Efficient Static Analysis and Family-based Model Checking of SPL Models

Maurice ter Beek

Formal Methods and Tools Lab

Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" Consiglio Nazionale delle Ricerche

joint work with Ferruccio Damiani, Luca Paolini, Giordano Scarso (University of Turin, IT), Michael Lienhardt (ONERA, FR) and Franco Mazzanti (FMT, ISTI–CNR, Pisa, IT)

SES'23, Tokyo, Japan, August 25^{th}, 2023

Who am I?

- Member of FMT lab at ISTI–CNR since '03, lab head since '19
- MSc ('96) and PhD ('03) degrees from Leiden University (NL)
- Positions in HU ('95-'96,'02), BE ('05), IT ('00-'01), NL ('12-'13,'15)

Who am I? What do I work on?

Formal Methods and Tools Lab

- Member of FMT lab at ISTI–CNR since '03, lab head since '19
- MSc ('96) and PhD ('03) degrees from Leiden University (NL)
- Positions in HU ('95-'96,'02), BE ('05), IT ('00-'01), NL ('12-'13,'15)
- Formal methods, model checking, SPLs, SOC, team automata, railway systems, ...

Why am I here?

General co-chair, with Paolo Arcaini, of SPLC 2023 (NII, next week)

Local chair: Fuyuki Ishikawa (NII)

https://2023.splc.net/

Why am I here?

General co-chair, with Paolo Arcaini, of SPLC 2023 (NII, next week)

Local chair: Fuyuki Ishikawa (NII)

Many tracks: research, industry, challenges & solutions, journal first, demonstrations & tools, doctoral symposium, hall of fame plus workshops, tutorials, MIP award, panel on Intelligent SPLs and keynotes

---

https://2023.splc.net/

Why am I here?

General co-chair, with Paolo Arcaini, of SPLC 2023 (NII, next week)

Local chair: Fuyuki Ishikawa (NII)

Many tracks: research, industry, challenges & solutions, journal first, demonstrations & tools, doctoral symposium, hall of fame plus workshops, tutorials, MIP award, panel on Intelligent SPLs and keynotes

David Benavides (U Seville, SP) Data-intensive PLs: Embracing past results and new variability challenges

Hideto Ogawa & Kentaro Yoshimura (Hitachi Ltd, JP) The 20-year journey of SPLE in Hitachi and the next

Marianne Huchard (U Montpellier, FR) Chronicles of concept lattices: Unveiling structure in the software world

---

`https://2023.splc.net/`

Outline

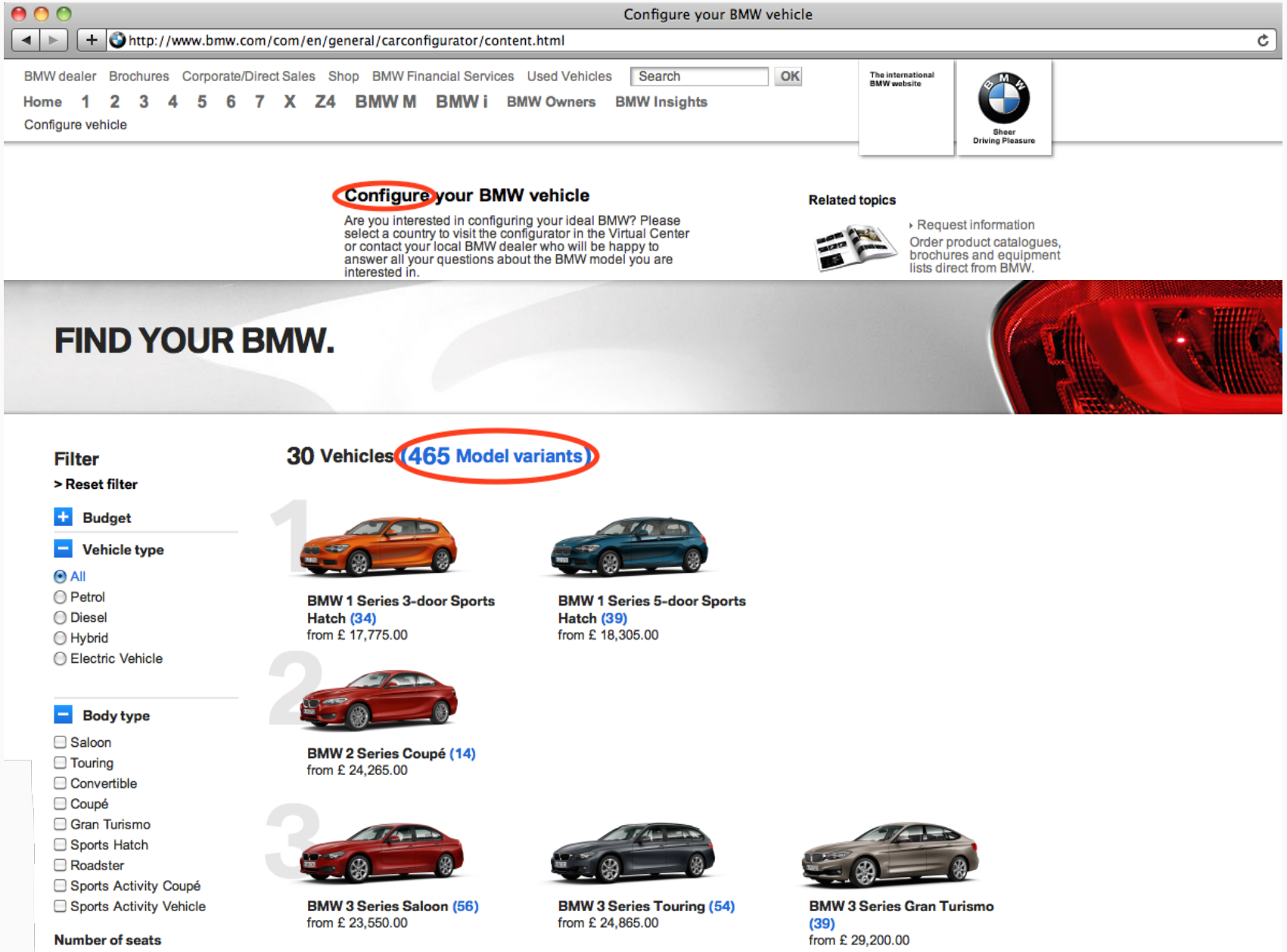
- Behavioural variability modelling and analysis of SPL models
  - Product- vs. family-based analysis
  - Featured Transition System (FTS)
  - Ambiguities: dead/false optional transitions, hidden deadlocks
- Automated static analysis of FTSs
  - Detecting ambiguities: criteria, SAT solving, implementation
  - Disambiguating FTSs: example and benchmark experiments
- Modal Transition System with variability constraints (MTSυ)
- VMC: family-based model checking of MTSυs
- FTS4VMC: family-based model checking of MTSυs *and* FTSs
- Conclusions and Future Work

Background

Software Product Lines (SPLs)

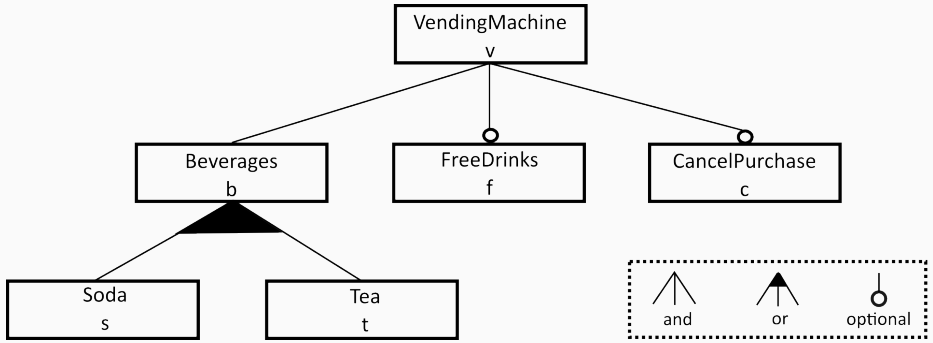
- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user

Product lines are everywhere



Feature models

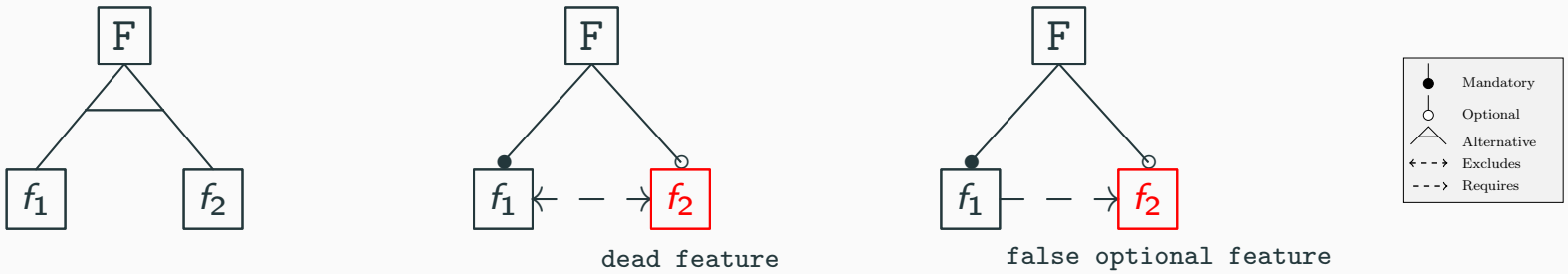
- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user



Feature Model (diagram) of a beverage vending machine

Feature models analysis

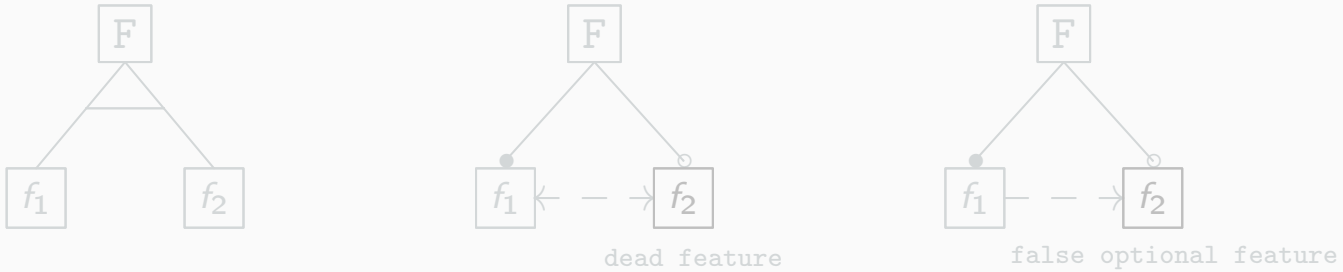
- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user



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

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 / critical systems domain: formal modelling and analysis techniques to prove SPL behaviour correct are widely studied

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

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

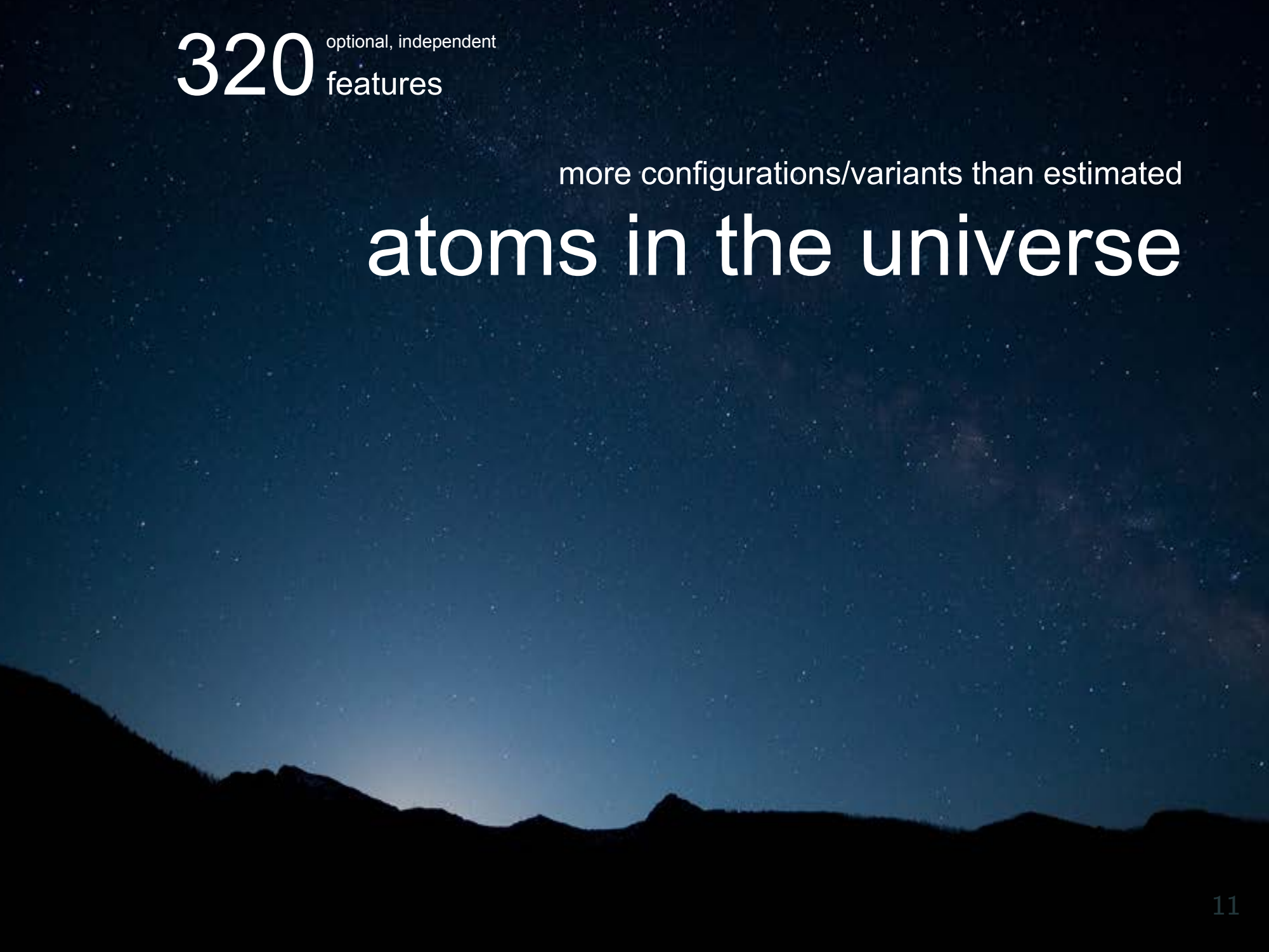
Scalability is an issue

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

33 optional, independent features

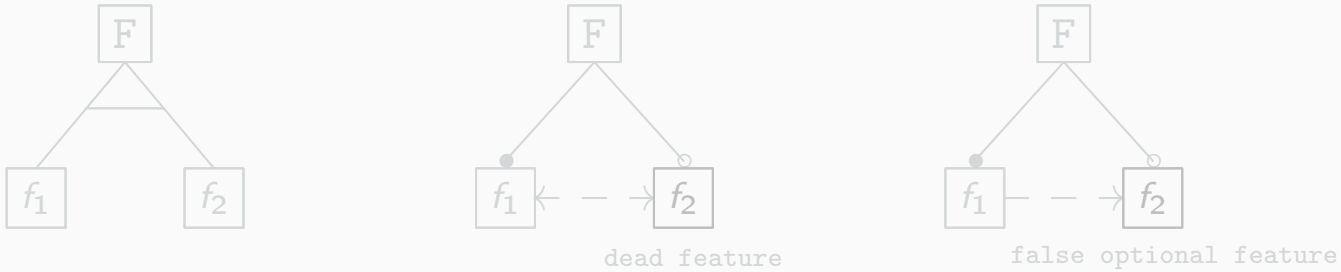
a unique configuration/variant for every

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 / critical systems domain: formal modelling and analysis techniques to prove SPL behaviour correct are widely studied

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

- Challenge known formal methods & 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- vs. family-based analysis

Product-based analysis:

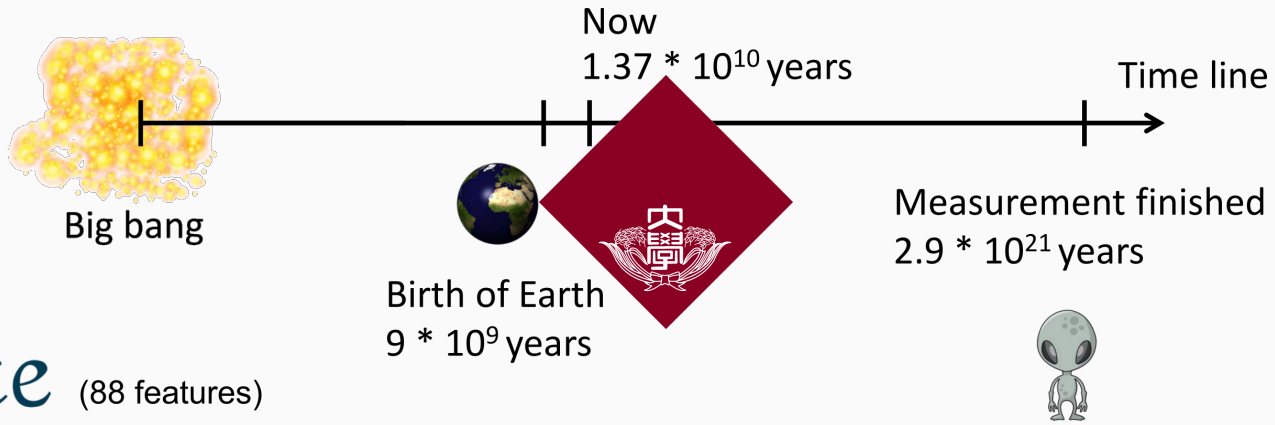
- 👍 Simple, brute-force approach
- 👍 Make use of standardly available, highly optimised analysis tools

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. FTS, MTSυ, Feature Net, 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. FTS, MTSυ, Feature Net, 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. FTS, MTSυ, Feature Net, 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

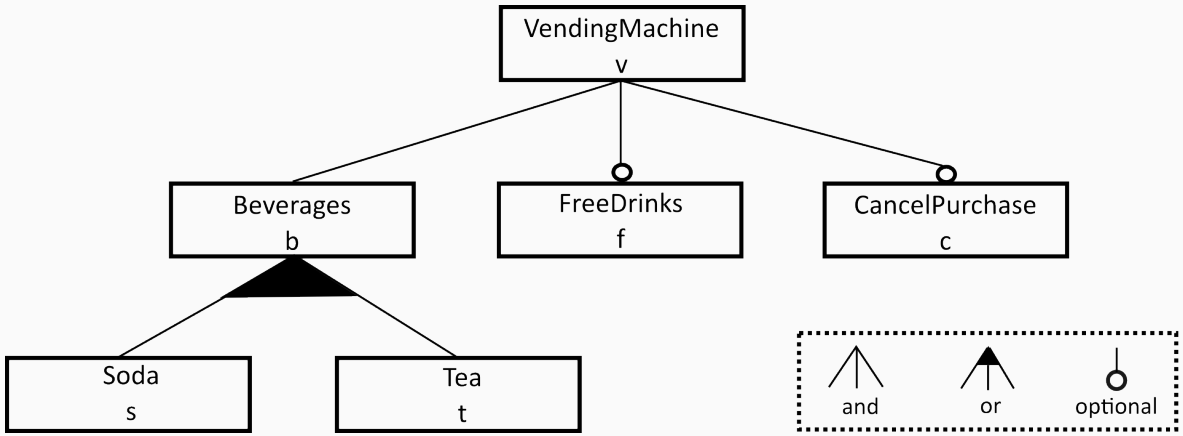
ter Beek *et al.*, Family-Based SPL Model Checking Using Parity Games with Variability @ FASE'20

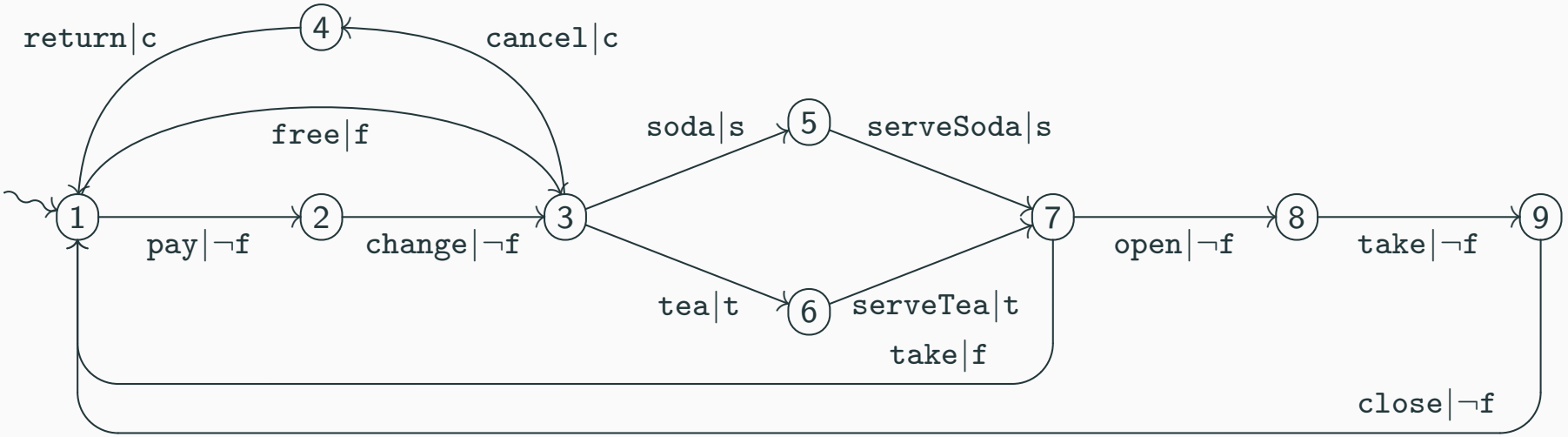
Featured Transition Systems (FTSs)

From LTS to FTS

An FTS adds to an LTS a feature model and so-called feature expressions

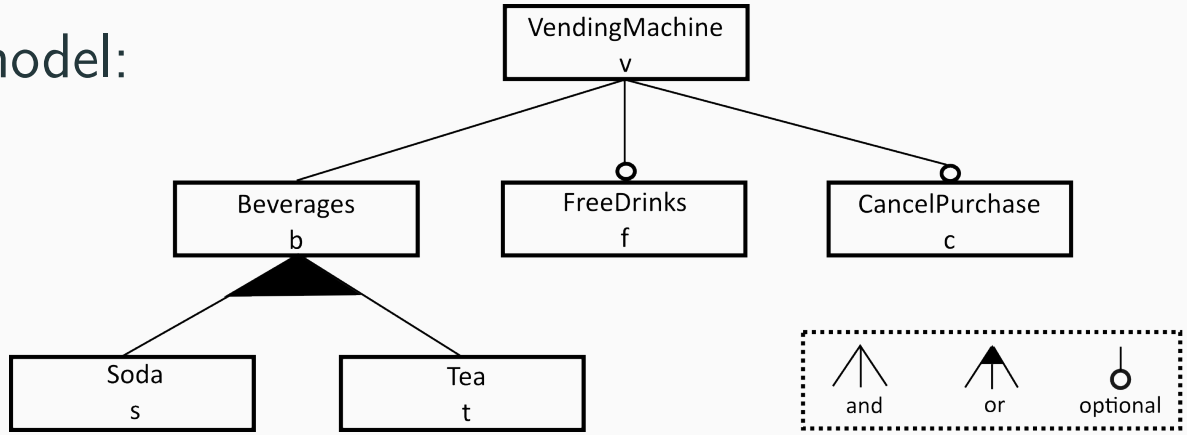
Classen *et al.*, Model checking lots of systems @ ICSE'10 (MIP Award @ SPLC'20), FTSs. *IEEE TSE*, 2013





FTS of example SPL: a vending machine

Feature model:

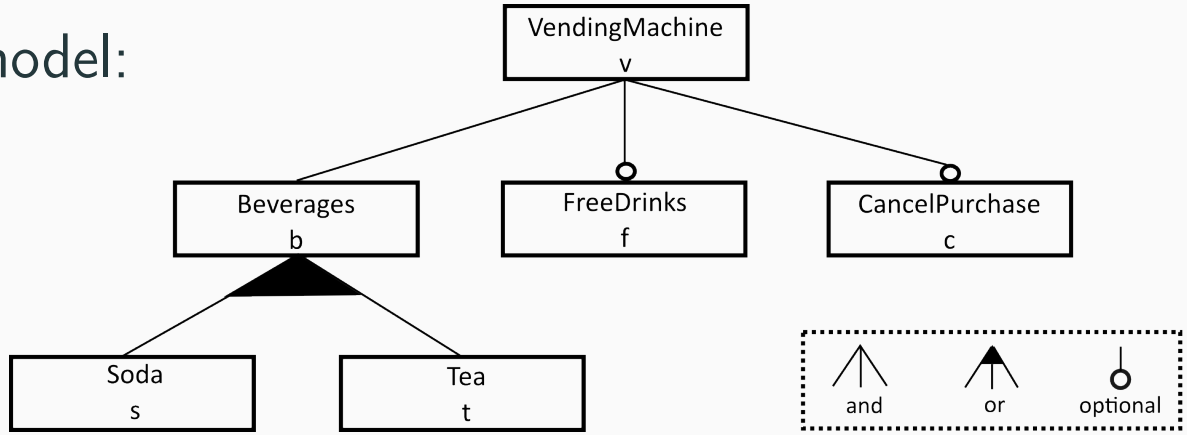


12 valid products

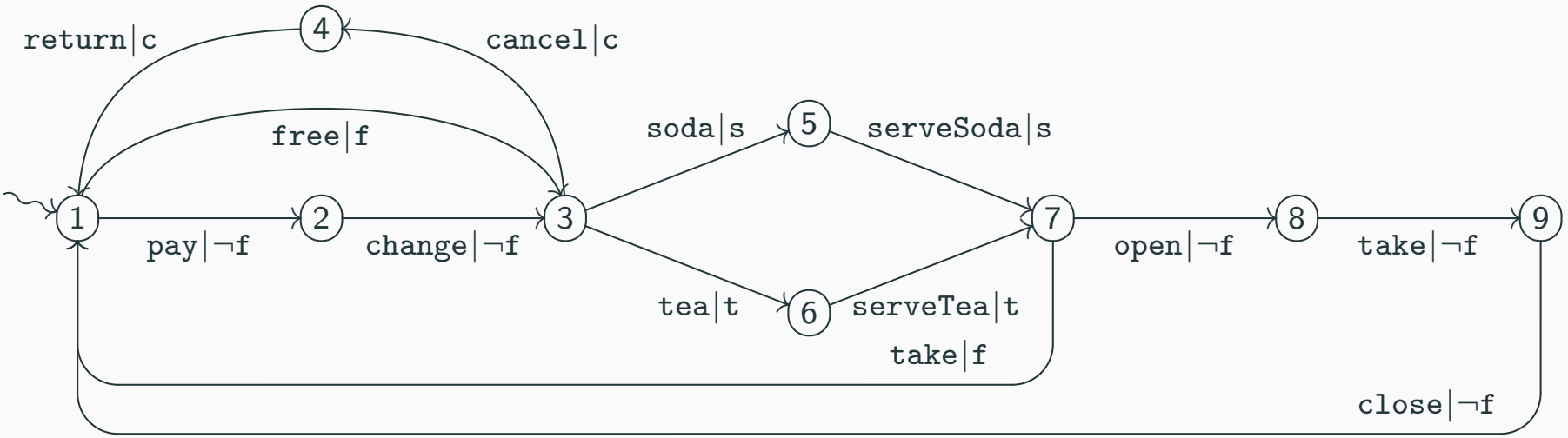
e.g., $\{\texttt{v,b,s,t}\}$, $\{\texttt{v,b,s,c}\}$

FTS of example SPL: a vending machine

Feature model:

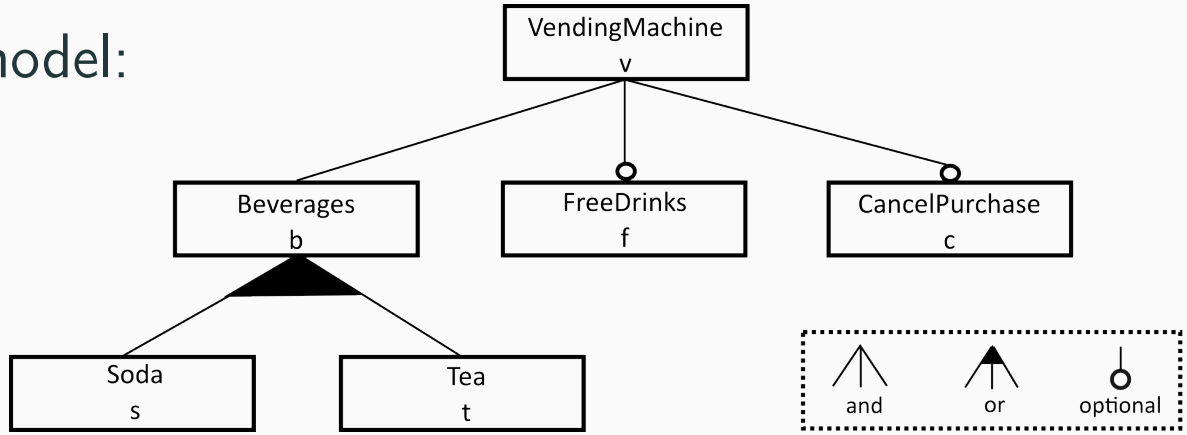


FTS of 12 valid products (LTSs) e.g., {v,b,s,t}, {v,b,s,c}

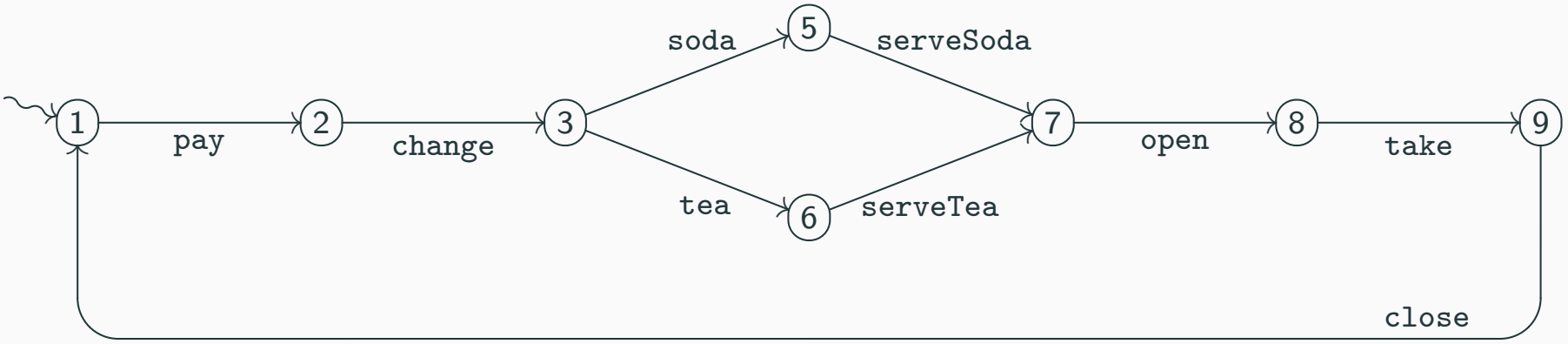


FTS of example SPL: a vending machine

Feature model:

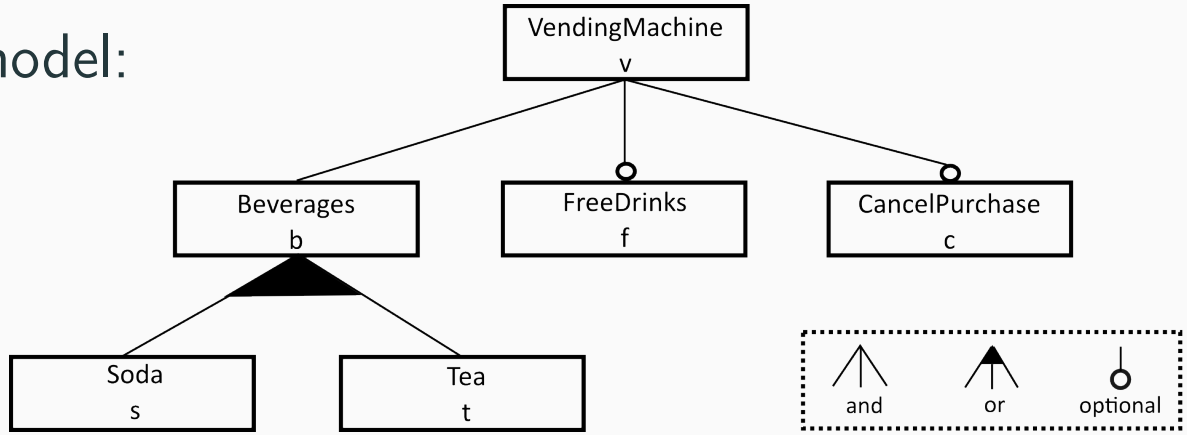


12 valid products (LTSs) e.g., {v,b,s,t}, {v,b,s,c}

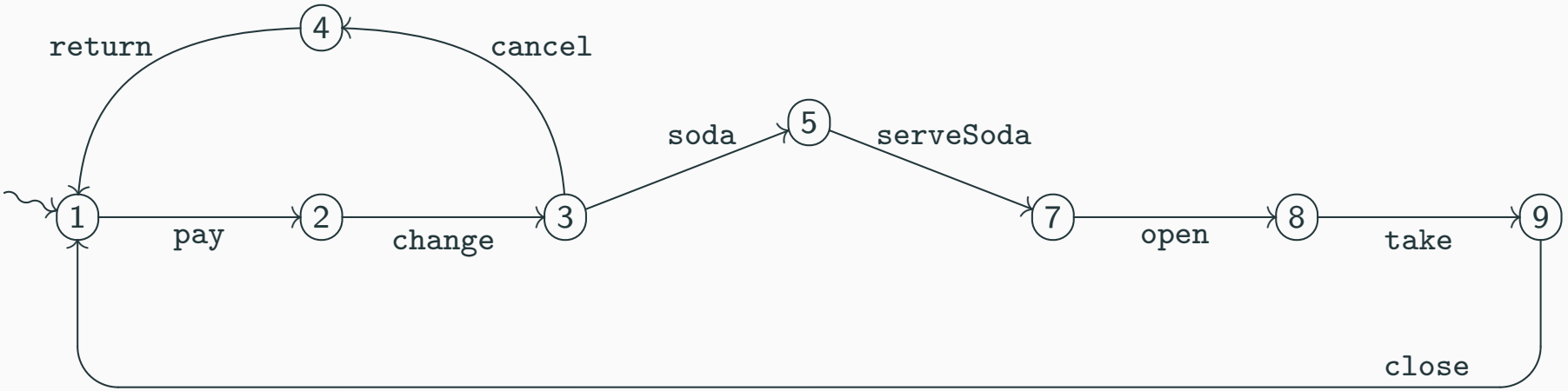


FTS of example SPL: a vending machine

Feature model:



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



Ambiguities in behavioural SPL models

Key idea and aim of our SPLC'19 paper

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 our SPLC'19 paper

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: provide unclear ideas of the SPLs
2. Unambiguous FTSs pave way to efficient family-based verification

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 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 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

---

Deadlock freedom is an important safety property: a system should not reach a state where no further action is possible, thus guaranteeing *progress* or liveness; for configurable systems, this extends to guaranteeing liveness for all system variants (products)

Ambiguous FTS $\longrightarrow$ unambiguous FTS [skip]

Transformation:

1. remove dead transitions

Ambiguous FTS $\longrightarrow$ unambiguous FTS [skip]

Transformation:

1. remove dead transitions

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

Ambiguous FTS $\longrightarrow$ unambiguous FTS [skip]

Transformation:

1. remove dead transitions

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

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 [skip]

Transformation:

1. remove dead transitions

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

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 with $\dagger \notin \Sigma$ and with a feature expression that negates the disjunction of the feature expressions of all outgoing transitions of s

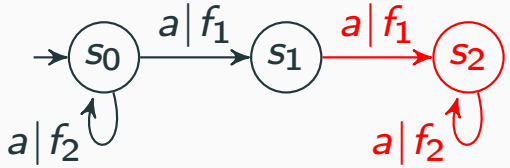
---

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)

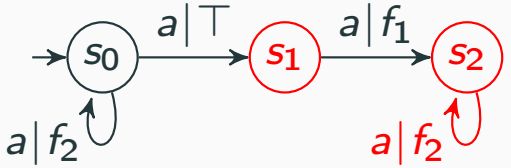
Example transformations [skip]

Feature Model: $f_1 \oplus f_2$

$\mathcal{F}$



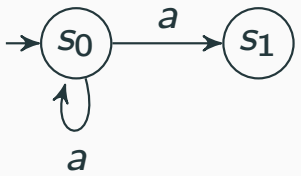
$\mathcal{F}'$



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

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

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

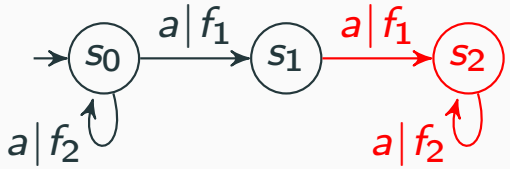


products $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$

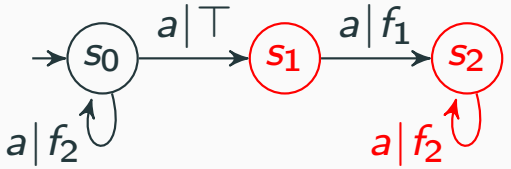
Example transformations [skip]

Feature Model: $f_1 \oplus f_2$

$\mathcal{F}$



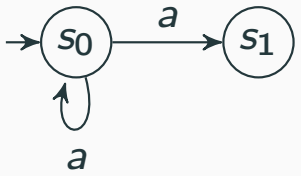
$\mathcal{F}'$



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

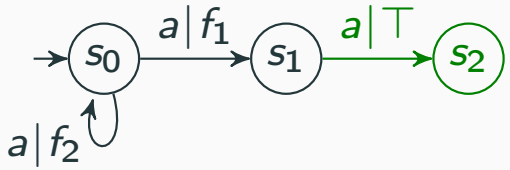
$\mathcal{F}|_{\lambda_2}$

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

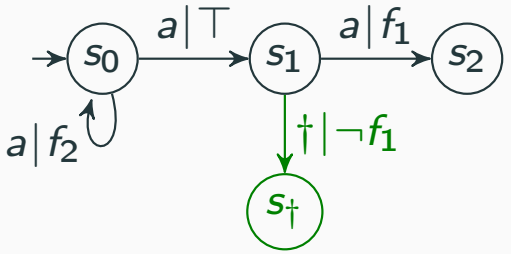


products $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$

$\mathcal{F}_*$



$\mathcal{F}'_*$



Efficient static analysis of FTSs

One-by-one vs. all-in-one [recall]

Recall: family-based (all-in-one) analysis

- The analysis of ambiguities for a single SPL's product is an expensive (feasible) task that can be automatised
- To apply the "brute force" by analysing all products of an SPL is too expensive to be used in concrete cases
- However products of an SPL share a common "piece of code", thus analysing each product individually (brute-force analysis) would involve a lot of redundancy

How to leverage this commonality and analyse the whole product line at once, bringing the total analysis time down, is a our goal!

Implementation in Z3

FTS ambiguity detection problem is NP-complete: we used a SAT tool!

Implementation in Z3

FTS ambiguity detection problem is NP-complete: we used a SAT tool!

- We implemented our algorithm in Z3
- We made our implementation publicly available, including all benchmark examples used in the rest of the talk
- Our artefact received the ACM reusable badge:

[SPLC'19]

Implementation in Z3

FTS ambiguity detection problem is NP-complete: we used a SAT tool!

- We implemented our algorithm in Z3
- We made our implementation publicly available, including all benchmark examples used in the rest of the talk
- Our artefact received the ACM reusable badge:

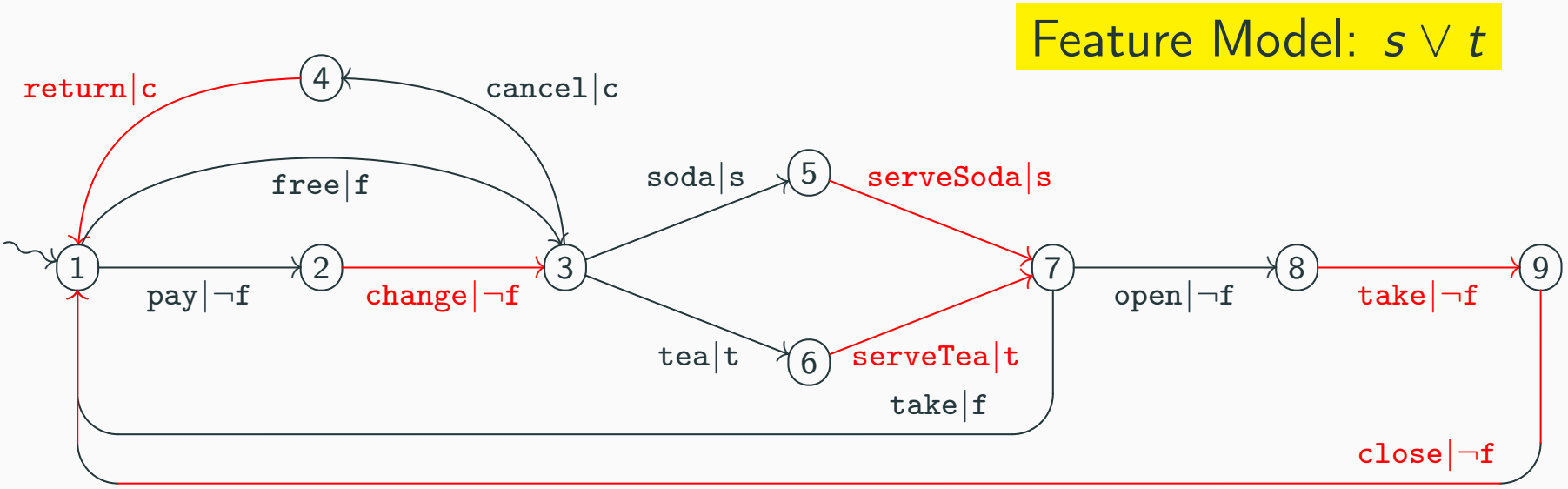
[SPLC'19]

Z3 is a cross-platform Satisfiability Modulo Theories (SMT) solver (that includes a SAT solver) developed by Microsoft:

- it is freely available under the MIT license
- it supports arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers
- its typical applications are static checking, test-case generation, and predicate abstraction

Example and benchmark experiments

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 = []
```

Benchmark experiments (1/3) [EMSE22]

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) [SPLC19] vs. [EMSE22]

FTS	characteristics			computational effort				results
				implementation [SPLC19]		implementation [EMSE22]		
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)

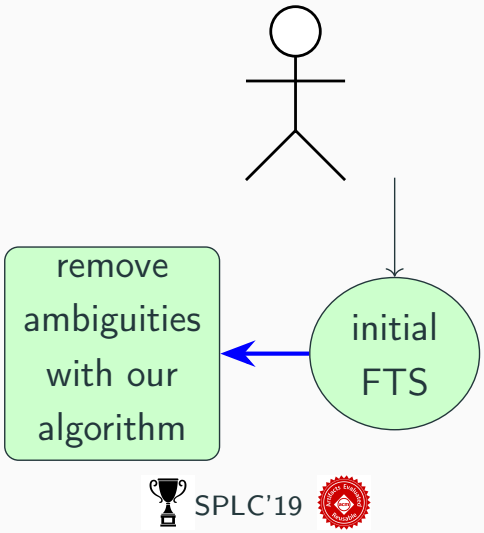
Benchmark experiments (3/3) [liveness only]

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%

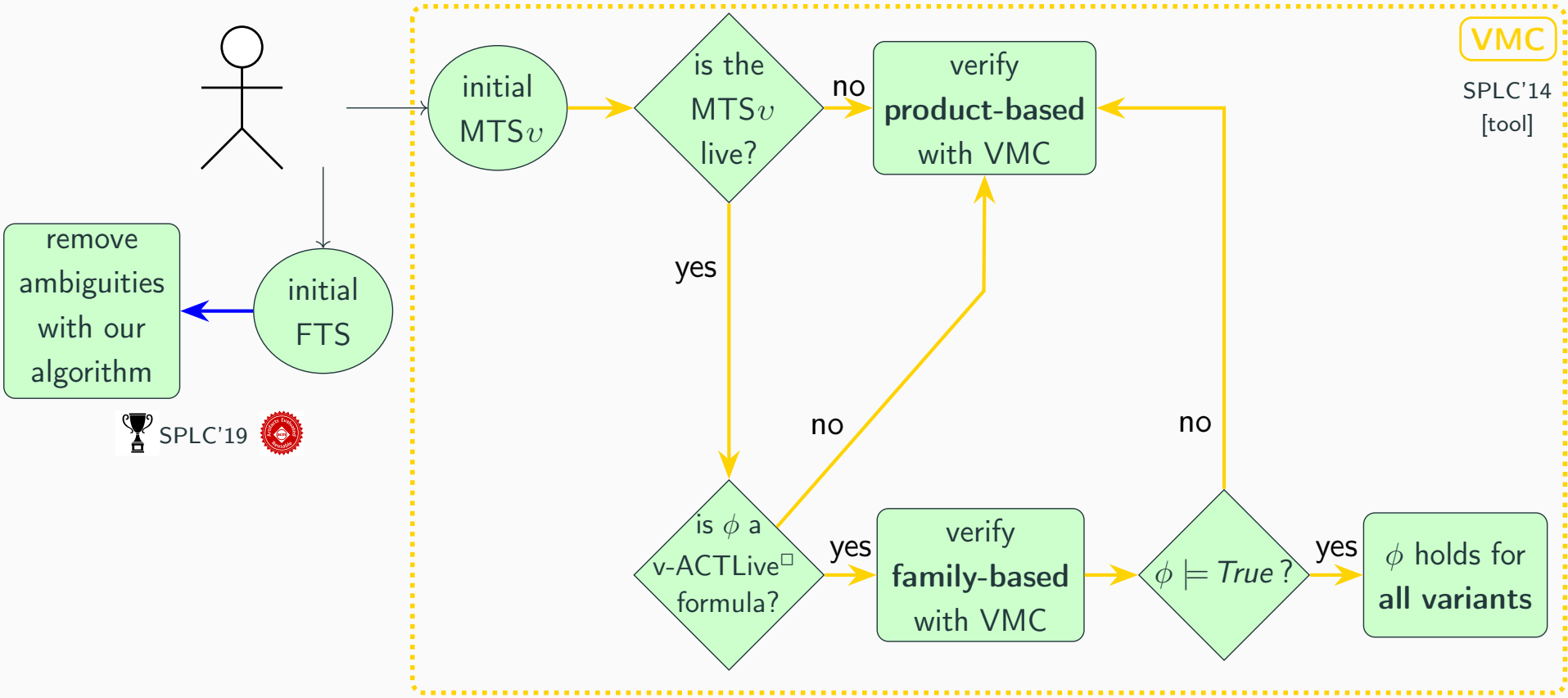
The experiments were performed on a virtual machine Gentoo 201905, CLI Version VirtualBox (VDI) 64 bit, with 2048 Mb of allocated memory on a Windows 10 Pro 64 bit with 16 Gb of RAM and CPU AMD Ryzen 7 1700X (8 core, 16 threads, 3.4 Ghz)

Wrap up so far and outlook for tooling

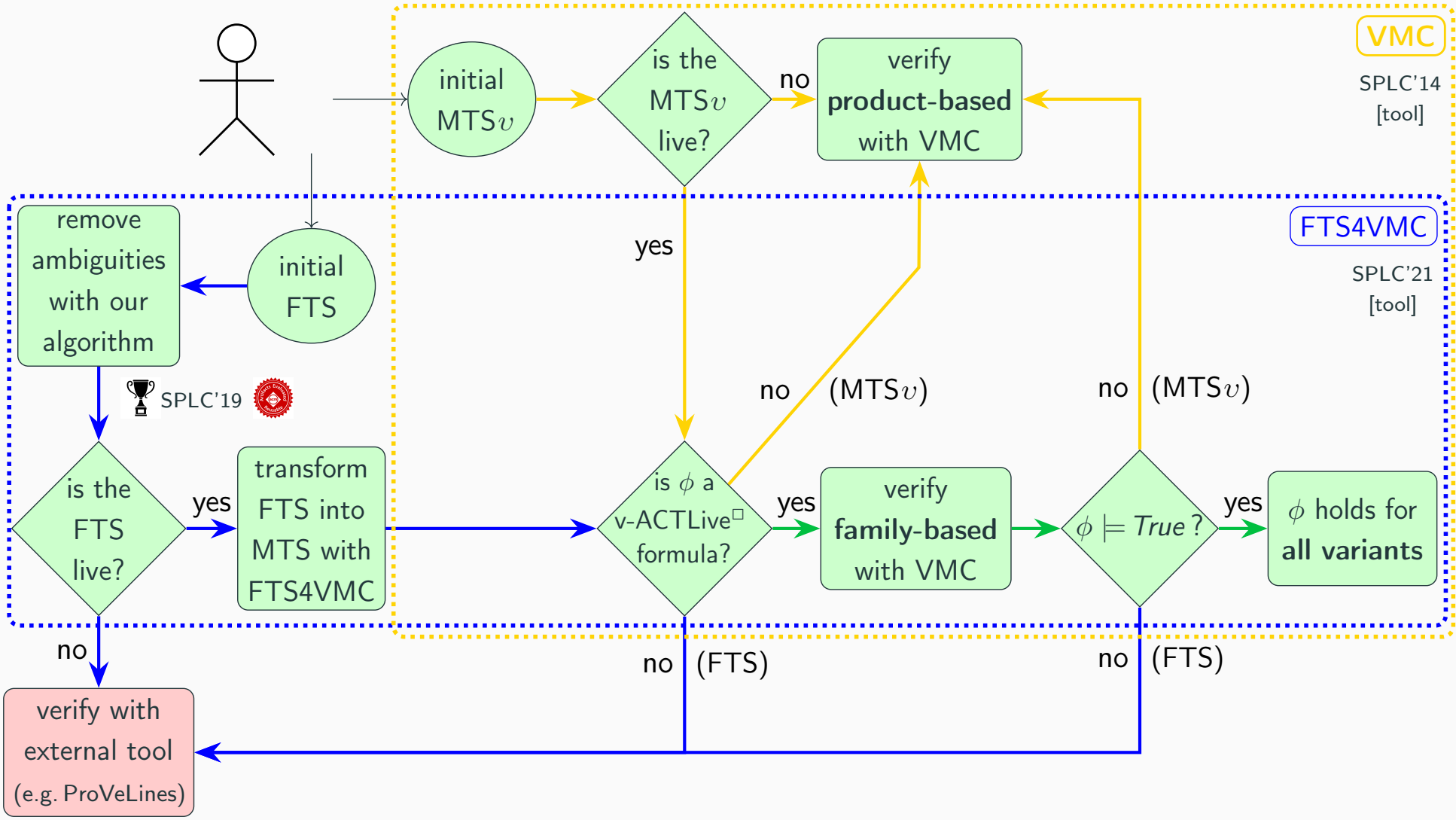
Detect and remove ambiguities



VMC: model checking MTSυ



FTS4VMC: front-end for VMC: model checking MTSυ/FTS



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

Modal Transition Systems (MTSs) with variability constraints (MTSυ)

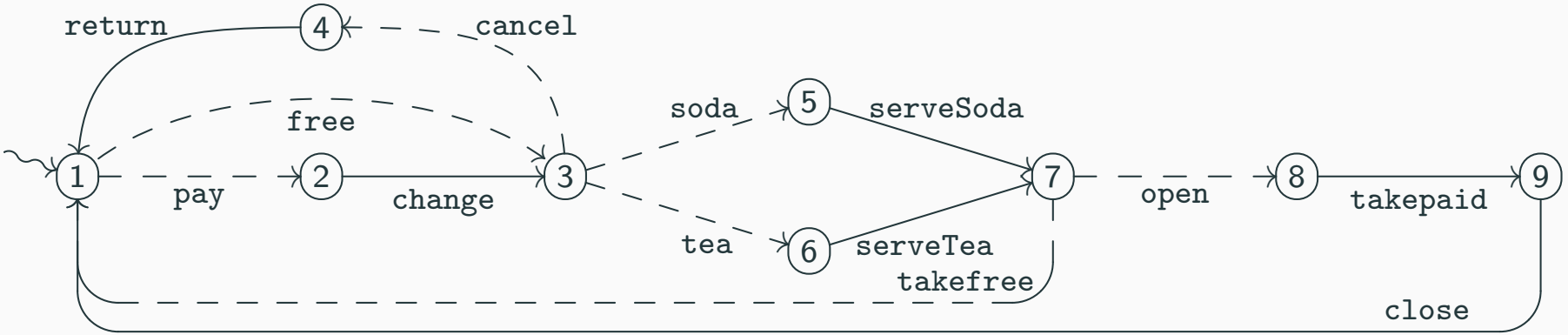
MTSυ of example SPL

An MTSυ adds to an LTS so-called may and must transitions

\+ (variability) constraints

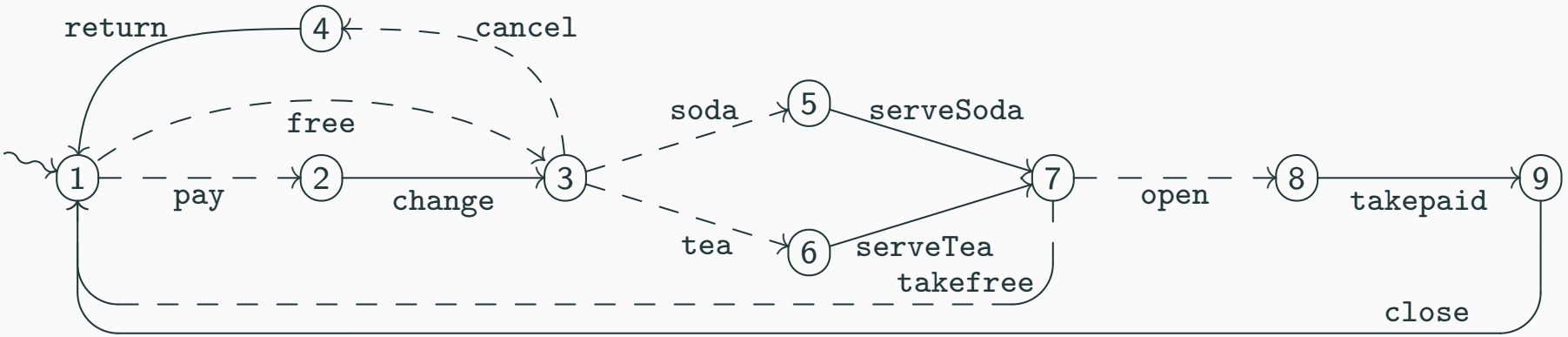
Asirelli *et al.*, Formal Description of Variability in Product Families @ SPLC'11

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

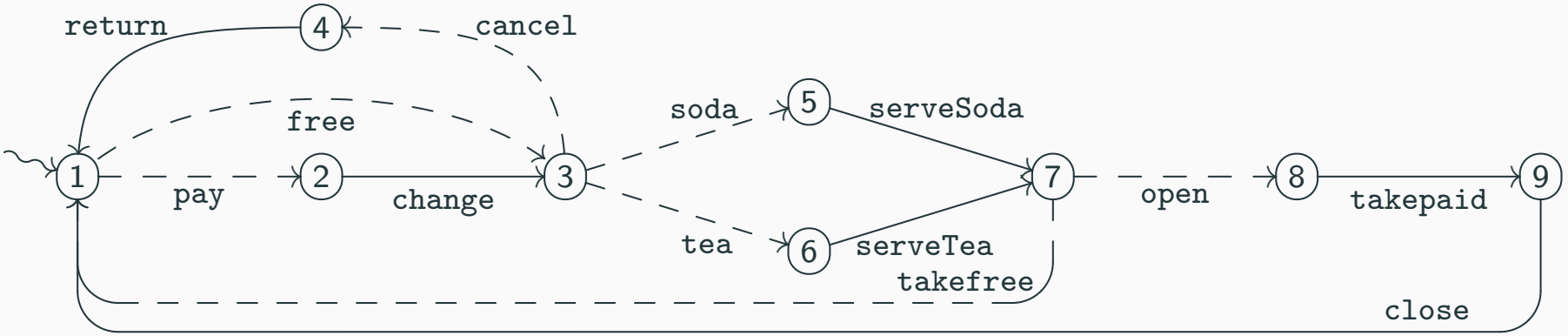



```
Constraints {
 pay ALT free        // precisely one must be present
 soda OR tea         // at least one must be present
 takefree IFF free   // either both or none are present
 open ALT takefree   // precisely one must be present
}
```

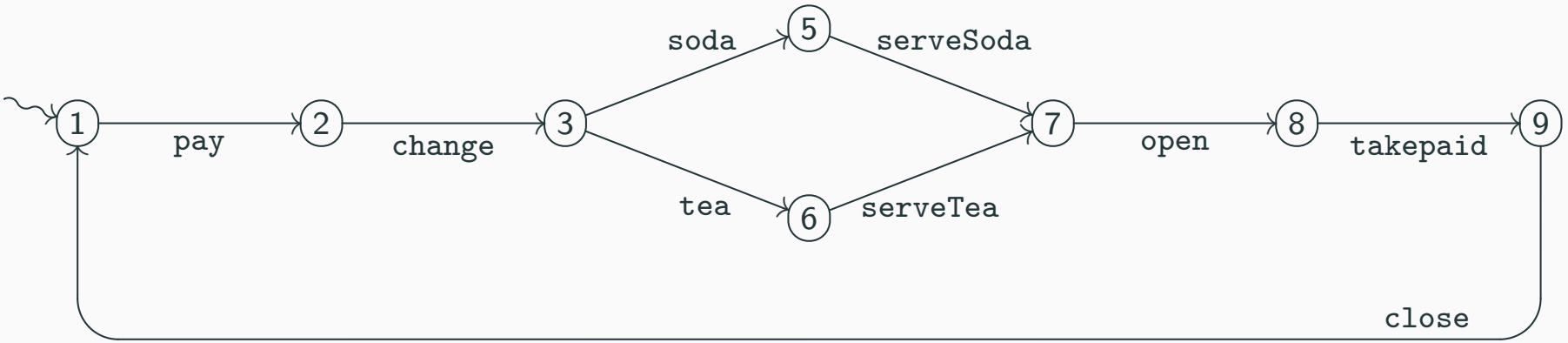
Product LTSs of MTSv of example SPL



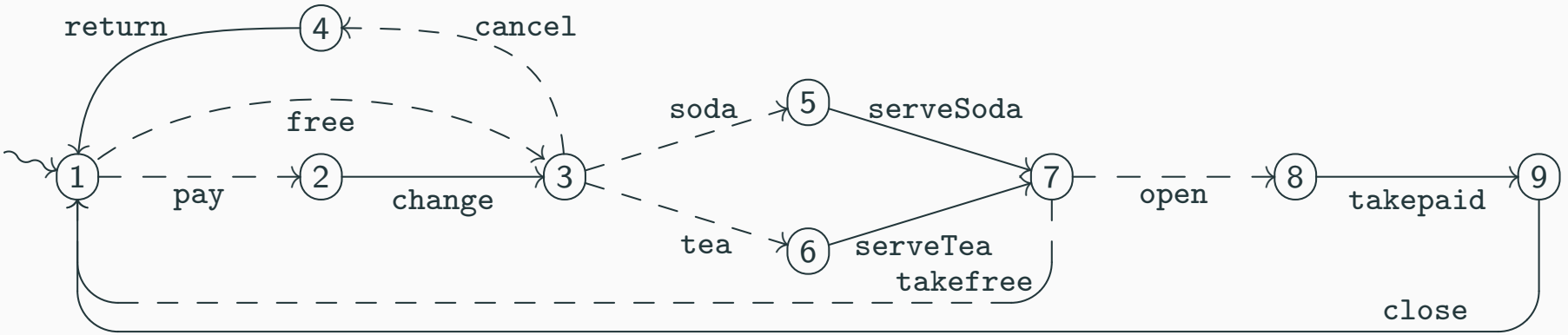
Product LTSs of MTSυ of example SPL



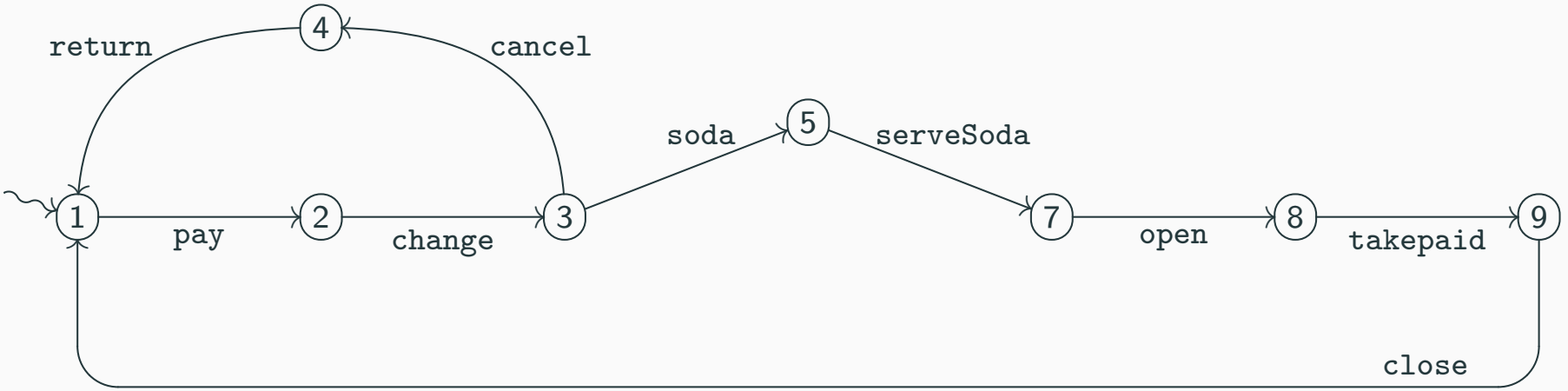
{pay,soda,tea,open}



Product LTSs of MTSυ of example SPL



{pay,cancel,soda,open}



VMC: family-based model checking MTSs

Variability analysis strategies supported by VMC

ter Beek & Mazzanti, VMC @ FM'12, SPLC'14

VMC:

- Input models in the form of a set of process-algebraic definitions
- Displays all possible family evolutions in the form of an MTS graph
- Supports a comprehensive temporal logic interpreted on L^2TSs

VMC is freely usable online: `http://fmt.isti.cnr.it/vmc/`

Variability analysis strategies supported by VMC

ter Beek & Mazzanti, VMC @ FM'12, SPLC'14

VMC:

- Input models in the form of a set of process-algebraic definitions
- Displays all possible family evolutions in the form of an MTS graph
- Supports a comprehensive temporal logic interpreted on L^2TSs

VMC offers product- and family-based model checking of MTSυs:

- Allows to verify (some of the) properties such that they hold for all LTS products of the MTS family . . . and to receive feedback when such properties do not hold ('family-based')
- Allows to generate all valid products of an MTS family . . . and to separately verify properties on them (product-based)

. . . with a linear complexity

VMC is freely usable online: `http://fmt.isti.cnr.it/vmc/`

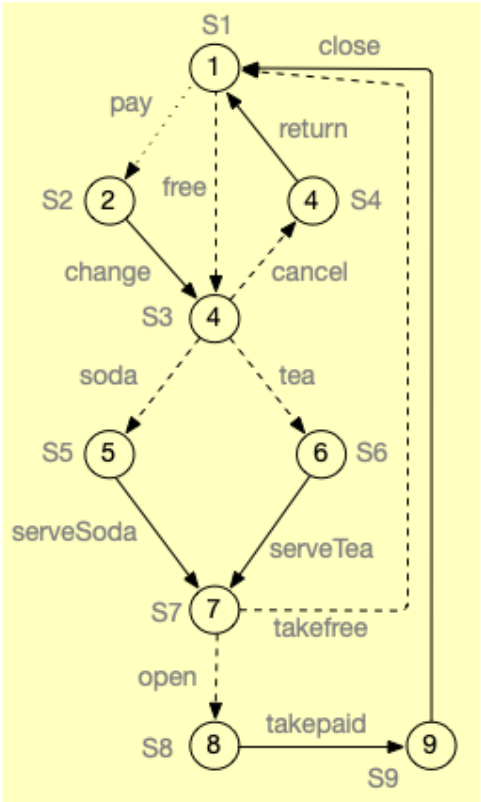
VMC input: textual model encoding as process definitions

VMC V6.5
(2019)

New Model ...
Load Current Model
Welcome
Quit


Kandinsky 1908

```
1
2   --  VENDING MACHINE  - MTSv version
3
4   S1 = pay(may) .S2
5        + free(may) .S3
6   S2 = change(must)  .S3
7   S3 = cancel(may) .S4
8        + soda(may) .S5
9        + tea(may) .S6
10  S4 = return(must) .S1
11  S5 = serveSoda(must) .S7
12  S6 = serveTea(must) .S7
13  S7 = takefree(may) .S1
14       + open(may) .S8
15  S8 = takepaid(must) .S9
16  S9 = close(must) .S1
17
18
19  net SYS = S1
20
21  Constraints {
22   pay ALT free
23   soda OR tea
24   takefree IFF free
25   open ALT takefree
26  }
27
28
29
30
31
```



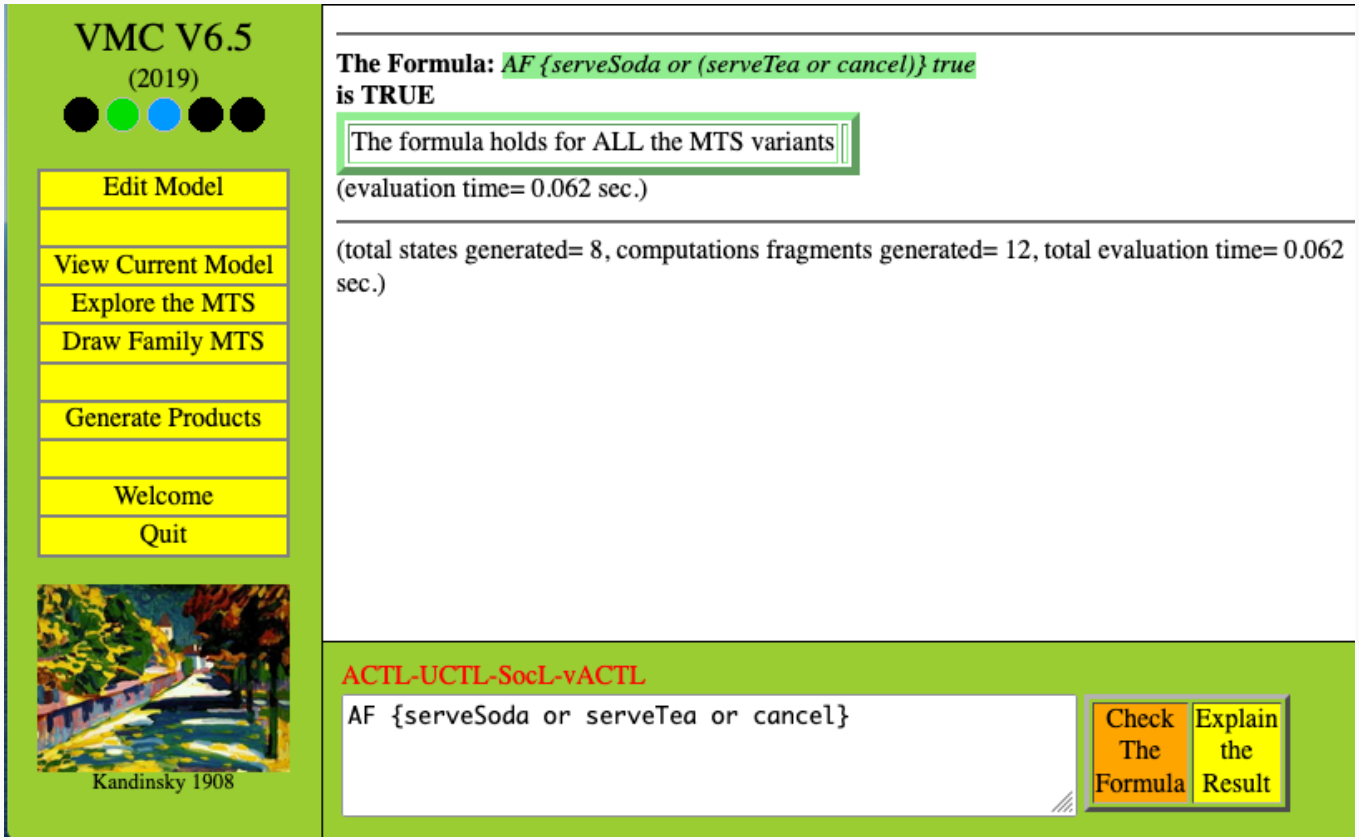
Vending Machine: family-based model checking with VMC

The properties preserved for all LTS products are those that hold for all possible executions—and use a live fragment of the MTS υ (live implies no hidden deadlocks!)

Vending Machine: family-based model checking with VMC

The properties preserved for all LTS products are those that hold for all possible executions—and use a live fragment of the MTS υ (live implies no hidden deadlocks!)

VMC notifies whenever preservation of an analysis result applies:



Vending Machine: product-based model checking with VMC

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

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+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

FTS4VMC: front-end for VMC (family-based model checking FTSs and FTSs)

Family-based model checking: v-ACTLive$^{\Box}$

v-ACTLive$^{\Box}$: fragment of Action-based CTL with dedicated variability-aware versions of temporal operators

Family-based model checking: v-ACTLive$^{\Box}$

v-ACTLive$^{\Box}$: fragment of Action-based CTL with dedicated variability-aware versions of temporal operators

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)

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

Family-based model checking: v-ACTLive$^{\Box}$

v-ACTLive$^{\Box}$: fragment of Action-based CTL with dedicated variability-aware versions of temporal operators

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 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 MTS υ to be live

Family-based model checking: v-ACTLive$^{\Box}$

v-ACTLive$^{\Box}$: fragment of Action-based CTL with dedicated variability-aware versions of temporal operators

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)

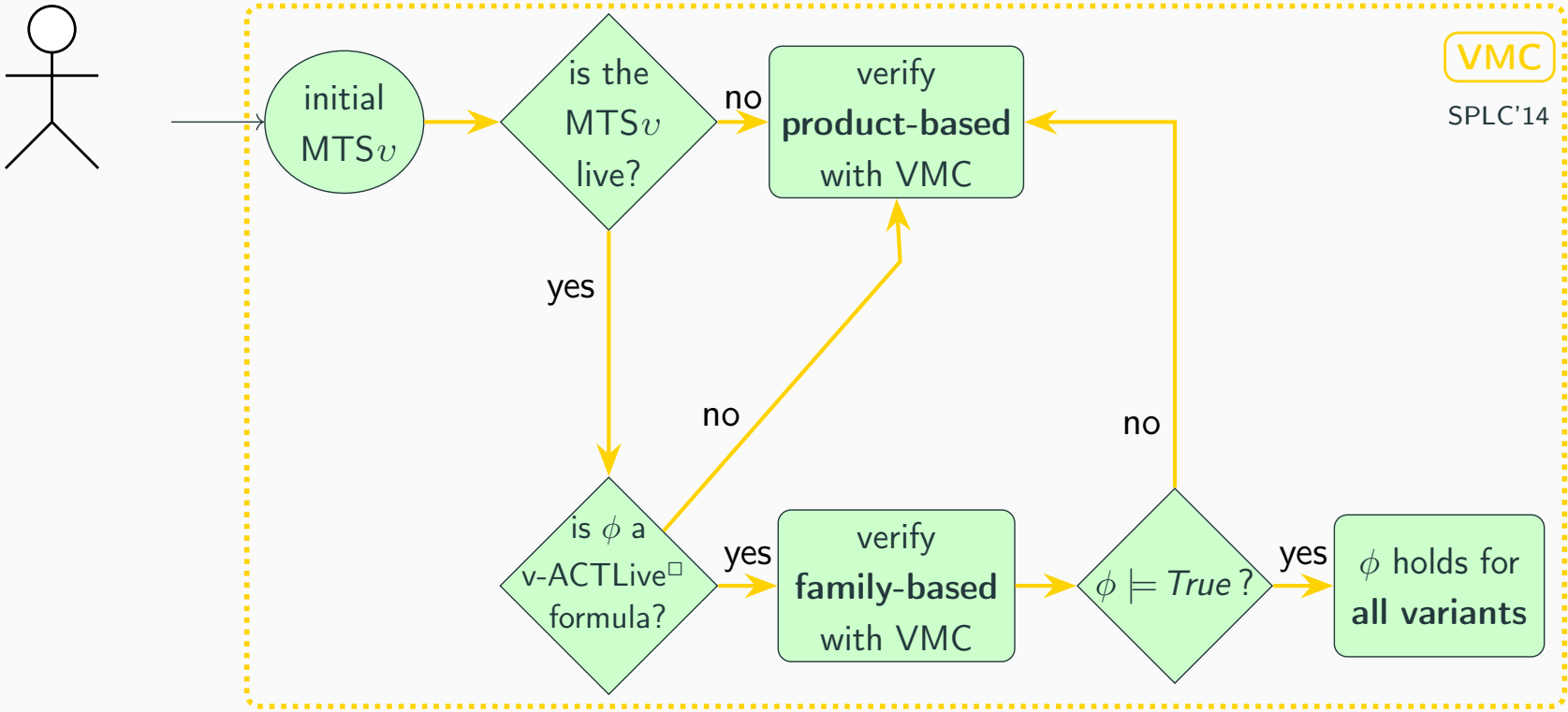
This allows to carry over a result for MTSυs to unambiguous FTSs:

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

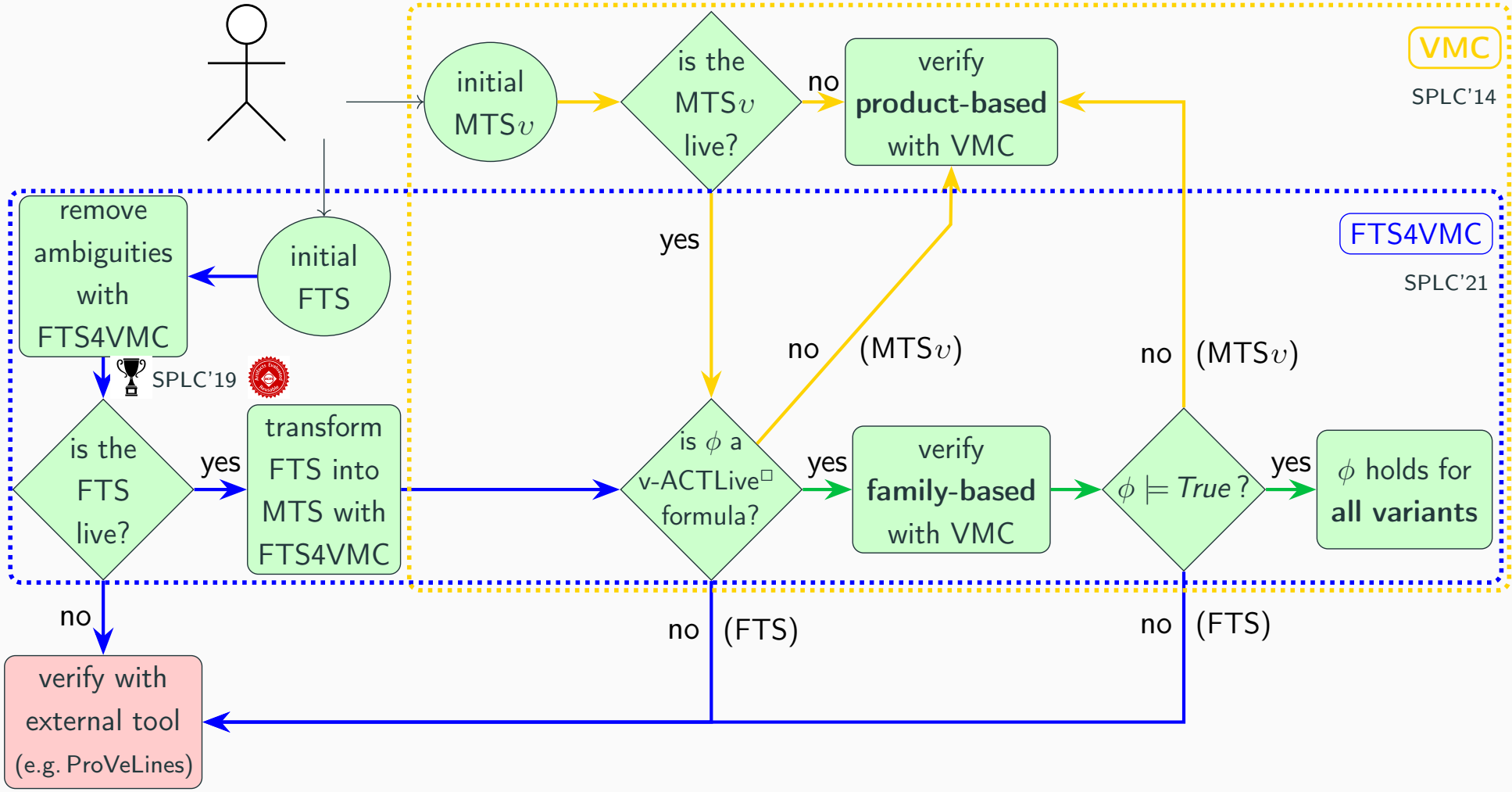
Any formula ϕ of v-ACTLive$^{\Box}$ is preserved by live FTSs: given a live FTS $\mathcal{F}$, whenever $\mathcal{F}_{MTS} \models \phi$, then $\mathcal{F}|_\lambda \models \phi$ for all products $\mathcal{F}|_\lambda$ of $\mathcal{F}$

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

Toolchain: family-based model checking of MTSυs



Toolchain: family-based model checking of MTSυs *and* FTSs



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

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

Conclusions and Future Work

Conclusion and Future Work

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness [EMSE22]
   - benchmark experiments

Conclusion and Future Work

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness [EMSE22]
   - benchmark experiments

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

Conclusion and Future Work

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness [EMSE22]
   - benchmark experiments

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

3. Automated by a toolchain:
   - publicly available front-end tool FTS4VMC

Conclusion and Future Work

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness [EMSE22]
   - benchmark experiments

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

3. Automated by a toolchain:
   - publicly available front-end tool FTS4VMC

4. Ongoing work:
   - integrating 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 23rd International Systems and Software Product Line Conference (SPLC'19), ACM, 2019, 39–51 (best paper) https://doi.org/10.1145/3336294.3336295

EMSE22 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* 22, 1 (2022), 10:1–10:43 https://doi.org/10.1007/s10664-020-09930-8

SPLC21 M.H. ter Beek, F. Mazzanti, F. Damiani, L. Paolini, G. Scarso, M. Valfrè, and M. Lienhardt, Static Analysis and Family-based Model Checking of Featured Transition Systems with VMC. In Proceedings 25th International Systems and Software Product Line Conference (SPLC'21), ACM, 2021 (tool demo) https://doi.org/10.1145/3461002.3473071

SCP22 M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, L. Paolini, and Giordano Scarso, FTS4VMC: A front-end tool for static analysis and family-based model checking of FTSs with VMC. *Science of Computer Programming* 224 (2022), 102879 (original software publication) https://doi.org/10.1016/j.scico.2022.102879

Family-based model checking (1/2): v-ACTLive$^{\Box}$ [recall]

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 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 (2/2): LTL

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

Model checking LTL formulas 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): LTL

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

Model checking LTL formulas 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): LTL

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

Model checking LTL formulas 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 . . .

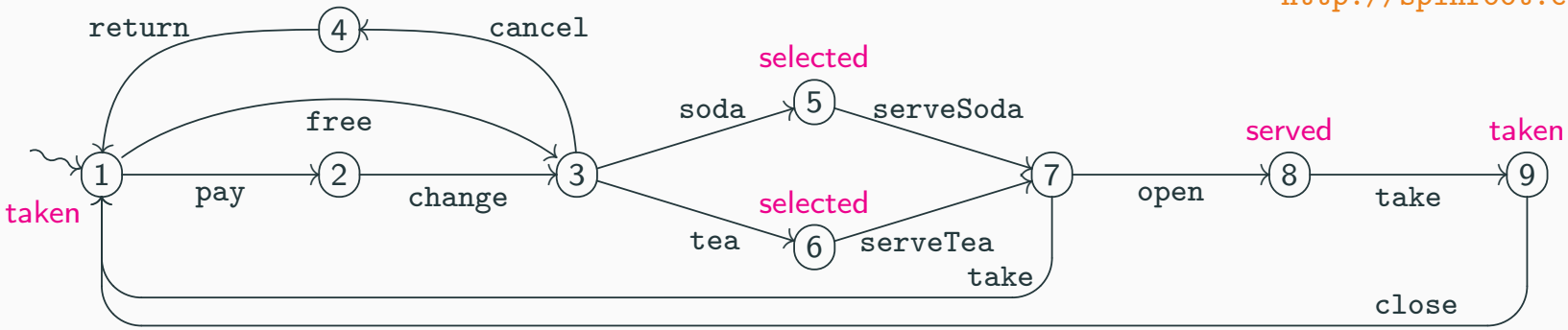
Example LTL formulas that can thus be verified with SPIN (for instance):

http://spinroot.com/

Family-based model checking with SPIN ...

Example LTL formulas that can thus be verified with SPIN (for instance):

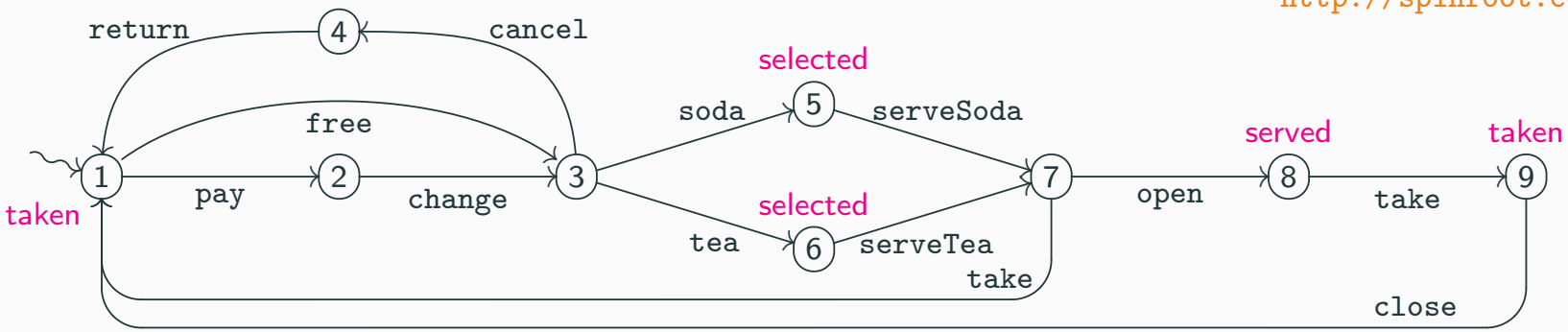
http://spinroot.com/



Family-based model checking with SPIN ...

Example LTL formulas that can thus be verified with SPIN (for instance):

http://spinroot.com/

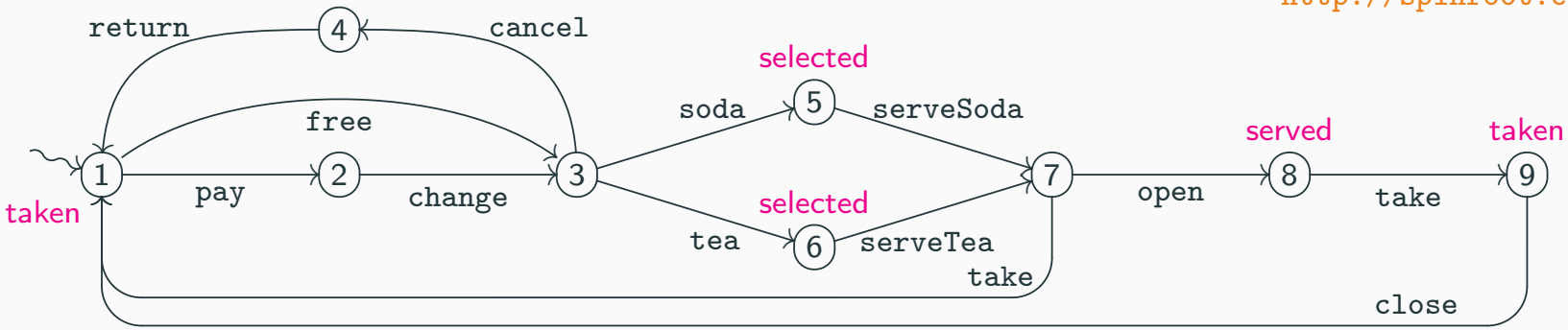


1. $G\,(selected \Rightarrow F\ served)$: *after a beverage has been selected, the vending machine will always eventually have served a beverage*
2. $G\,(served \Rightarrow F\ taken)$: *after a beverage has been served, a customer will always eventually have taken the beverage*

Family-based model checking with SPIN ...

Example LTL formulas that can thus be verified with SPIN (for instance):

http://spinroot.com/



1. $G\,(selected \Rightarrow F\ served)$: *after a beverage has been selected, the vending machine will always eventually have served a beverage*
2. $G\,(served \Rightarrow F\ taken)$: *after a beverage has been served, a customer will always eventually have taken the beverage*