

Engineering **Processes** that Engineer **Scalable Systems**

Leon J. Osterweil (ljo@cs.umass.edu)
University of Massachusetts Amherst

<http://laser.cs.umass.edu>

Key collaborators:

Lori A. Clarke

George Avrunin

Forrest Shull

Software at Scale Workshop
Berkeley, 18-19 August 2010

Systems that scale

Systems that scale

- Some Challenges
 - Size
 - Complexity
 - Speed
 - Human Intensiveness
 - Security
 - Composability
 - Evolvability

Systems that scale

- Some Challenges
 - Size
 - Complexity
 - Speed
 - Human Intensiveness
 - Security
 - Composability
 - Evolvability
- Some Process Approaches
 - Scrum
 - Test first
 - Team development
 - Pair programming
 - Daily build
 - Spiral Model
 - Continuous Integration

Some Obvious Questions

- What do these process *labels* mean?
- What properties does each one have?
- What are the various processes good for, not good for?
- How to use these to
 - Select appropriate processes
 - Compose and configure them

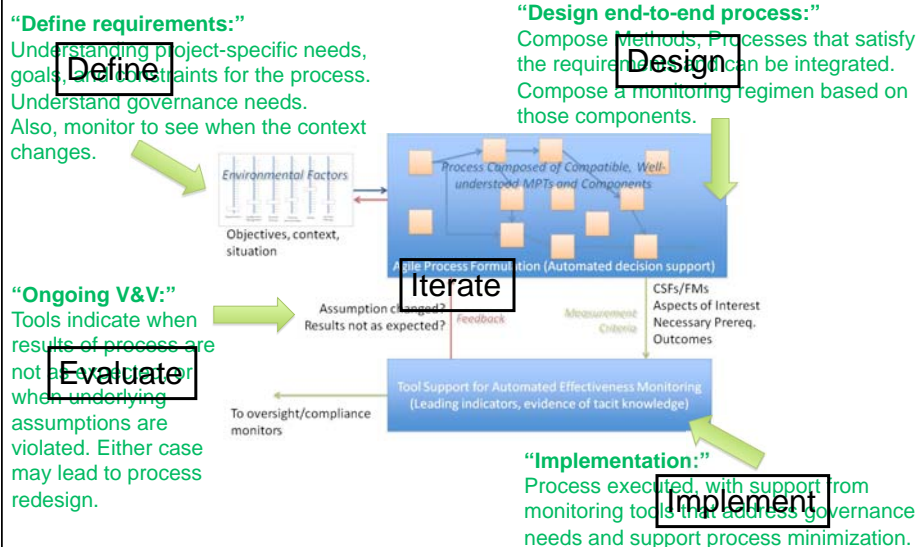
Which (combinations of)
process approaches
meet

Which (combinations of)
needs?

Needed: A Discipline of Process Engineering

- Define processes rigorously
 - Needed: Appropriate process languages
- Create process variants, customizations, syntheses
 - Needed: Support for process composition and synthesis
- Select and compose processes
 - Needed: Process evaluation and suitability measures
- Reason about process definitions to infer properties
 - Needed: Effective process analyzers (static and dynamic)
- Improve processes systematically and archive them
 - Needed: A continuous system process improvement environment integrating all of the above

Process Engineering: Treating Processes Like Software and Systems

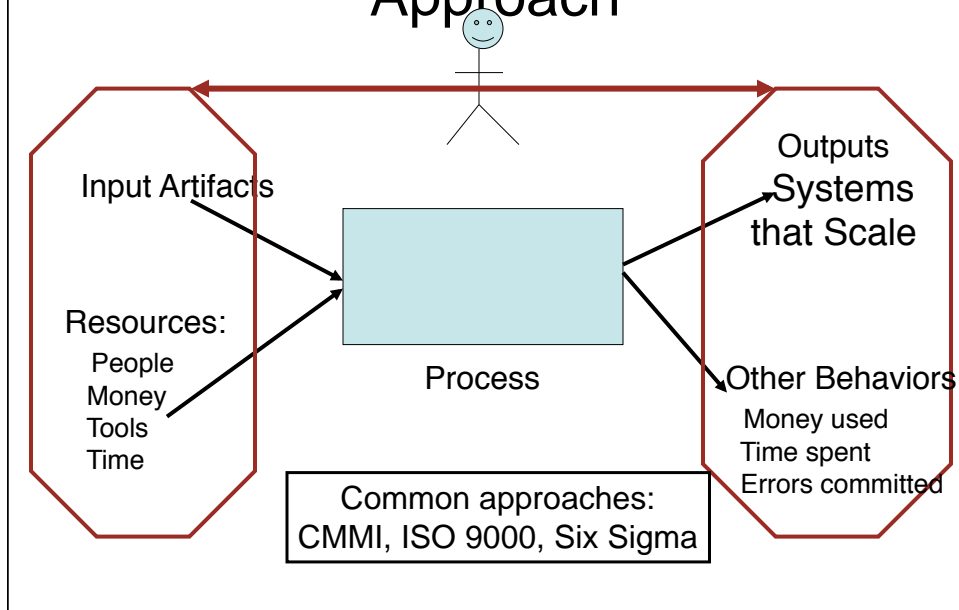


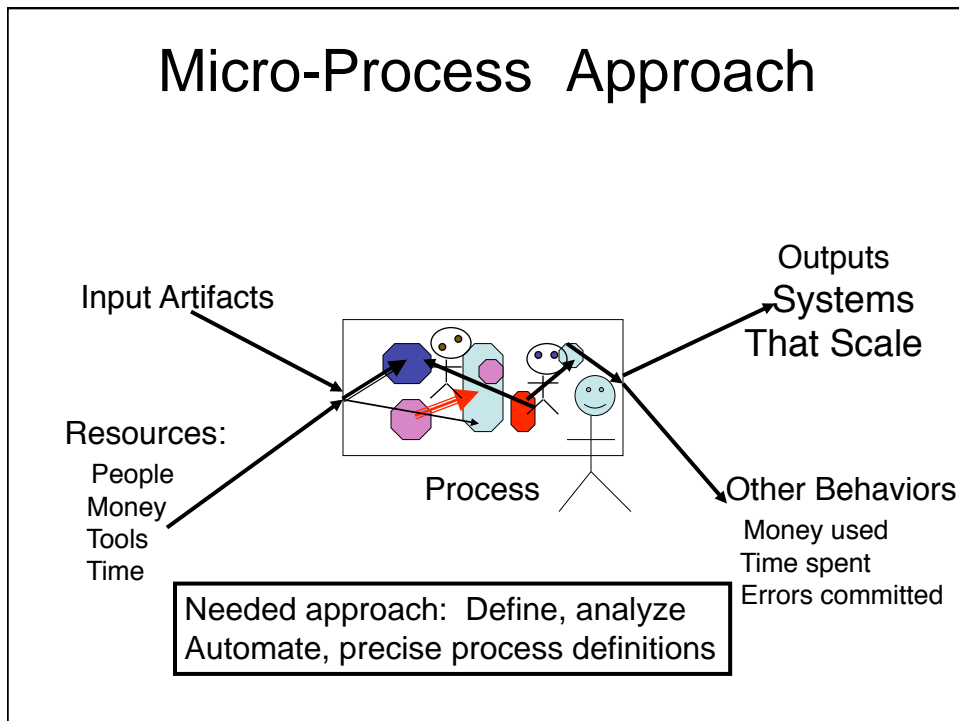
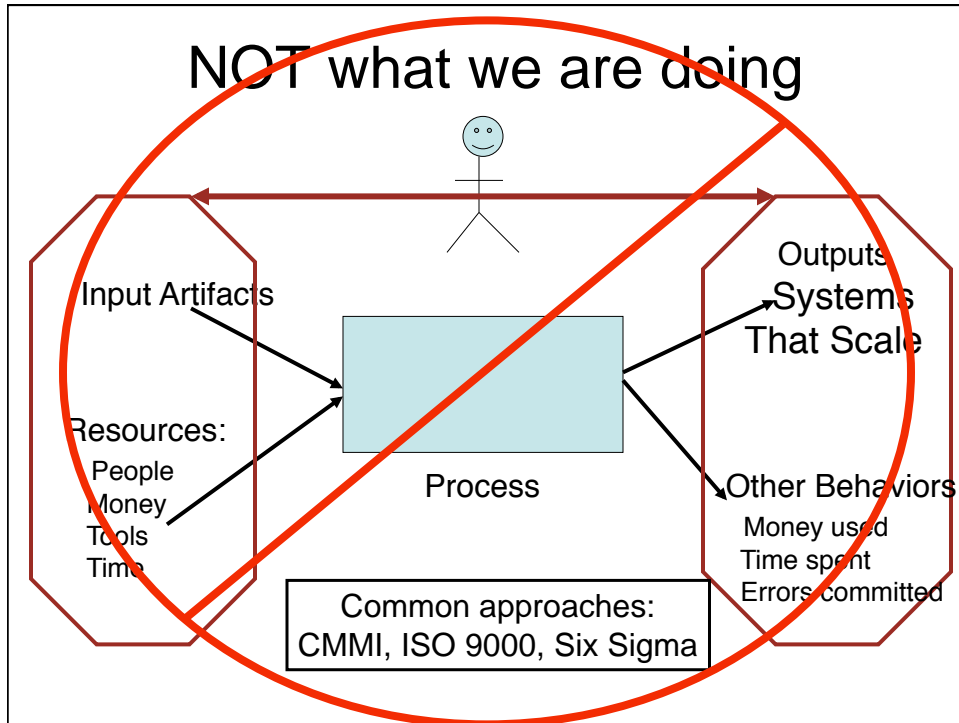
Early Steps in Needed Directions: Little-JIL definitions and analyses

- The Little-JIL process definition language
 - Rigorously defined executable semantics
 - Pictorial
- Process analyses
 - Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA)
 - Finite state verification (model checking)
 - Dynamic process testing and monitoring
 - Discrete event simulation
 - Scenario generation
- Integrations of the above

Costs are incremental: Initially modest, increasing cost of increasingly valuable details and insights.

Conventional Macro-Process Approach





“The” Scrum Process

- An “agile” SW development approach
 - Actually a family of processes
 - A well-accepted high-level characterization
 - A variety of lower-level elaborations
- Numerous imputed advantageous properties
- Can they be inferred from a rigorous scrum process definition?
 - Can lower level details undermine these?
 - How do you know if you are “doing scrum?”

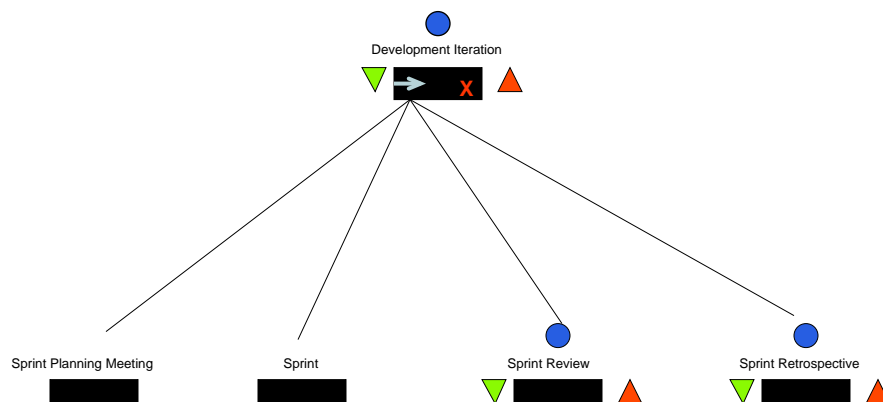
Reasoning about an example Scrum property for an **example** Scrum process

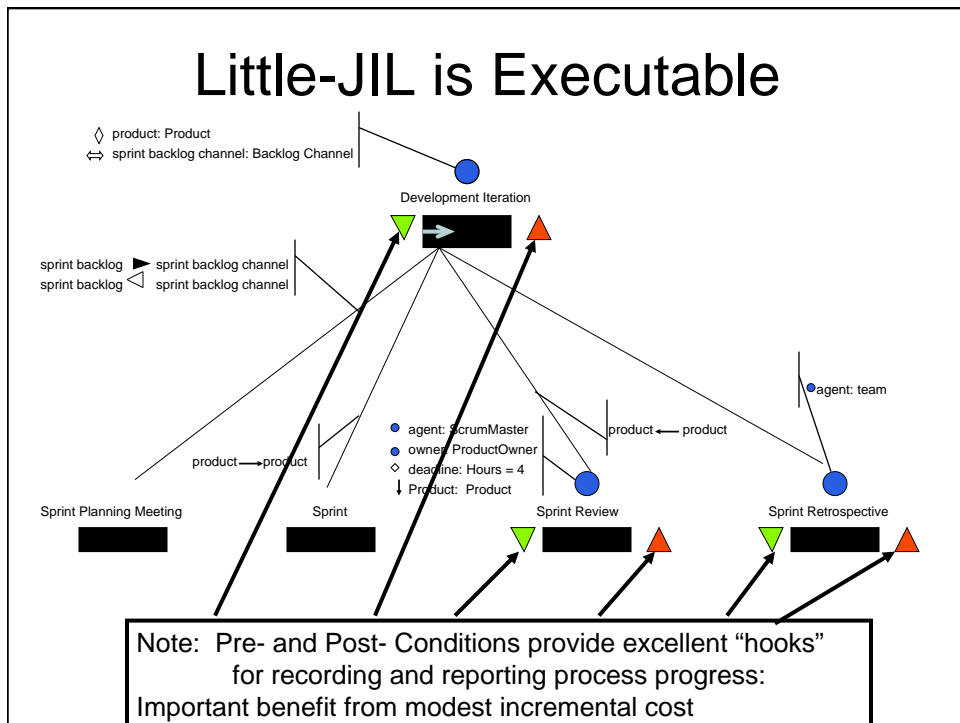
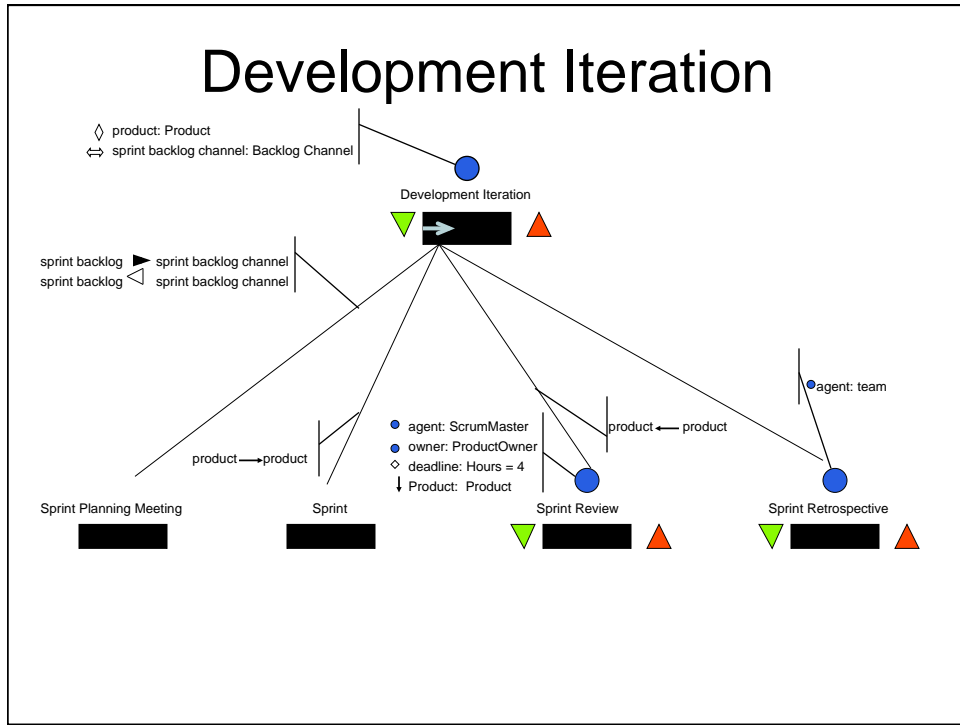
- Property: At the end of each sprint the Scrum Master will always be able to present a product that actually runs
 - This will be studied using a specific **example** scrum process definition
 - *Other members of the scrum process family will be different and may have different properties*
- Use FTA to determine which incorrect step performance(s) may endanger this property?

A Few Bare Essentials

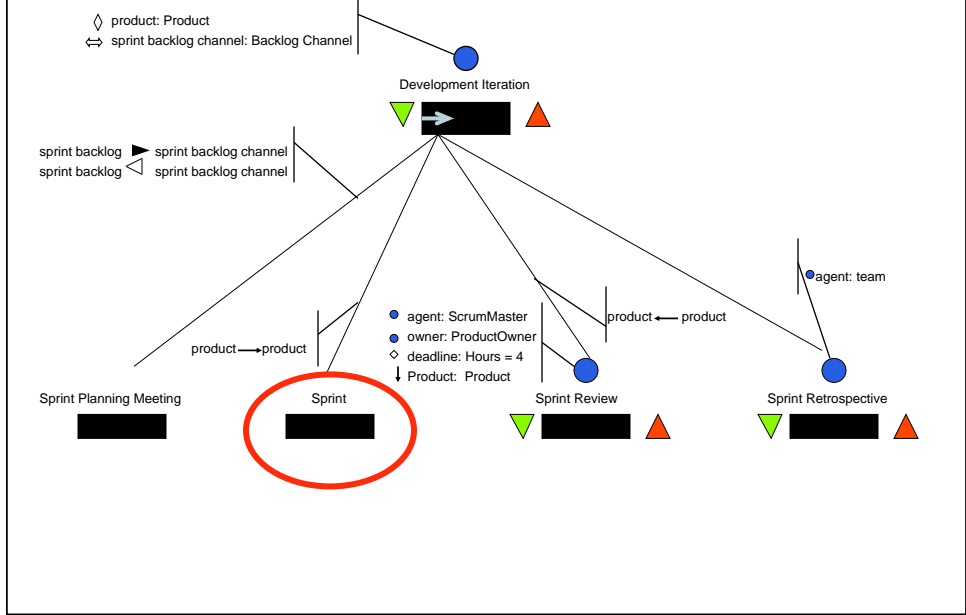
- Process definition is a hierarchical decomposition
- Think of steps as procedure invocations
 - They define scopes
 - Copy and restore argument semantics
- Encourages use of abstraction
 - Eg. process fragment reuse

Development Iteration: Activity Skeleton

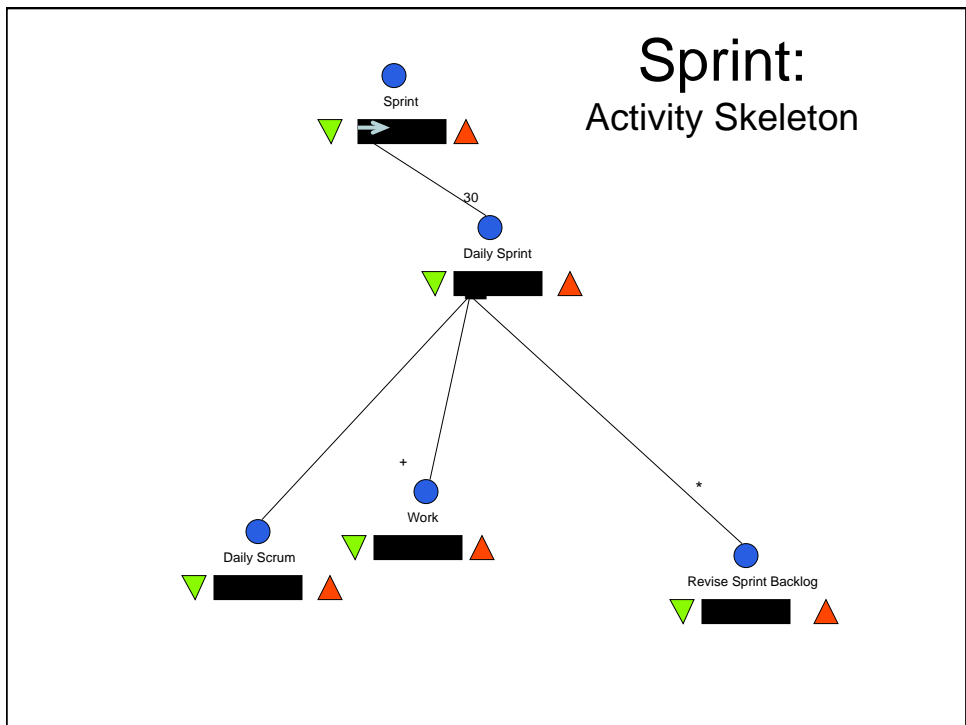


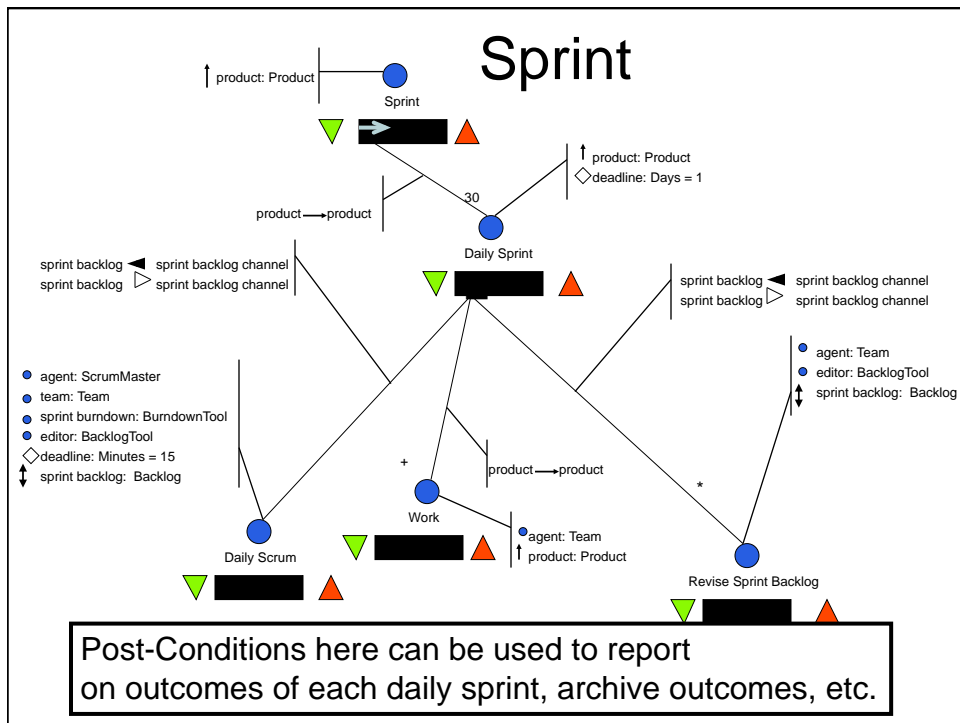
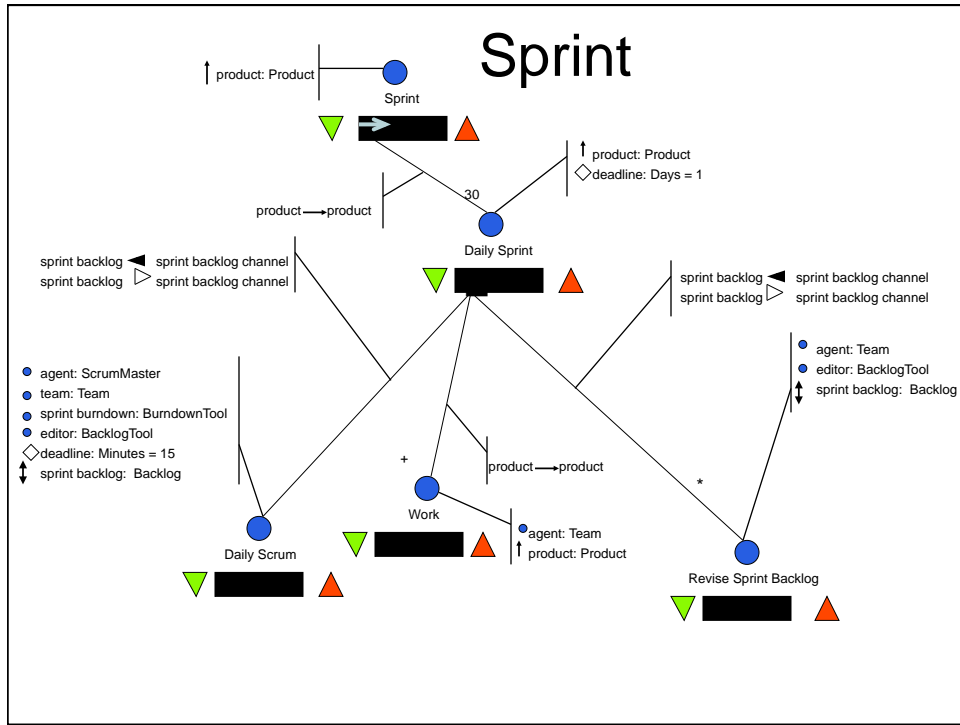


Now Elaborate on the Sprint Step



Sprint: Activity Skeleton





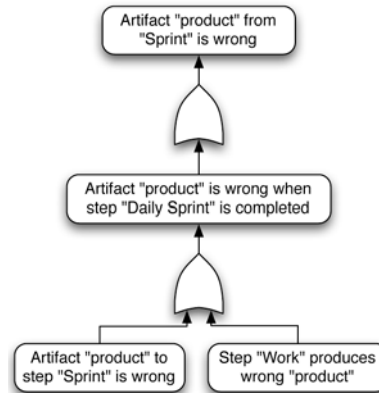
Recall the example Scrum property

- Property: At the end of each sprint the Scrum Master will always be able to present a product that actually runs
- What incorrect step performance(s) may endanger this property?
 - This can also be reported by making appropriate use of post-conditions

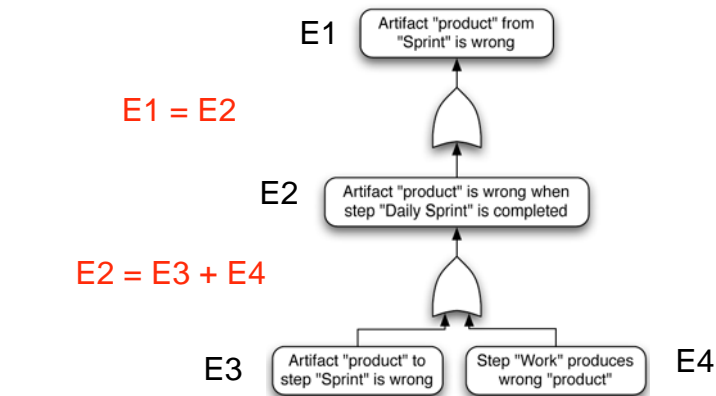
Using Fault Tree Analysis

- Helps determine where a process is vulnerable
- General approach
 - Specify a hazard that is of concern
 - Hazard: A condition in which a serious loss becomes possible
 - Create fault tree for that hazard
 - Derive Minimal Cut Sets (MCSs)--minimal event combinations that can cause the hazard
- Our approach
 - Automatically generate fault trees from the process definition
 - Manual fault tree derivation is time consuming and error prone for non-trivial processes

A Single Point of Failure in our Scrum process creates the hazard that the desired property may not be achieved by the process



A Single Point of Failure in our Scrum process creates the hazard that the desired property may not be achieved by the process



$$E1 = E2$$

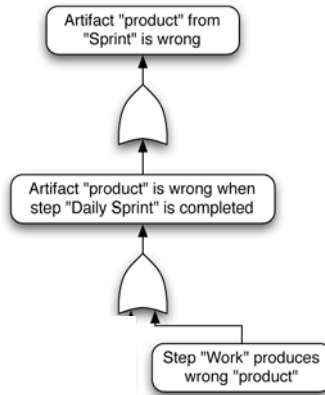
$$E2 = E3 + E4$$

$$\Rightarrow E1 = E3 + E4$$

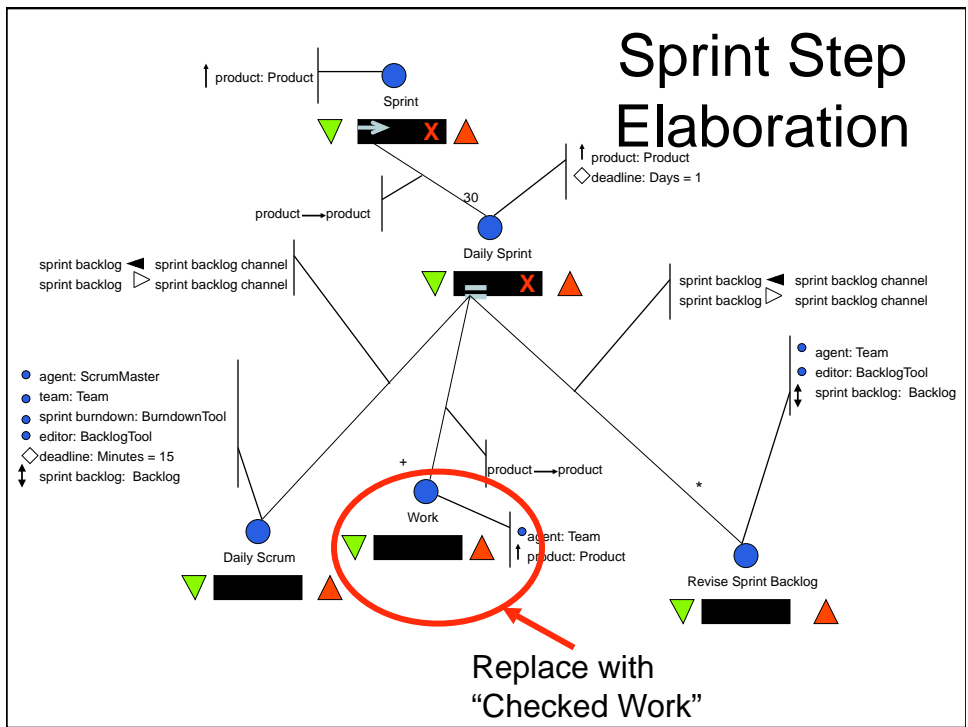
\Rightarrow E3 and E4 are single points of failure

Let's defer E3 for now, and focus on E4

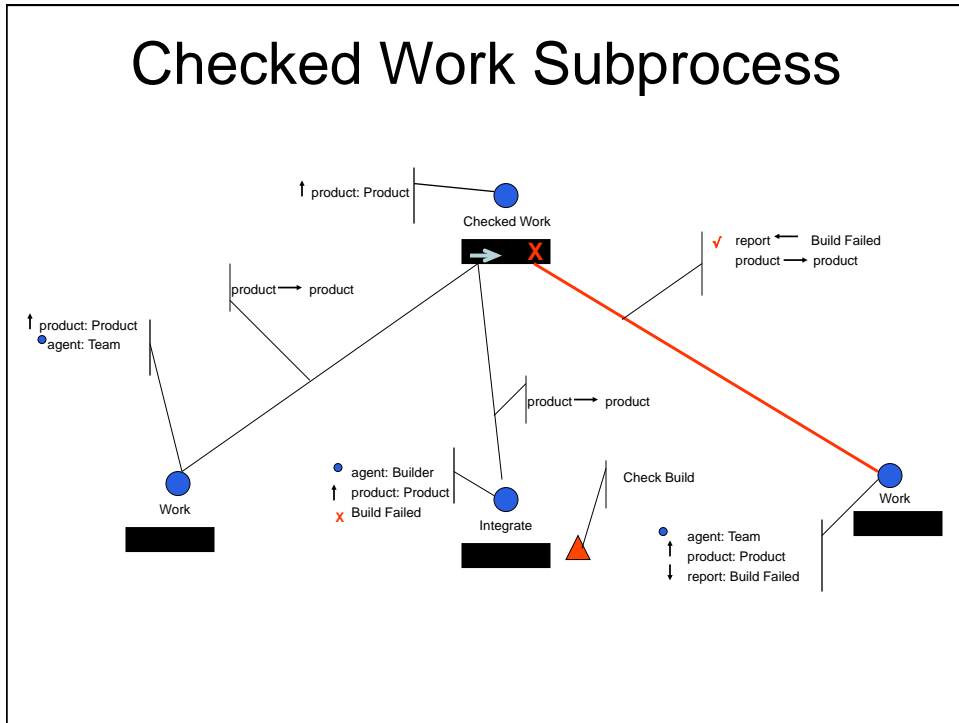
A Single Point of Failure in our Scrum process creates the hazard that the desired property may not be achieved by the process



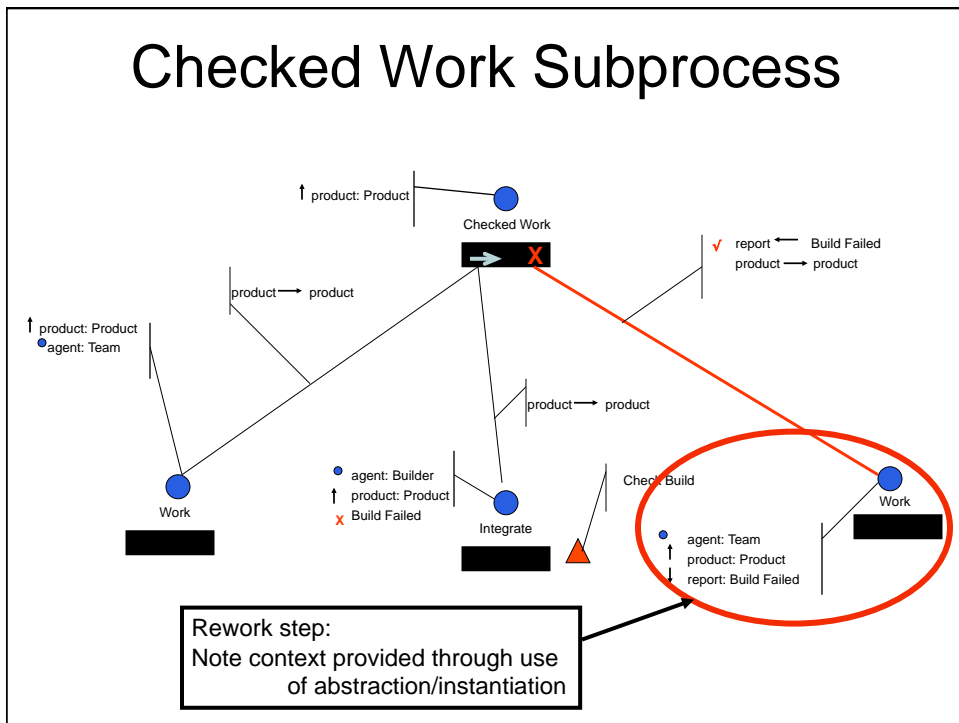
A process modification can remove the E4 SPF



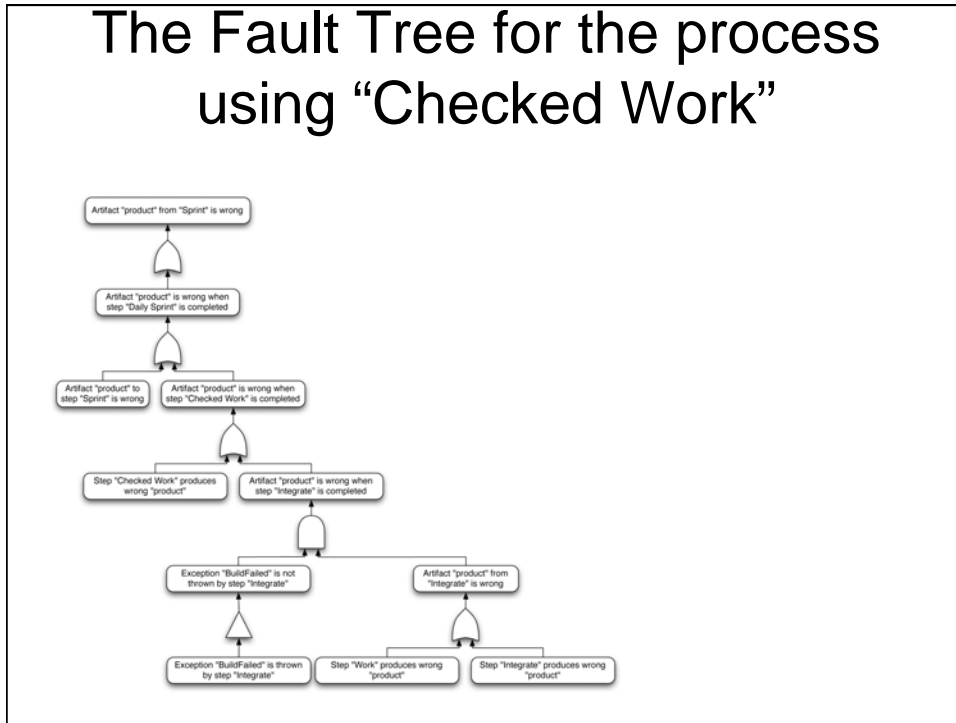
Checked Work Subprocess



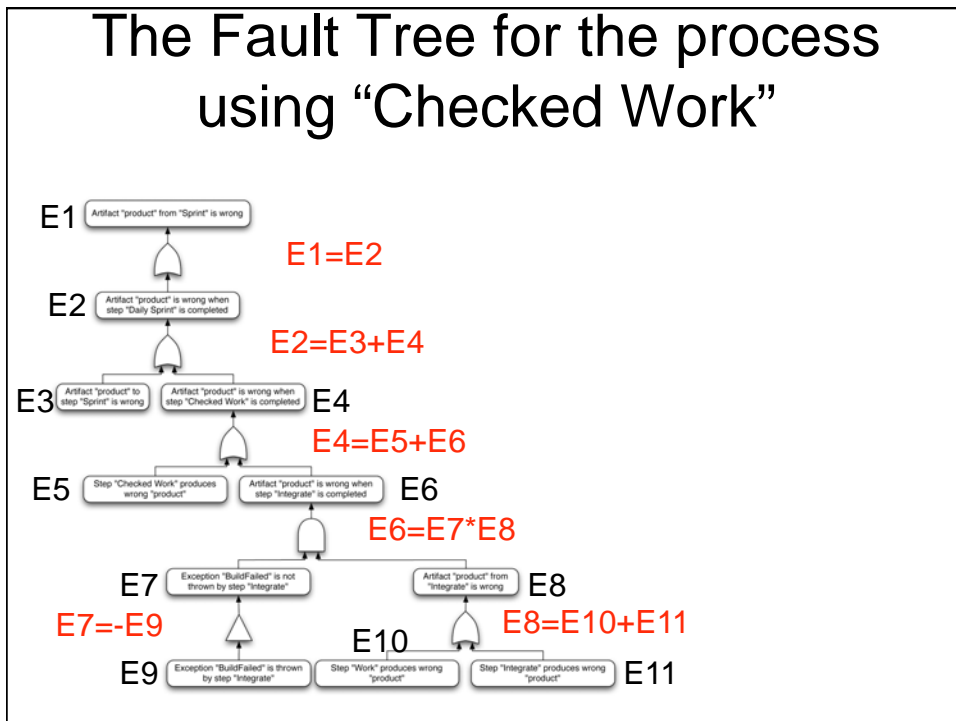
Checked Work Subprocess



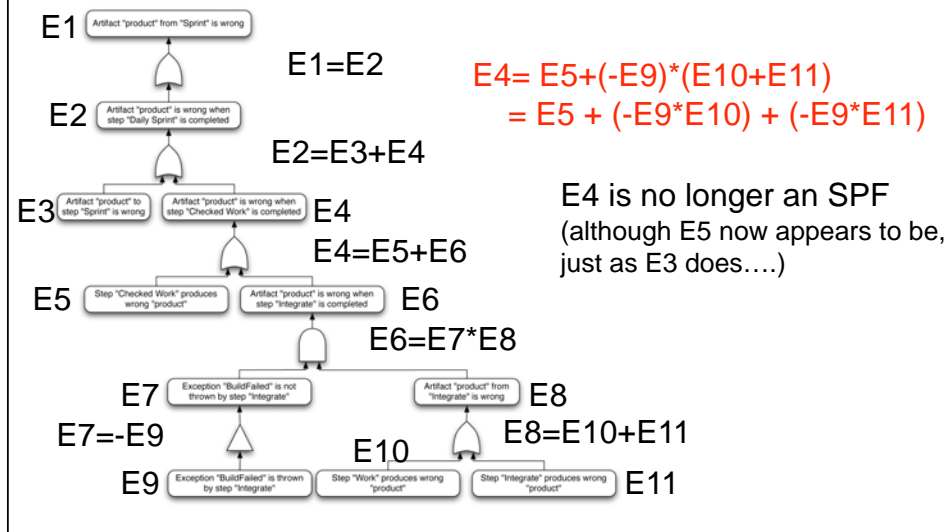
The Fault Tree for the process using "Checked Work"



The Fault Tree for the process using "Checked Work"



The Fault Tree for the process using "Checked Work"



Removing SPFs one at a time

- We removed the E4 SPF by adding a checking step
- We remove E3 and E5 by reasoning...
 - Both assume an incorrect product coming into the Sprint
 - But as the output from a previous sprint
 - But we have shown that the output from a sprint can only be incorrect by a multiple failure
- Improved FT generation will do this reasoning automatically
 - Incorporated into a more recent FT generation tool

Complementary Analysis Techniques

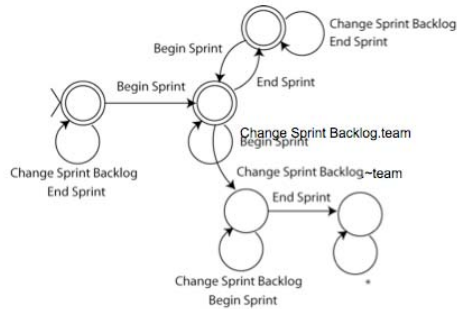
- Fault Tree Analysis assumes that the tasks/artifacts might be wrong and shows where the process is vulnerable if such problems arise
- Finite State Verification assumes tasks are done correctly, but detects when the order of events can lead to problems (as indicated in a property specification)
- Dynamic checking and monitoring supports real-time management/customer tracking, and can trigger desired interventions

Another desirable Scrum property:
During a Sprint, the Sprint Backlog can
only be changed by the team.

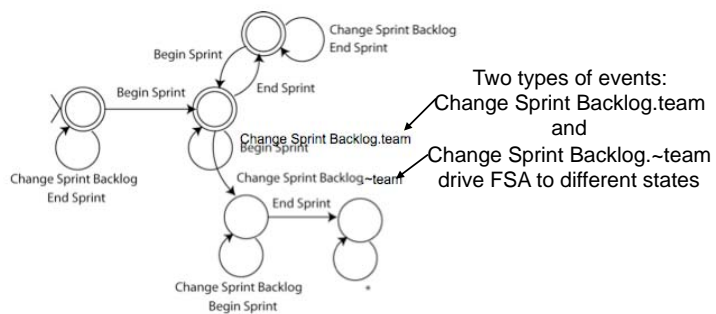
Is it always necessarily true that

“Change Sprint Backlog” occurs between a “Start Sprint” event and an “End Sprint” event only if “Change Sprint Backlog” is performed by the team?

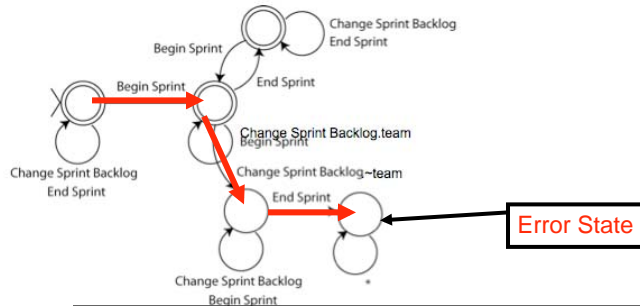
Define the potentially worrisome situation using an FSA



Define the potentially worrisome situation using an FSA

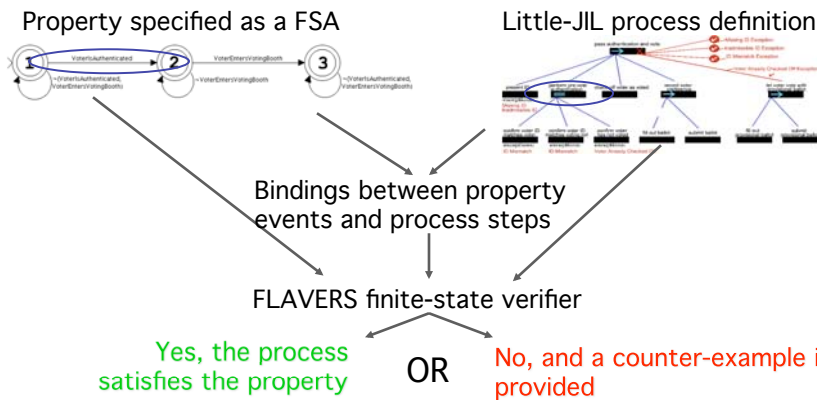


Define the potentially worrisome situation using an FSA



The worrisome state is clearly identified, and can be reached only by execution of the Change Sprint Backlog.~team event

Verifying that the process is consistent with the property



- To determine consistency, property events must be bound to process steps

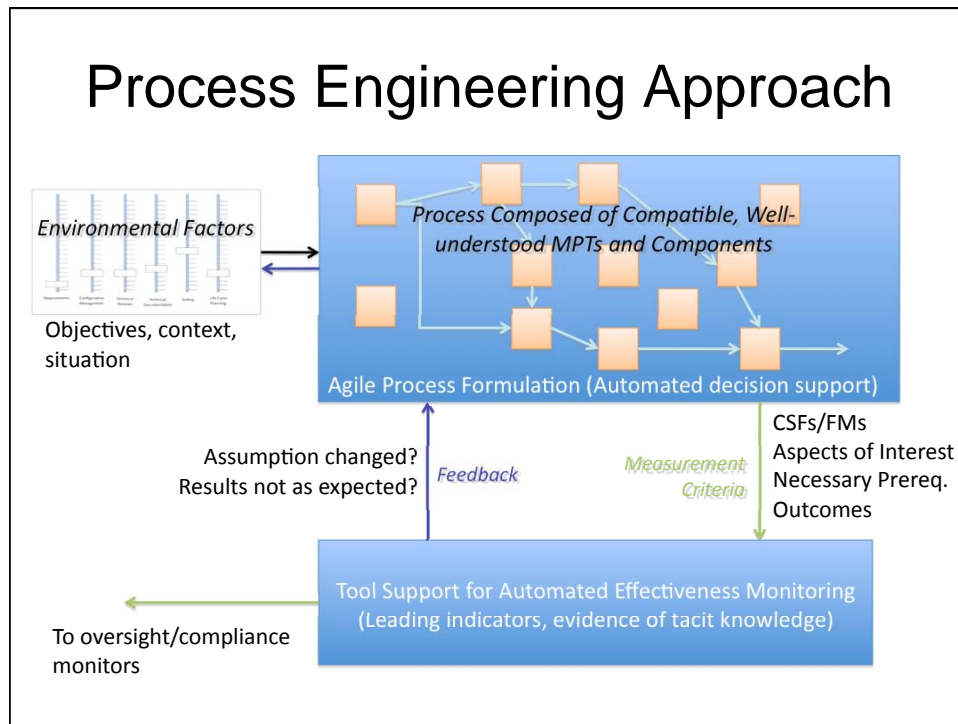
Finite-state verification with FLAVERS

- The FLAVERS verifier has been extended to automatically construct optimized models of Little-JIL process definitions
 - Applies a dataflow analysis algorithm to determine if the model is consistent with the property
- If the process is inconsistent with the property, a counter-example trace is produced

FLAVERS determines that there is a path whose execution causes a property violation

Future Directions

- Identify key relevant system development processes
- Evolve a repository of these processes with information/ attributes about their properties, performance, capabilities, weaknesses, histories, etc.
- Support execution of these processes
 - With collection of execution history data stored in repository
- Develop support for process selection and composition based on the these attributes and history data
- Apply additional analyses
 - Discrete event simulation
 - Scenario/use case generation
- Demonstrate cost-effectiveness of this approach
- Continuously improve all of the above:
 - Tools, processes, properties, repository, user guidance, selection/ composition, cost effectiveness



NOTE ALSO...

Apply this to all kinds of processes

- We are applying this to processes in:
 - Healthcare, negotiation, scientific data processing, elections, etc.
- Apply it also to DoD processes
 - E.g. troop deployment, weapon system firing, intelligence gathering and analysis
- A form of Model-Based System Development(?)
- Such system usage processes can define contexts
 - For inferring system requirements
 - Against which to evaluate suitability of components
 - And help document changes in these contexts
 - To show impacts of usage changes on systems and components

Four Questions

- How to support (dynamically changeable?) decisions about which processes to use when?
- What are the most useful characterizations of processes and how to derive them?
 - Including studies of the cost-effectiveness of deriving these characterizations
- How to monitor system development processes to mine data to support timely decision-making?
- How to build a repository of reusable, configurable, composable system processes? What would it look like internally and externally?

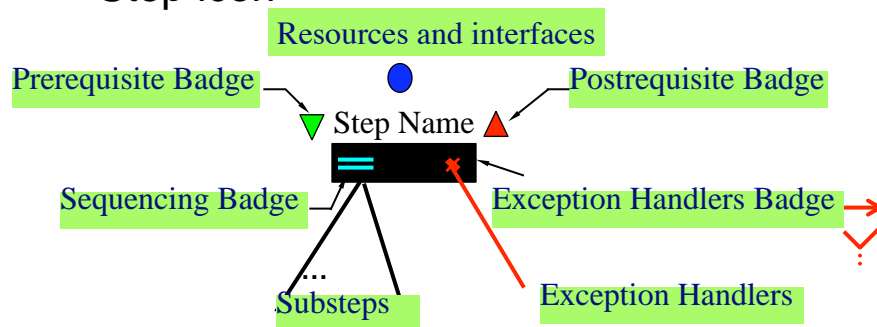
Backup Little-JIL Slides

The Little-JIL Process Language

- Vehicle for exploring language abstractions for
 - Reasoning (rigorously defined)
 - Automation (execution semantics)
 - Understandability (visual)
- Supported by
 - Visual-JIL graphical editor
 - Juliette interpreter
- Evaluation by application to broad domains
- A third-generation process language
- A “work in progress”

Little-JIL Overview





- Visual language for coordinating tasks
- Uses hierarchically decomposed steps
- Step icon







Hierarchy, Scoping, and Abstraction in Little-JIL

- Process definition is a hierarchical decomposition
- Think of steps as procedure invocations
 - They define scopes
 - Copy and restore argument semantics
- Encourages use of abstraction
 - Eg. process fragment reuse





Proactive Flow Specified by four Sequencing Kinds

-  • Sequential
 - In order, left to right
-  • Parallel
 - Any order (or parallel)
-  • Choice
 - Choose from Agenda
 - Only one choice allowed
-  • Try
 - In order, left to right

Proactive Flow Specified by four Sequencing Kinds

- These step kinds support human flexibility in process performance
-  • Sequential
 - In order, left to right
 -  • Parallel
 - Any order (or parallel)
 -  • Choice
 - Choose from Agenda
 - Only one choice allowed
 -  • Try
 - In order, left to right

Proactive Flow Specified by four Sequencing Kinds

-  • Sequential
 - In order, left to right
-  • Parallel
 - Any order (or parallel)
-  • Choice
 - Choose from Agenda
 - Only one choice allowed
-  • Try
 - In order, left to right

Iteration usually through recursion
 Alternation using pre/post requisites





Pre- and Post-requisites

- Steps guarded by (optional) pre- and post-requisites
- Are steps themselves
- Can throw exceptions
- May be executed by different agents
 - From each other
 - From the main step

Exception Handling: A Special Focus of Little-JIL

- Steps may have one or more exception handlers
- Handlers are steps themselves
 - With parameter flow
- React to exceptions thrown in descendent steps
 - By Pre- or Post-requisites
 - Or by Agents

Four different continuations on exception handlers

- Complete 
 - Handler was a “fixup”; substep is completed
- Continue 
 - Handler cleaned up; parent step is completed
- Restart 
 - Handler cleaned up; repeat substep (deprecated)
- Rethrow 
 - Rethrow to parent step

Artifact flow

- Primarily along parent-child edges
 - As procedure invocation parameters
 - Passed to exception handlers too
 - Often omitted from coordination diagrams to reduce visual clutter
- This is inadequate
 - Artifacts also need to flow laterally
 - And subtasks need to communicate with each other

Channels and Lateral flow

- Channel supports message passing
- Multiple steps can add artifacts
- And multiple steps that can take them
- Use for synchronization and passing artifacts

Resources

- Entities needed in order to perform step
- Step specifies resource needed as a type
 - Perhaps with attributes, qualifiers
- Resource instances bound at runtime
- Exception thrown when “resource unavailable”

Resources

- Entities needed in order to perform step
- Step specifies resource needed as a type
 - Perhaps with attributes, qualifiers
- Resource instances bound at runtime
- Exception thrown when “resource unavailable”

Much research is needed here

Agents

- Collection of all entities that can perform a step
 - Human or automated
- Process definition is orthogonal to assignments of agents to steps
 - Path to automation of process
- Have freedom to execute leaf steps in any way they want

Use an Example to Demonstrate This

- Define a part of a Scrum process
 - In detail
 - Using the Little-JIL process language
- Show the importance of details for understanding and coordination of efforts
- Apply rigorous analyzers to
 - Infer properties
 - Compare them to requirements
 - Identify weaknesses
 - Support monitoring and reporting
 - Suggest improvements
- Confirm effectiveness of improvements

Technology-based Process Improvement:
Engineer superior cost-benefits ratios