Efficient Static Analysis and Verification of Featured Transition Systems

Maurice ter Beek — ISTI–CNR, Pisa
Ferruccio Damiani — University of Turin
Michael Lienhardt — ONERA, France
Franco Mazzanti — ISTI–CNR, Pisa
Luca Paolini — University of Turin

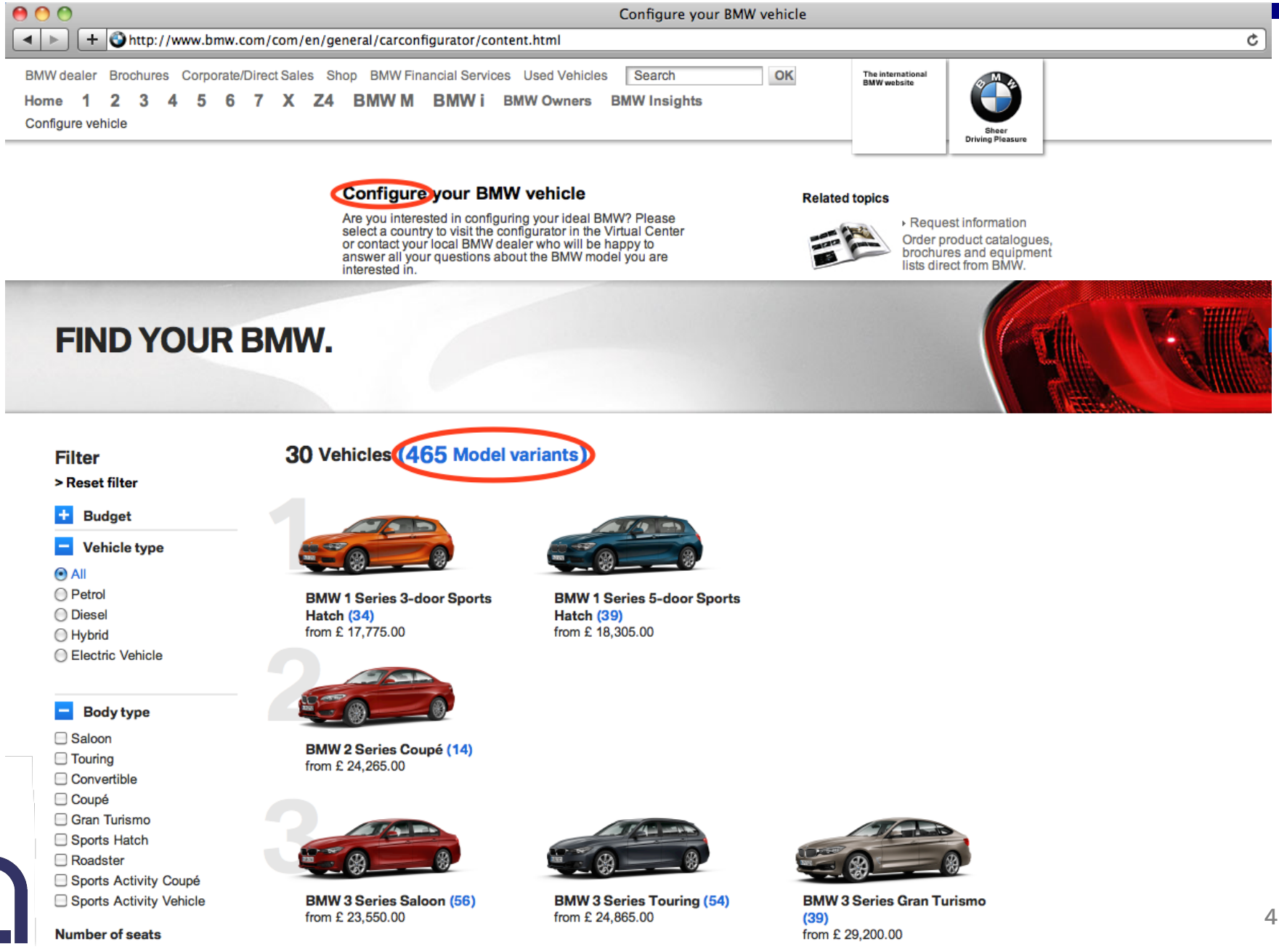
Informatics Seminars 2020/21
Leicester, UK
March 5, 2021

Outline

Software Product Lines (SPLs)

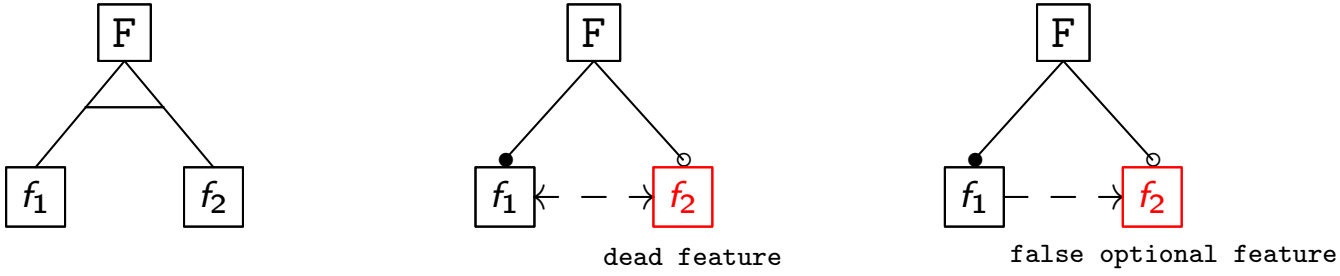
- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user

Product lines are everywhere



Feature models

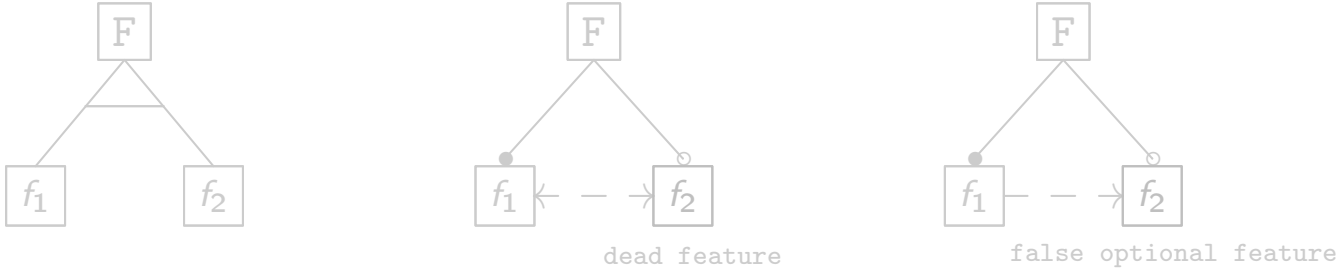
- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user



Benavides *et al.*, Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.*, 2010

SPL analysis

- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user



Benavides *et al.*, Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.*, 2010

- Popular in embedded and critical systems domain: formal modelling and analysis techniques for proving SPL behaviour correct are widely studied
  Thüm et al., A classification and survey of analysis strategies for SPLs. *ACM Comput. Surv.*, 2014
- Challenge existing formal methods and tools by potentially high number of different variants, each giving rise to a large state space, in general

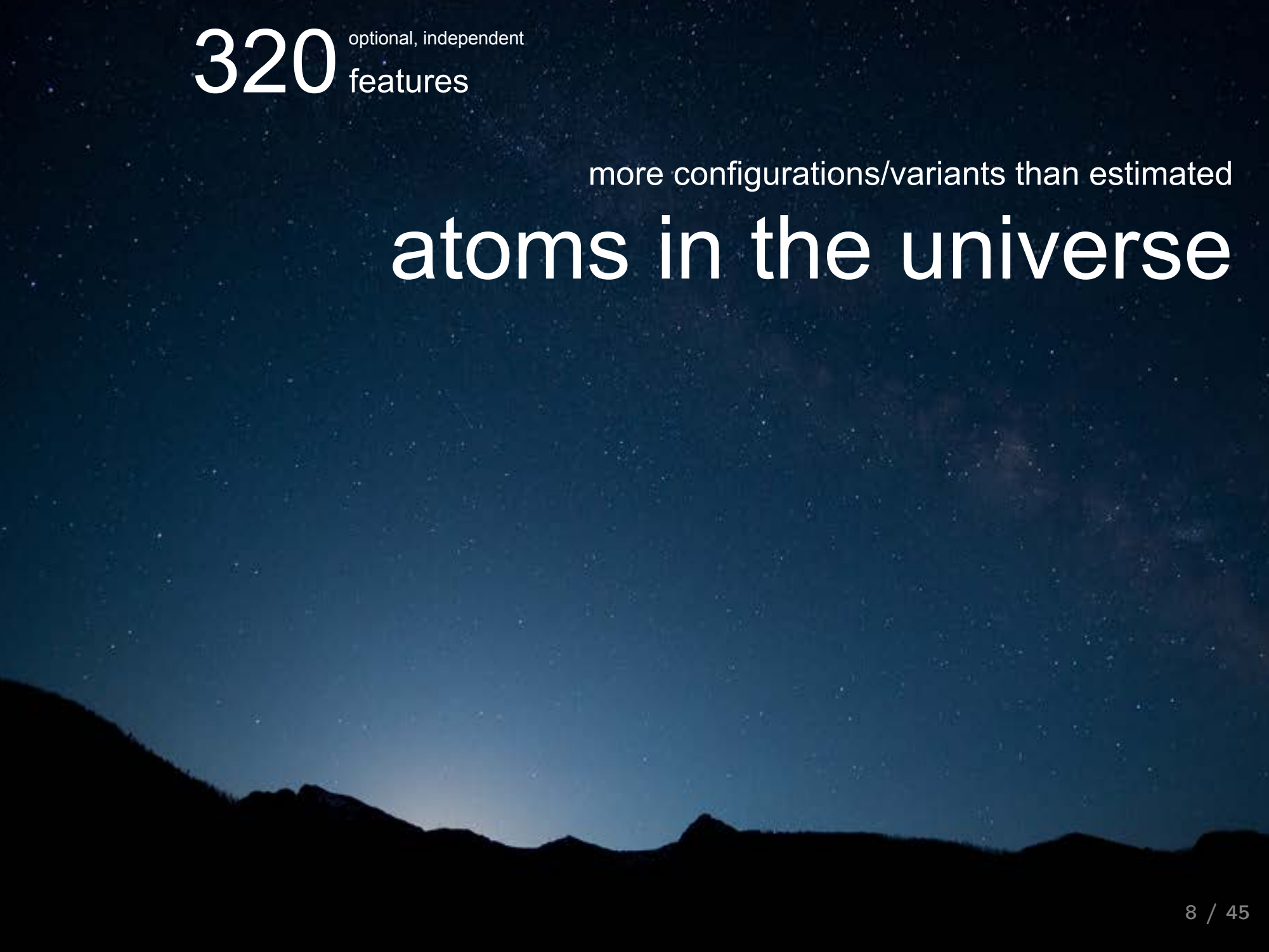
Scalability is an issue

Examples by C. Kästner (CMU, Pittsburgh, USA)

33 optional, independent features

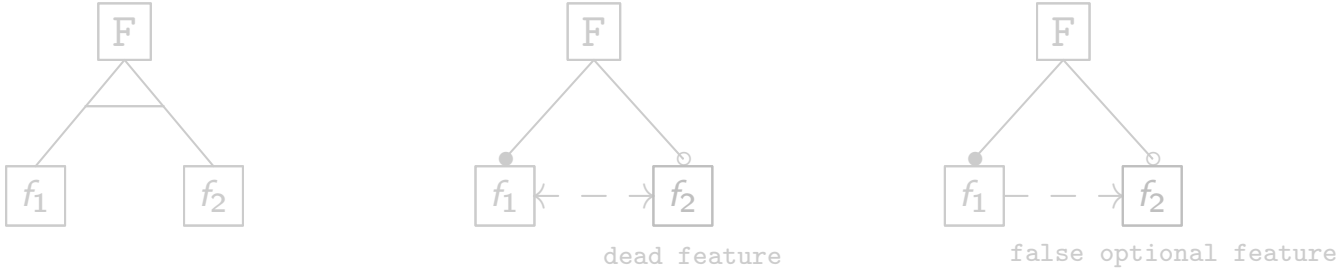
a unique configuration/variant for every

person on this planet


320 optional, independent features
more configurations/variants than estimated
atoms in the universe

SPL analysis

- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user



Benavides *et al.*, Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.*, 2010

- Popular in embedded and critical systems domain: formal modelling and analysis techniques for proving SPL behaviour correct are widely studied

Thüm et al., A classification and survey of analysis strategies for SPLs. *ACM Comput. Surv.*, 2014

- Challenge existing formal methods and tools by potentially high number of different variants, each giving rise to a large state space, in general

⇒ Lift success stories known for single systems (products) to sets of products (families) by exploiting variability modelling and analysis

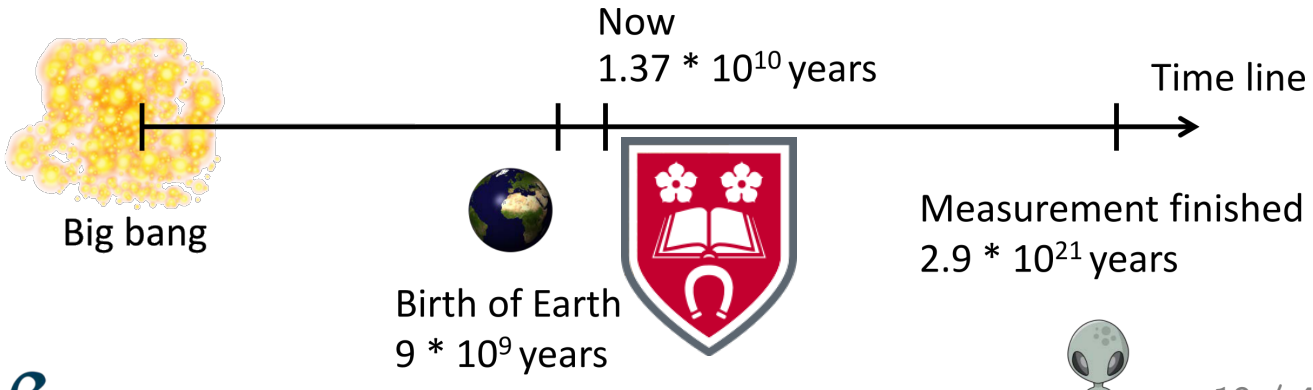
Product-based analysis:

- Simple, brute-force approach
- Make use of standardly available, highly optimised analysis tools

Product-based analysis: inefficient, if not infeasible (example by S. Apel, Saarland University, Germany)

- 👍 Simple, brute-force approach
- 👍 Make use of standardly available, highly optimised analysis tools

- 👎 Number of product variants is exponential in number of features
- 👎 Same piece of behaviour (or code) is verified numerous times, as many times as the number of variants that are able to execute it



SQLite (88 features)

Solution: family-based analysis

- Beneficial in case of many products with substantial similarities
- Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

Solution: family-based analysis

- 👍 Beneficial in case of many products with substantial similarities
- 👍 Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

- 👎 More complex analysis tasks
- 👎 Requires (compact) family models (superimposed, *150% models*)

Solution: family-based analysis

- Beneficial in case of many products with substantial similarities
- Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

- More complex analysis tasks
- Requires (compact) family models (superimposed, *150% models*)

- Dedicated variability-based models, logics, and model checkers (e.g. FTSs, Feature Nets, PL-CCS, fLTL, fCTL, v-ACTL, QFLan, SNIP, VMC)

Solution: family-based analysis

- Beneficial in case of many products with substantial similarities
- Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

- More complex analysis tasks
- Requires (compact) family models (superimposed, *150% models*)

- Dedicated variability-based models, logics, and model checkers (e.g. FTSs, Feature Nets, PL-CCS, fLTL, fCTL, v-ACTL, QFLan, SNIP, VMC)

- Dedicated model checkers need to be maintained and optimised

Solution: family-based analysis

- Beneficial in case of many products with substantial similarities
- Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it

- More complex analysis tasks
- Requires (compact) family models (superimposed, *150% models*)

- Dedicated variability-based models, logics, and model checkers (e.g. FTSs, Feature Nets, PL-CCS, fLTL, fCTL, v-ACTL, QFLan, SNIP, VMC)

- Dedicated model checkers need to be maintained and optimised

Dimovski et al., Family-based model checking without a family-based model checker @ SPIN'15
Chrszon *et al.*, Family-based modeling and analysis for probabilistic systems: featuring ProFeat @ FASE'16
ter Beek *et al.*, Family-Based Model Checking with mCRL2 @ FASE'17
Dimovski *et al.*, Variability-specific Abstraction Refinement for Family-based Model Checking @ FASE'17
Dimovski, Abstract Family-Based Model Checking Using Modal Featured Transition Systems @ FASE'18
ter Beek *et al.*, Family-Based SPL Model Checking Using Parity Games with Variability @ FASE'20
...

Key idea and aim of today's talk

Mimick anomaly detection known from feature model analysis in behavioural SPL models (FTSs) by automated static analysis:

1. dead transitions
2. false optional transitions
3. hidden deadlock states

Key idea and aim of today's talk

Mimick anomaly detection known from feature model analysis in behavioural SPL models (FTSs) by automated static analysis:

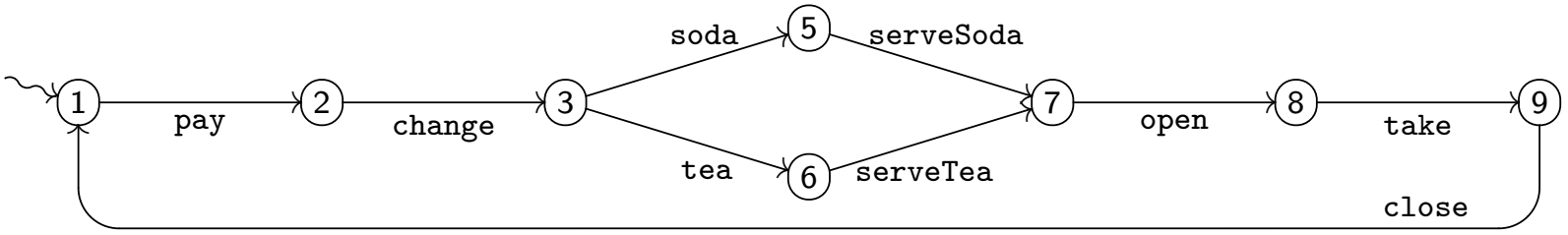
1. dead transitions
2. false optional transitions
3. hidden deadlock states

Catch and offer means to remove possible ambiguities in FTSs:

1. Ambiguous FTSs are undesired: they give unclear ideas of the SPLs
2. Unambiguous FTSs pave way to efficient family-based verification

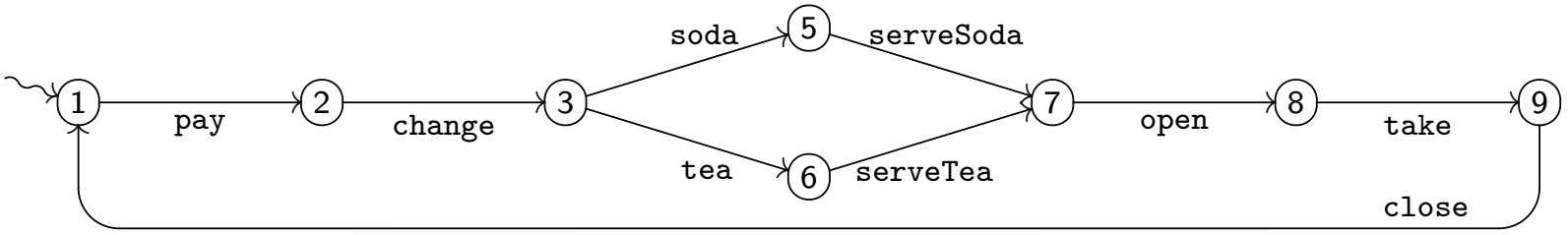
Featured Transition Systems (FTS)

A Labelled Transition System (LTS) is a quadruple (S, Σ, s_0, δ) with states S, actions Σ, initial state s_0, and transitions $\delta \subseteq S \times \Sigma \times S$



Featured Transition Systems (FTS)

A Labelled Transition System (LTS) is a quadruple (S, Σ, s_0, δ) with states S, actions Σ, initial state s_0, and transitions $\delta \subseteq S \times \Sigma \times S$



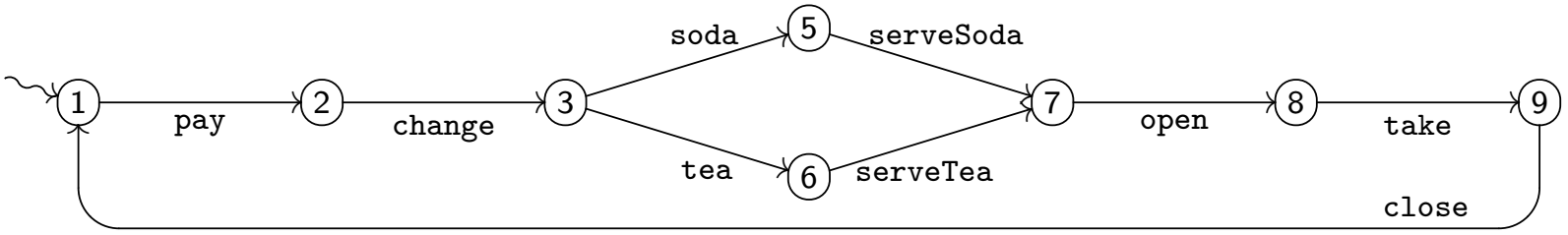
An FTS adds to this a feature model and feature expressions:

Classen *et al.*, Model checking lots of systems @ ICSE'10, FTSs. *IEEE TSE*, 2013

A Featured Transition System (FTS) is a sextuple $(S, \Sigma, s_0, \delta, F, \Lambda)$ with states S, actions Σ, initial state s_0, (featured) transitions $\delta \subseteq S \times \Sigma \times \mathbb{B}(F) \times S$, with Boolean (feature) expressions $\mathbb{B}(F)$ over features F, and (product) configurations $\Lambda \subseteq \{ \lambda : F \to \mathbb{B} \}$

Featured Transition Systems (FTS)

A Labelled Transition System (LTS) is a quadruple (S, Σ, s_0, δ) with states S, actions Σ, initial state s_0, and transitions $\delta \subseteq S \times \Sigma \times S$



An FTS adds to this a feature model and feature expressions:

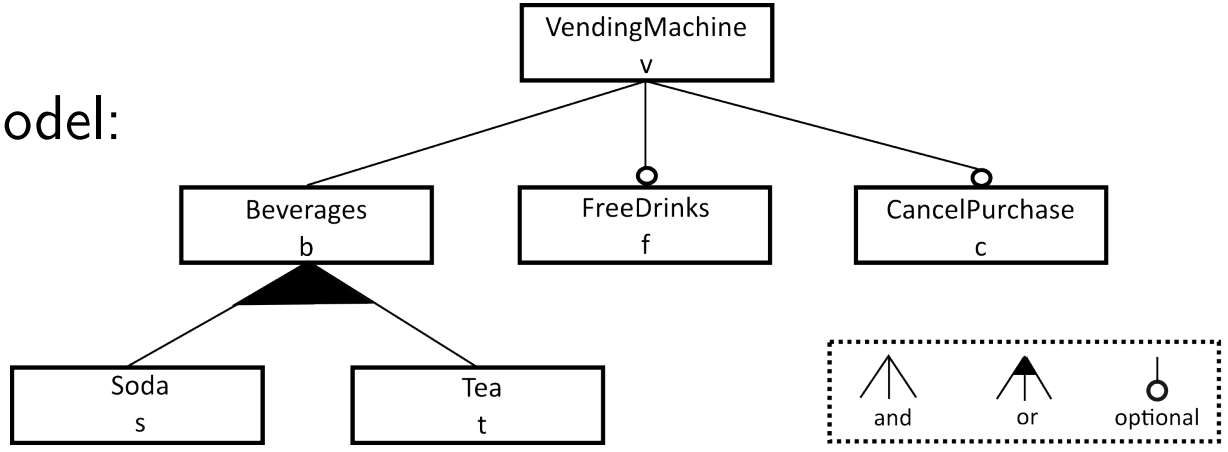
Classen *et al.*, Model checking lots of systems @ ICSE'10, FTSs. *IEEE TSE*, 2013

A Featured Transition System (FTS) is a sextuple $(S, \Sigma, s_0, \delta, F, \Lambda)$ with states S, actions Σ, initial state s_0, (featured) transitions $\delta \subseteq S \times \Sigma \times \mathbb{B}(F) \times S$, with Boolean (feature) expressions $\mathbb{B}(F)$ over features F, and (product) configurations $\Lambda \subseteq \{ \lambda : F \to \mathbb{B} \}$

LTS $\mathcal{F}|_\lambda$ specified by configuration $\lambda \in \Lambda$ is called a product of $\mathcal{F}$ [remove: 1) all featured transitions whose feature expressions are not satisfied by λ; 2) all unreachable states and their outgoing transitions]

FTS of example SPL: a vending machine

Feature model:

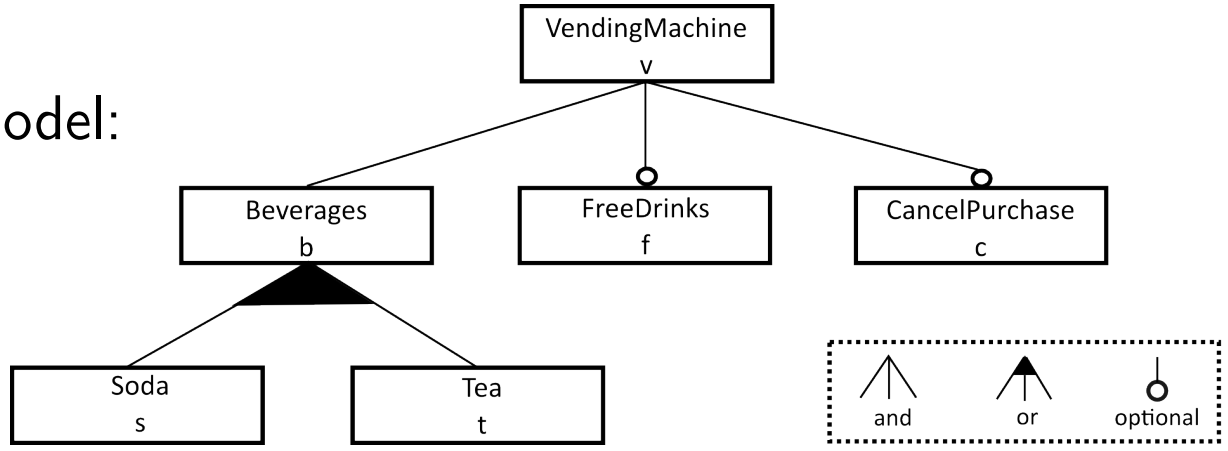


12 valid products

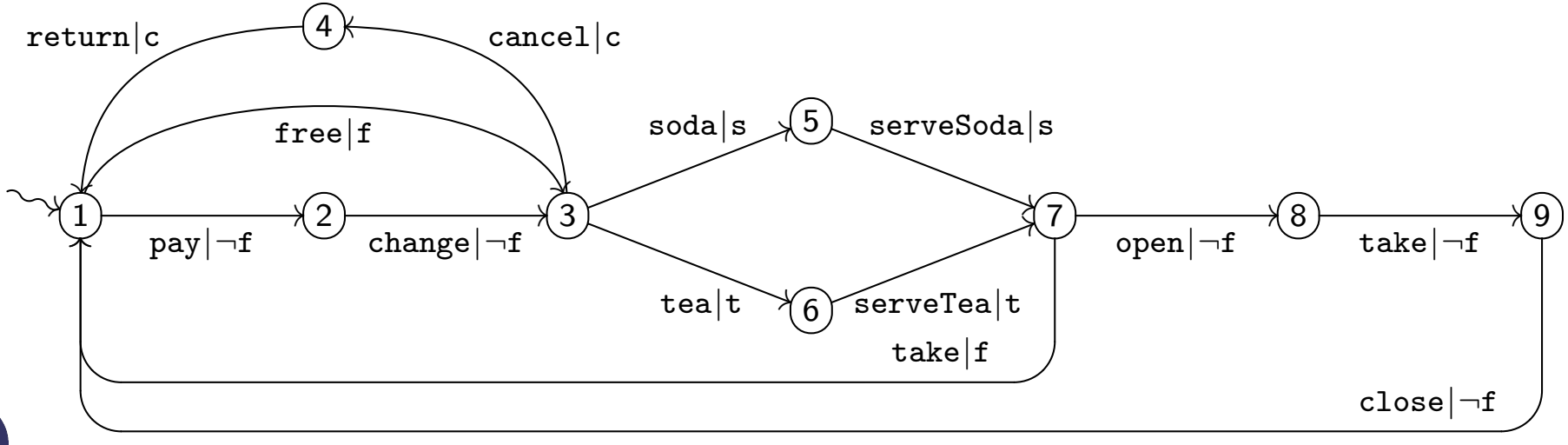
e.g., $\{\mathtt{v},\mathtt{b},\mathtt{s},\mathtt{t}\}$, $\{\mathtt{v},\mathtt{b},\mathtt{s},\mathtt{c}\}$

FTS of example SPL: a vending machine

Feature model:

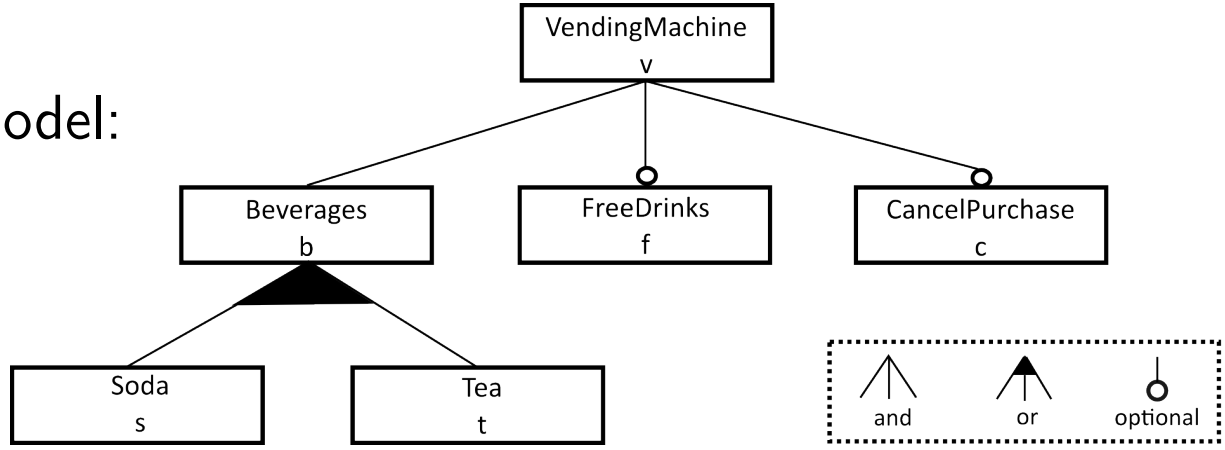


FTS of 12 valid products (LTSs) e.g., {v,b,s,t}, {v,b,s,c}

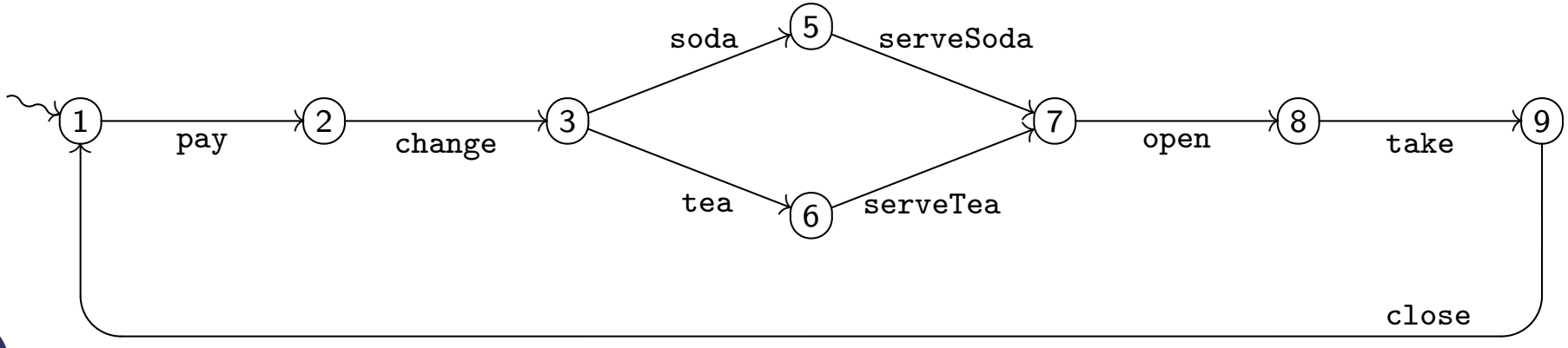


FTS of example SPL: a vending machine

Feature model:

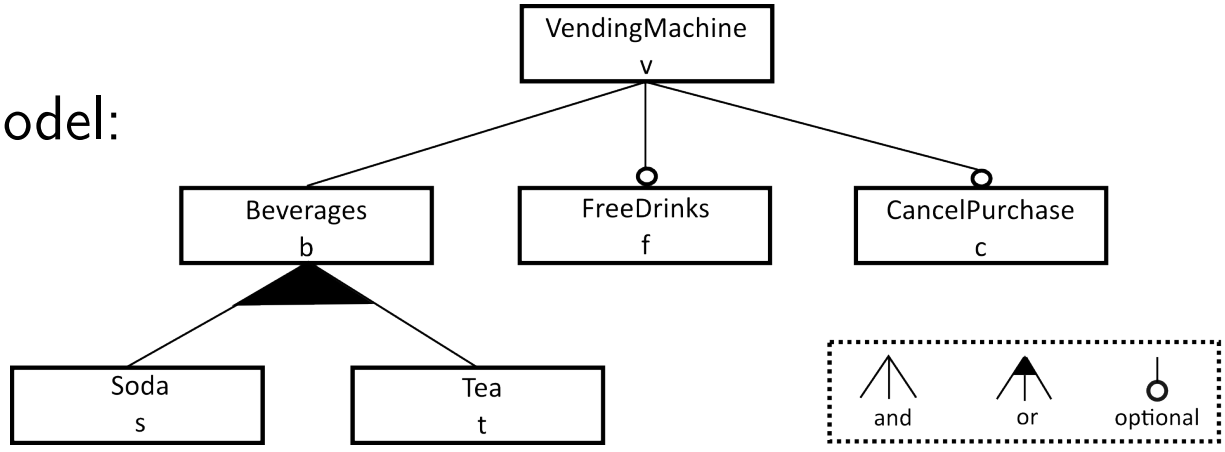


e.g., {v,b,s,t}, {v,b,s,c}

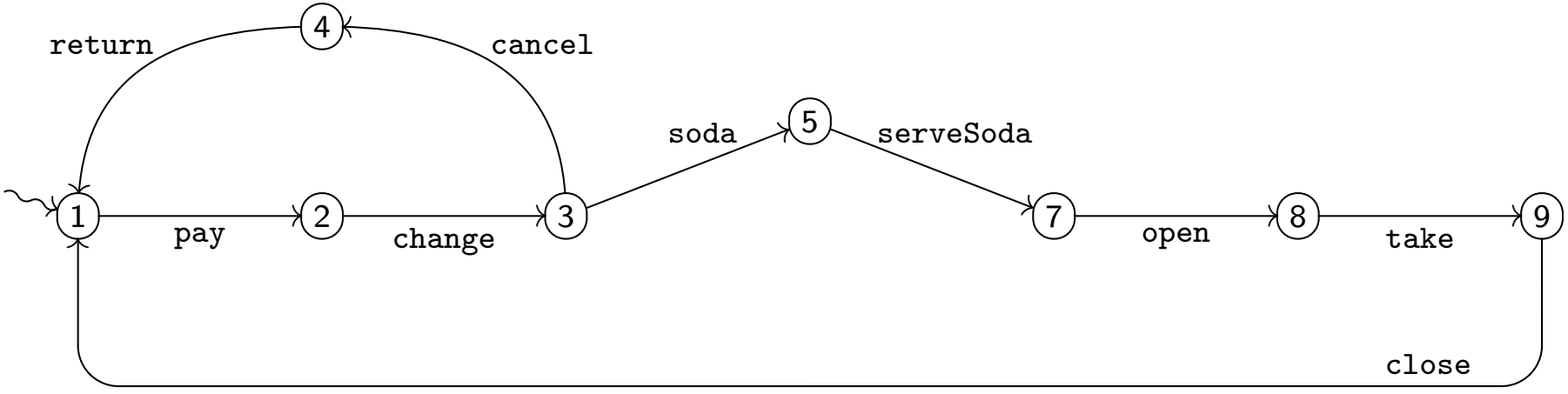


FTS of example SPL: a vending machine

Feature model:



e.g., {v,b,s,t}, {v,b,s,c}



Ambiguous FTS

dead transition an FTS transition not reachable in any product (LTS)

Ambiguous FTS

dead transition an FTS transition not reachable in any product (LTS)

false optional transition a featured FTS transition which is

1. not dead
2. not annotated with the feature expression $\top$ (true, i.e., selected)
3. present in every FTS product in which its source state is present

Ambiguous FTS

dead transition an FTS transition not reachable in any product (LTS)

false optional transition a featured FTS transition which is

1. not dead
2. not annotated with the feature expression $\top$ (true, i.e., selected)
3. present in every FTS product in which its source state is present

hidden deadlock state an FTS state which is

1. not a deadlock (i.e., it has outgoing transitions) in the FTS
2. a deadlock (i.e., no outgoing transitions) in some FTS product(s)

Important safety property: *deadlock freedom*, i.e., system should not reach a state in which no further action is possible, thus guaranteeing *progress* or *liveness*; for configurable systems, this notion is extended to guaranteeing liveness for each system variant (product)

Ambiguous FTS $\longrightarrow$ unambiguous FTS

Transformation:

1. remove dead transitions

Ambiguous FTS $\longrightarrow$ unambiguous FTS

Transformation:

1. remove dead transitions

2. turn false optional transitions into *must* transitions (i.e., labelled $\top$)

Ambiguous FTS $\longrightarrow$ unambiguous FTS

Transformation:

1. remove dead transitions
2. turn false optional transitions into *must* transitions (i.e., labelled $\top$)
3. make hidden deadlock states s explicit:

Step (3) must be performed only for the hidden deadlock states that have not yet become explicit deadlock states upon the removal of dead transitions in step (1)

Ambiguous FTS $\longrightarrow$ unambiguous FTS

Transformation:

1. remove dead transitions
2. turn false optional transitions into *must* transitions (i.e., labelled $\top$)
3. make hidden deadlock states s explicit:
   - 3.1 add a deadlock state $s_\dagger \notin S$

Step (3) must be performed only for the hidden deadlock states that have not yet become explicit deadlock states upon the removal of dead transitions in step (1)

Ambiguous FTS $\longrightarrow$ unambiguous FTS

Transformation:

1. remove dead transitions

2. turn false optional transitions into *must* transitions (i.e., labelled $\top$)

3. make hidden deadlock states s explicit:

   3.1 add a deadlock state $s_\dagger \notin S$

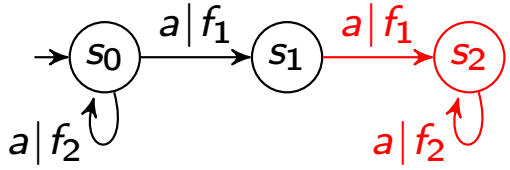
   3.2 $\forall s$: add a *deadlock transition* from s to $s_\dagger$ labelled by $\dagger \notin \Sigma$ and by a feature expression that negates the disjunction of the feature expressions of all outgoing transitions of s

Step (3) must be performed only for the hidden deadlock states that have not yet become explicit deadlock states upon the removal of dead transitions in step (1)

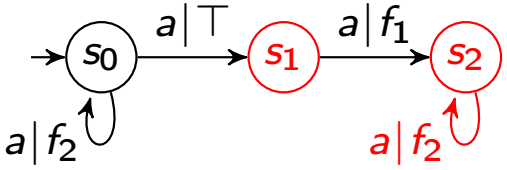
Example transformations

Feature Model: $f_1 \oplus f_2$

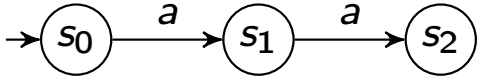
$\mathcal{F}$



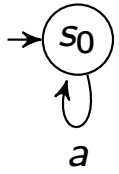
$\mathcal{F}'$



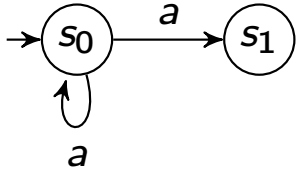
$\mathcal{F}|_{\lambda_1} = \mathcal{F}'|_{\lambda_1}$



$\mathcal{F}|_{\lambda_2}$



$\mathcal{F}'|_{\lambda_2}$

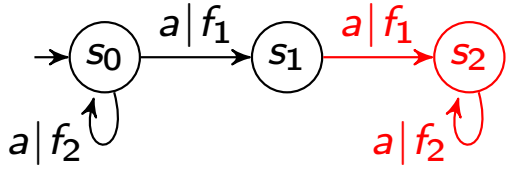


products $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$

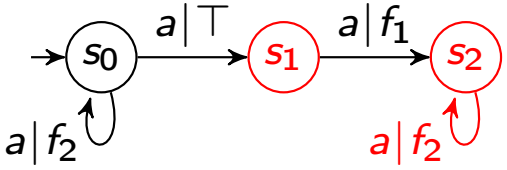
Example transformations

Feature Model: $f_1 \oplus f_2$

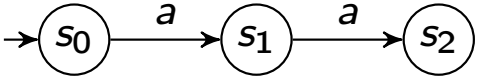
$\mathcal{F}$



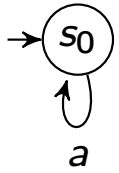
$\mathcal{F}'$



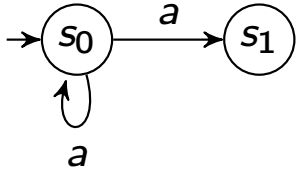
$\mathcal{F}|_{\lambda_1} = \mathcal{F}'|_{\lambda_1}$



$\mathcal{F}|_{\lambda_2}$

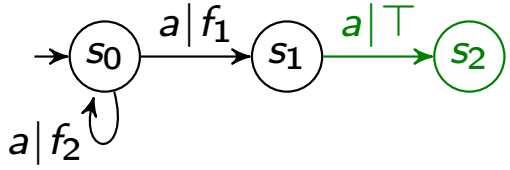


$\mathcal{F}'|_{\lambda_2}$

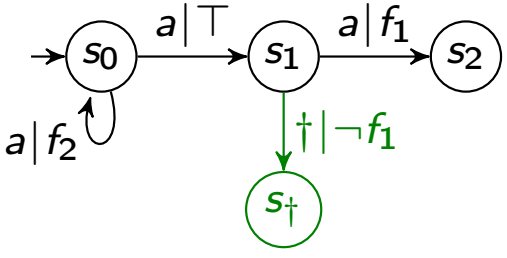


products $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$

$\mathcal{F}_*$



$\mathcal{F}'_*$



Detecting Ambiguities

- Based on expressing criteria for ambiguities as propositional formulae, thus reducing ambiguity detection to SAT solving

Detecting Ambiguities

- Based on expressing criteria for ambiguities as propositional formulae, thus reducing ambiguity detection to SAT solving

```
# fts contains the input FTS

for s in fts.states:
   if(s.out_trs = ∅):
      s.hdead ← False
   else:
      s.hdead ← check(exist_deadlock(s))

# for all states s, it holds that:
# (s.hdead ≡ "s is a hidden deadlock")
```

Detecting Ambiguities

- Based on expressing criteria for ambiguities as propositional formulae, thus reducing ambiguity detection to SAT solving

```
# fts contains the input FTS

for s in fts.states:
   if(s.out_trs = ∅):
      s.hdead ← False
   else:
      s.hdead ← check(exist_deadlock(s))

# for all states s, it holds that:
# (s.hdead ≡ "s is a hidden deadlock")
```

```
for s in fts.states:
   for t in s.in_trs:
      t.dead ← not check(is_not_dead_transition(t))
      if(t.dead or t.bx = ⊤):
         t.false_opt ← False
      else:
         t.false_opt ← not check(may_be_opt_transition(t))

# for all transitions t, it holds that: (t.dead ≡ "t is dead")
# and (t.false_opt ≡ "t is false optional")
```

Detecting Ambiguities

- Based on expressing criteria for ambiguities as propositional formulae, thus reducing ambiguity detection to SAT solving

```
# fts contains the input FTS

for s in fts.states:
   if(s.out_trs = ∅):
      s.hdead ← False
   else:
      s.hdead ← check(exist_deadlock(s))

# for all states s, it holds that:
# (s.hdead ≡ "s is a hidden deadlock")
```

```
for s in fts.states:
   for t in s.in_trs:
      t.dead ← not check(is_not_dead_transition(t))
      if(t.dead or t.bx = ⊤):
         t.false_opt ← False
      else:
         t.false_opt ← not check(may_be_opt_transition(t))

# for all transitions t, it holds that: (t.dead ≡ "t is dead")
# and (t.false_opt ≡ "t is false optional")
```

- We prove its correctness

Detecting Ambiguities

- Based on expressing criteria for ambiguities as propositional formulae, thus reducing ambiguity detection to SAT solving

```
# fts contains the input FTS

for s in fts.states:
   if(s.out_trs = ∅):
      s.hdead ← False
   else:
      s.hdead ← check(exist_deadlock(s))

# for all states s, it holds that:
# (s.hdead ≡ "s is a hidden deadlock")
```

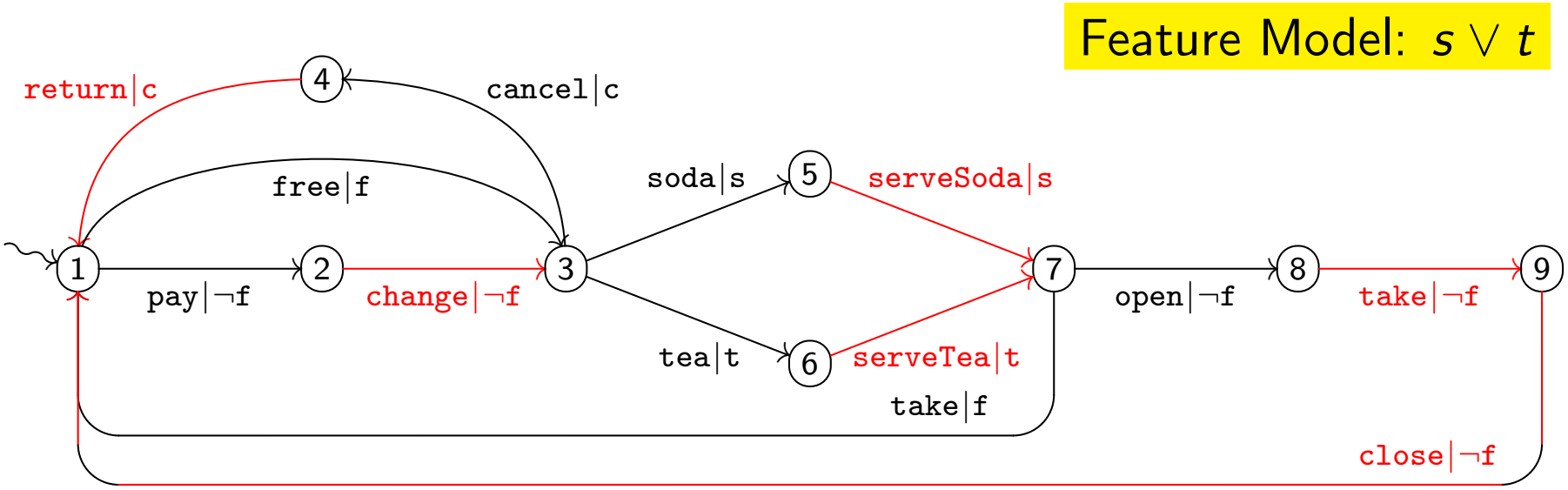
```
for s in fts.states:
   for t in s.in_trs:
      t.dead ← not check(is_not_dead_transition(t))
      if(t.dead or t.bx = ⊤):
         t.false_opt ← False
      else:
         t.false_opt ← not check(may_be_opt_transition(t))

# for all transitions t, it holds that: (t.dead ≡ "t is dead")
# and (t.false_opt ≡ "t is false optional")
```

- We prove its correctness

- Our implementation uses Z3

Python code publicly available, declared as reusable artifact @ SPLC'19

Example static analysis: vending machine

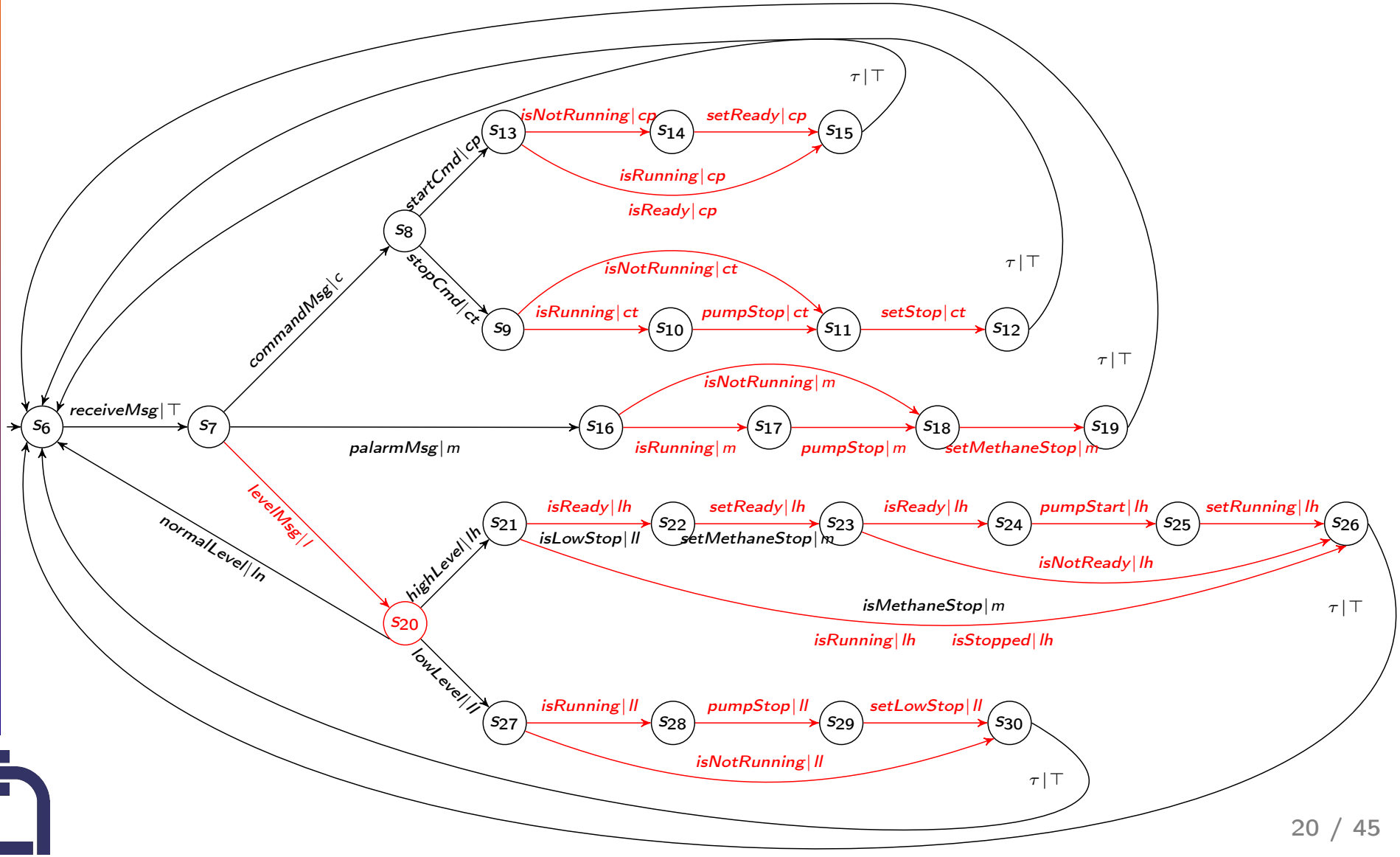


Result of static analysis on FTS

```
Vending Machine: live
 LIVE STATES = [1,2,3,4,5,6,7,8,9]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(2,3),(4,1),(5,7),(6,7),(8,9),(9,1)]
 HIDDEN DEADLOCK STATES = []
```

Example static analysis: mine pump (1/2)

Feature Model: $(c \leftrightarrow (ct \vee cp)) \wedge l$



Example static analysis: mine pump (2/2)

Result of static analysis on FTS

```
Mine Pump: not live
 LIVE STATES = [S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,
                S19,S21,S22,S23,S24,S25,S26,S27,S28,S29,S30]
 DEAD TRANSITIONS = []
 FALSE OPTIONAL TRANSITIONS = [(S10,S11),(S11,S12),(S13,S14),
      (S13,S15,isReady),(S13,S15,isRunning),(S14,S15),(S16,S17),
      (S16,S18),(S17,S18),(S18,S19),(S21,S22,isReady),
      (S21,S26,isRunning),(S21,S26,isStopped),(S22,S23,setReady),
      (S23,S24),(S23,S26),(S24,S25),(S25,S26),(S27,S28),(S27,S30),
      (S28,S29),(S29,S30),(S7,S20),(S9,S10),(S9,S11)]
 HIDDEN DEADLOCK STATES = [S20]
```

Benchmark Experiments (1/3)

FTS	characteristics			results of static analysis				computational effort	
Model	$\|S\|$	$\|\delta\|$	$\|\Sigma\|$	live-ness	# dead transitions	# false optional transitions	# hidden deadlock states	run-time (s)	memory usage (Mb)
Vending machine Classen, PhD thesis, 2011	9	13	12	yes	0	6	0	0.26	29.765
Coffee machine Asirelli *et al.* @ SPLC'11	14	23	15	yes	0	14	0	0.29	30.305
Soup component Belder *et al.* @ FMSPLE'15	13	28	18	yes	0	7	0	0.316	30.85
Mine pump (system) Classen, PhD thesis, 2011	25	41	22	no	0	25	1	0.344	31.704
Mine pump (controller) Classen, PhD thesis, 2011	77	104	22	no	0	59	4	0.548	36.295
Coffee/Soup machine Belder *et al.* @ FMSPLE'15	182	691	33	yes	8	284	0	37.766	119.427
Mine pump (complete) Classen, PhD thesis, 2011	418	1,255	26	yes	0	308	0	98.994	119.127
Claroline Devroey *et al.* @ VaMoS'14	107	11,236	106	yes	0	259	0	2413.8	2010.229

The experiments were performed on a virtual machine Gentoo 201905, CLI Version VirtualBox (VDI) 64 bit, with 2048 Mb of allocated memory on a Windows 10 Pro 64 bit with 16 Gb of RAM and CPU AMD Ryzen 7 1700X (8 core, 16 threads, 3.4 Ghz)

Benchmark Experiments (2/3)

FTS	characteristics			computational effort				results
				implementation [SPLC19]		implementation [EMSE21]		
Model	$\|S\|$	$\|\delta\|$	$\|\Sigma\|$	runtime (s)	memory usage (Mb)	runtime (s)	memory usage (Mb)	runtime speedup
Vending machine Classen, PhD thesis, 2011	9	13	12	0.92	38.230	0.26	29.765	3.54x
Coffee machine Asirelli *et al.* @ SPLC'11	14	23	15	2.822	40.140	0.29	30.305	9.72x
Soup component Belder *et al.* @ FMSPLE'15	13	28	18	2.544	40.870	0.316	30.85	8.05x
Mine pump (system) Classen, PhD thesis, 2011	25	41	22	2.192	41.899	0.344	31.704	6.37x
Mine pump (controller) Classen, PhD thesis, 2011	77	104	22	8.12	49.091	0.548	36.295	14.82x
Coffee/Soup machine Belder *et al.* @ FMSPLE'15	182	691	33	timeout	–	37.766	119.427	>7200.00x
Mine pump (complete) Classen, PhD thesis, 2011	418	1,255	26	timeout	–	98.994	119.127	>7200.00x
Claroline Devroey *et al.* @ VaMoS'14	107	11,236	106	timeout	–	2413.8	2010.229	>7200.00x

The experiments were performed on a virtual machine Gentoo 201905, CLI Version VirtualBox (VDI) 64 bit, with 2048 Mb of allocated memory on a Windows 10 Pro 64 bit with 16 Gb of RAM and CPU AMD Ryzen 7 1700X (8 core, 16 threads, 3.4 Ghz)

Benchmark Experiments (3/3)

FTS	characteristics			computational effort				results
				full-fledged implementation		specialised implementation		
Model	$\|S\|$	$\|\delta\|$	$\|\Sigma\|$	runtime (s)	memory usage (Mb)	runtime (s)	memory usage (Mb)	runtime fraction
Coffee/Soup machine Belder *et al.* @ FMSPLE'15	182	691	33	37.766	119.427	2.288	61.620	6.06%
Mine pump (complete) Classen, PhD thesis, 2011	417	1,255	26	98.994	119.127	2.948	68.969	2.97%
Claroline Devroey *et al.* @ VaMoS'14	107	11,236	106	2413.8	2010.229	86.752	551.888	3.59%

The experiments were performed on a virtual machine Gentoo 201905, CLI Version VirtualBox (VDI) 64 bit, with 2048 Mb of allocated memory on a Windows 10 Pro 64 bit with 16 Gb of RAM and CPU AMD Ryzen 7 1700X (8 core, 16 threads, 3.4 Ghz)

Intermezzo: MTSs for SPLE

Main ingredient: Modal Transition System (MTS)

- LTS distinguishing admissible (may) and necessary (must) transitions

  Larsen & Thomsen, A Modal Process Logic @ LICS'88

- Recognised as a useful formalism to describe in a compact way the possible behaviour of all the products (LTSs) of a product family

  Fischbein *et al.*, A foundation for behavioural conformance in SPL architectures @ ROSATEA'06

  Fantechi & Gnesi, A behavioural model for product families @ ESEC/FSE'07

Intermezzo: MTSs for SPLE

Main ingredient: Modal Transition System (MTS)

- LTS distinguishing admissible (may) and necessary (must) transitions

  Larsen & Thomsen, A Modal Process Logic @ LICS'88

- Recognised as a useful formalism to describe in a compact way the possible behaviour of all the products (LTSs) of a product family

  Fischbein *et al.*, A foundation for behavioural conformance in SPL architectures @ ROSATEA'06

  Fantechi & Gnesi, A behavioural model for product families @ ESEC/FSE'07

- ☞ MTSs cannot model variability constraints regarding alternative features, nor regarding requires / excludes inter-feature relations, resulting in several variants and extensions

  Larsen *et al.*, Modal I/O Automata for Interface and Product Line Theories @ ESOP'07

  Lauenroth *et al.*, Model Checking of Domain Artifacts in Product Line Engineering @ ASE'09

Intermezzo: MTSs for SPLE

Main ingredient: Modal Transition System (MTS)

- LTS distinguishing admissible (may) and necessary (must) transitions

  Larsen & Thomsen, A Modal Process Logic @ LICS'88

- Recognised as a useful formalism to describe in a compact way the possible behaviour of all the products (LTSs) of a product family

  Fischbein *et al.*, A foundation for behavioural conformance in SPL architectures @ ROSATEA'06

  Fantechi & Gnesi, A behavioural model for product families @ ESEC/FSE'07

- MTSs cannot model variability constraints regarding alternative features, nor regarding requires / excludes inter-feature relations, resulting in several variants and extensions

  Larsen *et al.*, Modal I/O Automata for Interface and Product Line Theories @ ESOP'07

  Lauenroth *et al.*, Model Checking of Domain Artifacts in Product Line Engineering @ ASE'09

- Our solution: add a set of variability constraints to the MTSs to be able to decide which derivable products (LTSs) are valid ones

  Asirelli *et al.*, Formal Description of Variability in Product Families @ SPLC'11

  ter Beek *et al.*, Modelling and analysing variability in product families. *JLAMP*, 2016

MTSυ: MTS with...

Recall:

A Labelled Transition System (LTS) is a quadruple $(Q, \Sigma, s_0, \rightarrow)$ with states S, actions A, initial state s_0, and transitions $\rightarrow \subseteq S \times \Sigma \times S$

Next we define MTSs with variability constraints:

MTSυ: MTS with...

Recall:

A Labelled Transition System (LTS) is a quadruple $(Q, \Sigma, s_0, \rightarrow)$ with states S, actions A, initial state s_0, and transitions $\rightarrow \subseteq S \times \Sigma \times S$

Next we define MTSs with variability constraints:

A Modal Transition System (MTS) is a quintuple $(S, \Sigma, s_0, \rightarrow_\Box, \rightarrow_\Diamond)$ such that $(S, \Sigma, s_0, \rightarrow_\Box \cup \rightarrow_\Diamond)$ is an LTS, called its underlying LTS

An MTS has two distinct transition relations

1. may transition relation $\rightarrow_\Diamond \subseteq S \times \Sigma \times S$: admissible transitions
2. must transition relation $\rightarrow_\Box \subseteq S \times \Sigma \times S$: necessary transitions

By definition, any necessary transition is also admissible: $\rightarrow_\Box \subseteq \rightarrow_\Diamond$
(denote $\dashrightarrow \equiv \rightarrow_\Diamond \setminus \rightarrow_\Box$: optional transitions)

. . . variability constraints (accepted by VMC)

Variability constraints of form ALTernative, EXCludes, REQuires, etc.

a_1 ALT $\cdots$ ALT a_n : precisely one among the $n \geq 2$ actions $a_1, \ldots, a_n$ is reachable in $\mathcal{L}$ (i.e. is the label of a reachable transition)

b_1 OR $\cdots$ OR b_n, where b_i is either a_i or $\neg\, a_i$: at least one among the conditions on $n \geq 2$ actions $b_1, \ldots, b_n$ holds, i.e. $b_i = a_i$ is reachable in $\mathcal{L}$ or $b_i = \neg\, a_i$ is not reachable in $\mathcal{L}$

a_1 EXC a_2 : at most one of the actions a_1 and a_2 is reachable in $\mathcal{L}$

a_1 REQ a_2 : action a_2 is reachable in $\mathcal{L}$ whenever a_1 is reachable in $\mathcal{L}$

a_1 REQ (a_2 ALT $\cdots$ ALT a_n) : precisely one among the $n \geq 2$ actions $a_2, \ldots, a_n$ is reachable in $\mathcal{L}$ if a_1 is reachable in $\mathcal{L}$

a_1 REQ (a_2 OR $\cdots$ OR a_n) : at least one among the $n \geq 2$ actions $a_2, \ldots, a_n$ is reachable in $\mathcal{L}$ if a_1 is reachable in $\mathcal{L}$

Derive products (implemented in VMC)

A product LTS is obtained from a family MTSυ in the following way:

1. include all (reachable) must transitions and
2. include subset of the (reachable) optional transitions, remove rest
3. satisfy assumptions of coherence and consistency
4. satisfy variability constraints

$\Rightarrow$ Each selection gives rise to a different variant

Derive products (implemented in VMC)

A product LTS is obtained from a family MTSυ in the following way:

1. include all (reachable) must transitions and
2. include subset of the (reachable) optional transitions, remove rest
3. satisfy assumptions of coherence and consistency
4. satisfy variability constraints

$\Rightarrow$ Each selection gives rise to a different variant

Let $(S, \Sigma, s_0, \delta^{\Diamond}, \delta^{\Box}, \Upsilon)$ be a coherent MTSυ, i.e. $\exists \dashrightarrow^{a} \implies \nexists \longrightarrow^{a}$

Derive products (implemented in VMC)

A product LTS is obtained from a family MTS υ in the following way:

1. include all (reachable) must transitions and
2. include subset of the (reachable) optional transitions, remove rest
3. satisfy assumptions of coherence and consistency
4. satisfy variability constraints

$\Rightarrow$ Each selection gives rise to a different variant

Let $(S, \Sigma, s_0, \delta^{\Diamond}, \delta^{\Box}, \Upsilon)$ be a coherent MTS υ, i.e. $\exists \xrightarrow{a} \implies \nexists \xrightarrow{a}$

The set $\{ \mathcal{P}_i = (S_i, \Sigma, s_0, \delta_i) \mid i > 0 \}$ of product LTSs is obtained by considering each pair of $S_i \subseteq S$ and $\delta_i \subseteq \delta^{\Diamond} \cup \delta^{\Box}$ to be defined s.t.

1. every $s \in S_i$ is reachable in $\mathcal{P}_i$ from s_0 via transitions from δ_i
2. there exists no $(s, a, s') \in \delta^{\Box} \setminus \delta_i$ such that $s \in S_i$
3. LTS is consistent: both $\xrightarrow{a} \rightsquigarrow \xrightarrow{a}$ and $\not\xrightarrow{a}$ not allowed
4. $\mathcal{P}_i$ satisfies all variability constraints in Υ

MTSυs are equally expressive as FTSs

Theorem (FTS2MTSυ transformation is sound and complete)

Let $\mathcal{F}$ be an FTS and let $\mathcal{M}$ be the MTSυ generated from $\mathcal{F}$ according to the FTS2MTSυ model transformation algorithm. Then the sets of derived variants $\mathsf{lts}(\mathcal{F})$ *and* $\mathsf{lts}(\mathcal{M})$ *coincide, up to dummy transitions and action relabelling.*

ter Beek *et al.*, From FTSs to MTSs with Variability Constraints @ SEFM'15

MTSυs are equally expressive as FTSs

Theorem (FTS2MTSυ transformation is sound and complete)

Let $\mathcal{F}$ be an FTS and let $\mathcal{M}$ be the MTSυ generated from $\mathcal{F}$ according to the FTS2MTSυ model transformation algorithm. Then the sets of derived variants $\mathsf{lts}(\mathcal{F})$ *and* $\mathsf{lts}(\mathcal{M})$ *coincide, up to dummy transitions and action relabelling.*

ter Beek *et al.*, From FTSs to MTSs with Variability Constraints @ SEFM'15

Theorem (MTSυ2FTS transformation is sound and complete)

Let $\mathcal{M}$ be an MTSυ and let $\mathcal{F}$ be the FTS generated from $\mathcal{M}$ according to the MTSυ2FTS model transformation algorithm. Then $\mathsf{lts}(\mathcal{M}) = \mathsf{lts}(\mathcal{F})$.

ter Beek *et al.*, On the Expressiveness of MTSs with Variability Constraints. *SCP*, 2019

Dedicated Variability Model Checker VMC

VMC profits from optimisations of KandISTI family of model checkers: bounded, on-the-fly model checking with linear complexity, providing clear and easily understandable explanations when the validation fails

ter Beek *et al.*, A state/event-based model-checking approach for the analysis of abstract system properties @ *SCP*, 2011

Dedicated Variability Model Checker VMC

VMC profits from optimisations of KandISTI family of model checkers: bounded, on-the-fly model checking with linear complexity, providing clear and easily understandable explanations when the validation fails

ter Beek *et al.*, A state/event-based model-checking approach for the analysis of abstract system properties @ *SCP*, 2011

VMC accepts as input a specification in (value-passing) modal process algebra, possibly with additional variability constraints:

- interactively explore the model (MTSυ)
- derive and explore (all) the model's valid products (LTSs)
- visualise the model/products graphically as MTSυ/LTSs
- verify branching-time v-ACTL properties over MTSυ/LTSs
- interactively explain why a property is (not) satisfied

Variability analysis strategies supported by VMC

VMC offers both product-based and family-based variability analyses:

Variability analysis strategies supported by VMC

VMC offers both product-based and family-based variability analyses:

1. The actual set of all valid product behaviour can explicitly be generated and the resulting LTSs can all be verified against one and the same logic property (expressed in Action-based CTL)

   De Nicola & Vaandrager, Three Logics for Branching Bisimulation. *J. ACM*, 1995

Variability analysis strategies supported by VMC

VMC offers both product-based and family-based variability analyses:

1. The actual set of all valid product behaviour can explicitly be generated and the resulting LTSs can all be verified against one and the same logic property (expressed in Action-based CTL)

   De Nicola & Vaandrager, Three Logics for Branching Bisimulation. *J. ACM*, 1995

2. A logic property (expressed in variability-aware ACTL) can directly be verified against the MTS υ, relying on the fact that under certain syntactic conditions validity over the MTS υ implies validity of the same property for all its derived products (more later)

   ter Beek *et al.*, Modelling and analysing variability in product families. *JLAMP*, 2016

Variability analysis strategies supported by VMC

VMC offers both product-based and family-based variability analyses:

1. The actual set of all valid product behaviour can explicitly be generated and the resulting LTSs can all be verified against one and the same logic property (expressed in Action-based CTL)

   De Nicola & Vaandrager, Three Logics for Branching Bisimulation. *J. ACM*, 1995

2. A logic property (expressed in variability-aware ACTL) can directly be verified against the MTS υ, relying on the fact that under certain syntactic conditions validity over the MTS υ implies validity of the same property for all its derived products (more later)

   ter Beek *et al.*, Modelling and analysing variability in product families. *JLAMP*, 2016

VMC is freely usable online:

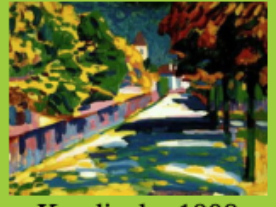
http://fmt.isti.cnr.it/vmc/

Vending Machine: family-based analysis

VMC notifies whenever preservation of an analysis result is applicable

VMC V6.5
(2019)

Edit Model
View Current Model
Explore the MTS
Draw Family MTS
Generate Products
Welcome
Quit


Kandinsky 1908

The Formula: *not E[true {not tea} U {serveTea}true]*
is TRUE

The formula holds for ALL the MTS variants

(evaluation time= 0.067 sec.)

(total states generated= 9, computations fragments generated= 17, total evaluation time= 0.067 sec.)

ACTL-UCTL-SocL-vACTL

```
not E [true {not tea} U {serveTea} true]
```

Check The Formula | Explain the Result

It is not possible that serveTea occurs without being preceded by tea

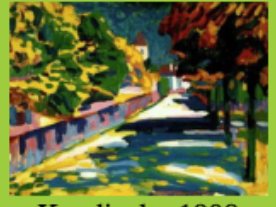
Vending Machine: product-based analysis

fmt.isti.cnr.it

VMC lists for each product the action labels of all may transitions that have been preserved (as must transitions) in that product

VMC V6.5
(2019)

New Model ...
Edit Current Model
Explore the MTS
View Current Model
Draw Family MTS
Generate Products
Welcome
Quit



Kandinsky 1908

Evaluation of the formula "[pay] AF {takePaid} true" on all family products

product+CancelPurchase+FreeDrinks+Soda+Tea_12	Formula evaluates	TRUE
product+CancelPurchase+FreeDrinks+Soda_08	Formula evaluates	TRUE
product+CancelPurchase+FreeDrinks+Tea_10	Formula evaluates	TRUE
product+CancelPurchase+Soda+Tea_06	Formula evaluates	FALSE
product+CancelPurchase+Soda_02	Formula evaluates	FALSE
product+CancelPurchase+Tea_04	Formula evaluates	FALSE
product+FreeDrinks+Soda+Tea_11	Formula evaluates	TRUE
product+FreeDrinks+Soda_07	Formula evaluates	TRUE
product+FreeDrinks+Tea_09	Formula evaluates	TRUE
product+Soda+Tea_05	Formula evaluates	TRUE
product+Soda_01	Formula evaluates	TRUE
product+Tea_03	Formula evaluates	TRUE

Logic Formula for all Products

[pay] AF {takePaid} true

Check The Formula | Explain the Result

Whenever pay occurs, eventually take(Paid) occurs

Family-based model checking (1/2)

Note

1. Any FTS $\mathcal{F}$ can trivially be transformed into an MTS $\mathcal{F}_{\text{MTS}}$ (must $\rightarrow$ necessary transitions, featured $\rightarrow$ optional transitions, all transitions admissible, remove all feature expressions)

2. If the FTS is unambiguous, then the corresponding MTS is live (as it has no hidden deadlocks, and all must transitions are necessary)

Family-based model checking (1/2)

Note

1. Any FTS $\mathcal{F}$ can trivially be transformed into an MTS $\mathcal{F}_{\text{MTS}}$
   (must $\rightarrow$ necessary transitions, featured $\rightarrow$ optional transitions,
   all transitions admissible, remove all feature expressions)

2. If the FTS is unambiguous, then the corresponding MTS is live
   (as it has no hidden deadlocks, and all must transitions are necessary)

This allows us to carry over a result for MTSυs to unambiguous FTSs:

ter Beek *et al.*, Modelling and analysing variability in product families. *JLAMP*, 2016

Novel feature of VMC v6.5: `Constraints { LIVE }` explicitly declares an MTSυ to be live

Family-based model checking (1/2)

Note

1. Any FTS $\mathcal{F}$ can trivially be transformed into an MTS $\mathcal{F}_{\text{MTS}}$ (must $\rightarrow$ necessary transitions, featured $\rightarrow$ optional transitions, all transitions admissible, remove all feature expressions)

2. If the FTS is unambiguous, then the corresponding MTS is live (as it has no hidden deadlocks, and all must transitions are necessary)

This allows us to carry over a result for MTSυs to unambiguous FTSs:

ter Beek *et al.*, Modelling and analysing variability in product families. *JLAMP*, 2016

Any formula ϕ of v-ACTLive$^{\Box}$ is preserved by live FTSs: given a live FTS $\mathcal{F}$, whenever $\mathcal{F}_{MTS} \models \phi$, then $\mathcal{F}|_{\lambda} \models \phi$ for all products $\mathcal{F}|_{\lambda}$ of $\mathcal{F}$

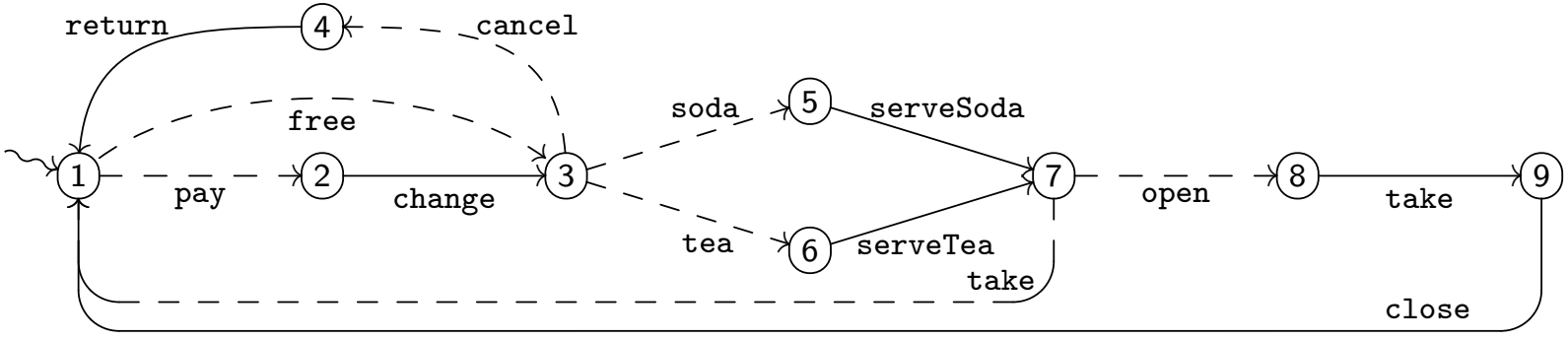
Novel feature of VMC v6.5: `Constraints { LIVE }` explicitly declares an MTS υ to be live

Family-based model checking with VMC

Example v-ACTLive$^{\Box}$ formulae that could now be verified with VMC:
(with a linear complexity)

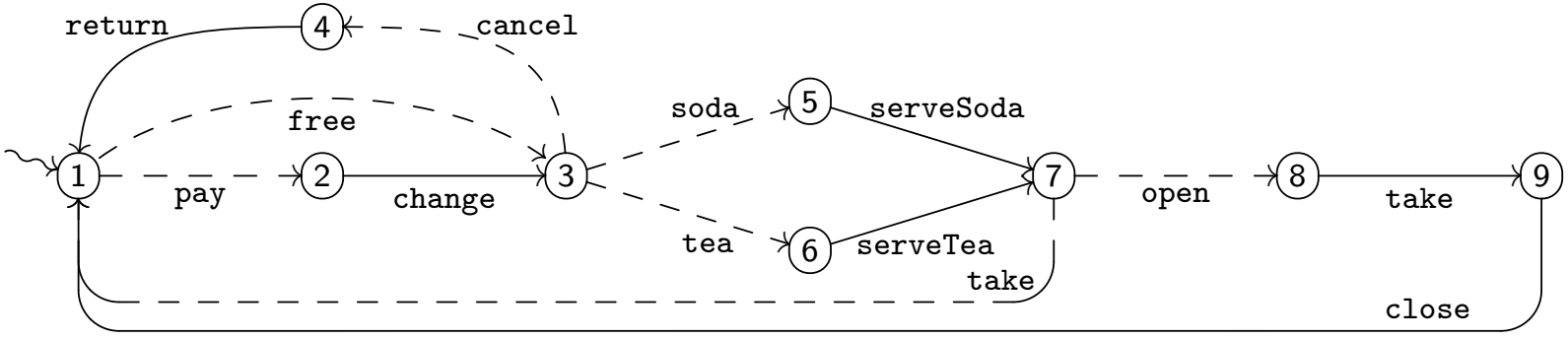
Family-based model checking with VMC

Example v-ACTLive$^{\square}$ formulae that could now be verified with VMC:
(with a linear complexity)



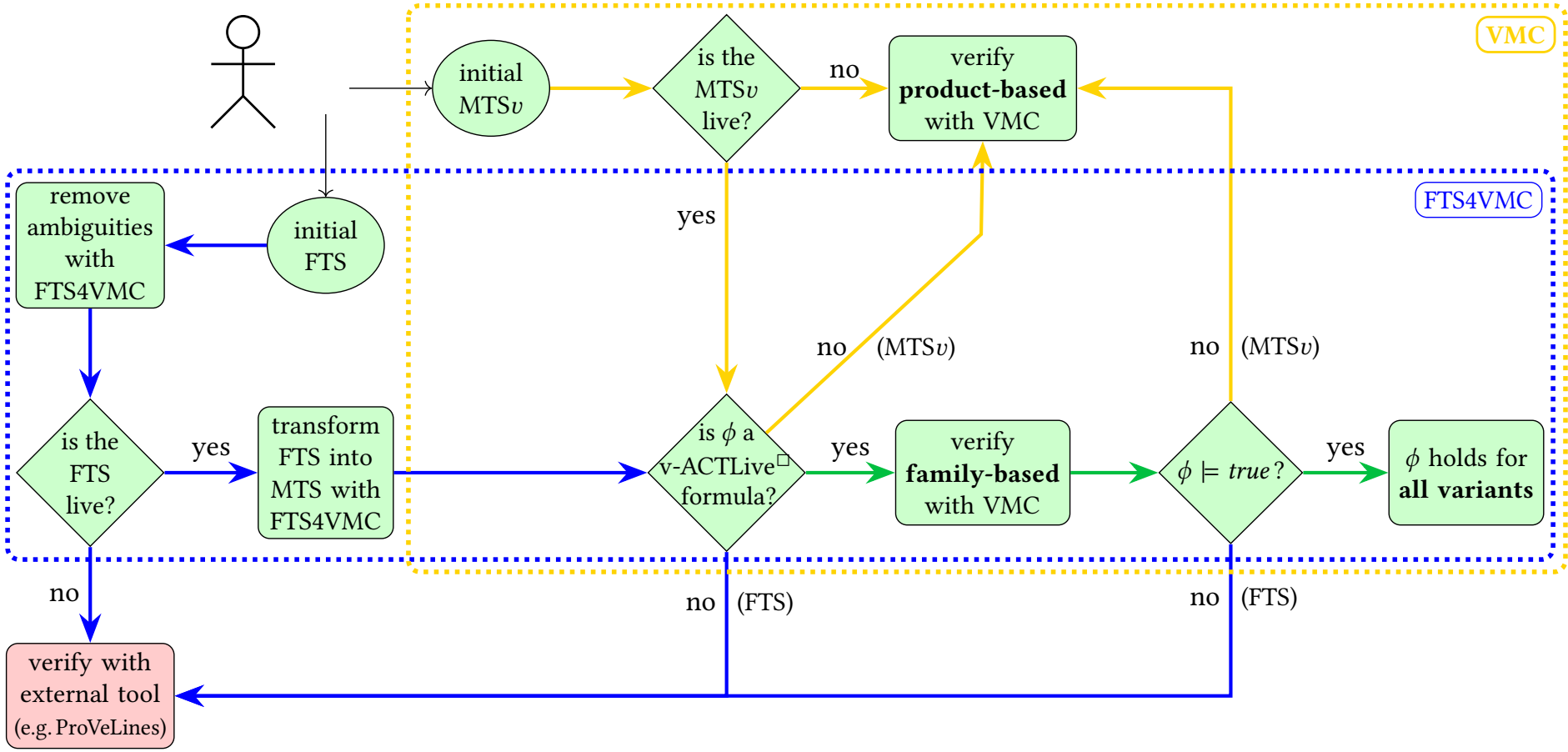
Family-based model checking with VMC

Example v-ACTLive$^{\square}$ formulae that could now be verified with VMC:
(with a linear complexity)



1. $AG\ AF_{pay \vee free}\ \top$: *infinitely often, either action pay or action free is executed*
2. $AG\ [open]\ AF_{close}\ \top$: *it is always the case that the execution of action open is eventually followed by that of action close*
3. $AG\ AF_{cancel \vee serveSoda \vee serveTea}\ \top$: *infinitely often, either action cancel or action serveSoda or action serveTea is executed*
4. $\neg E\ [\top\ {}_{\neg tea}U_{serveTea}\ \top]$: *it is not possible that action serveTea is executed without being preceded by an execution of action tea*
5. $[pay]\ AF_{take \vee cancel}\ \top$: *whenever action pay is executed, eventually also either action take or action cancel is executed*

FTS4VMC: front-end tool for VMC

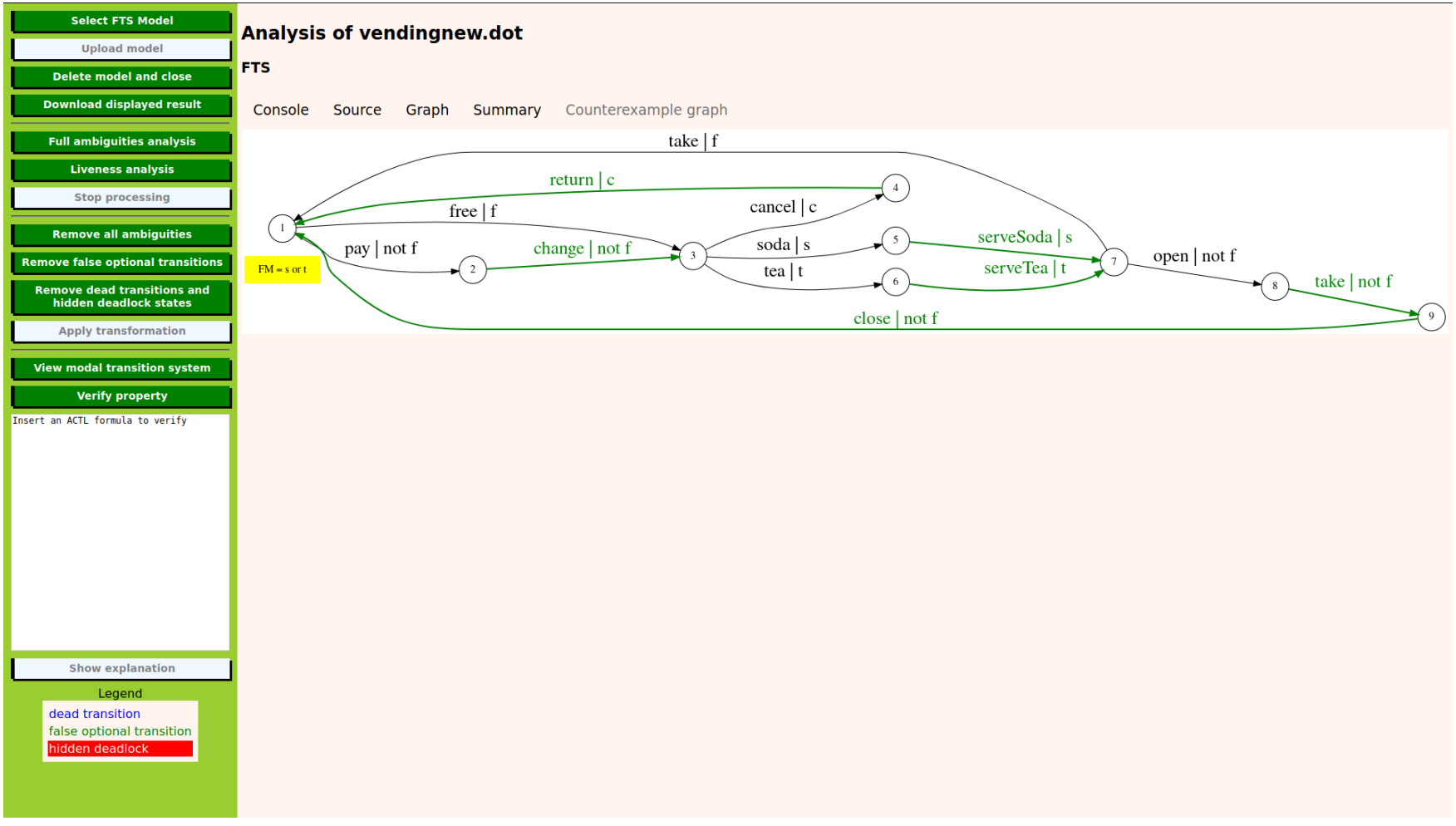


The green blocks are automated by the toolchain (FTS4VMC+VMC)

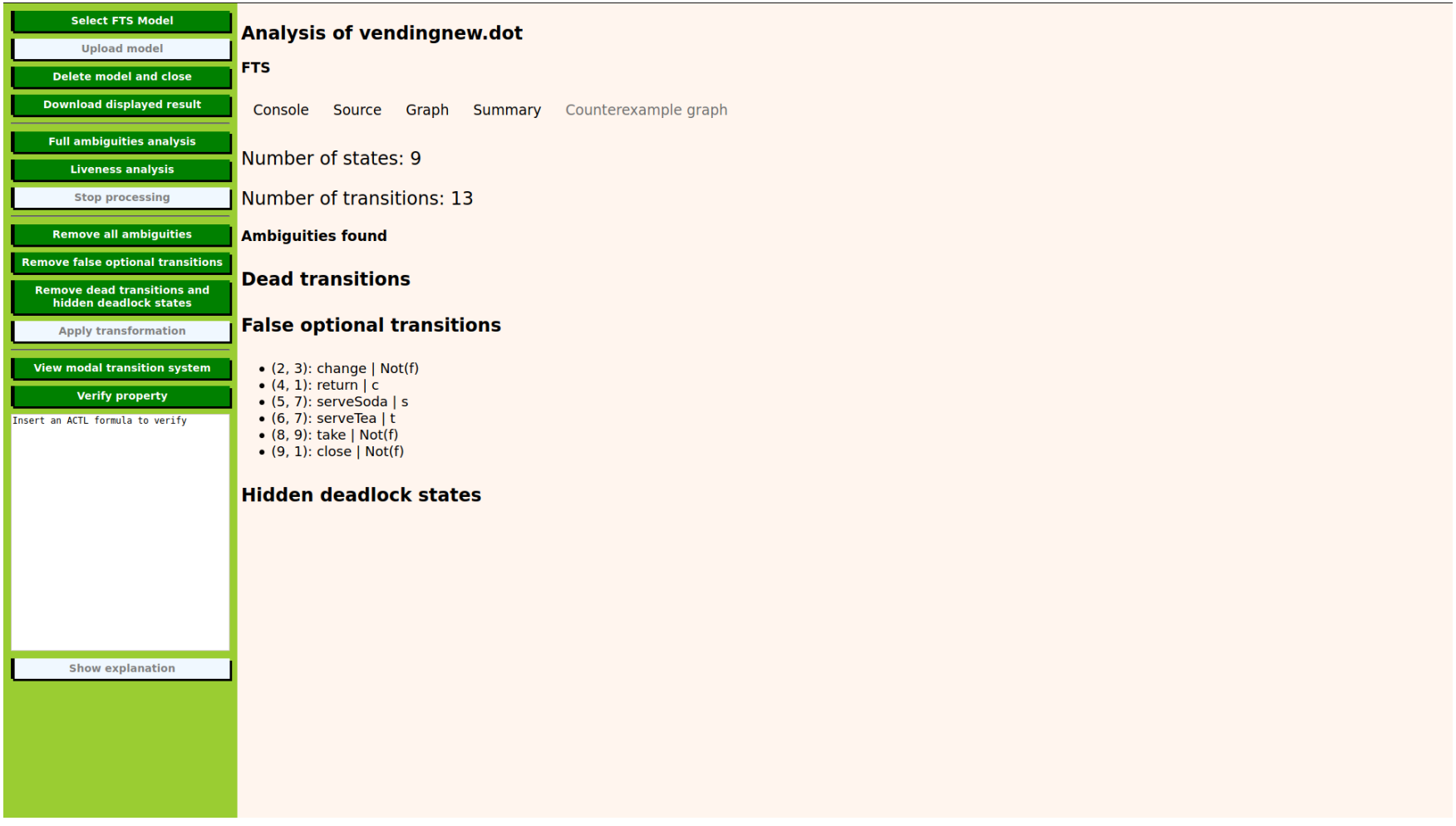
The blue and green steps (applied to FTSs) are realised by FTS4VMC

The FTS4VMC implementation is freely available online: https://github.com/fts4vmc/FTS4VMC

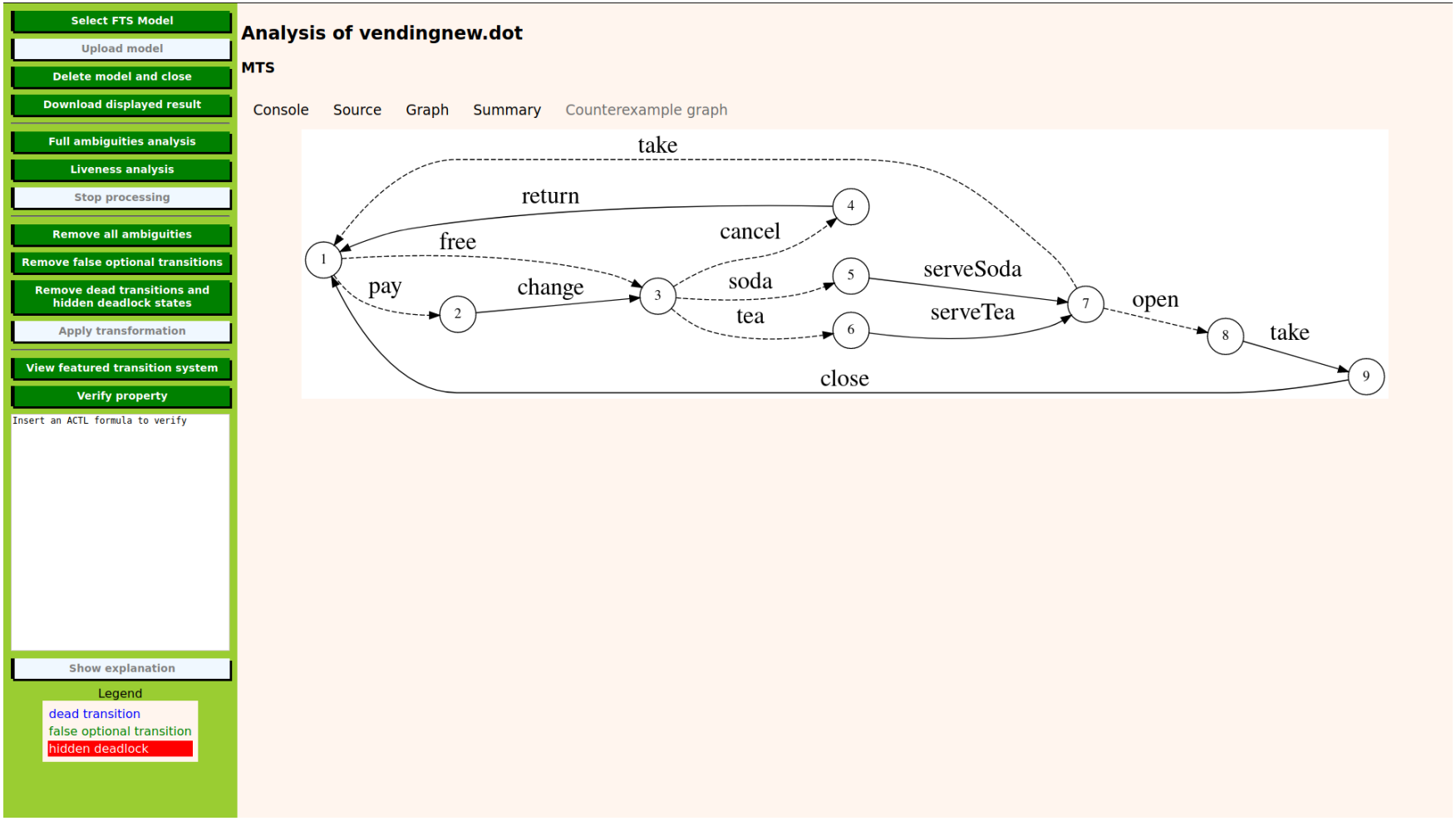
FTS4VMC at work (1/4)



FTS4VMC at work (2/4)

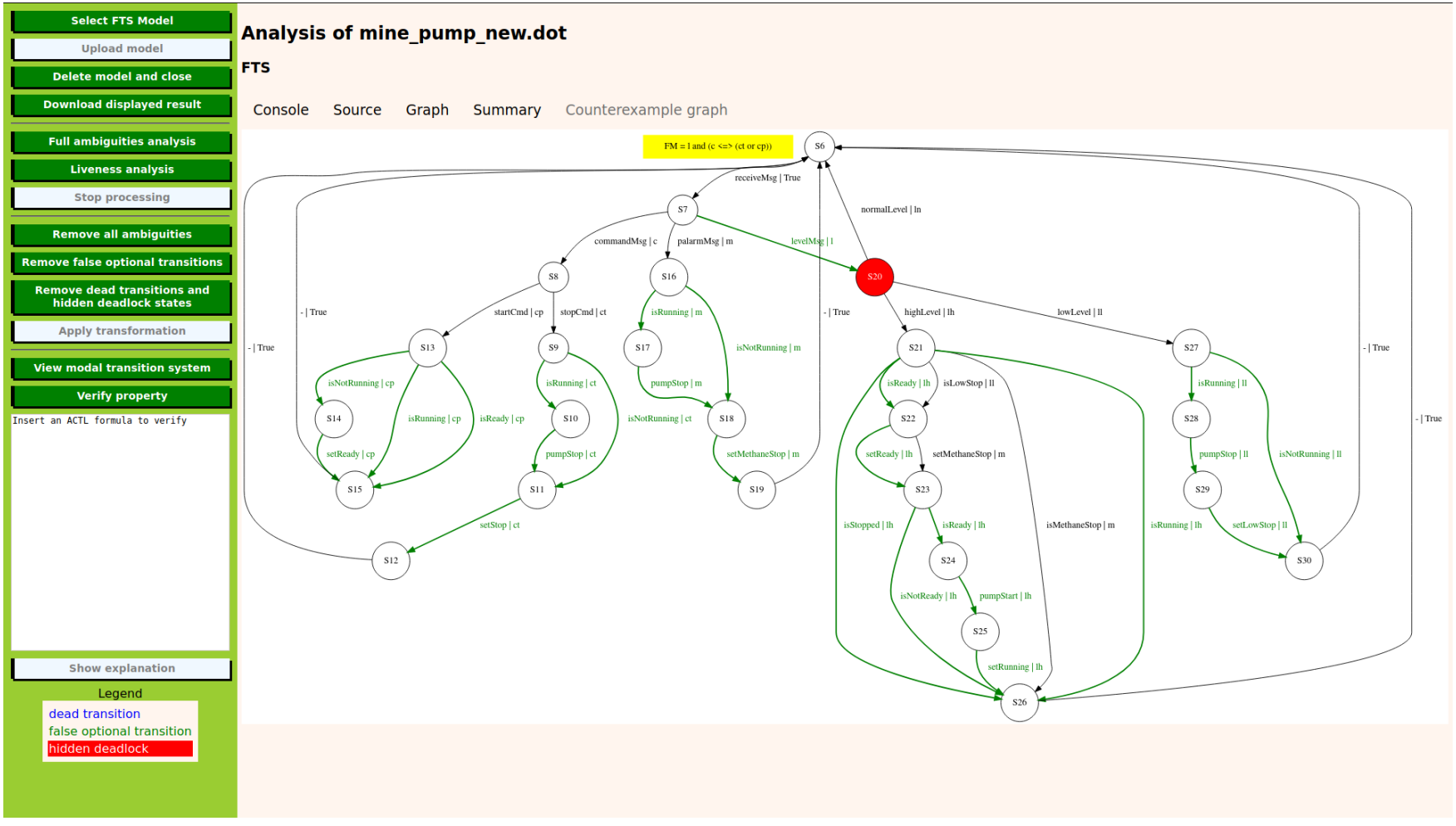


FTS4VMC at work (3/4)



FTS4VMC at work (4/4)

fmt.isti.cnr.it



Family-based model checking (2/2)

A path in an LTS is *maximal* if it cannot be extended further

Model checking LTL formulae on an LTS reduces to analysing its maximal paths: an LTL formula is valid if it holds for all maximal paths

Family-based model checking (2/2)

A path in an LTS is *maximal* if it cannot be extended further

Model checking LTL formulae on an LTS reduces to analysing its maximal paths: an LTL formula is valid if it holds for all maximal paths

Note

1. Trivially carries over to FTSs by ignoring the feature expressions

2. If the FTS is live, then the set of maximal paths of any product is a subset of the set of maximal paths of the FTS

Family-based model checking (2/2)

A path in an LTS is *maximal* if it cannot be extended further

Model checking LTL formulae on an LTS reduces to analysing its maximal paths: an LTL formula is valid if it holds for all maximal paths

Note

1. Trivially carries over to FTSs by ignoring the feature expressions

2. If the FTS is live, then the set of maximal paths of any product is a subset of the set of maximal paths of the FTS

Thus:

Any formula ϕ of LTL is preserved by live FTSs: given a live FTS $\mathcal{F}$, whenever $\mathcal{F}_{LTS} \models \phi$, then $\mathcal{F}|_\lambda \models \phi$ for all products $\mathcal{F}|_\lambda$ of $\mathcal{F}$

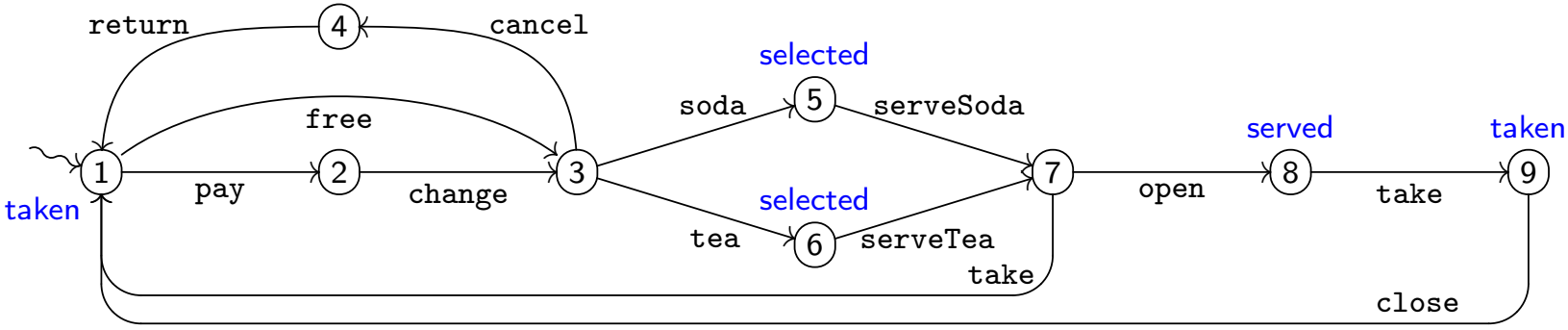
Family-based model checking with SPIN

Example LTL formulae that could now be verified with SPIN (for instance):

`http://spinroot.com/`

Example LTL formulae that could now be verified with SPIN (for instance):

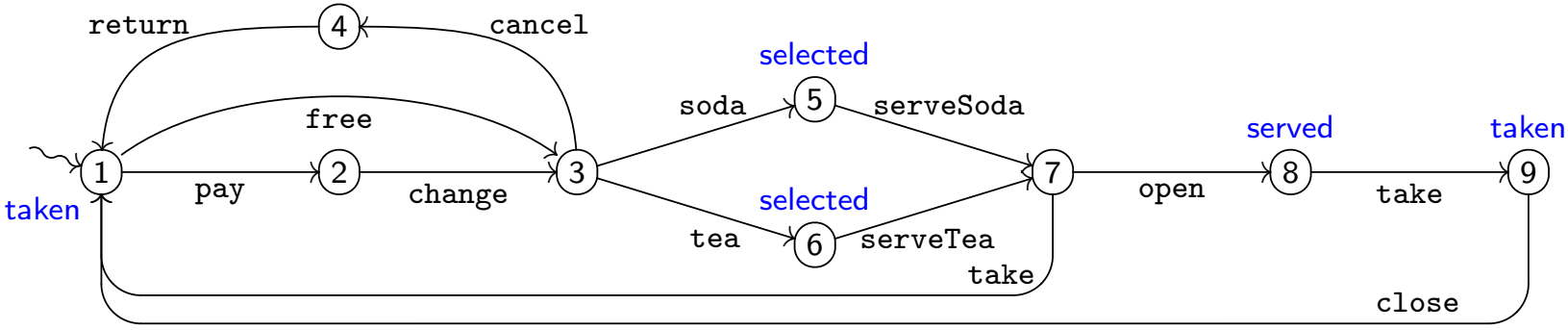
http://spinroot.com/



Family-based model checking with SPIN

Example LTL formulae that could now be verified with SPIN (for instance):

http://spinroot.com/

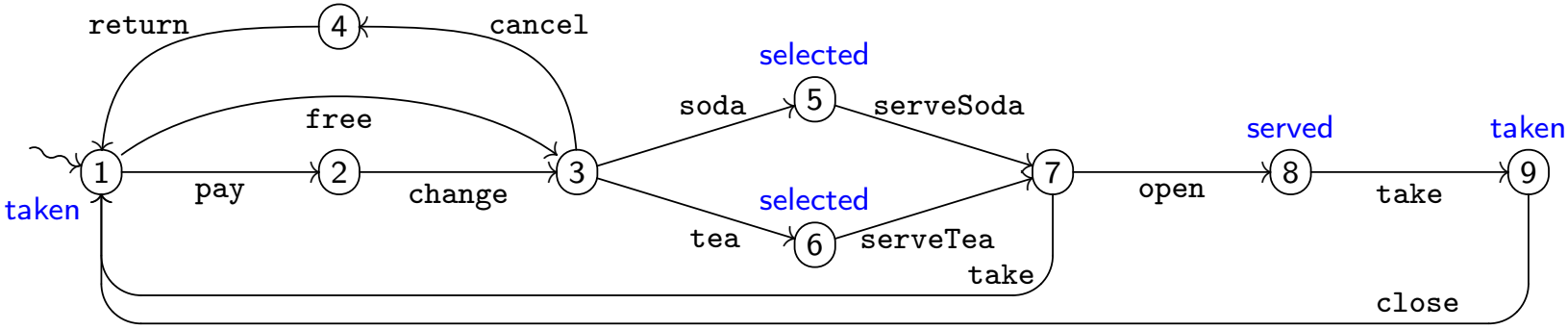


1. $\Box(selected \Rightarrow \Diamond served)$: *after a beverage is selected, the vending machine will always eventually serve a beverage*
2. $\Box(served \Rightarrow \Diamond taken)$: *after a beverage is served, a customer will always eventually take the beverage*

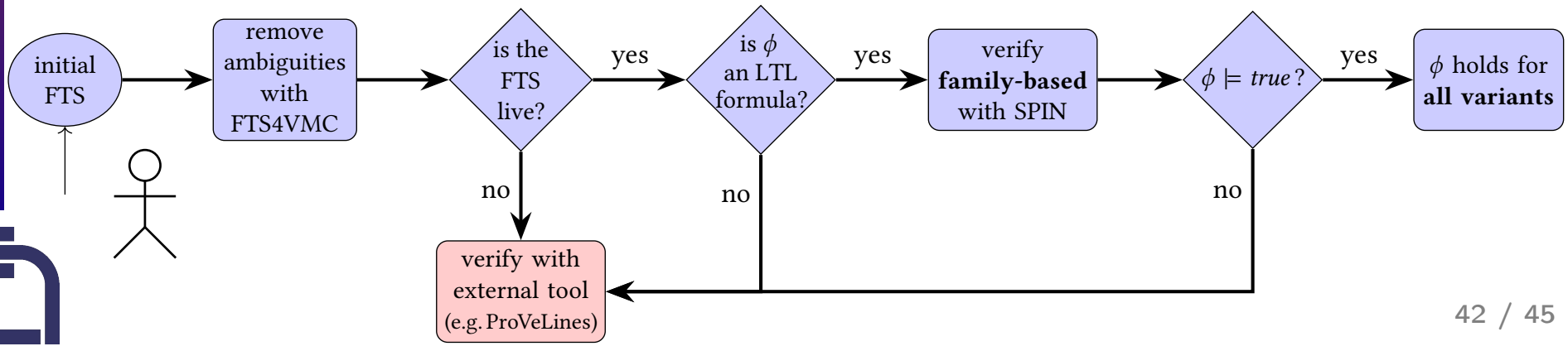
Family-based model checking with SPIN

Example LTL formulae that could now be verified with SPIN (for instance):

http://spinroot.com/



1. $\Box(selected \Rightarrow \Diamond served)$: *after a beverage is selected, the vending machine will always eventually serve a beverage*
2. $\Box(served \Rightarrow \Diamond taken)$: *after a beverage is served, a customer will always eventually take the beverage*



Conclusion and Outlook

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness
   - benchmark experiments

Conclusion and Outlook

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness
   - benchmark experiments

2. Efficient verification of FTSs:
   - a kind of family-based model checking
   - both linear- and branching-time properties

Conclusion and Outlook

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness
   - benchmark experiments
2. Efficient verification of FTSs:
   - a kind of family-based model checking
   - both linear- and branching-time properties
3. Automated by a toolchain:
   - front-end tool FTS4VMC

Conclusion and Outlook

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness
   - benchmark experiments
2. Efficient verification of FTSs:
   - a kind of family-based model checking
   - both linear- and branching-time properties
3. Automated by a toolchain:
   - front-end tool FTS4VMC
4. Future work:
   - integrate FTS2PROMELA transformation

References

SPLC19 M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, and L. Paolini, Static Analysis of Featured Transition Systems. In Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19), Paris, France, ACM Press, New York, 2019, 39–51 (best paper)

EMSE21 M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, and L. Paolini, Efficient Static Analysis and Verification of Featured Transition Systems. *Empirical Software Engineering* (2021) (to appear)

SPLC21 M.H. ter Beek, F. Damiani, F. Mazzanti, G. Scarso, and M. Valfrè, Static Analysis and Family-based Model Checking of Featured Transition Systems with VMC. Zenodo (February 2021) (to be submitted) `http://doi.org/10.5281/zenodo.4497888`

Thanks for your attention!

Hope to see you all at SPLC 2021 'here' in Leicester:

25th Systems and Software Product Line Conference

Submission deadline research papers: April 23, 2021

PC chairs:

- Maurice H. ter Beek, ISTI–CNR, Pisa, Italy
- Ina Schaefer, TU Braunschweig, Germany

Many more tracks (industry, challenges, demos & tools, journal first, doctoral symposium, hall of fame) plus workshops and tutorials

Organised by Mohammad Mousavi and his team

https://splc2021.net/