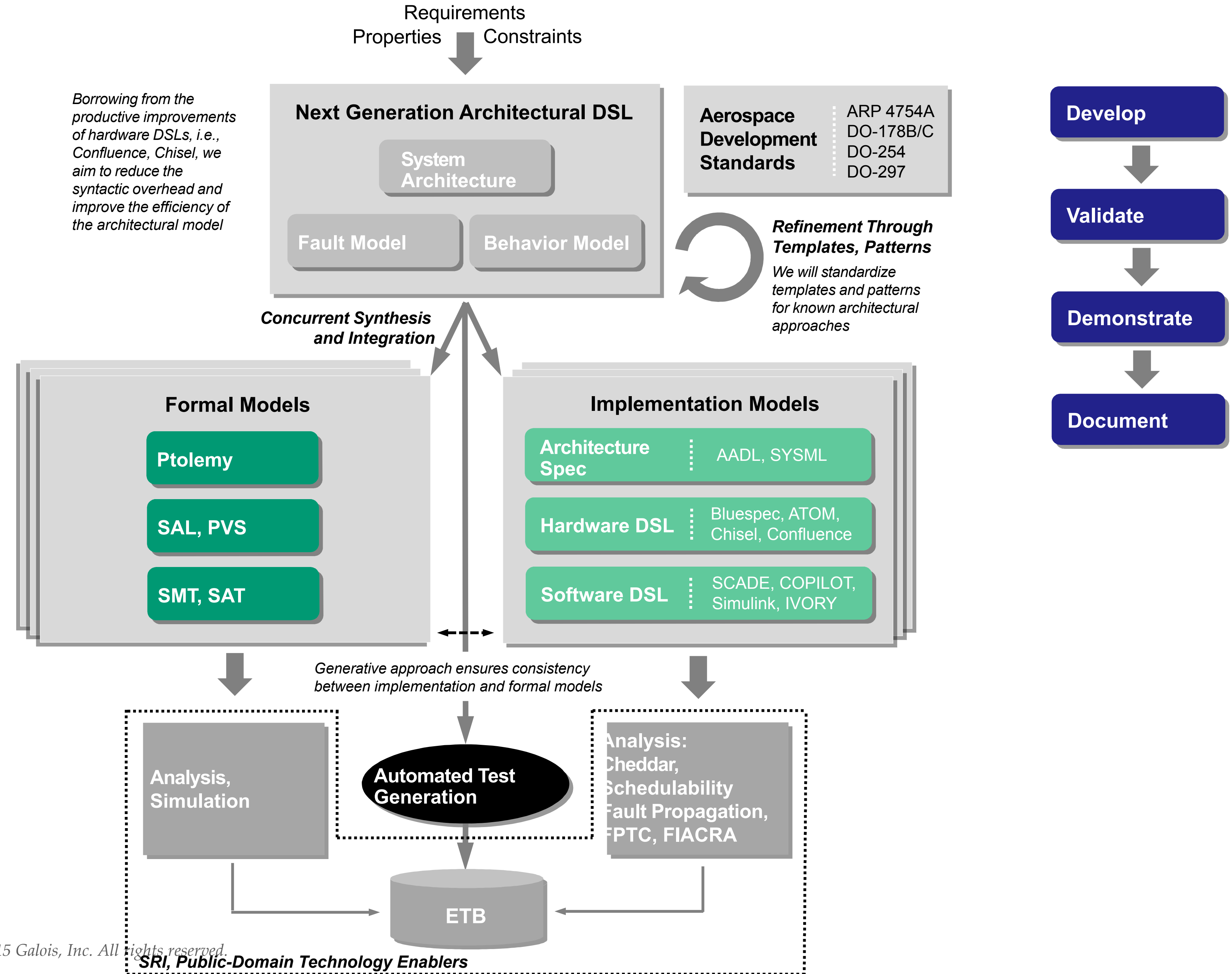
AFFIRM Tag-Up

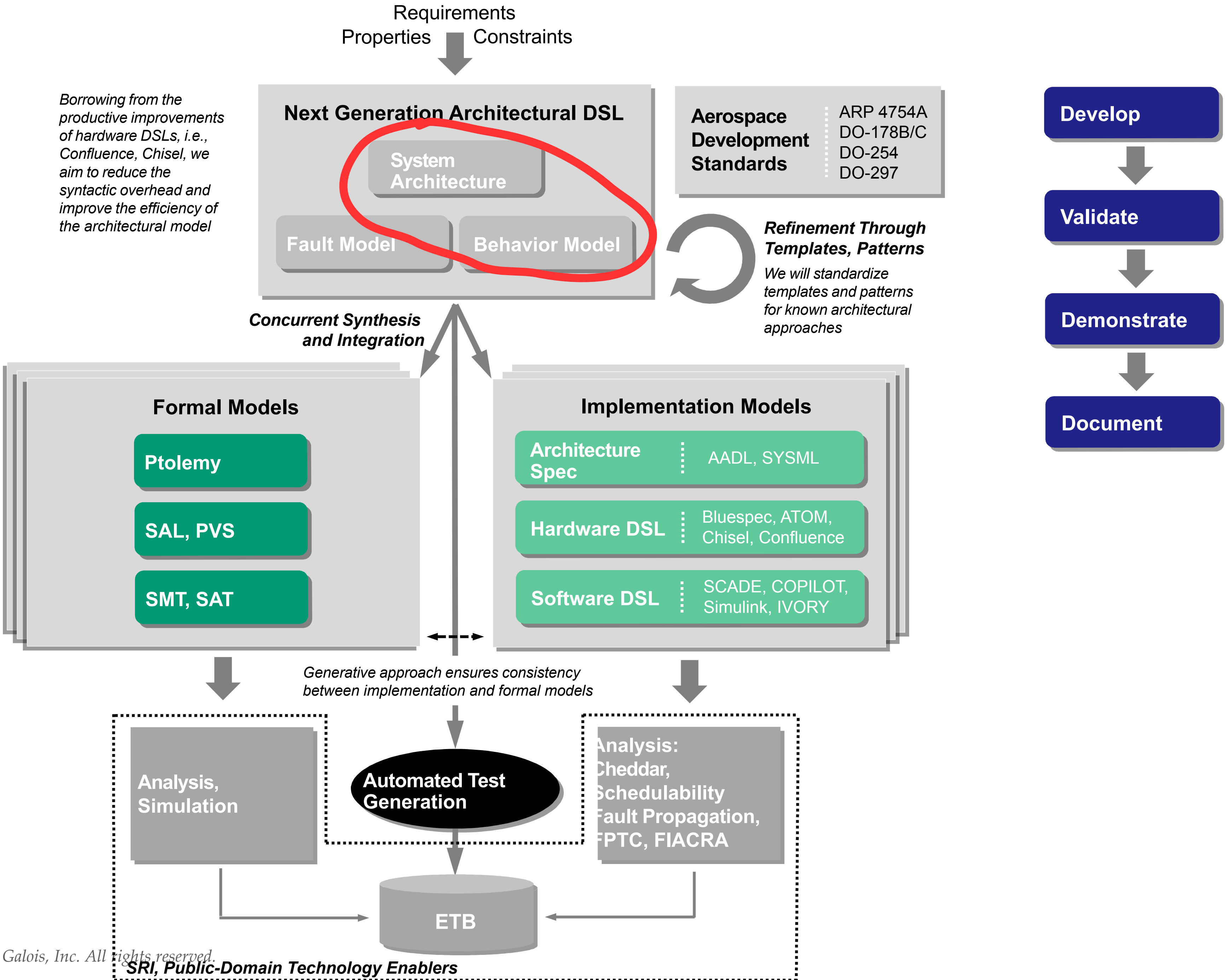
Lee Pike (Galois)
Ben Jones (Galois)
Brendan Hall (Honeywell)
Srivatsan Varadarajan (Honeywell)

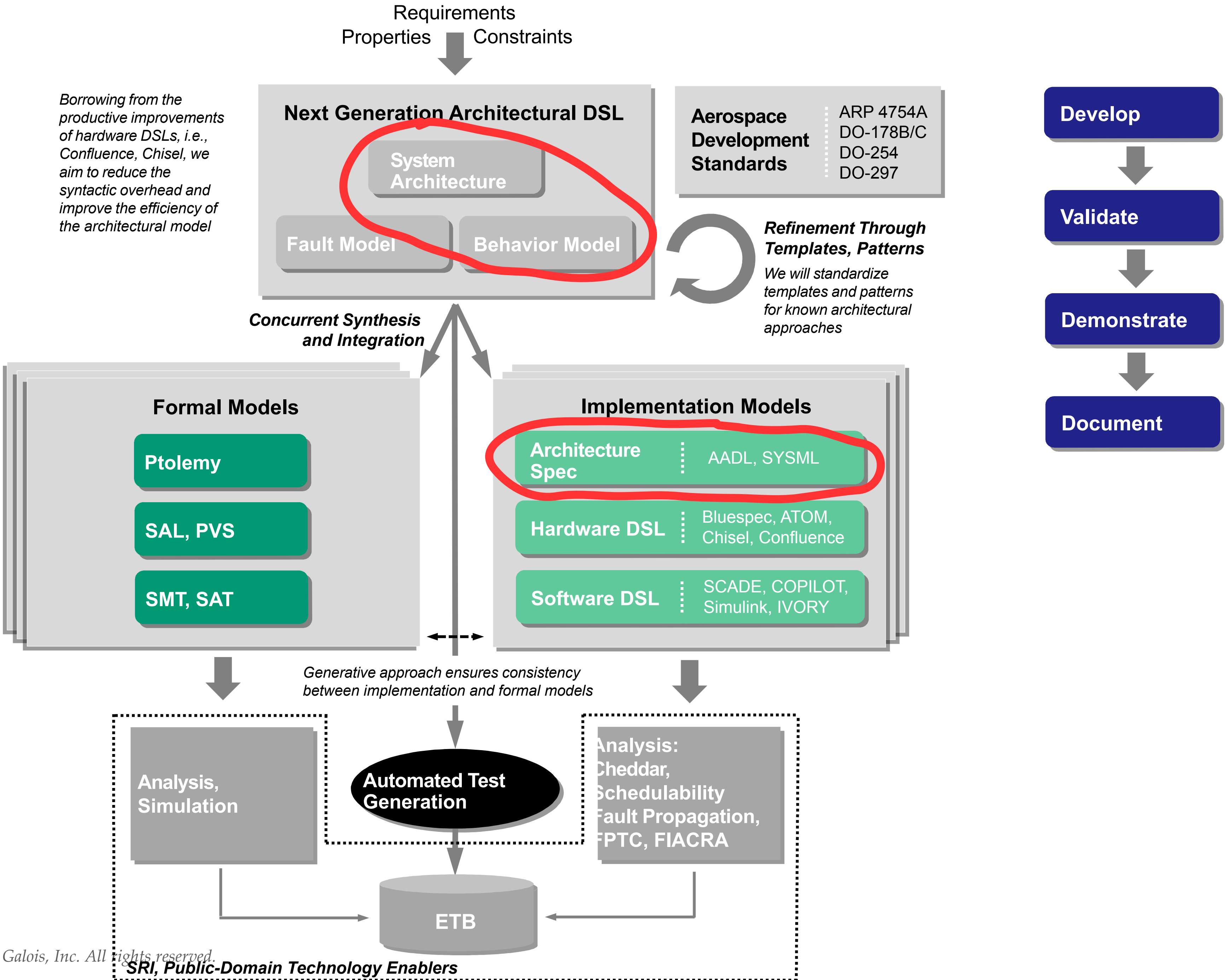
Feb, 2015

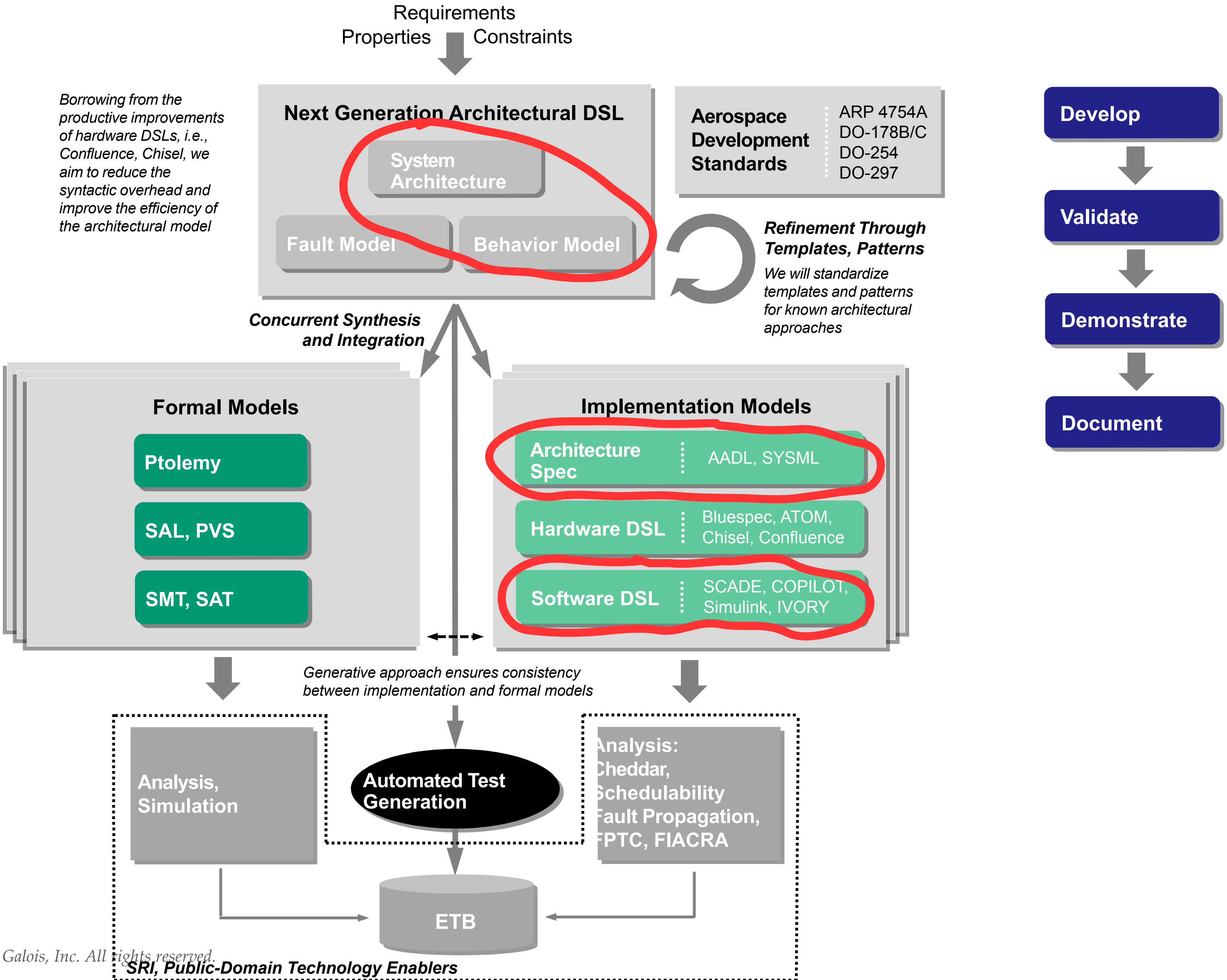
Honeywell

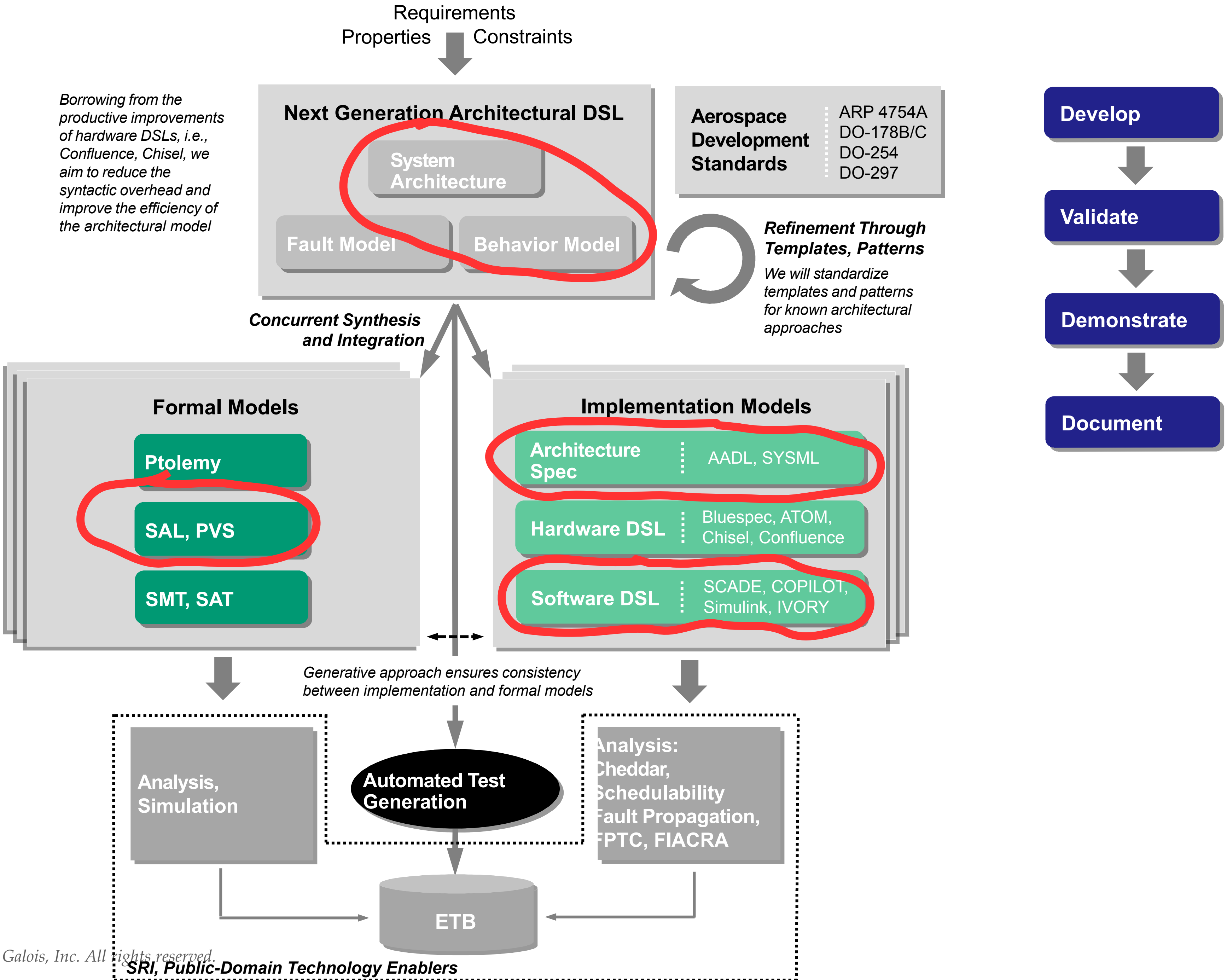
| galois |


Requirements
Properties
Constraints
Borrowing from the productive improvements of hardware DSLs, i.e., Confluence, Chisel, we aim to reduce the syntactic overhead and improve the efficiency of the architectural model
Next Generation Architectural DSL
System Architecture
Fault Model
Behavior Model
Aerospace Development Standards
ARP 4754A
DO-178B/C
DO-254
DO-297
Refinement Through Templates, Patterns
We will standardize templates and patterns for known architectural approaches
Develop
Validate
Demonstrate
Document
Concurrent Synthesis and Integration
Formal Models
Ptolemy
SAL, PVS
SMT, SAT
Implementation Models
Architecture Spec
AADL, SYSML
Hardware DSL
Bluespec, ATOM, Chisel, Confluence
Software DSL
SCADE, COPILOT, Simulink, IVORY
Generative approach ensures consistency between implementation and formal models
Analysis, Simulation
Automated Test Generation
Analysis: Cheddar, Schedulability Fault Propagation, FPTC, FIACRA
ETB
SRI, Public-Domain Technology Enablers


Requirements
Properties
Constraints
Borrowing from the productive improvements of hardware DSLs, i.e., Confluence, Chisel, we aim to reduce the syntactic overhead and improve the efficiency of the architectural model
Next Generation Architectural DSL
System Architecture
Fault Model
Behavior Model
Aerospace Development Standards
ARP 4754A
DO-178B/C
DO-254
DO-297
Refinement Through Templates, Patterns
We will standardize templates and patterns for known architectural approaches
Develop
Validate
Demonstrate
Document
Concurrent Synthesis and Integration
Formal Models
Ptolemy
SAL, PVS
SMT, SAT
Implementation Models
Architecture Spec
AADL, SYSML
Hardware DSL
Bluespec, ATOM, Chisel, Confluence
Software DSL
SCADE, COPILOT, Simulink, IVORY
Generative approach ensures consistency between implementation and formal models
Analysis, Simulation
Automated Test Generation
Analysis: Cheddar, Schedulability Fault Propagation, FPTC, FIACRA
ETB
SRI, Public-Domain Technology Enablers


Requirements
Properties
Constraints
Borrowing from the productive improvements of hardware DSLs, i.e., Confluence, Chisel, we aim to reduce the syntactic overhead and improve the efficiency of the architectural model
Next Generation Architectural DSL
System Architecture
Fault Model
Behavior Model
Aerospace Development Standards
ARP 4754A
DO-178B/C
DO-254
DO-297
Refinement Through Templates, Patterns
We will standardize templates and patterns for known architectural approaches
Develop
Validate
Demonstrate
Document
Concurrent Synthesis and Integration
Formal Models
Ptolemy
SAL, PVS
SMT, SAT
Implementation Models
Architecture Spec
AADL, SYSML
Hardware DSL
Bluespec, ATOM, Chisel, Confluence
Software DSL
SCADE, COPILOT, Simulink, IVORY
Generative approach ensures consistency between implementation and formal models
Analysis, Simulation
Automated Test Generation
Analysis: Cheddar, Schedulability Fault Propagation, FPTC, FIACRA
ETB
SRI, Public-Domain Technology Enablers


Requirements
Properties
Constraints
Borrowing from the productive improvements of hardware DSLs, i.e., Confluence, Chisel, we aim to reduce the syntactic overhead and improve the efficiency of the architectural model
Next Generation Architectural DSL
System Architecture
Fault Model
Behavior Model
Aerospace Development Standards
ARP 4754A
DO-178B/C
DO-254
DO-297
Refinement Through Templates, Patterns
We will standardize templates and patterns for known architectural approaches
Develop
Validate
Demonstrate
Document
Concurrent Synthesis and Integration
Formal Models
Ptolemy
SAL, PVS
SMT, SAT
Implementation Models
Architecture Spec
AADL, SYSML
Hardware DSL
Bluespec, ATOM, Chisel, Confluence
Software DSL
SCADE, COPILOT, Simulink, IVORY
Generative approach ensures consistency between implementation and formal models
Analysis, Simulation
Automated Test Generation
Analysis: Cheddar, Schedulability Fault Propagation, FPTC, FIACRA
ETB
SRI, Public-Domain Technology Enablers


Requirements
Properties
Constraints
Borrowing from the productive improvements of hardware DSLs, i.e., Confluence, Chisel, we aim to reduce the syntactic overhead and improve the efficiency of the architectural model
Next Generation Architectural DSL
System Architecture
Fault Model
Behavior Model
Aerospace Development Standards
ARP 4754A
DO-178B/C
DO-254
DO-297
Refinement Through Templates, Patterns
We will standardize templates and patterns for known architectural approaches
Develop
Validate
Demonstrate
Document
Concurrent Synthesis and Integration
Formal Models
Ptolemy
SAL, PVS
SMT, SAT
Implementation Models
Architecture Spec
AADL, SYSML
Hardware DSL
Bluespec, ATOM, Chisel, Confluence
Software DSL
SCADE, COPILOT, Simulink, IVORY
Generative approach ensures consistency between implementation and formal models
Analysis, Simulation
Automated Test Generation
Analysis: Cheddar, Schedulability Fault Propagation, FPTC, FIACRA
ETB
SRI, Public-Domain Technology Enablers

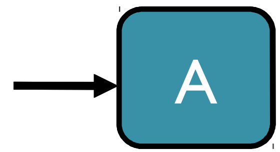
Jan. - Feb. Results

- Abstractions & AADL (Tower) modelling
  - Related work: COMPASS, BLESS
- Tower-to-SAL translation
  - Focus on calendar-automata modeling
- Pencil & paper proof of bi-simulation between
  - Shared-state semantics of ADSL (efficient simulation)
  - Message-passing semantics of ADSL (distributed systems)
- Co-routines for mode-control

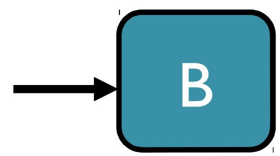
https://wiki.sei.cmu.edu/aadl/images/b/b7/COMPASSTutorialOct2011.pdf
https://ti.arc.nasa.gov/m/events/nfm2013/pubs/BLESS.pdf

Modes

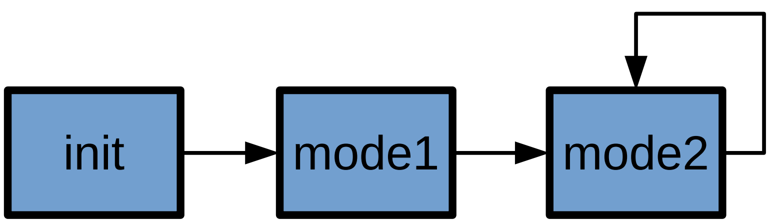
- Orthogonal to tasks
- Persistent—handlers are functions on state and inputs
- Lots of state
- Complex control-flow
- Bugs: who updates mode?




```
if(m == init) {
  ...
} elseif (m == mode1) {
  ...
} elseif (m == mode2) {
  ...
```




```
if(m == init) {
  ...
} elseif (m == mode1) {
  ...
} elseif (m == mode2) {
  ...
```



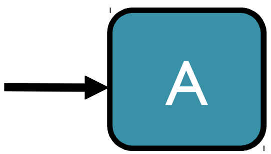
(Semi)-Coroutines

|galois|

```
def go(n):
    yield "a"
    while (n < 2):
        yield "b"
        n += 1
```

```
> for i in go(0):
     print i
> “a”
  “b”
  “b”
```

Mode Control with Coroutines




```
if(m == init) {
  …
} elseif (m == mode1) {
  …
} elseif (m == mode2) {
  ...
```

```
…
yield // Done with initialization
…
yield // Done with mode 1
…
forever {
  …
  yield // Done with mode 2
}
```

- Doesn't require global state (state variable m)
- Caller in charge of mode control

Tower to SAL

Generating logical specifications from an architectural DSL

Premise: We want to generate (**SAL**) models of a system that is specified in our DSL (**Tower**) using abstractions appropriate to the domain of fault tolerant distributed systems.

Example

Consider a toy example system:

- one node labeled “A”
- A’s state consists of one integer variable
- there is a typed input channel to A, “rx”, carrying integers
- A updates its state integer by adding each received integer to it

Toy Example Specified in Tower

The node A is represented by a monitor that contains a handler listening to the input channel. The handler calls an `update` function upon receiving a message.

```
monitor "A" $ do

  st <- state "st"   -- local state
  store st 0         -- initialization

  handler rx "rx" $ do  -- handle channel "rx"
    callback (\m -> update m st)
```

Toy Example (continued..)

The update function specifies the low-level details of A's state transition.

```
update m st = do
  m' <- deref m           -- get msg
  st' <- deref st         -- get current state
  store st (st' + m')     -- add msg to state and store
```

SAL Model

To generate a SAL model from the Tower code:

- generate a SAL MODULE for each monitor node
- map monitor state variables to SAL module LOCAL variables
- map channel inputs, clocks, and signals to module INPUTs
- map channel outputs to module OUTPUTs
- generate a TRANSITION from the asynchronous composition of the handlers

Toy Example in SAL

SAL module definition is straightforward. The variables `time` and `cal` are used to model message passing in a real-time system.

```
monitor[i : IDENTITY]: MODULE =
  INPUT  time : TIME       -- current time
  GLOBAL cal  : CALENDAR   -- event calendar
  LOCAL  st   : INTEGER    -- local state
  INITIALIZATION
    st = 0
  {- TRANSITION ... -}
END
```

State Transition

The elided calendar functions tell a node when a new message has arrived.

```
TRANSITION
  [
    pending?(cal, i) AND time = event_time(cal, i) -->
      st’ = st + get_msg(cal, i);
      cal’ = consume_msg(cal, i)
  []
    ELSE -->
  ]
```

Abstraction

The SAL module above attempts to model our toy example faithfully, including all the details of the state machine at each node.

. . .

However, we may want to reason about the system at a different level of abstraction.

Update Function Abstracted

We can use and extend the "requires / ensures" framework from Ivory in order to generate *abstract* transition systems in our SAL model.

In **Tower**:

```
callback $ \m ->
  requires (0 <=? m) $
  ensures (\r -> st <=? r) $
    update m st = {- original update code -}
```

SAL Transition Abstracted

In the *abstract* transition, the new state value is drawn from the set of possible new states according to our `ensures` annotation.

```
TRANSITION
  [
    pending?(cal, i) AND time = event_time(cal, i) -->
      IF get_msg(cal, i) >= 0
        THEN st' IN { x : INTEGER | st <= x }
        ELSE signal(cal, i, time, undefined_behavior)
      ENDIF
      cal' = consume_msg(cal, i)
  []
    ELSE -->
  ]
```

Concrete Steps

Short-term plans for implementing the ideas we've presented:

- **Implement SAL syntax in Haskell** and an embedded language of constructors and combinators for generating native SAL syntax
  `https://github.com/benjaminfjones/sal-lang`
- **Map Tower to SAL** using the requires/ensures framework to abstract state machine details
- Explore using **fault annotations** on channels
- Explore using the **synchronous observer model** for specifying system properties