

Filtering Test Models to Support Incremental Testing

Antti Jääskeläinen

Tampere University of Technology
Department of Software Systems

Outline

- Background and Theory
- Filtering
- Connectivity Algorithm
- Example
- Conclusion

Background and Theory

Model-Based Testing

- Automates the generation of tests
- Tests generated from a formal test model
- Test model describes the functionality to be tested
- Off-line testing: tests are generated first and executed later
- Online testing: tests are executed as they are generated

Model Formalism

- Our models are LSTS (labeled state transition system) state machines
- Events encoded into actions (transition labels)
- State labels used for auxiliary information
- Models are strongly connected
 - Test generation never ends in a deadlock

Parallel Composition

- Realistic systems are too large to model in a single state machine
- We create several smaller model components and combine them with parallel composition
- In parallel composition some actions of individual model components are executed synchronously
 - For example, actions of the same name in different model components always executed together

Rule-based composition

- Synchronized actions defined explicitly
- For example $\{(a, \surd, 1a), (\surd, b, 2b), (c, c, c)\}$
 - Rules for two model components
 - First model component executes a alone as $1a$
 - Second model component executes b alone as $2b$
 - Both model components execute c together as c

Filtering

Motivation

- Models in product lifecycle
 - Models can be completed before the SUT
 - Complete test model may generate tests for features not yet implemented
 - Unimplemented features must be avoided
- Bugs in the SUT
 - Test execution terminates when a bug is found
 - Until the bug is fixed, further tests may terminate in the same place
 - Bugged features must be avoided

Solution

- Obtain a *filtered* view of the model with troublesome parts removed
- Performed by banning individual actions (and related transitions)
- Tests generated from the filtered model will not encounter unexecutable features

Retaining Connectivity

- Banning arbitrary actions can break strong connectivity
- Other actions must also be banned to restore the connectivity
- Test model may be too large for proper connectivity calculation
- Additional actions to be banned must be found by other means

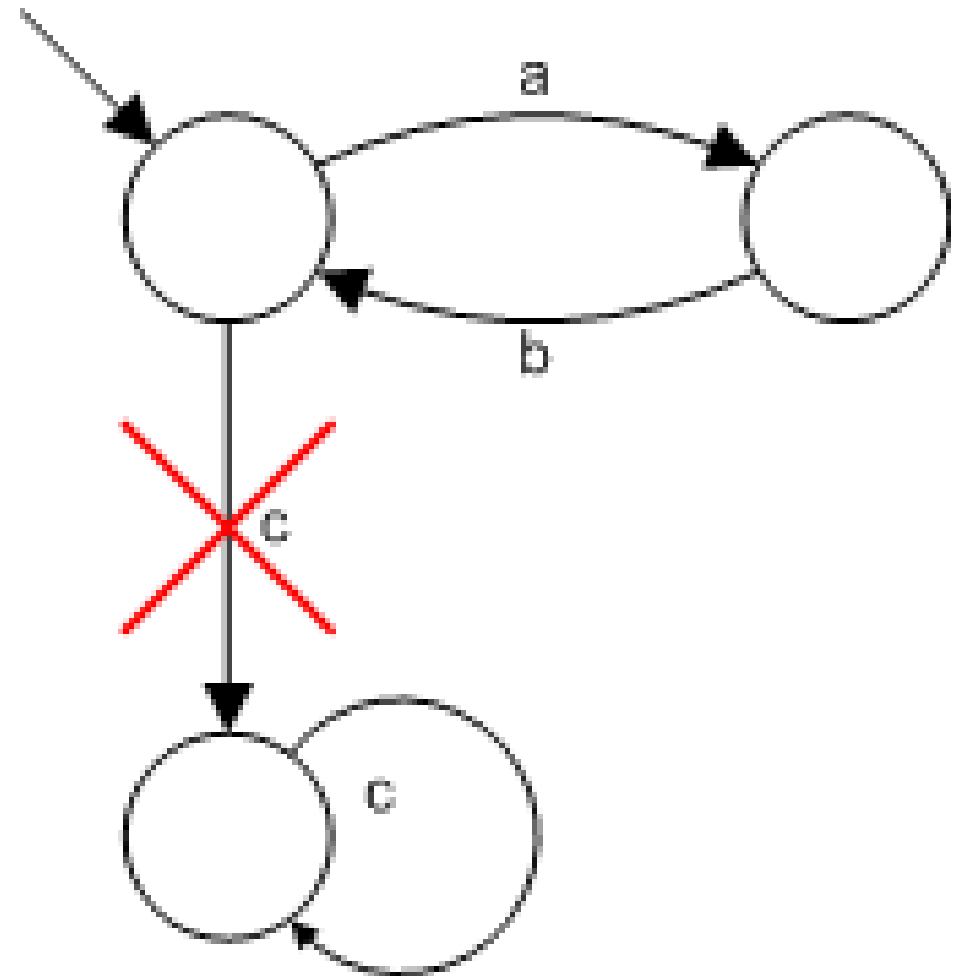
Connectivity Algorithm

Purpose and Methodology

- Algorithm seeks to ban all actions whose execution may lead to violation of strong connectivity
- Based on four methods for finding actions to be banned
- Methods applied in turns until no further progress can be made
- Methods work on model components, not composed model

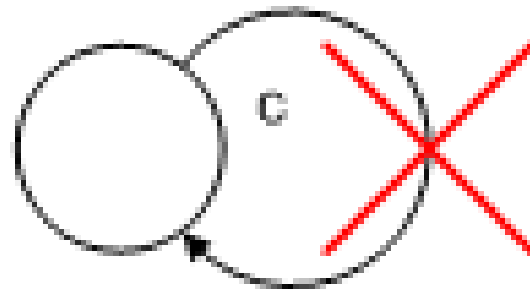
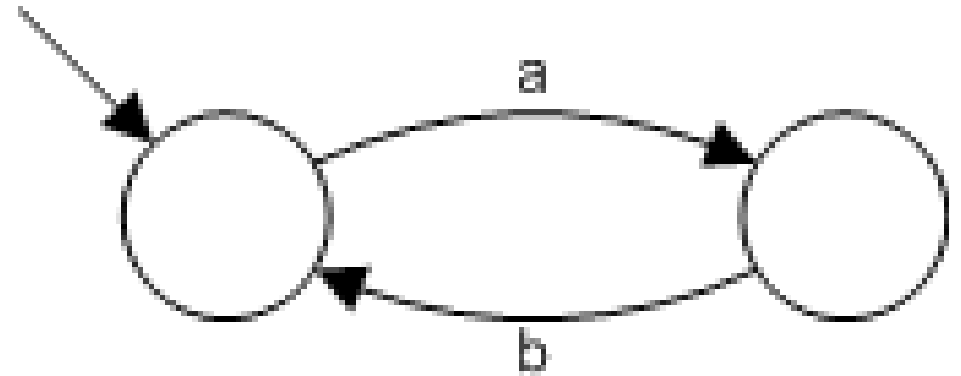
Actions Breaking Connectivity

- Actions leading out of the initial strong component (the strong component containing the initial state) must be banned
- The most important of the methods



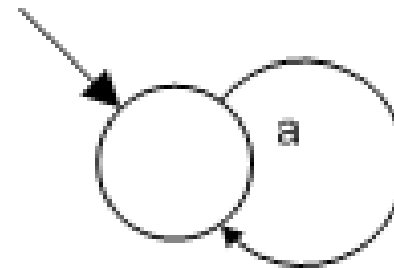
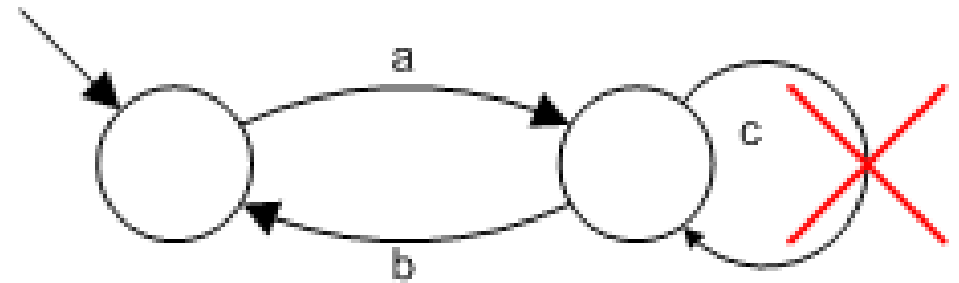
Unreachable Actions

- Unreachable actions may be banned
- Not useful in itself, but may allow other actions to be banned later on



Actions without Rules

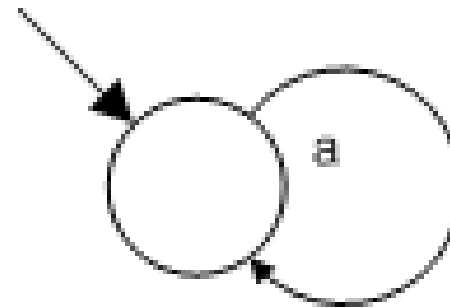
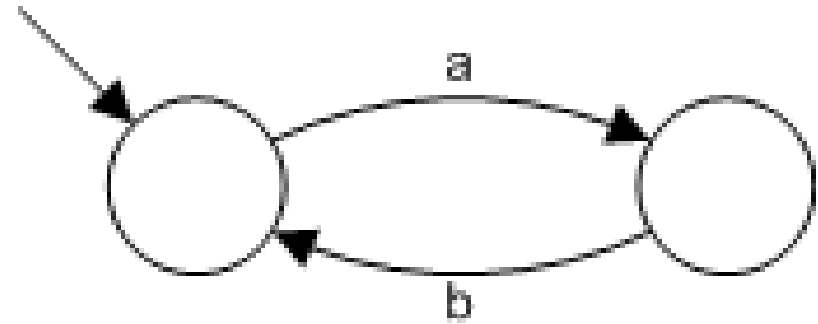
- Actions may be banned if there are no rules which allow their execution



$$R = \{(a, a, a), \\ (b, \surd, b)\}$$

Rules without Actions

- Composition rule may be removed if one of its actions is banned or otherwise unavailable



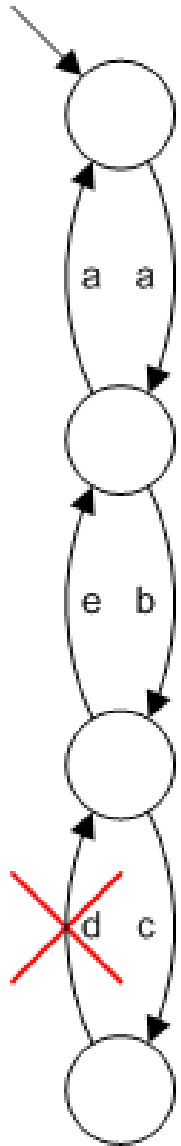
$$R = \{(a, a, a), \\ \text{~~(b, b, b)~~\}$$

End Result

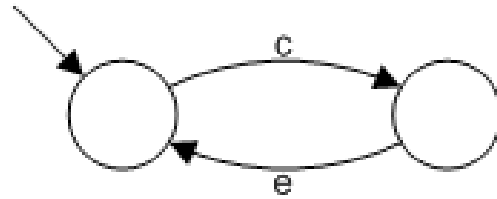
- Algorithm calculates an upper bound for initial strong component
 - Some remaining actions may still break connectivity
- Modeler has to make sure that the bound will be accurate
- In our experience this is not difficult to ensure

Example

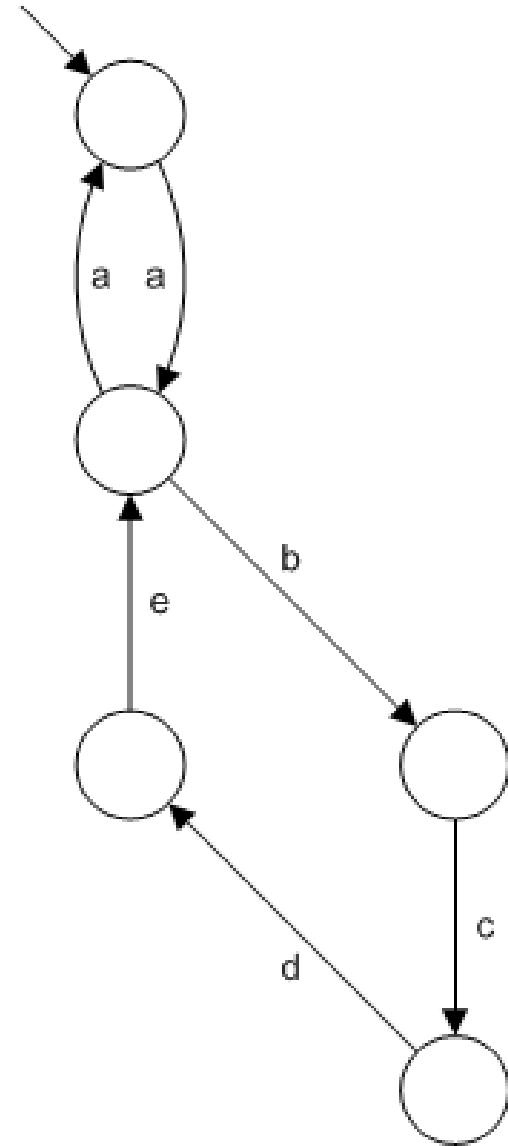
Model 1



Model 2

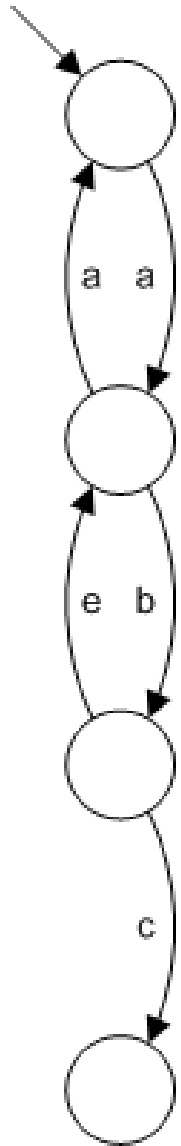


Model 1 x Model 2

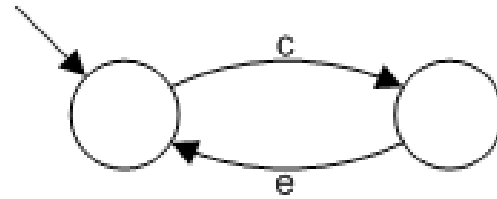


$R = \{(a, \surd, a),$
 $(b, \surd, b),$
 $(c, c, c),$
 $(d, \surd, d),$
 $(e, e, e)\}$

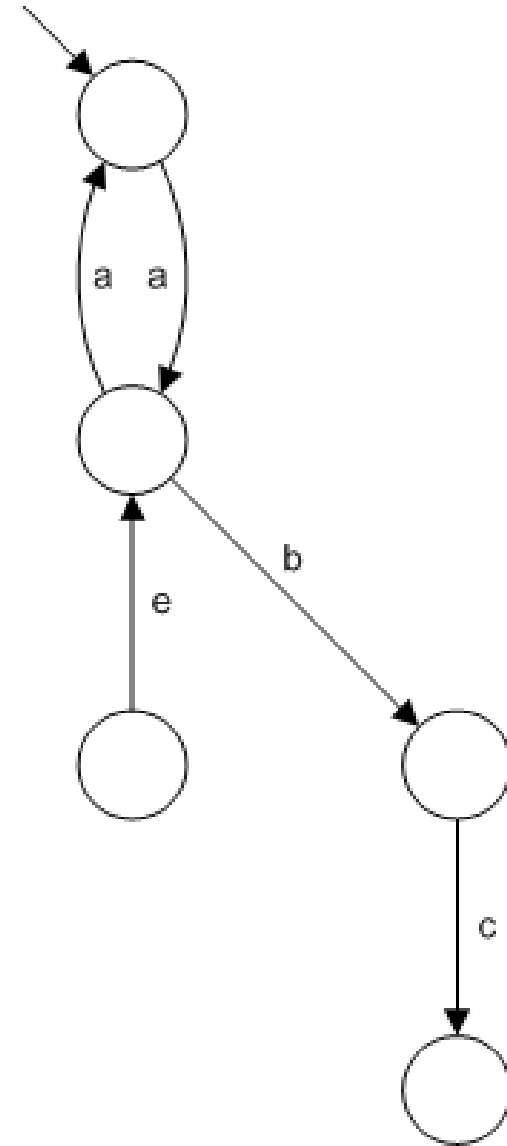
Model 1



Model 2

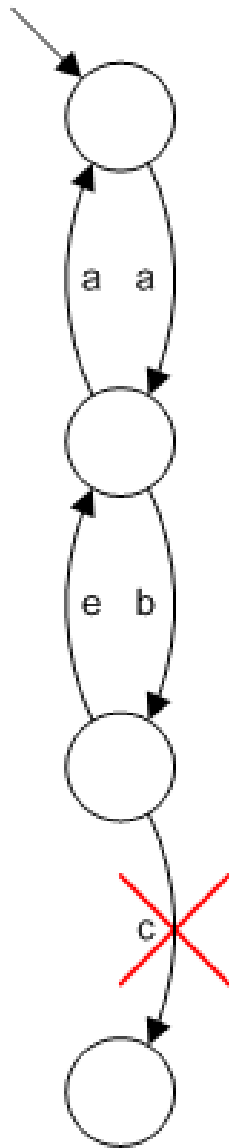


Model 1 x Model 2

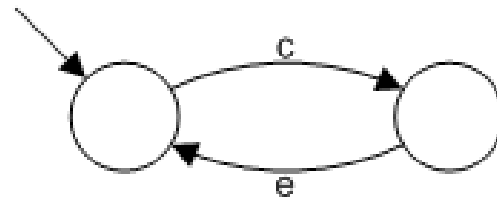


$R = \{(a, \checkmark, a),$
 $(b, \checkmark, b),$
 $(c, c, c),$
 $(\cancel{d}, \checkmark, \cancel{d}),$
 $(e, e, e)\}$

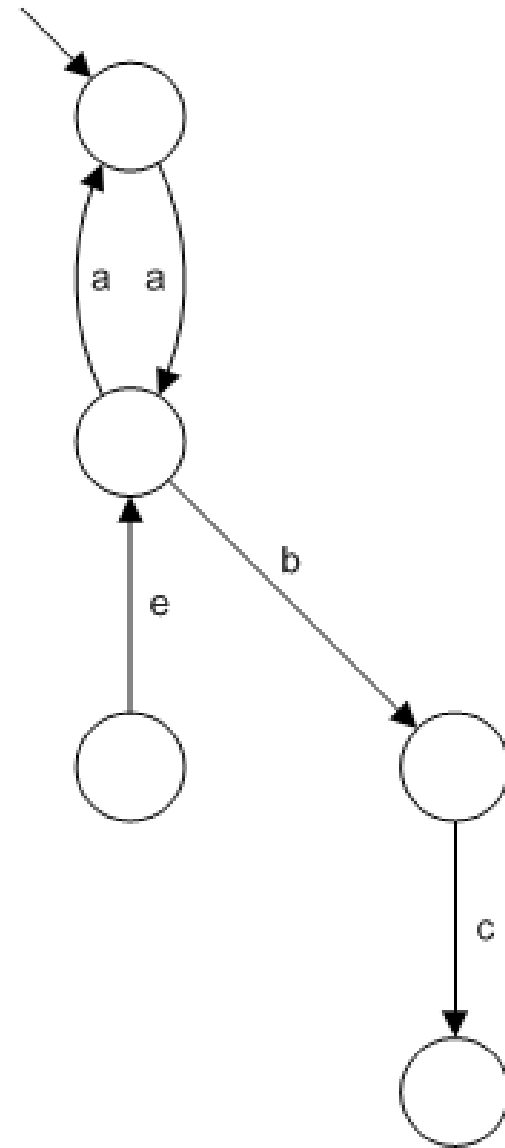
Model 1



Model 2

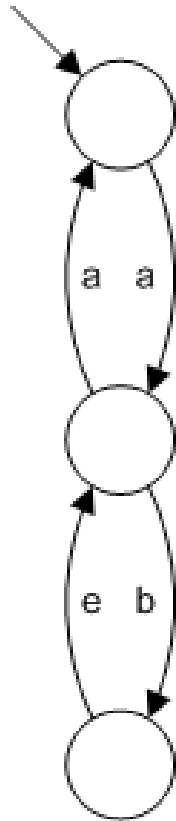


Model 1 x Model 2

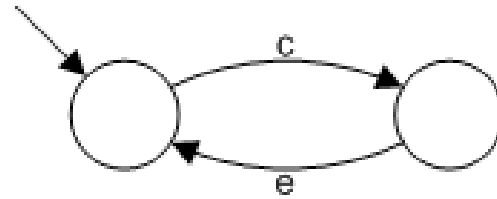


$$R = \{(a, \surd, a), (b, \surd, b), (c, c, c), (e, e, e)\}$$

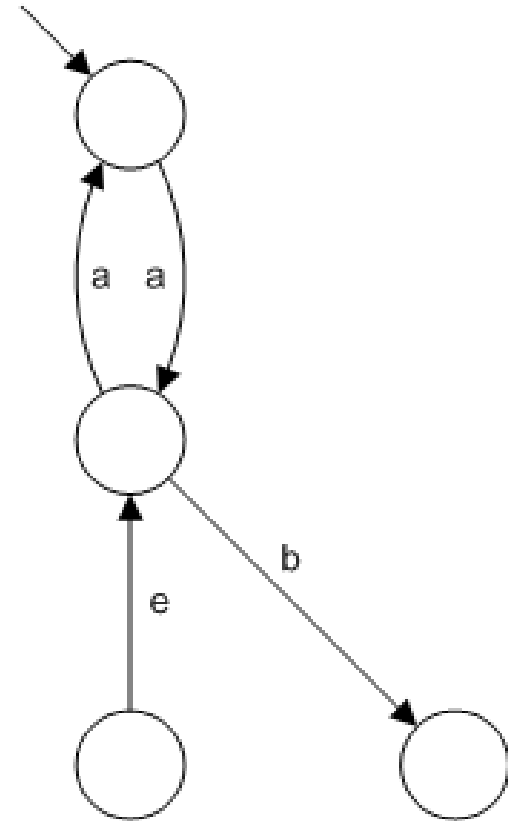
Model 1



Model 2



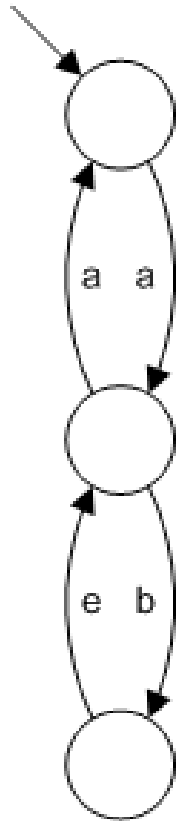
Model 1 x Model 2



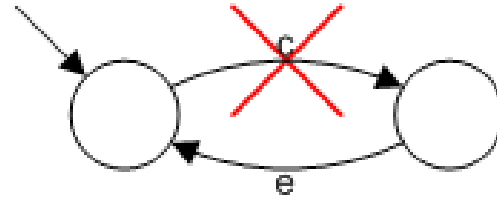
$$R = \{(a, \checkmark, a), (b, \checkmark, b), \cancel{(c, c, c)}, (e, e, e)\}$$



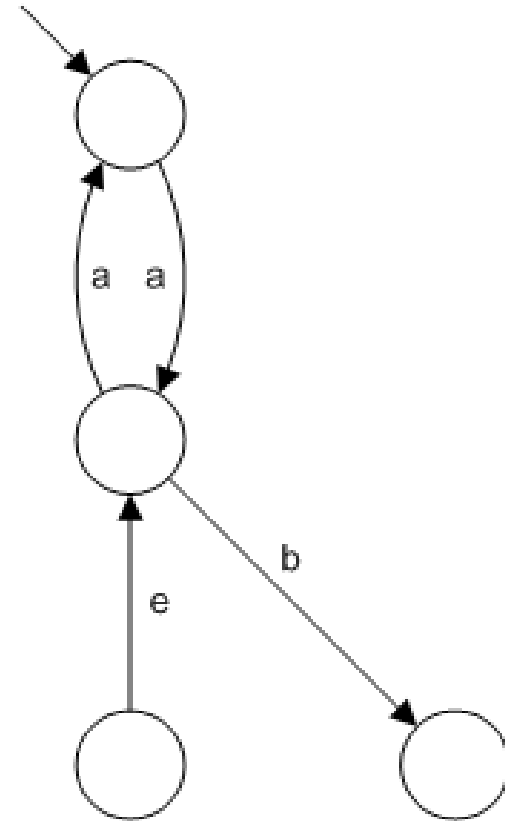
Model 1



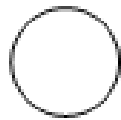
Model 2



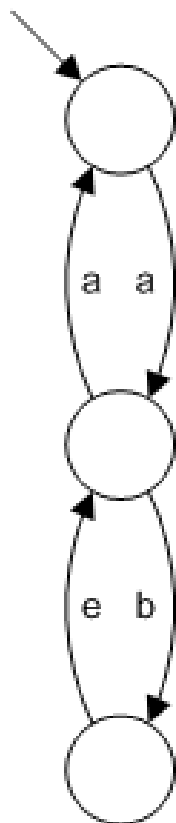
Model 1 x Model 2



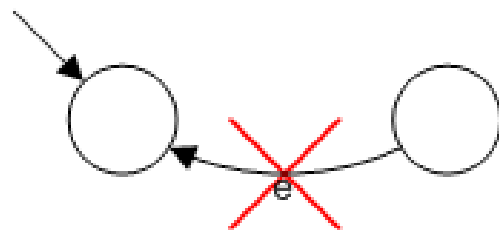
$$R = \{(a, \checkmark, a), (b, \checkmark, b), (e, e, e)\}$$



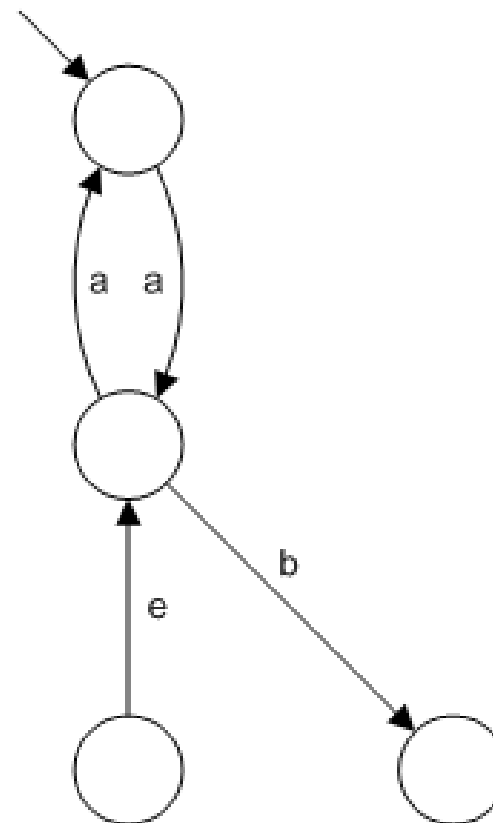
Model 1



Model 2



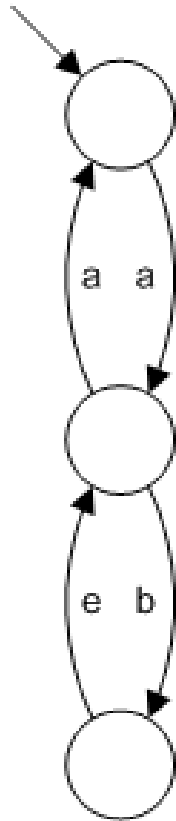
Model 1 x Model 2



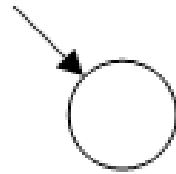
$$R = \{(a, \checkmark, a), \\ (b, \checkmark, b), \\ (e, e, e)\}$$



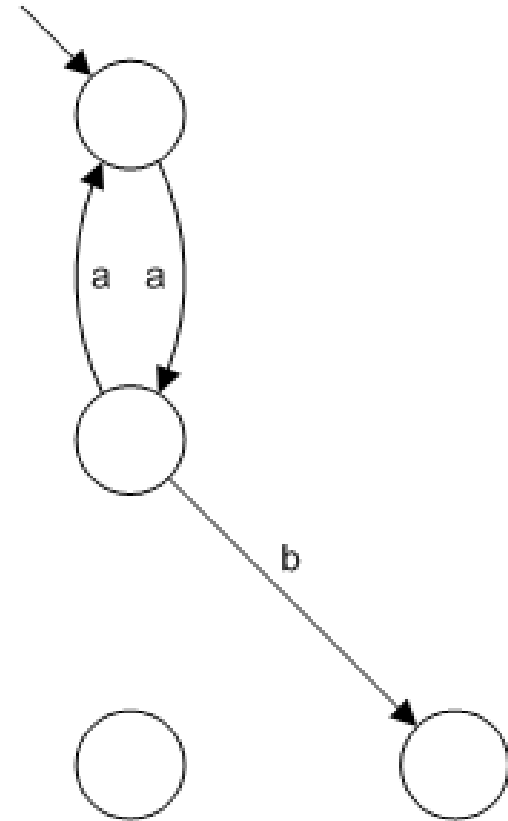
Model 1



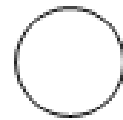
Model 2



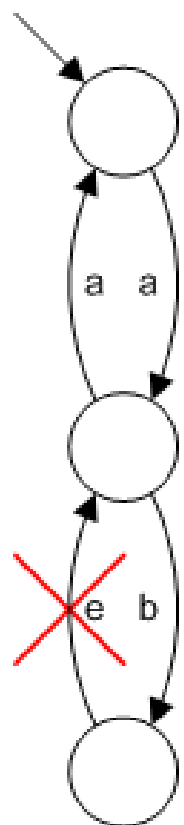
Model 1 x Model 2



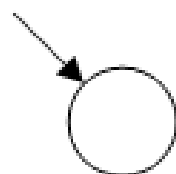
$$R = \{(a, \checkmark, a), (b, \checkmark, b), (e, e, e)\}$$



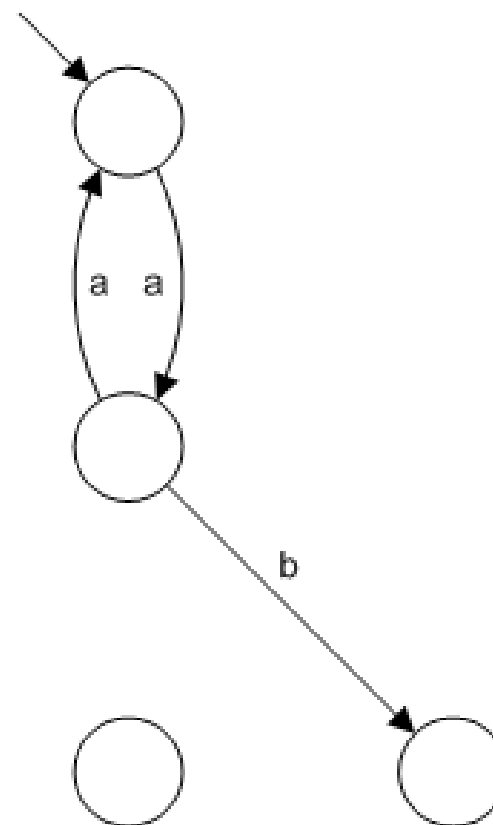
Model 1



Model 2



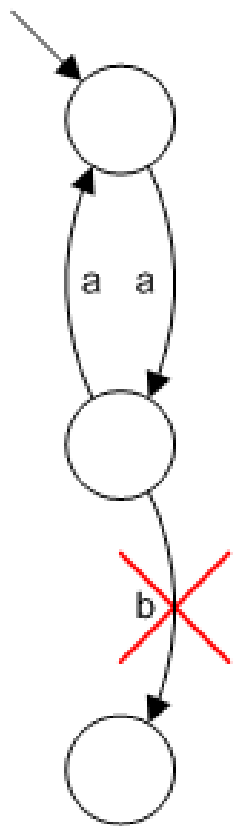
Model 1 x Model 2



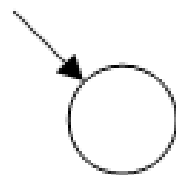
$$R = \{(a, \checkmark, a), (b, \checkmark, b)\}$$



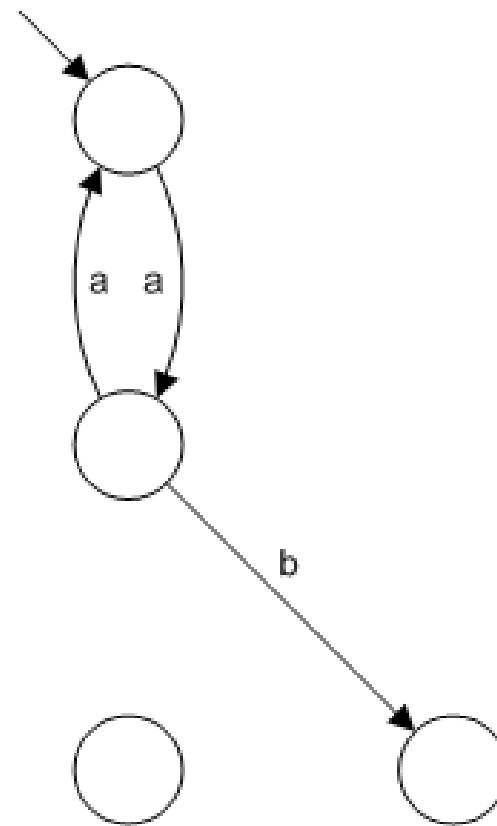
Model 1



Model 2



Model 1 x Model 2



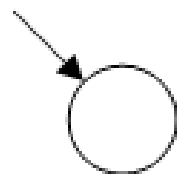
$$R = \{(a, \vee, a), (b, \vee, b)\}$$



Model 1



Model 2



Model 1 x Model 2



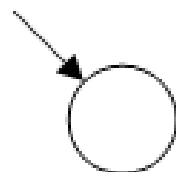
$$R = \{(a, \checkmark, a), \\ (b, \checkmark, b)\}$$



Model 1



Model 2



Model 1 x Model 2



$$R = \{(a, \vee, a)\}$$



Conclusion

Results

- Filtering methodology can be used generate tests that avoid unexecutable functionality
- Models must be restored to strong connectivity when actions are removed; this can be mostly automated
- Some effort in modeling is required to ensure compatibility

Future Work

- Applying the filtering methodology to other forms of parallel composition
- Filtering non-behavioral models and data

Thank You