Modeling Complex Systems in Rewriting Logic

Carolyn Talcott, SRI International

LAP

2022 September

Motivation

- Smart, (semi) autonomus systems are everywhere. robots, factory automation, warehouse management, self-driving vehicles, wearbles, medical devices, …

- Assuring safety and security is often critical.

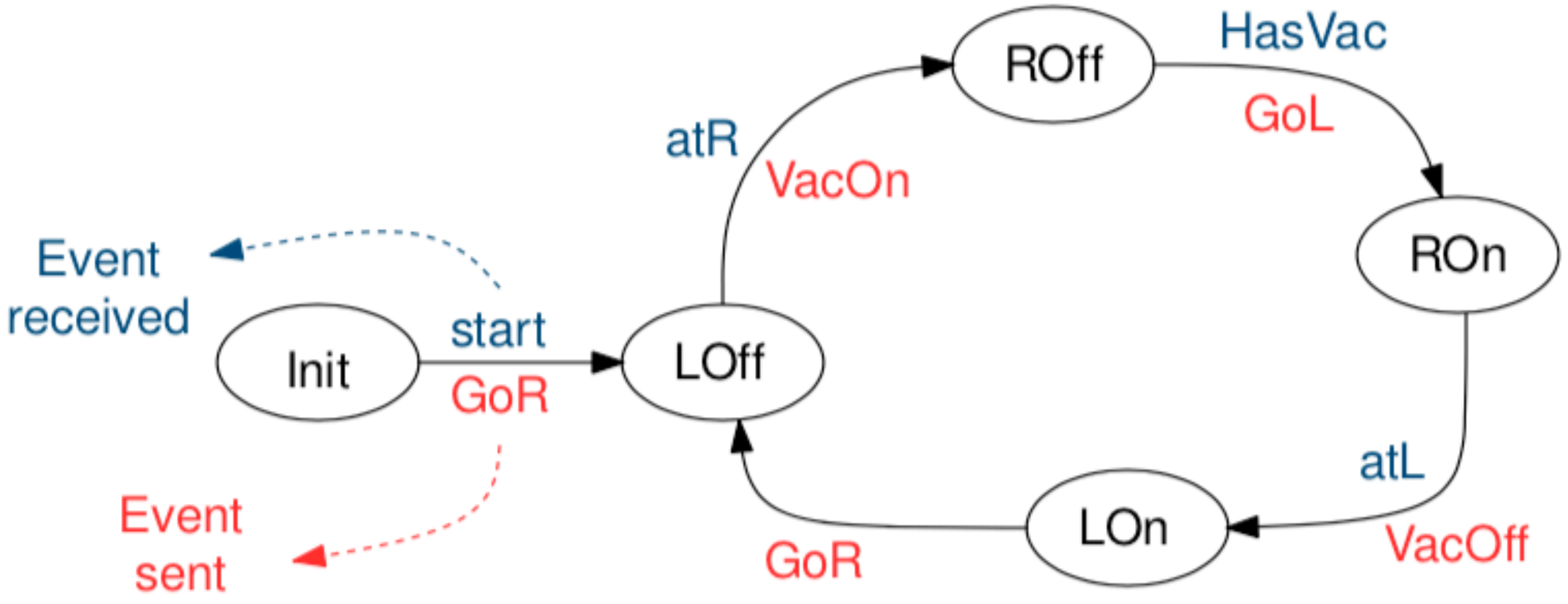
- Accurate models are complex and likely intractible. Small abstract models give useful insights but getting the big picture is a challenge.

- As an example: Industry 4.0 is a paradigm of manufacturing where devices (IoT devices, sensors, …) are highly networked, to form cyber-physical systems — think smart factories.

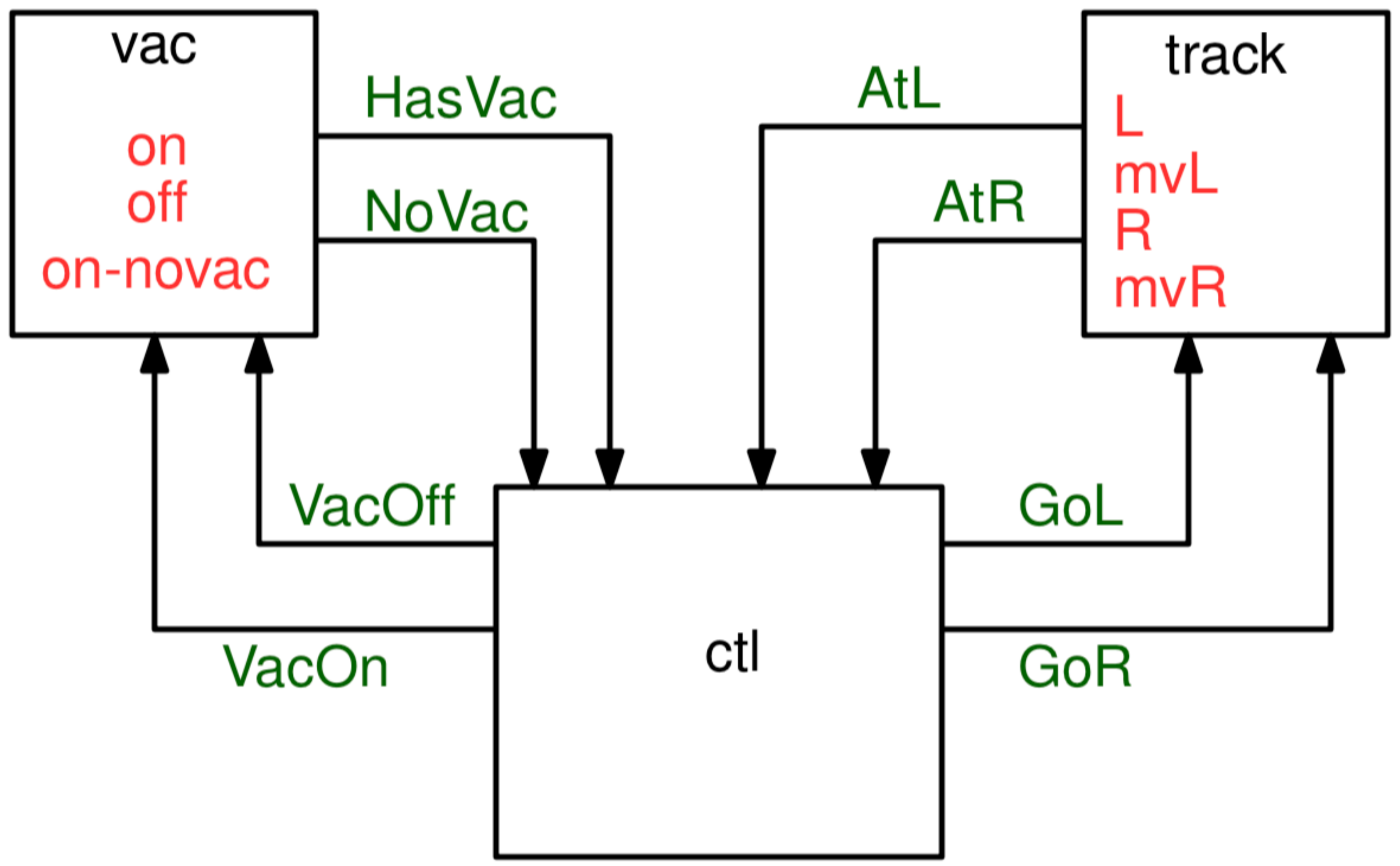
Example Pick n Place

- Pick n place (PnP) is typical example of a basic I4.0 system
- Elements
  - An Arm -- positioned on a track (move, sense end points)
  - A Gripper -- with state on/off (gripping or not) — for example a vacuum
  - Coordinator -- enforces pickup at one end, drop at the other end

  - A designer might start with component models as Mealy automata -- receive signal, change state, send signal

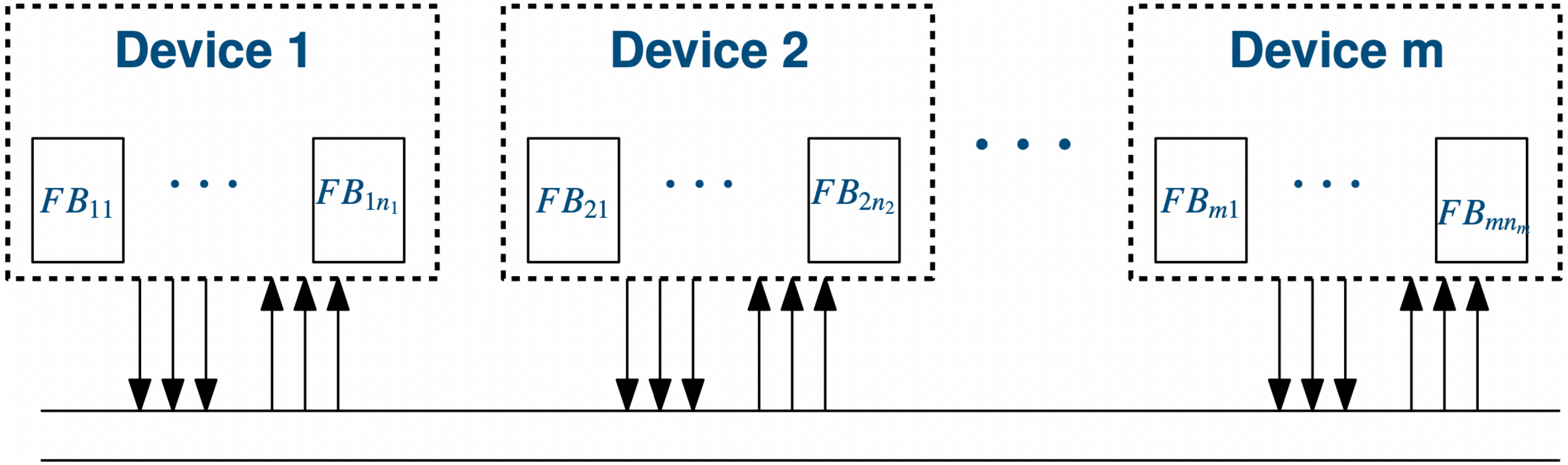


Example **Pick n Place continued**



PnP Application

- Application Design -- a collection of interacting automata
- Can do reachability analysis for correct, safe operation — search for bad states.
- The system engineer needs to make it run on hardware:
  - deploying automata (implementations) on physical devices
  - connected by a network



Oops

- Components communicating over a network are open to attacks — tampering with inserting or deleting messages.
- We add attacks to the models to find which messages are vulnerable— cause safety violations
- Once we find the sensitive communications they can be protected by digital signatures. This may influence the choice of deployment to devices.
- At the end we have a bunch of models for different purposes.
  - Complex models are more faithful to the running system.
  - Simpler models are more practical to analyze.
  - The models may not be consistent with each other.
  - What if some component design changes?
  - What if a new threat is discovered?   =>

Solution idea

- Connect models by formal patterns — model transformations that preserve properties of interests and achieve some goal.
- In fact application of the patterns generates models and their connections.
- Sample transformations types -- informally
  - Addition of threat or fault models — may add new (undesirable) behaviors
  - Converting concrete models into symbolic models (and vv) — preserves traces and trace based properties — reason about forall rather than for one.
  - Converting abstract designs into deployment models
    - communicating automata -> networked devices
    - stuttering bisimilar, preserves TL/next properties
  - model -> Wrap(model) — preserve some properties, add new properties, prevent undesired behaviors

In the remainder of the talk we
show how this is done in RWL / Maude
for the Pick n Place example

Rewriting Logic (RWL)

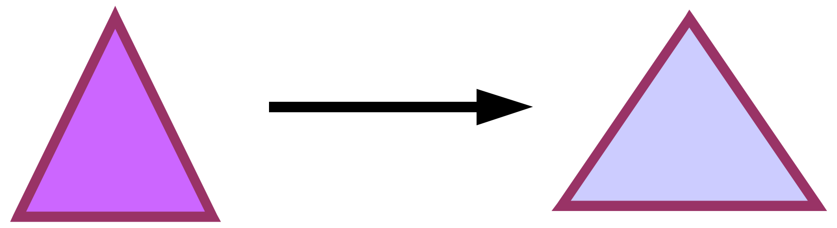
- Rewriting Logic is a logical formalism that is based on two simple ideas
  - states of a system are represented as elements of an equationally specified algebraic data type
  - the behavior of a system is given by local transitions between states described by rewrite rules
- It is a logic for executable specification and analysis of software systems, that may be concurrent, distributed, or even mobile.
- Can also model physical aspects and cyberphysical systems
- It is also a (meta) logic for specifying and reasoning about formal systems, including itself (reflection!)

Rewriting Logic Formally

- Rewrite theory: (Signature, Labeled Rules)
- Signature: (Sorts, Ops, Eqns) -- an equational theory
  - Specifies data types and functions that represent structure of system state and operations on state.
  - Sorts are partially ordered
  - Axioms < Eqns
- Rules have the form label : t => t' if cond
- Rewriting operates modulo equations
  - rules apply locally, matching modulo axioms
  - rule application generates computations (pathways, proofs)

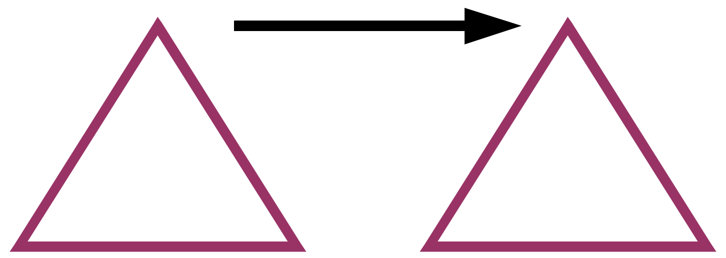
Rewrite rule application in pictures

one step rewrite:

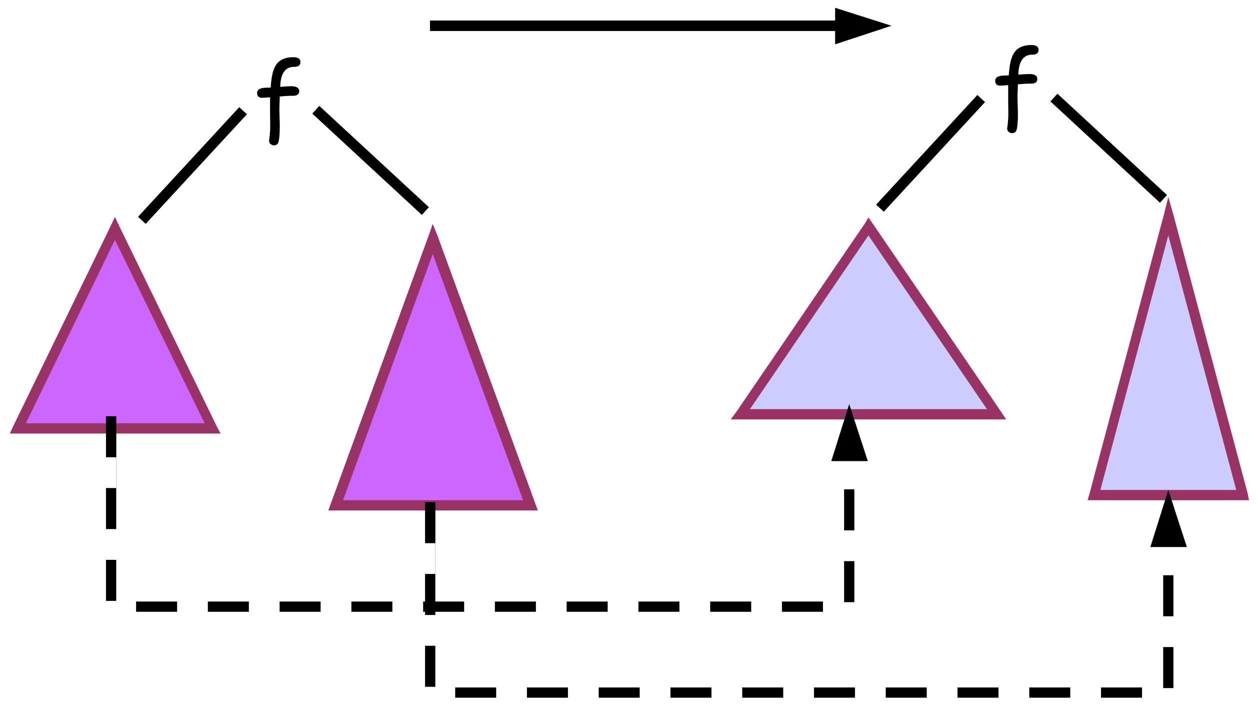


closed under

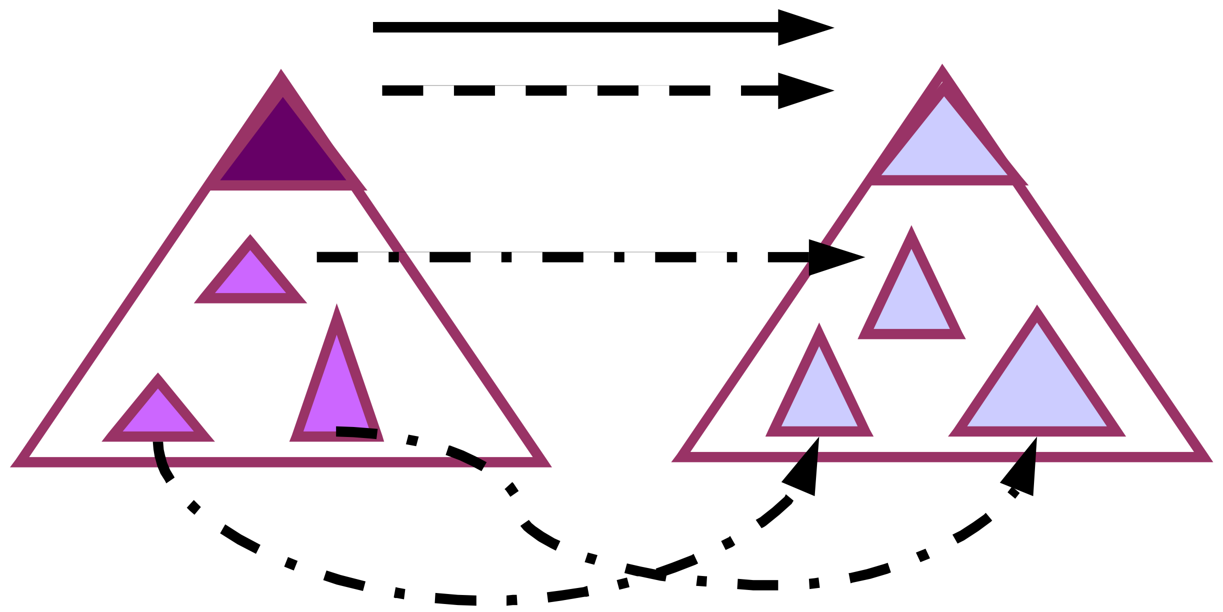
reflexivity:



congruence:



replacement:



Rewriting Semantics

- The semantics of the equational theory (Sorts, Ops, Eqns) is its initial model.
  - Isomorphic to the algebra of equivalence classes wrt Eqns.
- Equivalence classes are represented by their canonical form.

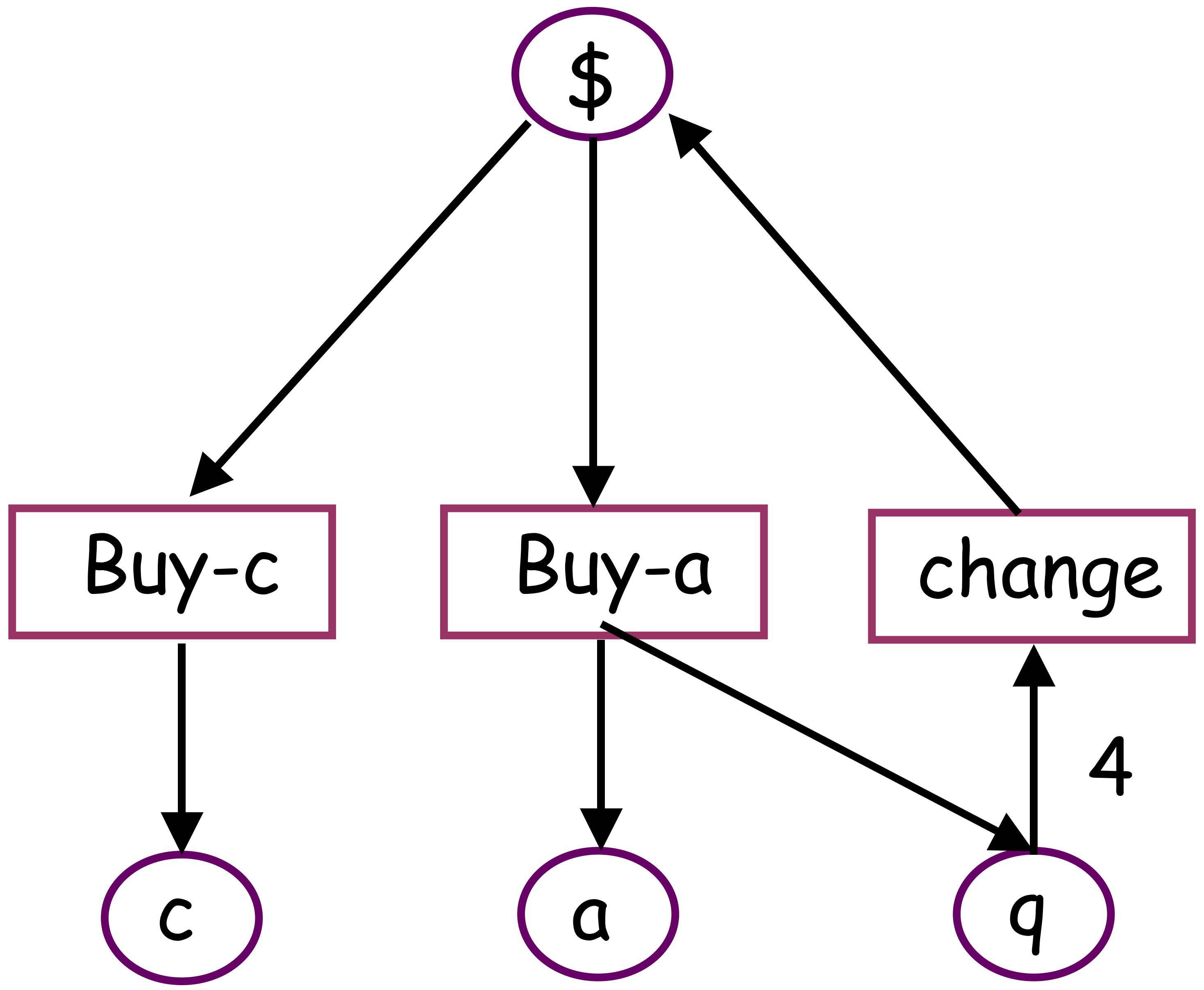
- The base rewrite relation: t0 -> t1
- if there is
  - a rule: l => r if cond
  - a subterm u0 of t0
  - a substitution s such that s(l) =[Ax] t0 and s[cond] holds.
- and t1 = t0[u0 <- s(r)]
- We restrict attention to addmissible rewrite theories -- intuitively this means reduction to canonical form commutes' with rewriting.
- Thus the full rewrite relation operates one canonical forms.
  - t -Eqn-> tc -rules-> t` -Eqn-> tc'

Maude

- Maude is a language and tool based on rewriting logic
- Available at: http://maude.cs.uiuc.edu
- Features:
  - High performance engine
  - {ACI} matching
  - position /rule/object fair rewriting
  - Modularity and parameterization
  - Builtins -- booleans, number hierarchy, strings, SMT solving
  - Reasoning: rewriting, search and model-checking
  - Reflection -- using descent and ascent functions!)

Maude Model of a Vending Machine

A introduction to modeling in Maude




```
mod VENDING-MACHINE is
  sorts Coin Item Place Marking .
  subsorts Coin Item < Place < Marking .
  op null : -> Marking .
              *** empty marking
  ops $ q : -> Coin .
  ops a c : -> Item .
  op _ _ : Marking Marking -> Marking
           [assoc comm id: null] .
           *** multiset using axioms
  rl[buy-c]: $ => c .
  rl[buy-a]: $ => a q .
  rl[change]: q q q q => $ .
endm
```

Using Maude to analyze the vending machine

- What is one way to use 3 $s? Use the rewrite command:

```
Maude> rew $ $ $ .
result Marking: q a c c
```

- How can I get 2 apples with 3 $s? Usethe search command

```
Maude> search $ $ $ =>! a a M:Marking .

Solution 1 (state 8)
M:Marking --> q q c

Solution 2 (state 9)
M:Marking --> q q q a

No more solutions.
states: 10  rewrites: 12)
```

Sampling of the Maude PnP model — function block

- Vacuum (gripper) function block automata initial state:

```
[fbId:Id : vac | state : st("off") ; ticked : false ;
                 iEvEffs : none ; oEvEffs : none] .
```

- A vacuum transition

```
tr(st("off"), st("on"),  inEv("VacOn") is ev("VacOn"),
                         outEv("HasVac") :~ ev("HasVac"))
```

- Application of the transition

```
[fbId:Id : vac | state : st("off") ; ticked : false ;
                 iEvEffs : inEv("VacOn") :~ ev("VacOn") ; oEvEffs : none] .
=>
[fbId:Id : vac | state : st("on") ; ticked : true ;
                 iEvEffs : none ; oEvEffs : outEv("HasVac") :~ ev("HasVac"))] .
```

Sampling of the Maude PnP model — application

- Application composed of function blocks and an intial message:

```
pnpInit(emsg) = [id("pnp") | (fbs : (vacInit(id("vac")) trackInit(id("track")) ctlInit(id("ctl"))) ) ;
            iEMsgs : emsg ; oEMsgs : none ; ssbs : none ] .
```

- The rewrite that fires enabled transitions

```
crl[app-exe1]: [appId |  fbs : ([fbId : fbCid | (state : st) ; (ticked : false) ;
                                                   oEvEffs : none ; fbAttrs] fbs1) ;
      iEMsgs : (emsgsO iemsgs) ; oEMsgs : oemsgs ;  ssbs : ssbsO ; appAttrs ]
=>
[appId |  fbs : ([fbId : fbCid | (state : st1) ; (ticked : true) ;
                              oEvEffs : oeffs ; fbAttrs]  fbs1) ;
 iEMsgs : (iemsgs[[ssbs1]]) ; oEMsgs : oemsgs ;  ssbs : (ssbsO ssbs1) ;  appAttrs ]
 if symtr(st, st1,[css] csss,oeffs) symtrs  := symtrsFB(fbCid,st)
 /\ size(emsgsO) = size(css)
 /\({ssbs1} ssbss) := genSol1(fbId,emsgsO,css) .
```

- There are two more rules in the Application+Intruder model

```
**** all enabled transitions fired; collect and deliver output
**** intruder injects a message
```

Sampling of the Maude PnP model — analysis

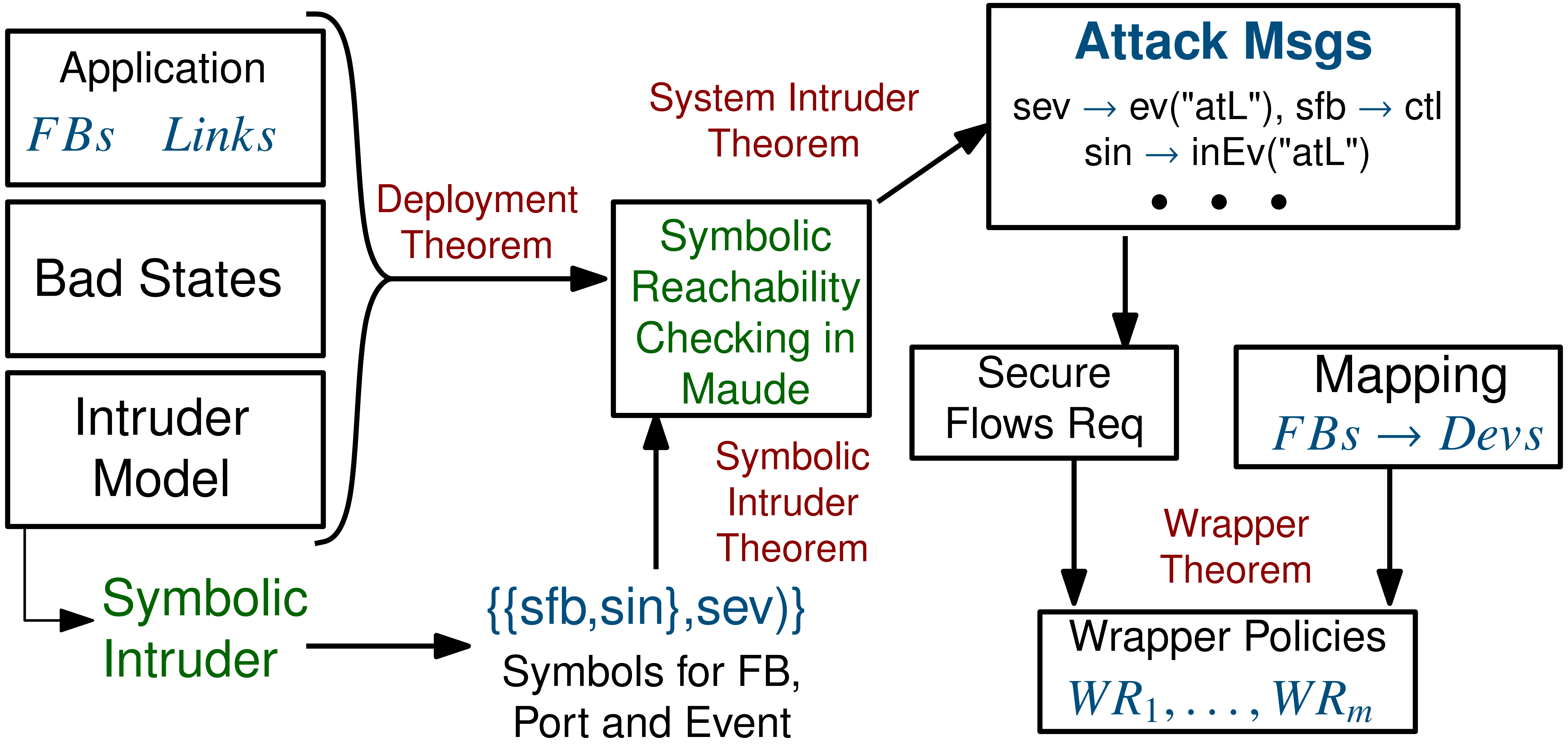
- Specification of a bad state -- the arm drops its load mid traversal

```
ceq badState(vacFB trackFB fbs) = true
 if cidOf(vacFB) == vac
 /\ cidOf(trackFB) == track
 /\ (getState(vacFB) == st("off")  or getState(vacFB) == st("on-novac"))
 /\ getState(trackFB) == st("mvL") .

eq badState(fbs) = false [owise] . .
```

- Command to check if PnP can reach a bad state.

```
search [1] in PNP-SCENARIO : pnpInit(emsgStart) =>+ app:Application
       such that badState(app:Application) = true .

No solution.
states: 24  rewrites: 7326 in 3ms cpu (3ms real) (2121019 rewrites/second)
```

Formal methodology



Space of PnP Models

- Model ~ (Module,state)
  - (App, A) A ~ [appId | fbs,iemsgs,oemsgs] — design level
  - (AppI,Ac) — with concrete intruder
  - (AppI,As) —with symbolic intruder
  - (Sys, S) — system model S ~ [sysId | devs, imsgs, omsgs]
  - (SysI,Si) — system model with intruder (ground)

Transform to add Intruders

- addIc(App,A,n) ~ (AppI, [A,allMsgs,n]) — concrete intruder
- addIs (App,A,n) ~ (AppI,[A,smsgs]) -- |smsgs| = n , symbolic intruder
  - AppI is App + rule to deliver intruder messages
- <u>Symbolic Intruder Theorem:</u> (AppI, [Ac,cmsgs,n]) ~ (AppI,[As,smsgs]) *Each execution of an application A in a symbolic intruder environment has a corresponding execution of A in the concrete intruder environment with the same bound, and conversely.*
- The key to this result is the soundness and completeness of the symbolic match generation used to enumerate deliverable messages.
- Thus search for attacks in either model the finds same attacks.

Deployment transformation

- deployApp(sysId,A,idmap) = S ~ [sysId | devs, imsgs, omsgs]
  - idmap maps function blocks to devices
- deployAppM(App,idmap) = Sys -- App + rules for gathering and distributing messages between devices
- <u>Deployment Theorem</u>: Executions of an application A and a deployment S of A are in close correspondence (stuttering bisimilar). In particular, the underlying function block transitions are the same and thus properties that depend only on function block states are preserved.
- <u>deployApp</u> provides the correspondence between states of A and of S

Deployment with Intruder

- deployAppI(sysId,(A,emsgs),idmap) =

  [deployApp[sysId,A,idmap], deployMsgs(emsgs,appLinks(A),idmap)]

- deployAppIM(Sys,idmap)

  = deployAppI(Sys,idmap) + rl[app-intruder] lifted to rl[sys-intruder]

- <u>System Intruder Theorem:</u> Let (App,A) be an application model and (Sys,S) be a deployment of (App,A).

  1. For any execution of S in an intruder environment there is a corresponding execution of A in that environment; and

  2. For any execution of A in an intruder environment that does not deliver any intruder messages that should flow on links internal to some device, has a corresponding execution from S in that environment.

Wrapping to secure vulnerable communications

- We define the function getBadEMsgs([A,smsgs]) that returns the set of injected message sets that lead to badState. This function uses reflection to enumerate search paths reflecting the command
    - search [A,smsgs] =>+ appInt:AppIntruder such that badState(appInt:AppIntruder) .
- wrapApp(A,smsgs,idmap) = wrapSys(deployApp(sysId,A,idmap),flatten(getBadEMsgs([A,smsgs])))
- wrapAppM(App,idmap) = wSys — Sys plus Policies for message signing.
- <u>Wrapper Theorem</u> : Let A be an application, S a deployment of A, and emsgs a set of messages containing the attack messages enumerated by symbolic search with an n bounded intruder. The wrapped system wrap(S,emsgs) is resistant to attacks by an n bounded intruder.

Summary

- We presented a simple example of the power of formal patterns to supprt high quality system design and correct by construction implementations.
- Some other examples of patterns
  - PALS — Physically Asynchronous Logically Synchronous architectural pattern for design of distributed real-time systems including medical devices.
  - Distribution transform — concurrent model to distributed message passing
  - Probablistic transform — for performance analysis
- Future directions include various forms of symbolic rewriting

Some References

- Meseguer, J., 2014. Taming distributed system complexity through formal patterns. Sci. Comput. Program. 83, 3–34. doi:10.1016/j.scico.2013.07.004
  - General theory and many examples
- Vivek Nigam and Carolyn Talcott. Automated construction of security integrity wrappers for Industry 4.0 applications. In The 13th International Workshop on Rewriting Logic and its Applications, LNCS, 2020. Journal version to appear in JLAMP.
  - Details of the PnP case study and experimental results.

- The Maude code along with documentation, scenarios, and sample runs can be found at
  - https://github.com/SRI-CSL/WrapPat.git.

?? Questions ??