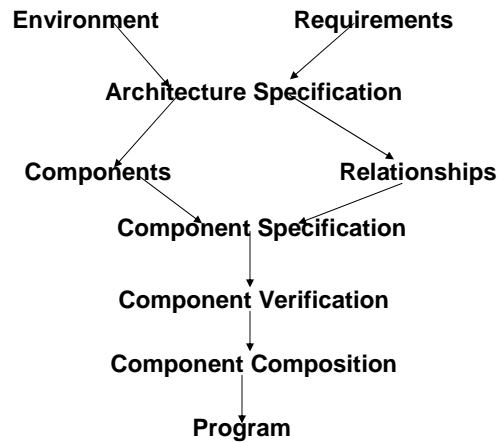


9/10/2007

Design For Verification

9

Design/Development Flow



9/10/2007

Design For Verification

10

Architecture Definition

1. Architectures/Templates/Patterns – An architecture is a composition of components implementing an input/output specification and conforming to a set of properties.
2. An architecture is often defined in terms of layers of functionality/abstraction.
3. There are architectures, templates or patterns associated with each level of abstraction.
4. Each layer/level has a set of components.
5. An instance of the architecture is defined by a choice of execution environment and initialization.
6. Components are either selected from libraries or developed for a specific application.

9/10/2007

Design For Verification

11

Components - Definition

1. A component is a functional specification, a set of properties and an implementation.
2. A component provides one or more logical functions at the level of abstraction at which it is defined.

9/10/2007

Design For Verification

12

Components - Design

1. Design (and development) begins with writing functional specifications and deriving properties for the component from the functional specifications.
2. The control flow of a component is defined by a state machine.
3. The actions associated with each state (Moore) or transition (Mealy) must be "run to completion".
4. The states in the properties of the component must be explicitly visible in the component/program design.
5. Transitions among states in the property specifications should be explicitly visible in the component/program design.
8. Data which is shared among a set of components should be encapsulated in a component.
9. The inputs and outputs sequences (environment) for a component should be specifiable as a state machine.

9/10/2007

Design For Verification

13

Relationships - Definition

- Connector is a word often used in the software architecture literature to denote the relationship among components in an architecture.
- Connectors may be defined explicitly in a language or implicitly through matching of component specifications
- An interaction is an execution of a connector/relationship
- Examples:
 - Procedure invocations
 - Continuations – data flow
 - Callbacks

9/10/2007

Design For Verification

14

Top Down Phase of Design

1. Specify the environment in which the system will be defined. The specification may be in terms of message sequence charts, state machines, or temporal logic properties.
2. Specify the required properties for the system in an appropriate temporal logic.
3. Specify layers of abstraction within which components may be defined. (There may be only a single layer.)
4. Specify components within each layer.
5. Specify the relationships (interactions or connectors) within each layer and across layers as “event streams,” dependence graphs and/or call/return invocations including sequencing of interactions in the relationship. (Steps 2, 3, 4 and 5 define a classical software architecture.)
6. Derive those properties of components which are required as a result of system properties and the relationships among components.

9/10/2007

Design For Verification

15

Top Down Phase of Design

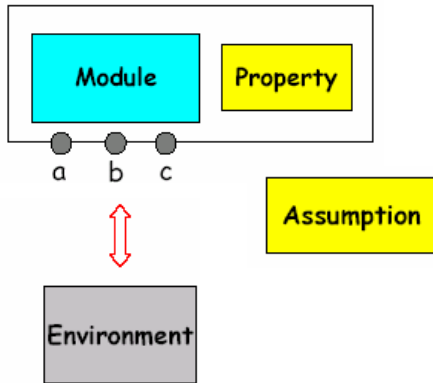
7. Identify the sources of components in libraries and/or as application specific.
8. The properties of a component are the union of the properties derived with the functionality of the component and the requirements imposed by the architecture.
9. Verify the properties of the “primitive” components using the architecture to derive the environment of the “primitive” components.
10. Use the architecture to determine pairs or sets of components which can or should be composed into components visible in the architecture.
11. Follow the procedure given following to compose components into larger components.

9/10/2007

Design For Verification

16

Component Specification and Verification



Modules may require context
information to satisfy a
property

$\text{Assumption} \parallel \text{Module} \Rightarrow$
 Property

*(assume/guarantee
reasoning)*

**How are assumptions
obtained?**

Developer encodes them

Abstractions of environment,
if known

Automatically generate *exact*
assumption A

9/10/2007

Design For Verification

17

Component Composition Phase Overview

1. Verify properties with respect to environment derived from architecture
2. Compose components at lower levels of abstraction following architecture and conformance of properties to get components at higher levels of abstraction

9/10/2007

Design For Verification

18

Component Composition Phase

1. Construct architecture with components and relationships following process given previously.
2. Implement components following the process given previously.
3. Use the architecture to define the environment for verification of component properties.
4. Verify the properties of the components by appropriate methods.

9/10/2007

Design For Verification

19

Component Composition Phase

5. Use architecture to identify compositions of components.
6. Compose components into “larger” components including derivation of specifications and properties.
7. Continue steps 5 and 6 until architecture is realized.

9/10/2007

Design For Verification

20

Comments

- Architectures may be designed from scratch or be examples of patterns
- Architectures are nearly always implemented with a mixture of library and application specific components.
- Incorporation of components from libraries requires special consideration since specifications, source and properties may not be available.

9/10/2007

Design For Verification

21

Example – Simple Microwave Oven

Microwave Oven Specification

This simple oven has a single control button. When the oven door is closed and the user presses the button, the oven will cook (that is, energize the power tube) for 1 minute.

There is a light inside the oven. Any time the oven is cooking, the light must be turned on, so you can peer through the window in the oven's door and see if your food is bubbling. Any time the door is open, the light must be on, so you can see your food or so you have enough light to clean the oven.

When the oven times out (cooks until the desired preset time), it turns off both the power tube and the light. It then emits a warning beep to signal that the food is ready.

The user can stop the cooking by opening the door. Once the door is opened, the timer resets to zero.

Closing the oven door turns out the light.

9/10/2007

Design For Verification

22

Properties

- If the door is open, the tube is not on or If the Door is open then the Tube is off
- If the button is pushed while the Door is closed the Light and the Tube will turn on.
- If the Door is open then the Light is on
- If the Light is on then the Door is open or the Tube is on

9/10/2007

Design For Verification

23

State Table for Microwave Oven

	buttonPressed	doorOpened	doorClosed	timerTimesOut
Ready To Cook	Cooking	Door Open		
Cooking	<i>Cooking Extended</i>	Cooking interrupted		Cooking Complete
Cooking Complete	Cooking	Door Open		
Cooking Interrupted			Ready To Cook	
Door Open			Ready To Cook	
Cooking Extended	<i>Cooking Extended</i>	<i>Cooking Interrupted</i>		<i>Cooking Complete</i>

9/10/2007

Design For Verification

24

State Table for Microwave Oven

	buttonPressed	doorOpened	doorClosed	timerTimesOut
Ready To Cook	Cooking	Door Open	<i>Can't Happen</i>	Event Ignored
Cooking	Cooking Extended	Cooking Interrupted	<i>Can't Happen</i>	Cooking Complete
Cooking Complete	Cooking	Door Open	Can't Happened	<i>Can't Happen</i>
Cooking Interrupted	Event Ignored	<i>Can't Happen</i>	Ready To Cook	Event Ignored
Door Open	Event ignored	<i>Can't Happen</i>	Ready To Cook	<i>Can't Happen</i>
Cooking Extended	Cooking Extended	Cooking Interrupted	<i>Can't Happen</i>	Cooking Complete

9/10/2007

Design For Verification

25

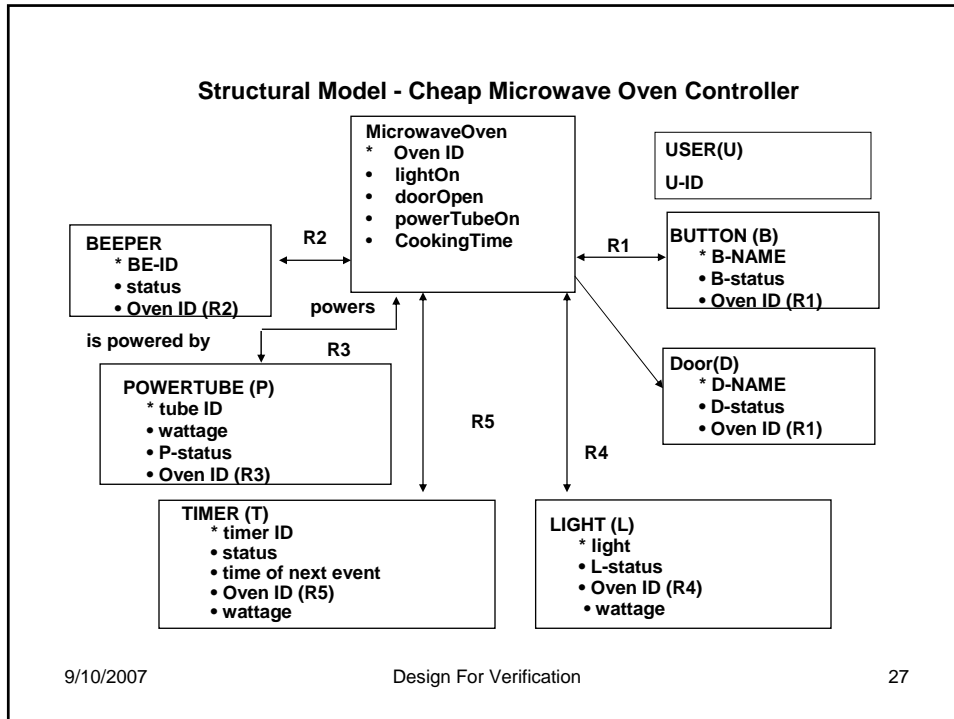
Components

- Door (open, closed)
- Power tube (on, off)[wattage, etc]
- Light (on, off) [wattage, etc]
- Timer (on and counting down, off and zero)
- Beeper (on, off)
- Button – Event generator

9/10/2007

Design For Verification

26



Component Composition Detail

- Assume we have components where we have the requirements specifications and the implementations.
- Assume we have derived properties for the components.
- Assume we have chosen an execution model for the system.

Component Model for Verification

- A component, C , is a four-tuple, (E, I, V, P) :
 - E is an executable representation of C ;
 - I is an interface through which C interacts with other components;
 - V is a set of variables defined in E and references by properties defined in P ;
 - P is a set of properties defined on I and V , and have been verified on E .

9/10/2007

Design For Verification

29

Component Property

- A temporal property in P is a pair, $(p, A(p))$,
 - p is a temporal formula defined on I and V ;
 - $A(p)$ is a set of temporal formulas defined on I and V
- p holds on C if $A(p)$ holds on the environment of C .
- The environment of C
 - is the set of components that C interacts with
 - varies in different compositions.

9/10/2007

Design For Verification

30

Component Composition

- (E, I, V, P) from (E_i, I_i, V_i, P_i) , $0 \leq i < n$:
 - E is derived by connecting E_0, \dots, E_{n-1} via their interfaces.
 - I is derived from I_0, \dots, I_{n-1} : An operation in I_i , $0 \leq i < n$, is included in I iff it is used as C interacts with environment.
 - V is a subset of $\cup V_i$, $0 \leq i < n$. A variable in $\cup V_i$ is included in V iff it is referenced by the properties defined in P .
 - P is a set of temporal properties defined on I and V , and verified on E by utilizing the properties in P_0, \dots, P_{n-1} .

9/10/2007

Design For Verification

31

Asynchronous Message-passing Interleaving Model (AIM)

- A component is a process
- A system consists of a set of processes.
- Processes interact through asynchronous message passing.
- At any moment, only one process executes.

9/10/2007

Design For Verification

32

Instantiation of Component Model on AIM Computation Model

- A component, $C = (E, I, V, P)$:
 - E is an executable representation of C with semantics conforming to the AIM model,
 - E.g., A model in xUML, an executable dialect of UML;
 - I is a messaging interface and a pair, (R, S) :
 - R is set of input message types;
 - S is set of output message types.

9/10/2007

Design For Verification

33

AIM Instantiation of Component Composition

- (E, I, V, P) from $(E_i, I_i, V_i, P_i), 0 \leq i < n$
 - Deriving E from E_0, \dots, E_{n-1}
 - mapping output message types in S_0, \dots, S_{n-1} to input message types in R_0, \dots, R_{n-1} ;
 - Deriving I from I_0, \dots, I_{n-1}
 - An output (or input) message type in $\cup R_i$ (or $\cup S_i$) is included in R (or S) if it may be used when C interacts with its environment.

9/10/2007

Design For Verification

34

Case Study: TinyOS

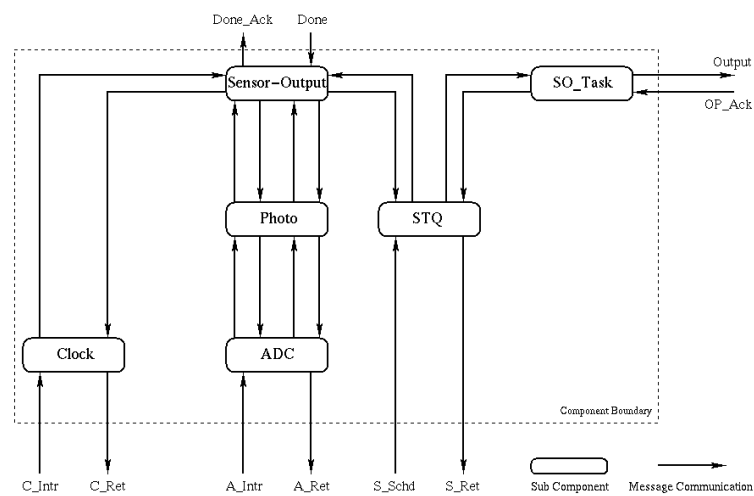
- A run-time system for network sensors;
- Is component-based
 - Requirements of different network sensors differ;
 - Physical limitations of network sensors;
- Requires high reliability
 - To support concurrency-intensive operations;
 - Loaded to a large number of network sensors.

9/10/2007

Design For Verification

35

Sensor Component



9/10/2007

Design For Verification

36

Sensor Component (cont.)

- The messaging interface, I:
 - $R = \{C_Intr, A_Intr, S_Schd, OP_Ack, Done\}$;
 - $S = \{C_Ret, A_Ret, S_Ret, Output, Done_Ack\}$;
- The set, V, of referenced variables:
 - $\{ADC.Pending, STQ.Empty\}$.

9/10/2007

Design For Verification

37

Sensor Component (cont.)

Properties:

Repeatedly (Output);
After (Output) Never (Output) UntilAfter (OP_Ack);
After (Done) Eventually (Done_Ack);
Never (Done_Ack) UntilAfter (Done);
After (Done_Ack) Never (Done_Ack) UntilAfter(Done);

Assumptions:

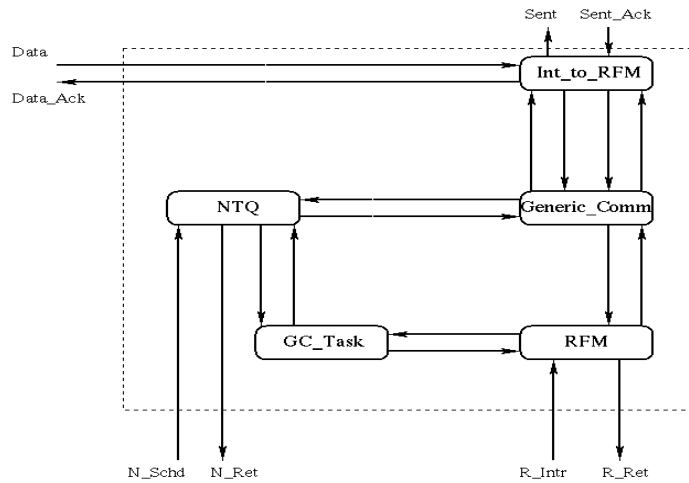
After (Output) Eventually (OP_Ack);
Never (OP_Ack) UntilAfter (Output);
After (OP_Ack) Never (OP_Ack) UntilAfter (Output);
After (Done) Never (Done) UntilAfter (Done_Ack);
Repeatedly (C_Intr);
After (C_Intr) Never (C_Intr + A_Intr + S_Schd) UntilAfter (C_Ret);
After (ADC.Pending) Eventually (A_Intr);
After (A_Intr) Never (C_Intr + A_Intr + S_Schd) UntilAfter (A_Ret);
After (STQ.Empty = FALSE) Eventually (S_Schd);
After (S_Schd) Never (C_Intr + A_Intr + S_Schd) UntilAfter (S_Ret);

9/10/2007

Design For Verification

38

Network Component



9/10/2007

Design For Verification

39

Network Component (cont.)

Properties:

IfRepeatedly (Data) Repeatedly (RFM.Pending);
 IfRepeatedly (Data) Repeatedly (Not RFM.Pending);
 After (Data) Eventually (Data_Ack); Never (Data_Ack) UntilAfter (Data);
 After (Data_Ack) Never (Data_Ack) UntilAfter (Data);
 After (Sent) Never (Sent) UntilAfter (Sent_Ack);

Assumptions:

After (Data) Never (Data) UntilAfter (Data_Ack);
 After (Sent) Eventually (Sent_Ack); Never (Sent_Ack) UntilAfter (Sent);
 After (Sent_Ack) Never (Sent_Ack) UntilAfter (Sent);
 After (NTQ.Empty = FALSE) Eventually (N_Schd);
 After (N_Schd) Never (N_Schd +R_Intr) UntilAfter (N_Ret);
 After (RFM.Pending) Eventually (R_Intr);
 After (R_Intr) Never (N_Schd +R_Intr) UntilAfter (R_Ret);

9/10/2007

Design For Verification

40

9/10/2007

Design For Verification

41