

Efficient Static Analysis and Verification of Featured Transition Systems

Maurice ter Beek
Franco Mazzanti

Ferruccio Damiani
Luca Paolini
Giordano Scarso

Michael Lienhardt

ISTI-CNR, Pisa, IT

University of Turin, IT

ONERA, France

Tutorial @ SPLC'21, Leicester, UK, September 7th, 2021

Tutorialists

Maurice ter Beek



- Senior researcher, FMT lab, ISTI-CNR, IT, lab head since 2019
- M.Sc. ('96) and Ph.D. ('03) degrees from Leiden University, NL
- Formal methods, model checking (tools), team automata, SOC, SPLE, railway systems

Franco Mazzanti



- Senior researcher, FMT lab, ISTI-CNR, IT
- Main developer of KandISTI formal verification framework, including the model-checking tools CMC, FMC, UMC & VMC
- Analysis of concurrent, asynchronous systems using state- and event-based branching-time temporal logics, and by exploiting formal methods and tools diversity

Tutorialists (2/3)

Ferruccio Damiani



- Associate professor, UNITO, IT, head MoVeRe research group
- Computational models and languages; concurrent, distributed, and mobile systems; variability modelling and SPLs

Luca Paolini



- Associate professor, UNITO, IT, member of MoVeRe group
- Formal analysis of SPLs; development of programming languages; innovative and unconventional computing models

Giordano Scarso



- M.Sc. student, UNITO, IT, B.Sc. ('20) degree from UNITO, IT
- Thesis on and development of FTS4VMC supervised by Damiani, ter Beek & Mazzanti

Michael Lienhardt



- Research engineer in the field of applied formal methods in computer science, French Space Lab ONERA, FR
- Design of a new generation of aerodynamics simulators
- Concurrency, static analysis and software engineering

Part I

- Behavioural variability modelling and analysis
 - 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: examples and benchmark experiments

Part I

- Behavioural variability modelling and analysis
 - 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: examples and benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

Part I

- Behavioural variability modelling and analysis
 - 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: examples and benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

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

Configure your BMW vehicle

http://www.bmw.com/en/general/carconfigurator/content.html

BMW dealer Brochures Corporate/Direct Sales Shop BMW Financial Services Used Vehicles Search

Home 1 2 3 4 5 6 7 X Z4 BMW M BMW i BMW Owners BMW Insights

Configure vehicle

The International BMW website



BMW
Never Stop Driving.

Configure your BMW vehicle

Are you interested in configuring your ideal BMW? Please select a country to visit the configurator in the Virtual Center or contact your local BMW dealer, who will be happy to answer all your questions about the BMW model you are interested in.

Related topics



Request information
Order product catalogues,
brochures and equipment
lists direct from BMW.

FIND YOUR BMW.



Filter

> Reset filter

- Budget
- Vehicle type
- All
- Petrol
- Diesel
- Hybrid
- Electric Vehicle

Body type

- Saloon
- Touring
- Convertible
- Coupé
- Gran Turismo
- Sports Hatch
- Roadster
- Sports Activity Coupé
- Sports Activity Vehicle

Number of seats

30 Vehicles **465 Model variants**



BMW 1 Series 3-door Sports Hatch (24)
from € 17,775.00



BMW 1 Series 5-door Sports Hatch (29)
from € 18,305.00



BMW 2 Series Coupé (14)
from € 24,265.00



BMW 3 Series Saloon (56)
from € 23,550.00



BMW 3 Series Touring (54)
from € 24,865.00



BMW 3 Series Gran Turismo (39)
from € 29,200.00

Feature models

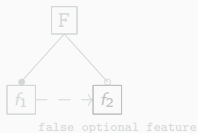
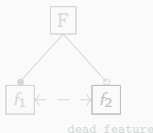
- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user



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 / 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

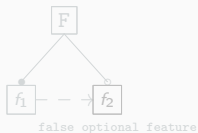
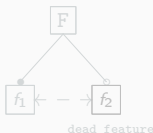
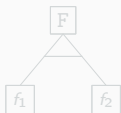
320 optional, independent
features

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

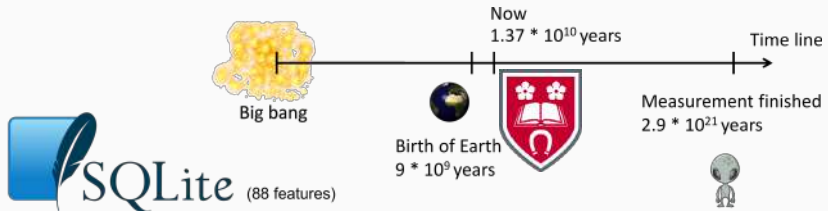
Product-based analysis:

- 👍 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





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

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

Featured Transition Systems (FTSs)

Part I

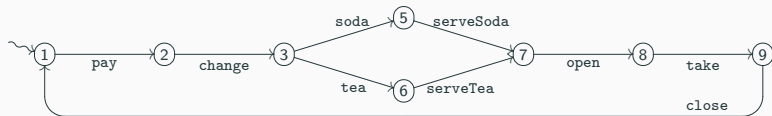
- Behavioural variability modelling and analysis
 - Product- vs family-based analysis
 - Featured Transition System (FTS)
 - Ambiguities: dead/false optional transitions, hidden deadlocks
- Automated static analysis of FTSs
 - Detecting ambiguities: criteria, SAT solving, implementation
 - Disambiguating FTSs: examples and benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

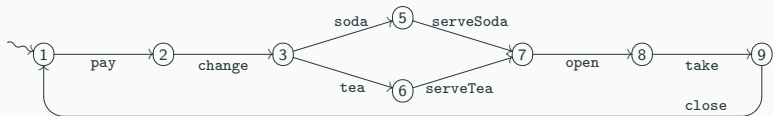
Labelled Transition System (LTS)

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



Labelled Transition System (LTS)

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



S ① ② ③ ... ⑦ ⑧ ⑨

Σ pay change ... open take close

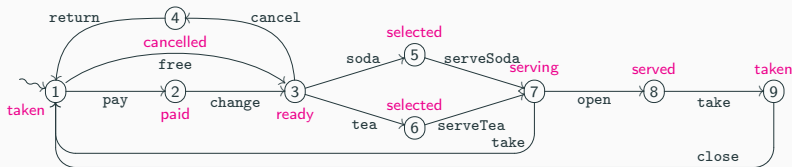
s_0 

δ ① $\xrightarrow{\text{pay}}$ ② ② $\xrightarrow{\text{change}}$ ③ ... ⑦ $\xrightarrow{\text{open}}$ ⑧ ⑧ $\xrightarrow{\text{take}}$ ⑨
⑨ $\xrightarrow{\text{close}}$ ①

Doubly-Labelled Transition System (L^2TS)

A **Doubly-Labelled Transition System** (L^2TS) is a sextuple $(S, \Sigma, s_0, \delta, AP, L)$ in which (S, Σ, s_0, δ) is an LTS, AP is a set of **atomic propositions**, and $L: S \rightarrow 2^{AP}$ is a function labelling each state with a set of AP (here singleton)

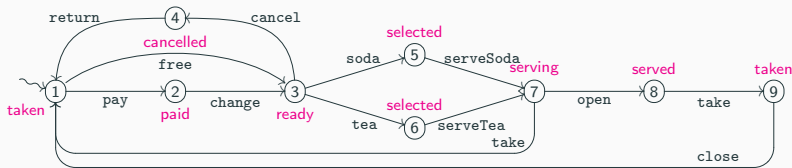
De Nicola & Vaandrager, Three Logics for Branching Bisimulation. *J. ACM*, 1995



Doubly-Labelled Transition System (L^2TS)

A **Doubly-Labelled Transition System** (L^2TS) is a sextuple $(S, \Sigma, s_0, \delta, AP, L)$ in which (S, Σ, s_0, δ) is an LTS, AP is a set of **atomic propositions**, and $L: S \rightarrow 2^{AP}$ is a function labelling each state with a set of AP (here singleton)

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



AP paid ready ... served taken

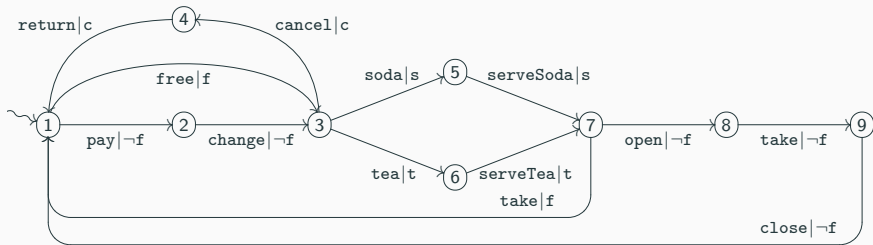
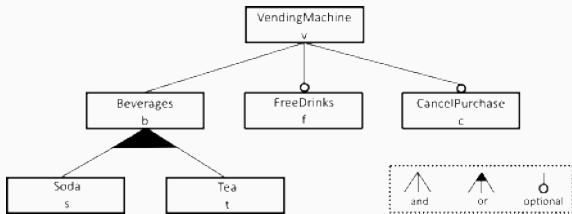
L (2) (3) ... (8) (9)
 paid ready served taken

Kripke structure: only state labelling, no transition labelling

From LTS to FTS

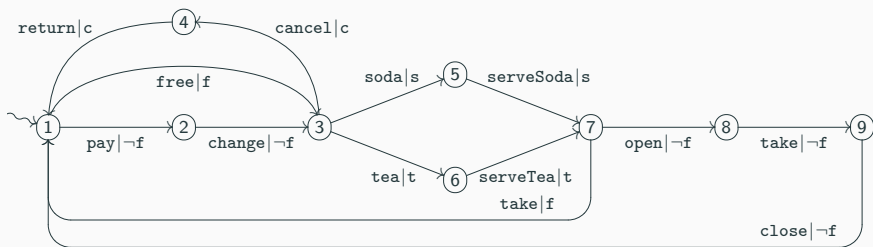
An FTS adds to an LTS a feature model and so-called **feature expressions**

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



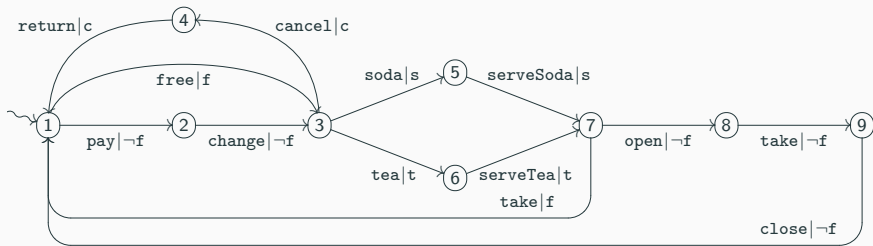
Featured Transition System (FTS)

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 $\Lambda \subseteq \{\lambda : F \rightarrow \mathbb{B}\}$



Featured Transition System (FTS)

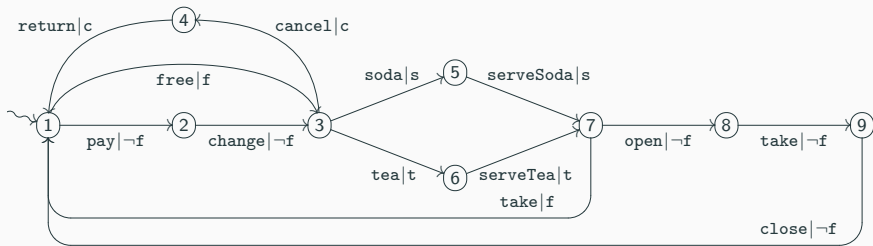
A **Featured Transition System** (FTS) is a sextuple $(S, \Sigma, s_0, \delta, F, \Lambda)$ with states S , actions Σ , initial state s_0 , (featured) transitions $\delta \subseteq S \times \Sigma \times \mathbb{B}(F) \times S$, with Boolean (feature) expressions $\mathbb{B}(F)$ over features F , and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



δ	$(1) \xrightarrow{\text{pay} \neg f} (2)$	$(1) \xrightarrow{\text{free} f} (3)$...
F	v b f	c s t	
Λ	$\{v, b, s, t\}$	$\{v, b, s, c\}$...

Featured Transition System (FTS)

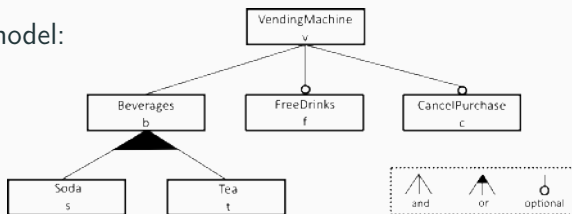
A **Featured Transition System** (FTS) is a sextuple $(S, \Sigma, s_0, \delta, F, \Lambda)$ with states S , actions Σ , initial state s_0 , (featured) transitions $\delta \subseteq S \times \Sigma \times \mathbb{B}(F) \times S$, with Boolean (feature) expressions $\mathbb{B}(F)$ over features F , and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



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:

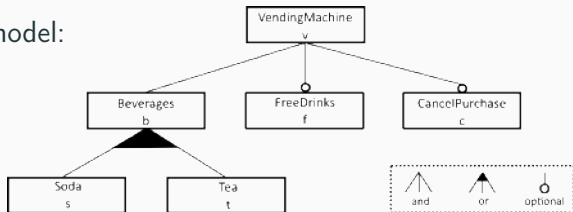


12 valid products

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

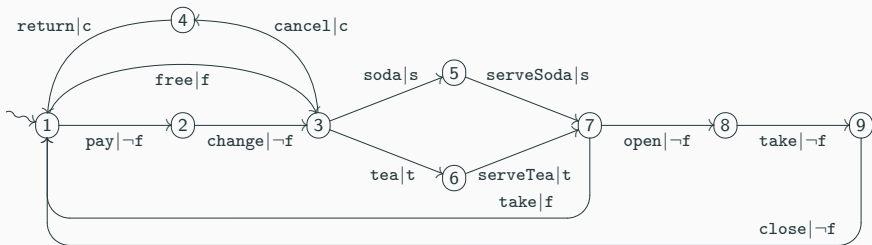
FTS of example SPL: a vending machine

Feature model:



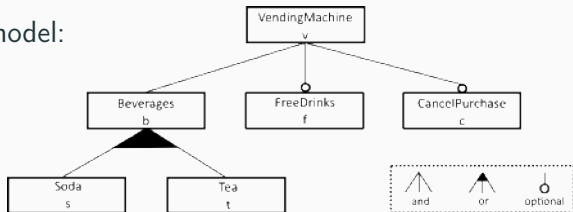
FTS of 12 valid products (LTSs)

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

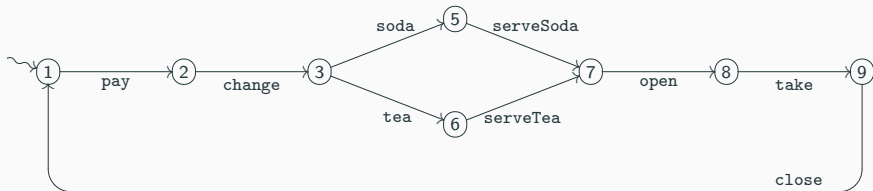


FTS of example SPL: a vending machine

Feature model:

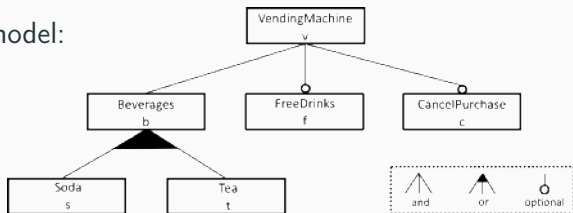


e.g., $\{v, b, s, t\}$, $\{v, b, s, c\}$

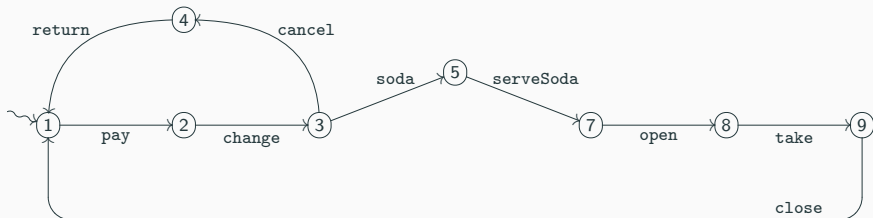


FTS of example SPL: a vending machine

Feature model:



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



Ambiguities in behavioural models

Part I

- Behavioural variability modelling and analysis
 - 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: examples and benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

Ambiguous FTS

dead transition

an FTS transition not reachable in any product (LTS)

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

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 \rightarrow unambiguous FTS

Transformation:

1. remove dead transitions

Ambiguous FTS \rightarrow unambiguous FTS

Transformation:

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

Ambiguous FTS \rightarrow unambiguous FTS

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 \rightarrow unambiguous FTS

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 $s_{\dagger} \notin S$

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 \rightarrow unambiguous FTS

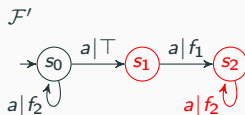
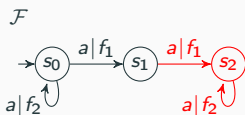
Transformation:

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

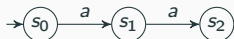
Step (3) must be performed only for the hidden deadlock states that have not yet become explicit deadlock states upon the removal of dead transitions in step (1)

Example transformations

Feature Model: $f_1 \oplus f_2$



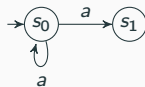
$\mathcal{F}|_{\lambda_1} = \mathcal{F}'|_{\lambda_1}$



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



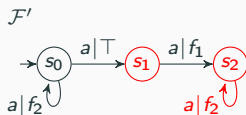
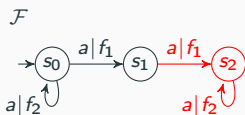
$\mathcal{F}'|_{\lambda_2}$



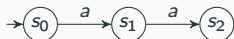
products $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$

Example transformations

Feature Model: $f_1 \oplus f_2$



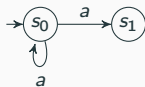
$\mathcal{F}|_{\lambda_1} = \mathcal{F}'|_{\lambda_1}$



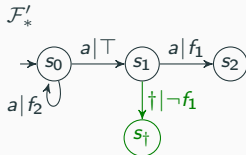
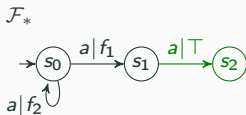
$\mathcal{F}|_{\lambda_2}$



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



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



Efficient static analysis of FTSs

Part I

- Behavioural variability modelling and analysis
 - 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: examples and benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

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

Satisfiability

We used a SAT tool!

- The **satisfiability problem** (SAT problem) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula
- A **SAT solver** is a program that takes as input a formula and that returns as output TRUE iff the formula is satisfiable (in the positive case a witness interpretation is returned)
- To discover the discussed ambiguities we use a SAT solver (we used **Z3** in experiments)

1. SAT solving is an **expensive task**, since each additional propositional-atom added to the considered formula doubles up the number of possible interpretations

1. SAT solving is an **expensive task**, since each additional propositional-atom added to the considered formula doubles up the number of possible interpretations
2. On the theoretical side, SAT problems fall in the family of **intractable** problems because of their exponential complexity; Indeed, NP-complete

SAT solving

1. SAT solving is an **expensive task**, since each additional propositional-atom added to the considered formula doubles up the number of possible interpretations
2. On the theoretical side, SAT problems fall in the family of **intractable** problems because of their exponential complexity; Indeed, NP-complete
3. Luckily, the use of SAT solvers for software verification is currently quite satisfying, and **successfully used in practice**

SAT solver technologies

- SAT solvers have **countless applications**, spanning from software verification to artificial intelligence, algorithms, hardware design, . . .
- Since 2002, **the International SAT Competition** is organised, which is a competitive affair for evaluating the progress in the state-of-the-art for SAT solvers and to highlight the development of SAT tools
- Most SAT solvers are based on developments of the original **DP and DPLL** algorithms but include heuristics, stochastic approaches . . . new ideas

M. Davis & H. Putnam, A Computing Procedure for Quantification Theory. *J. ACM*, 1960
M. Davis & G. Logemann & D. Loveland, A Machine Program for Theorem Proving. *C. ACM*, 1962

FTS representation

Our algorithm assumes that the considered FTS is represented by the **global data structure** `fts` that includes 4 fields:

1. `states` stores the set of all states in the FTS
2. `transitions` stores the set of all transitions in the FTS
3. `initial` stores the initial state of the FTS
4. `fm` stores the formula `fm`, which is a formula in $\mathbb{B}(F)$ that represents the feature model of the FTS

FTS representation (state)

Each `state` is represented by a data structure that includes 3 fields:

1. `in_trs` stores the set of incoming transitions of this state
2. `out_trs` stores the set of outgoing transitions of this state
3. `hdead` is a Boolean flag used to record whether this state is a hidden deadlock

FTS representation (transition)

Each **transition** is represented by a data structure including 4 fields:

1. **bx** stores the feature expression labelling the transition, i.e., a propositional formula in $\mathbb{B}(F)$
2. **source** stores the source state of the transition
3. **dead** is a Boolean flag used to record whether this transitions is dead
4. **false_opt** is a Boolean flag used to record whether this transitions is false optional

Notation

- Let T be the set of the names of the transitions of the FTS;
Recall that an interpretation for a propositional formula in $\mathbb{B}(F \cup S \cup T)$ is a function $\mathcal{I} : (F \cup S \cup T) \rightarrow \{\top, \perp\}$
- Let $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ be an FTS:
 - `fts.states` is used for S
 - `fts.transitions` is used for T
 - `fts.initial` is used for s_0
- Let `inner_states` denote the set `fts.states \ {fts.initial}`;
An *initial path* is a path that starts from the initial state

Useful formulas

- ϕ_{initial} is the formula `fts.initial`

Useful formulas

- ϕ_{initial} is the formula `fts.initial`
- ϕ_{inner} is $\bigwedge_{s \in \text{inner_states}} (s \Rightarrow \text{atLeastOneTransitionOf}(s.\text{in_trs}))$, where `atLeastOneTransitionOf(X)` is a placeholder for
$$\bigvee_{t \in X} (t.\text{bx} \wedge t \wedge t.\text{source})$$

Useful formulas

- ϕ_{initial} is the formula `fts.initial`
- ϕ_{inner} is $\bigwedge_{s \in \text{inner_states}} (s \Rightarrow \text{atLeastOneTransitionOf}(s.\text{in_trs}))$, where $\text{atLeastOneTransitionOf}(X)$ is a placeholder for
$$\bigvee_{t \in X} (t.\text{bx} \wedge t \wedge t.\text{source})$$
- ϕ_{single} is the formula $\bigwedge_{s \in \text{fts.states}} \text{atMostOneOf}(s.\text{out_trs})$, where $\text{atMostOneOf}(X)$ is a placeholder for
$$\bigwedge_{t \in X} t \Rightarrow (\bigwedge_{t \in X \setminus \{t\}} \neg t)$$

Useful formulas

- ϕ_{initial} is the formula `fts.initial`
- ϕ_{inner} is $\bigwedge_{s \in \text{inner_states}} (s \Rightarrow \text{atLeastOneTransitionOf}(s.\text{in_trs}))$, where $\text{atLeastOneTransitionOf}(X)$ is a placeholder for
$$\bigvee_{t \in X} (t.\text{bx} \wedge t \wedge t.\text{source})$$
- ϕ_{single} is the formula $\bigwedge_{s \in \text{fts.states}} \text{atMostOneOf}(s.\text{out_trs})$, where $\text{atMostOneOf}(X)$ is a placeholder for
$$\bigwedge_{t \in X} t \Rightarrow (\bigwedge_{t \in X \setminus \{t\}} \neg t)$$
- $\text{end}(s)$ is the formula $s \wedge (\bigwedge_{t \in s.\text{out_trs}} \neg t)$

Lemma

Let fts represent the FTS and let \mathcal{I} be an interpretation; Then:

1. $\mathcal{I} \models fts.fm$ iff $\lambda_{\mathcal{I}} \in \Lambda$
2. $\mathcal{I} \models \phi_{initial}$ iff $\mathcal{I}(fts.initial) = \top$
3. $\mathcal{I} \models \phi_{inner}$ iff, for all $s \in inner_states$, if $\mathcal{I}(s) = \top$ then there is at least a transition $t \in s.in_trs$ such that:
 $\mathcal{I} \models t.bx$, $\mathcal{I}(t) = \top$, and $\mathcal{I}(t.source) = \top$
4. $\mathcal{I} \models \phi_{single}$ iff, for all $s \in inner_states$, there is at most one transition $t \in s.out_trs$ such that $\mathcal{I}(t) = \top$
5. $\mathcal{I} \models end(s)$ iff, $\mathcal{I}(s) = \top$ and $\mathcal{I}(t) = \perp$, for all $t \in s.out_trs$, where $s \in fts.states$

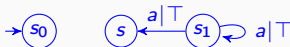
Formula `is_useful_state(s)`

`is_useful_state(s)` is the formula `fts.fm` \wedge ϕ_{initial} \wedge ϕ_{inner} \wedge ϕ_{single} \wedge `end(s)`

This formula is satisfiable (i.e., valid in some interpretation \mathcal{I}) iff in at least one LTS product there is a simple path (i.e., a path with no repeated states) that starts from the initial state and ends in s

Formula `is_useful_state(s)` (example)

Consider the FTS on the right:



It has no features and just one product configuration which yields the LTS consisting of the initial state s_0 ; Namely, `fts.fm = T`

The formulas `is_useful_state(s)` and `is_useful_state(s1)` are not satisfiable

To see this, let us call:

- t the transition from s_1 to s
- t_1 the transition from s_1 to s_1

Ambiguities encoding

We can straightforwardly define the formulas for checking the behavioural ambiguities by exploiting the formula `is_useful_state(s)`

- `exists_deadlock(s)` is the formula

$$\text{is_useful_state}(s) \wedge \bigwedge_{t \in s.\text{out_trs}} \neg t.\text{bx}$$

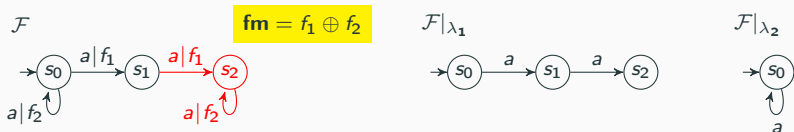
- `is_not_dead_transition(t)` is the formula

$$\text{is_useful_state}(t.\text{source}) \wedge t.\text{bx}$$

- `may_be_opt_transition(t)` is the formula

$$\text{is_useful_state}(t.\text{source}) \wedge \neg t.\text{bx}$$

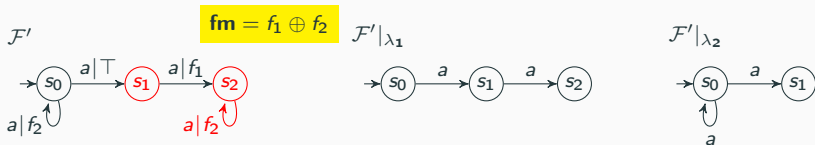
Ambiguities encoding (example 1)



Let t_0 , t_1 , t_2 , and t_3 be the transitions of \mathcal{F} from left to right:

- `is_not_dead_transition(t_3)` is not satisfiable (therefore t_3 is dead)
- `exists_deadlock(s_2)` is satisfiable (therefore state s_2 is a hidden deadlock)
- `is_not_dead_transition(t_2)` is satisfiable (therefore t_2 is not dead)
- `may_be_opt_transition(t_2)` is satisfiable (therefore t_2 is false optional)

Ambiguities encoding (example 2)



Let t_0 , t_1 , t_2 , and t_3 be the transitions of \mathcal{F}' from left to right:

- `is_not_dead_transition(t_3)` is not satisfiable (therefore t_3 is dead)
- both formulas `exists_deadlock(s_1)` and `exists_deadlock(s_2)` are satisfiable (therefore both s_1 and s_2 are hidden deadlocks)

Theorem (Correctness of the formulas for checking the behavioural ambiguities)

Let fts represent the FTS $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ and s be a state:

1. $exists_deadlock(s)$ is satisfiable iff there is a configuration $\lambda \in \Lambda$ s.t. the state s is a deadlock in $\mathcal{F}|_\lambda$

Theorem (Correctness of the formulas for checking the behavioural ambiguities)

Let f represent the FTS $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ and s be a state:

1. $exists_deadlock(s)$ is satisfiable iff there is a configuration $\lambda \in \Lambda$ s.t. the state s is a deadlock in $\mathcal{F}|_\lambda$
2. $is_not_dead_transition(t)$ is satisfiable iff there is $\lambda \in \Lambda$ s.t. the LTS transition corresponding to transition t is reachable in $\mathcal{F}|_\lambda$

Theorem (Correctness of the formulas for checking the behavioural ambiguities)

Let $f_t s$ represent the FTS $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ and s be a state:

1. $exists_deadlock(s)$ is satisfiable iff there is a configuration $\lambda \in \Lambda$ s.t. the state s is a deadlock in $\mathcal{F}|_\lambda$
2. $is_not_dead_transition(t)$ is satisfiable iff there is $\lambda \in \Lambda$ s.t. the LTS transition corresponding to transition t is reachable in $\mathcal{F}|_\lambda$
3. $may_be_opt_transition(t)$ is satisfiable iff there is a $\lambda \in \Lambda$ s.t. the state $t.source$ is reachable in $\mathcal{F}|_\lambda$ and the LTS transition corresponding to transition t is not reachable in $\mathcal{F}|_\lambda$

Ambiguities discovery algorithm

```
1 for s in fts.states:
2     if(s.out_trs =  $\emptyset$ ):
3         s.hdead  $\leftarrow$  False
4     else:
5         s.hdead  $\leftarrow$  check(exist_deadlock(s))
6
```

Ambiguities discovery algorithm

```
1  for s in fts.states:
2      if(s.out_trs =  $\emptyset$ ):
3          s.hdead  $\leftarrow$  False
4      else:
5          s.hdead  $\leftarrow$  check(exist_deadlock(s))
6
7  for s in fts.states:
8      for t in s.in_trs:
9          t.dead  $\leftarrow$  not check(is_not_dead_transition(t))
10         if(t.dead or t.bx =  $\top$ ):
11             t.false_opt  $\leftarrow$  False
12         else:
13             t.false_opt  $\leftarrow$  not check(may_be_opt_transition(t))
```

FTS ambiguity detection: NP-hard

For every propositional formula ϕ with variables in F , the FTS

$$(\{s_0, s\}, \{a\}, s_0, \{s_0 \xrightarrow{a|\phi} s\}, F, 2^F)$$

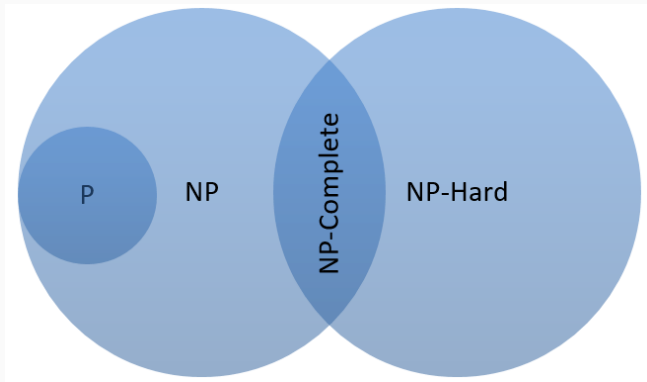
is such that its (unique) transition is:

- (i) dead iff ϕ is not satisfiable (i.e. $\neg\phi$ is valid)
- (ii) false optional iff $\neg\phi$ is not satisfiable (i.e. ϕ is valid)

Moreover, state s_0 is a hidden deadlock iff ϕ is not satisfiable

Thus, the FTS ambiguity detection problem is **NP-hard**

FTS ambiguity detection: NP-hard



NP-hard is formed by complexity problems that are informally **at least** as hard as the hardest problems in NP

FTS ambiguity detection: NP-complete

Consider this slight modification of our algorithm:

```
1 for s in fts.states:
2     if(s.out_trs =  $\emptyset$ ):
3         s.hdead_formula  $\leftarrow$  False
4     else:
5         s.hdead_formula  $\leftarrow$  exist_deadlock(s)
6
7 for s in fts.states:
8     for t in s.in_trs:
9         t.dead_formula  $\leftarrow$  is_not_dead_transition(s)
10        t.false_opt_formula  $\leftarrow$  may_be_opt_transition(s)
```

FTS ambiguity detection: NP-complete

Consider this slight modification of our algorithm:

```
1  for s in fts.states:
2      if(s.out_trs = ∅):
3          s.hdead_formula ← False
4      else:
5          s.hdead_formula ← exist_deadlock(s)
6
7  for s in fts.states:
8      for t in s.in_trs:
9          t.dead_formula ← is_not_dead_transition(s)
10         t.false_opt_formula ← may_be_opt_transition(s)
```

This algorithm reduces in polynomial time (in the size of `fts`) the ambiguity detection problem for \mathcal{F} to $n + 2 \times m$ SAT problem

Therefore, the FTS ambiguity detection problem is **NP-complete**

Implementation in Z3

- We implemented our algorithm in Z3
- We made our implementation publicly available, including all examples used in the rest of these slides
- Our original artefact received the following 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

Examples and Experiments

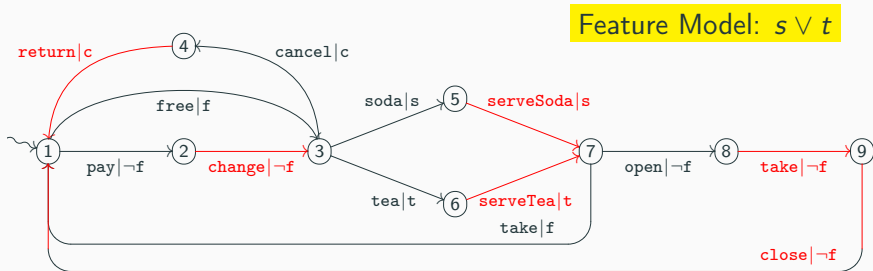
Part I

- Behavioural variability modelling and analysis
 - 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: examples and benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

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]

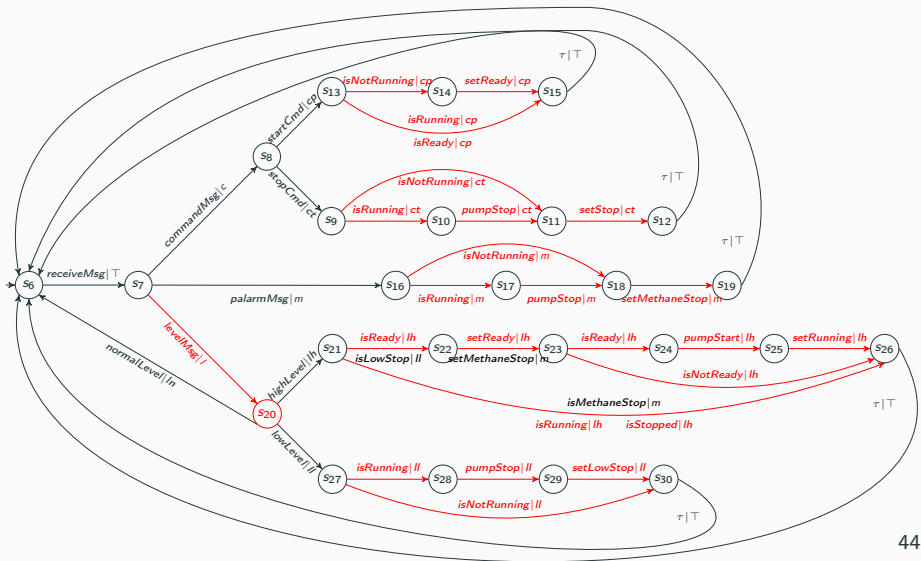
DEAD TRANSITIONS = []

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: $(c \leftrightarrow (ct \vee cp)) \wedge I$



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	
	S	$ \delta $	$ \Sigma $	live-ness	# dead transitions	# false optional transitions	# hidden deadlock states	run-time (s)	memory usage (Mb)
Vending machine <i>Classen, PhD thesis, 2011</i>	9	13	12	yes	0	6	0	0.26	29.765
Coffee machine <i>Asirelli et al. @ SPLC'11</i>	14	23	15	yes	0	14	0	0.29	30.305
Soup component <i>Belder et al. @ FMSPL'15</i>	13	28	18	yes	0	7	0	0.316	30.85
Mine pump (system) <i>Classen, PhD thesis, 2011</i>	25	41	22	no	0	25	1	0.344	31.704
Mine pump (controller) <i>Classen, PhD thesis, 2011</i>	77	104	22	no	0	59	4	0.548	36.295
Coffee/Soup machine <i>Belder et al. @ FMSPL'15</i>	182	691	33	yes	8	284	0	37.766	119.427
Mine pump (complete) <i>Classen, PhD thesis, 2011</i>	418	1,255	26	yes	0	308	0	98.994	119.127
Claroline <i>Devroey et al. @ VaMoS'14</i>	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	δ	Σ	runtime (s)	memory usage (Mb)	runtime (s)	memory usage (Mb)
Vending machine <i>Classen, PhD thesis, 2011</i>					9	13	12	0.92
Coffee machine <i>Asirelli et al. @ SPLC'11</i>	14	23	15	2.822	40.140	0.29	30.305	9.72x
Soup component <i>Belder et al. @ FMSPLE'15</i>	13	28	18	2.544	40.870	0.316	30.85	8.05x
Mine pump (system) <i>Classen, PhD thesis, 2011</i>	25	41	22	2.192	41.899	0.344	31.704	6.37x
Mine pump (controller) <i>Classen, PhD thesis, 2011</i>	77	104	22	8.12	49.091	0.548	36.295	14.82x
Coffee/Soup machine <i>Belder et al. @ FMSPLE'15</i>	182	691	33	timeout	–	37.766	119.427	>7200.00x
Mine pump (complete) <i>Classen, PhD thesis, 2011</i>	418	1,255	26	timeout	–	98.994	119.127	>7200.00x
Claroline <i>Devroey et al. @ VaMoS'14</i>	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)

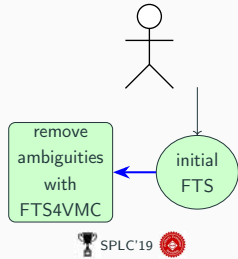
Benchmark experiments (3/3)

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

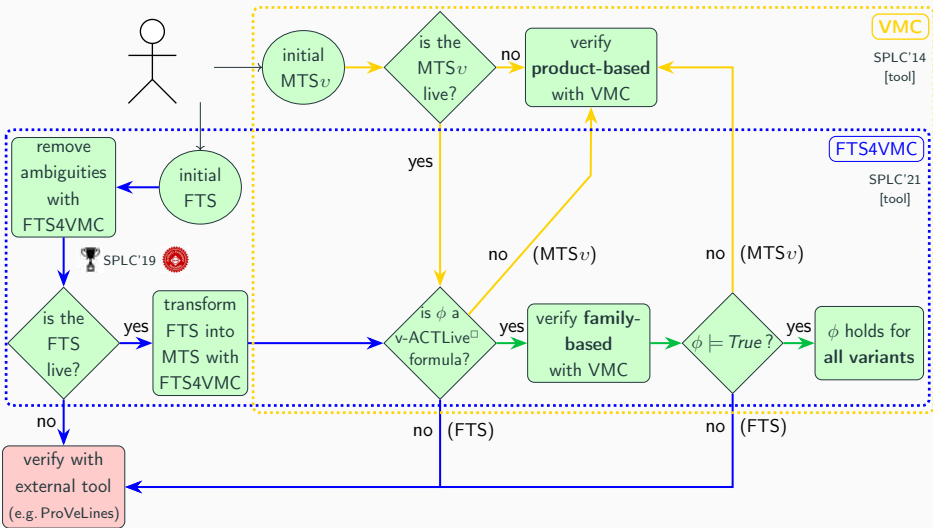
The experiments were performed on a virtual machine Gentoo 201905, CLI Version VirtualBox (VDI) 64 bit, with 2048 Mb of allocated memory on a Windows 10 Pro 64 bit with 16 Gb of RAM and CPU AMD Ryzen 7 1700X (8 core, 16 threads, 3.4 Ghz)

Wrap up of Part I

Detect and remove ambiguities



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



Part II

Part I

- Behavioural variability modelling and analysis
 - 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: benchmark experiments

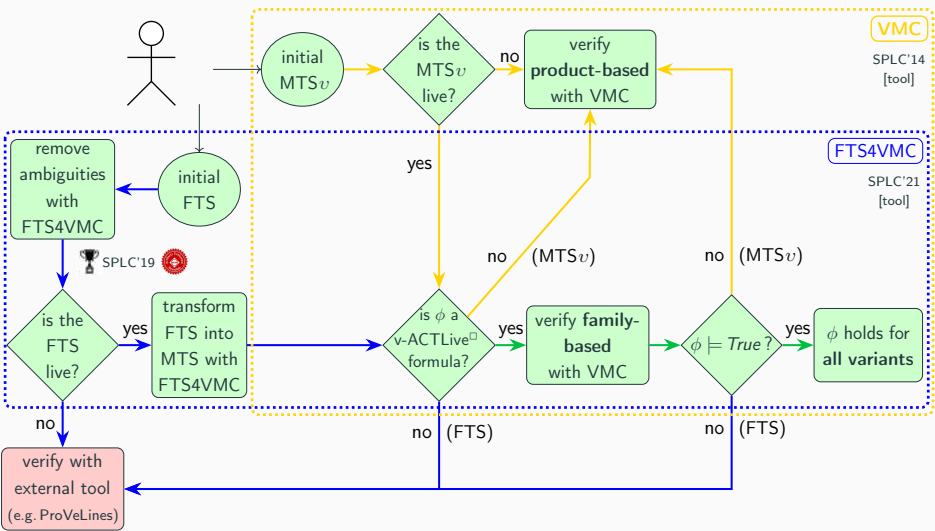
Part I

- Behavioural variability modelling and analysis
 - 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: benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

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



Modal Transition Systems (MTSs) with variability constraints (MTS v)

Part I

- Behavioural variability modelling and analysis
 - 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: benchmark experiments

Part II

- **Modal Transition System with variability constraints (MTS_v)**
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

Another behavioural variability model: MTSs for SPLE

Main ingredient: Modal Transition System (MTS)

- LTS recognising admissible (**may**) and necessary (**must**) transitions

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

- Shown to be 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

Another behavioural variability model: MTSs for SPLE

Main ingredient: Modal Transition System (MTS)

- LTS recognising admissible (**may**) and necessary (**must**) transitions

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

- Shown to be 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



MTS 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

Another behavioural variability model: MTSs for SPLE

Main ingredient: Modal Transition System (MTS)

- LTS recognising admissible (**may**) and necessary (**must**) transitions

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

- Shown to be 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



MTS 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 MTS 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

MTS_v: MTS with. . .

Recall:

An LTS $(Q, \Sigma, s_0, \rightarrow)$ with states, actions, initial state, and transitions

A **Modal Transition System with variability constraints** (MTS_v) is a sextuple $(S, \Sigma, s_0, \rightarrow_{\square}, \rightarrow_{\diamond}, \Upsilon)$ such that $(S, \Sigma, s_0, \rightarrow_{\square} \cup \rightarrow_{\diamond})$ is an LTS

MTS_v: MTS with. . .

Recall:

An LTS $(Q, \Sigma, s_0, \rightarrow)$ with states, actions, initial state, and transitions

A **Modal Transition System with variability constraints** (MTS_v) is a sextuple $(S, \Sigma, s_0, \rightarrow_{\square}, \rightarrow_{\diamond}, \Upsilon)$ such that $(S, \Sigma, s_0, \rightarrow_{\square} \cup \rightarrow_{\diamond})$ is an LTS

Formally, 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 $\rightarrow_{\square} \subseteq S \times \Sigma \times S$: **necessary** transitions

By definition, any necessary transition is also admissible: $\rightarrow_{\square} \subseteq \rightarrow_{\diamond}$

We denote the transitions $\dashrightarrow \equiv \rightarrow_{\diamond} \setminus \rightarrow_{\square}$ as **optional** transitions

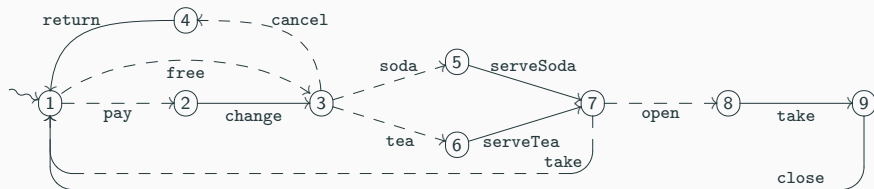
MTS_v: MTS with...

Recall:

An LTS $(Q, \Sigma, s_0, \rightarrow)$ with states, actions, initial state, and transitions

A **Modal Transition System with variability constraints** (MTS_v) is a sextuple $(S, \Sigma, s_0, \rightarrow_{\square}, \rightarrow_{\diamond}, \Upsilon)$ such that $(S, \Sigma, s_0, \rightarrow_{\square} \cup \rightarrow_{\diamond})$ is an LTS

Practically:



plus an additional set of variability constraints Υ , defined next

Variability constraints Υ of form **ALT**ernative, **EXC**ludes, **REQ**uires, ...

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

b_1 **OR** \dots **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, \dots, b_n holds, i.e.
 $b_i = a_i$ is reachable or $b_i = \neg a_i$ is not reachable (in \mathcal{L})

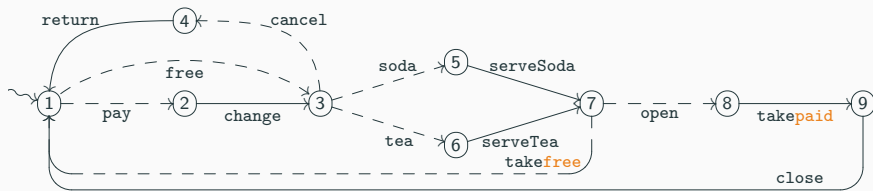
a_1 **EXC** a_2 : at most one of the actions a_1 and a_2 is reachable (in \mathcal{L})

a_1 **REQ** a_2 : action a_2 is reachable whenever a_1 is reachable (in \mathcal{L})

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

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

VMC accepts a_j **IFF** a_k as shorthand for $(a_j$ **REQ** $a_k) \wedge (a_k$ **REQ** $a_j)$



S1 = pay(may).S2 + free(may).S3

S2 = change(must).S3

S3 = cancel(may).S4 + soda(may).S5 + tea(may).S6

S4 = return(must).S1

S5 = serveSoda(must).S7

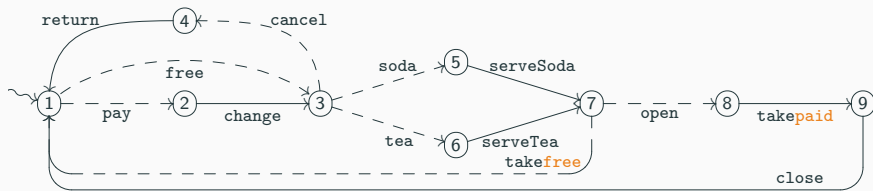
S6 = serveTea(must).S7

S7 = takefree(may).S1 + open(may).S8

S8 = takepaid(must).S9

S9 = close(must).S1

net SYS = S1 // initial state



...

```
net SYS = S1          // initial state
```

```
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
}
```

Intuitively, a **product LTS** is obtained from a family MTS_v as follows:

1. include **all** (reachable) must transitions
 2. include **subset** of (reachable) optional transitions, remove rest
 3. satisfy assumptions of **coherence** and **consistency**
 4. satisfy **variability constraints**
- ⇒ Each selection gives rise to a different variant

Intuitively, a **product LTS** is obtained from a family MTS_v as follows:

1. include **all** (reachable) must transitions
 2. include **subset** of (reachable) optional transitions, remove rest
 3. satisfy assumptions of **coherence** and **consistency**
 4. satisfy **variability constraints**
- ⇒ Each selection gives rise to a different variant

Formally, let $(S, \Sigma, s_0, \delta^\diamond, \delta^\square, \Upsilon)$ be a coherent MTS_v , i.e. $\exists \xrightarrow{a} \implies \nexists \xrightarrow{a}$

Intuitively, a **product LTS** is obtained from a family MTS_v as follows:

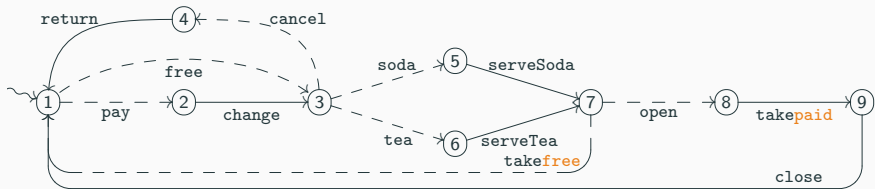
1. include **all** (reachable) must transitions
 2. include **subset** of (reachable) optional transitions, remove rest
 3. satisfy assumptions of **coherence** and **consistency**
 4. satisfy **variability constraints**
- ⇒ Each selection gives rise to a different variant

Formally, let $(S, \Sigma, s_0, \delta^\diamond, \delta^\square, \Upsilon)$ be a coherent MTS_v , i.e. $\exists \xrightarrow{a} \implies \nexists \xrightarrow{a}$

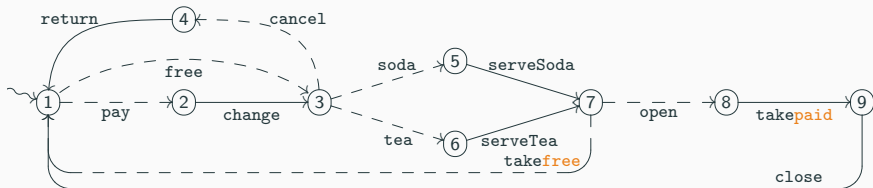
Then the set $\{\mathcal{P}_i = (S_i, \Sigma, s_0, \delta_i) \mid i > 0\}$ of **product LTSs** is obtained by considering each pair of $S_i \subseteq S$ and $\delta_i \subseteq \delta^\diamond \cup \delta^\square$ to be defined s.t.

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

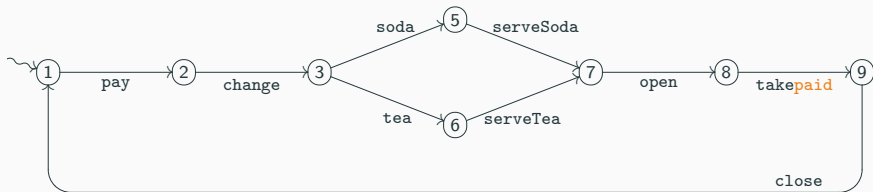
Product LTSs of MTS_v of example SPL



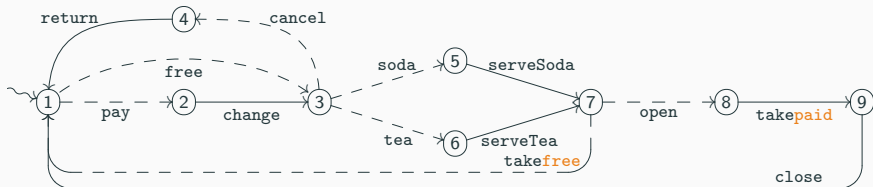
Product LTSs of MTS_v of example SPL



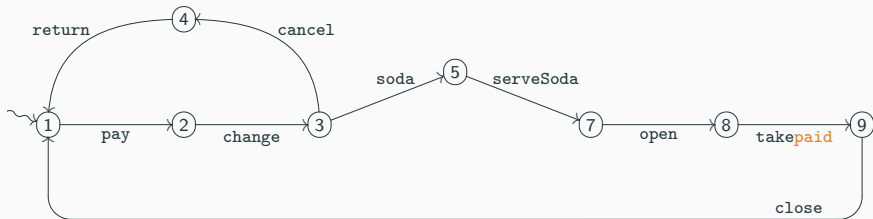
⇓ {pay,soda,tea,open}



Product LTSs of MTS_v of example SPL



⇓ {pay, cancel, soda, open}



MTS_vs are equally expressive as FTSs

Theorem (FTS2MTS_v transformation sound and complete)

Let \mathcal{F} be an FTS and let \mathcal{M} be the MTS_v generated from \mathcal{F} according to the FTS2MTS_v model transformation algorithm.

Then the sets of derived variants $\text{lts}(\mathcal{F})$ and $\text{lts}(\mathcal{M})$ coincide, up to dummy transitions and action relabelling.

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

MTS_vs are equally expressive as FTSs

Theorem (FTS2MTS_v transformation sound and complete)

Let \mathcal{F} be an FTS and let \mathcal{M} be the MTS_v generated from \mathcal{F} according to the FTS2MTS_v model transformation algorithm.

Then the sets of derived variants $\text{lts}(\mathcal{F})$ and $\text{lts}(\mathcal{M})$ coincide, up to dummy transitions and action relabelling.

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

Theorem (MTS_v2FTS transformation sound and complete)

Let \mathcal{M} be an MTS_v and let \mathcal{F} be the FTS generated from \mathcal{M} according to the MTS_v2FTS model transformation algorithm.

Then $\text{lts}(\mathcal{M}) = \text{lts}(\mathcal{F})$.

Model checking principles

Part I

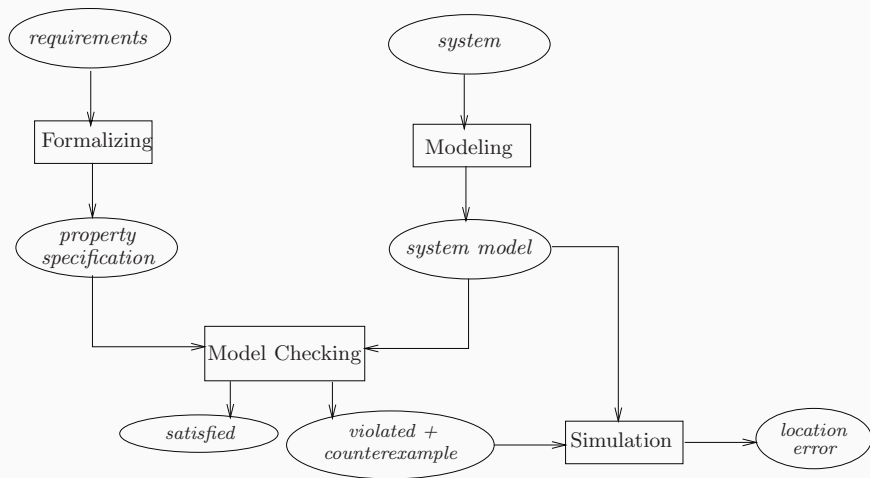
- Behavioural variability modelling and analysis
 - Product- vs family-based analysis
 - Featured Transition System (FTS)
 - Ambiguities: dead/false optional transitions, hidden deadlocks
- Automated static analysis of FTSs
 - Detecting ambiguities: criteria, SAT solving, implementation
 - Disambiguating FTSs: benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- **Model checking principles**, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

What is model checking?

(taken from Baier & Katoen, *Principles of Model Checking*, MIT, 2008)



Model checking process

(taken from Baier & Katoen, *Principles of Model Checking*, MIT, 2008)

Modelling phase:

- model the system under consideration using the model description language of the model checker at hand
- as a first sanity check and quick assessment of the model perform some simulations
- formalise the property to be checked using the property specification language

Running phase:

- run the model checker to check the validity of the property in the system model

Analysis phase:

- property satisfied? → check next property (if any)
- property violated? →
 1. analyse generated counterexample by simulation
 2. refine the model, design or property
 3. repeat the entire procedure
- out of memory? → try to reduce the model and try again

Strengths and weaknesses

(taken from Baier & Katoen, *Principles of Model Checking*, MIT, 2008)

- **General** verification approach, applicable to a wide range of applications (e.g., embedded systems, software engineering, hardware design)
- Supports **partial** verification, i.e. properties can be checked individually, allowing to focus on essential properties first (no complete requirement specification needed)
- Not vulnerable to likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects
- Provides **diagnostic info** in case a property is invalidated: very useful for debugging
- **Potentially a “push-button” technology**, requiring neither a high degree of user interaction nor a high degree of expertise
- Enjoys a rapidly increasing **interest by industry**: several companies have started their in-house verification labs (e.g., Amazon), frequent job offers with required skills in model checking, and commercial model checkers have become available
- Easy **integration** in existing development cycles: its learning curve is not very steep, empirical studies indicate that it may lead to shorter development times
- **Sound and mathematical underpinning**, based on theory of graph algorithms, data structures and logic

Strengths and weaknesses

(taken from Baier & Katoen, *Principles of Model Checking*, MIT, 2008)

- General verification approach, applicable to a wide range of applications (e.g., embedded systems, software engineering, hardware design)
- Supports partial verification, i.e. properties can be checked individually, allowing to focus on essential properties first (no complete requirement specification needed)
- Not vulnerable to likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects
- Provides diagnostic info in case a property is invalidated: very useful for debugging
- **Potentially a “push-button” technology**, requiring neither a high degree of user interaction nor a high degree of expertise
- Enjoys a rapidly increasing interest by industry: several companies have started their in-house verification labs (e.g., Amazon), frequent job offers with required skills in model checking, and commercial model checkers have become available
- Easy integration in existing development cycles: its learning curve is not very steep, empirical studies indicate that it may lead to shorter development times
- Sound and mathematical underpinning, based on theory of graph algorithms, data structures and logic

Strengths and weaknesses

(taken from Baier & Katoen, *Principles of Model Checking*, MIT, 2008)

- Mainly appropriate to **control-intensive** applications and less suited for data-intensive applications as data typically ranges over infinite domains
- Applicability subject to **decidability** issues: for infinite-state systems or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable
- Verifies a **system model**, and not the actual system (product or prototype) itself: **any obtained result is thus as good as the system model** (need complementary techniques, like testing, to find fabrication faults (for HW) or coding errors (for SW))
- Checks only **stated requirements**, i.e. **no guarantee of completeness**, since the validity of properties that are not checked cannot be judged
- Suffers from **state space explosion** problem: number of states needed to model the system accurately may easily exceed the amount of available computer memory (despite the development of several very effective methods to combat this problem, models of realistic systems may still be too large to fit in memory)
- Its usage requires some **expertise** in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used
- Correct results not guaranteed: like any tool, it may contain **software defects**
- Does not allow checking **generalizations**: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated

Strengths and weaknesses

(taken from Baier & Katoen, *Principles of Model Checking*, MIT, 2008)

- Mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains
- Applicability subject to decidability issues: for infinite-state systems or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable
- Verifies a **system model**, and not the actual system (product or prototype) itself: **any obtained result is thus as good as the system model** (need complementary techniques, like testing, to find fabrication faults (for HW) or coding errors (for SW))
- Checks only **stated requirements**, i.e. **no guarantee of completeness**, since the validity of properties that are not checked cannot be judged
- Suffers from state space explosion problem: number of states needed to model the system accurately may easily exceed the amount of available computer memory (despite the development of several very effective methods to combat this problem, models of realistic systems may still be too large to fit in memory)
- Its usage requires some expertise in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used
- Correct results not guaranteed: like any tool, it may contain software defects
- Does not allow checking generalizations: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated

Temporal logic: LTL, (A)CTL, v-CTL

Part I

- Behavioural variability modelling and analysis
 - 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: benchmark experiments

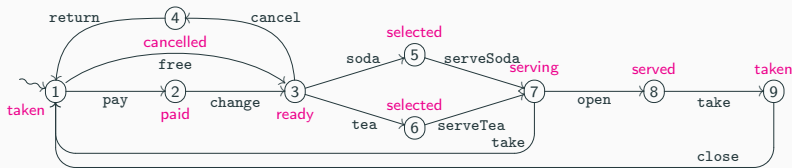
Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

Doubly-Labelled Transition System (L^2TS) [recall]

A **Doubly-Labelled Transition System** (L^2TS) is a sextuple $(S, \Sigma, s_0, \delta, AP, L)$ in which (S, Σ, s_0, δ) is an LTS, AP is a set of **atomic propositions**, and $L: S \rightarrow 2^{AP}$ is a function labelling each state with a set of AP (here singleton)

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



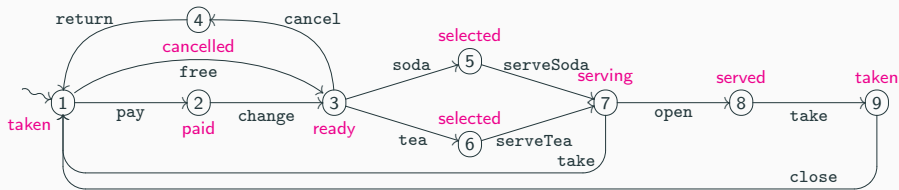
AP paid ready ... served taken

L (2) (3) ... (8) (9)
 paid ready served taken

Kripke structure: only state labelling, no transition labelling

Temporal logic

We have seen that possible system evolutions can be described by a directed graph (LTS, L^2TS , FTS, ...), where nodes and edges may be labelled with properties and events



We can reason on the properties of such graphs by evaluating, starting from the initial state, temporal logic formulas

Temporal logic operators

Temporal logic formulas are formed by composing temporal operators, the most classical of which are of the form:

- ALWAYS** $G \phi$, meaning ϕ is Globally true from now on
- EVENTUALLY** $F \phi$, meaning ϕ is eventually true in the Future
- NEXT** $X \phi$, meaning ϕ is true in the neXt state
- UNTIL** $\phi_1 U \phi_2$, meaning ϕ_2 is eventually true, and
Until that point ϕ_1 is always true

Temporal logic operators

Temporal logic formulas are formed by composing temporal operators, the most classical of which are of the form:

ALWAYS	$G \phi$, meaning ϕ is Globally true from now on
EVENTUALLY	$F \phi$, meaning ϕ is eventually true in the Future
NEXT	$X \phi$, meaning ϕ is true in the neXt state
UNTIL	$\phi_1 U \phi_2$, meaning ϕ_2 is eventually true, and Until that point ϕ_1 is always true

Depending on the formal verification framework, formulas are evaluated:

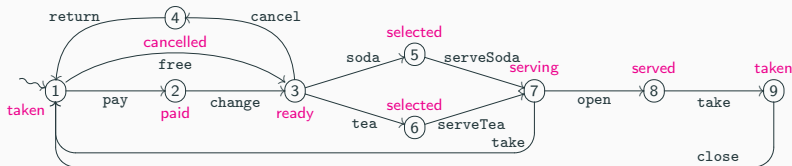
- Independently, on all elements of the set of possible execution sequences, without considering the branching structure of the system evolutions
- Or considering the actual branching structure of the system evolutions

And these operators can refer to just the names of events, to just the properties of states, or to both

Temporal logic operators: linear-time examples

Linear Temporal Logic (LTL)

Pnueli, The temporal logic of programs @ FOCS'77

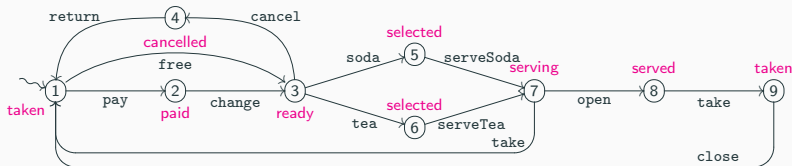


$G (\text{servng} \Rightarrow (F \text{ taken}))$: for all paths, it is always (globally) true that a state satisfying **servng** is eventually followed by a (future) state satisfying **taken**

Temporal logic operators: linear-time examples

Linear Temporal Logic (LTL)

Pnueli, The temporal logic of programs @ FOCS'77



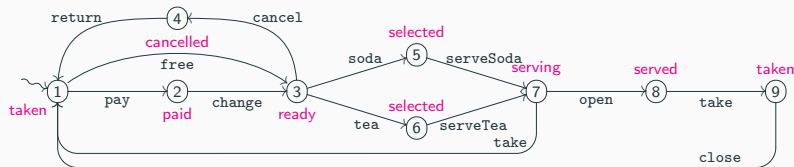
$G (\text{serving} \Rightarrow (F \text{ taken}))$: for all paths, it is always (globally) true that a state satisfying **servicing** is eventually followed by a (future) state satisfying **taken**

$(F \text{ taken}) \Rightarrow (F \text{ servicing})$: for all paths, if the path eventually contains a (future) state satisfying **taken**, then it must contain also a (future) state satisfying **servicing**

Temporal logic operators: branching-time examples

Computation Tree Logic (CTL)

Clarke & Emerson, Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic @ Logic of Programs Workshop'81

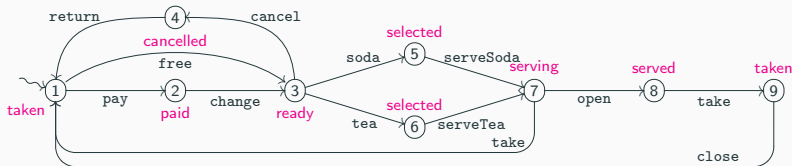


AG EF taken: for all (A) paths it is globally (G) true that there exists (E) a possibility to eventually (F) reach a (future) state satisfying **taken**

Temporal logic operators: branching-time examples

Action-based Computation Tree Logic (ACTL)

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



$AG EF \text{ taken}$: for all (A) paths it is globally (G) true that there exists (E) a possibility to eventually (F) reach a (future) state satisfying **taken**

$AF_{\text{take} \vee \text{cancel}}$: all (A) paths eventually (F) lead to a (future) transition for which **take** or **cancel** holds

$[\text{pay}] EX_{\text{change}}$: if **pay** holds for one of the next transitions, then after it, there must exist (E) a next (X) transition for which **change** holds

$[\text{act}] \phi$ is a shorthand for $\neg EX_{\text{act}} \neg \phi$

v-ACTL: variability-aware ACTL

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

ACTL + dedicated **variability-aware** versions of temporal operators, e.g.:

$F^{\square} \phi$, meaning ϕ is eventually true in the Future, and until that point, all transitions are **must** transitions

$F_{\chi}^{\square} \phi$, meaning the same as $F^{\square} \phi$, but that point is reached by executing an action satisfying χ

$X^{\square} \phi$, meaning ϕ is true in the neXt state, reached by a **must** transition

$X_{\chi}^{\square} \phi$, meaning the same as $X^{\square} \phi$, but that neXt state is reached by a **must** transition labelled with an action satisfying χ

v-ACTL: variability-aware ACTL

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

ACTL + dedicated **variability-aware** versions of temporal operators, e.g.:

$F^{\square} \phi$, meaning ϕ is eventually true in the Future, and until that point, all transitions are **must** transitions

$F_{\chi}^{\square} \phi$, meaning the same as $F^{\square} \phi$, but that point is reached by executing an action satisfying χ

$X^{\square} \phi$, meaning ϕ is true in the neXt state, reached by a **must** transition

$X_{\chi}^{\square} \phi$, meaning the same as $X^{\square} \phi$, but that neXt state is reached by a **must** transition labelled with an action satisfying χ

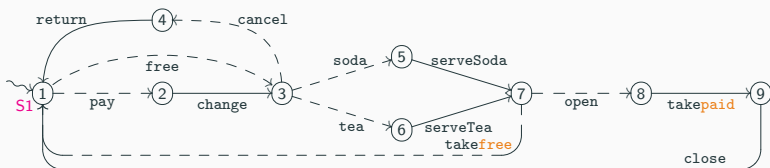
v-ACTLive[□] is a specific subset of v-ACTL with the following property: any formula that is true for a **live** MTS v , is also true for all product LTSs

1. Universal (A) properties that hold for all execution paths (thus also for the subset of MTS_v paths corresponding to a product)
2. Properties that hold for **must** paths (thus present in all products)
3. Negations of existential (E) properties that do not hold on any path

v-ACTLive[□]: family-based model checking with VMC

1. Universal (A) properties that hold for all execution paths (thus also for the subset of MTS_v paths corresponding to a product)
2. Properties that hold for **must** paths (thus present in all products)
3. Negations of existential (E) properties that do not hold on any path

Example v-ACTLive[□] formulas of properties that can thus be verified in a family-based manner with VMC (with a **linear complexity**)



1. $AG AF_{takefree \vee takepaid \vee cancel} \top$
2. $[pay] EX_{change}^{\square}$ (also $[pay] AX_{change}$, but not $[pay] EX_{change}$ seen before)
3. $AG [tea] \neg E [cancel U S1]$

VMC

Part I

- Behavioural variability modelling and analysis
 - 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: benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

Dedicated Variability Model Checker VMC

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

- VMC is not an industry-ready production/verification tool
- VMC is an academic prototype for research and education

Dedicated Variability Model Checker VMC

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

- VMC is not an industry-ready production/verification tool
- VMC is an academic prototype for research and education

VMC:

- Part of the KandISTI framework, developed by the FMT lab at ISTI-CNR (<https://fmt.isti.cnr.it/kandisti>)
- Freely usable online (<https://fmt.isti.cnr.it/vmc>)
- Command line based executables freely downloadable (Linux, Mac, Windows)
- Available as a local web server for MacOS

Variability analysis strategies supported by VMC

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

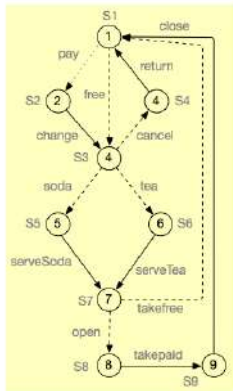
VMC offers product- and family-based model checking of MTS_{vs} :

- 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 the valid products of an MTS family ... and to separately verify properties on them (product-based)

Textual model encoding as process definitions [recall]



```
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
```



Graphical model displayed as MTS_v

VMC V6.5

(2019)



Edit Model

View Current Model

Explore the MTS

Draw Family MTS

Modelcheck MTS ...

Generate Products

Welcome

Quit

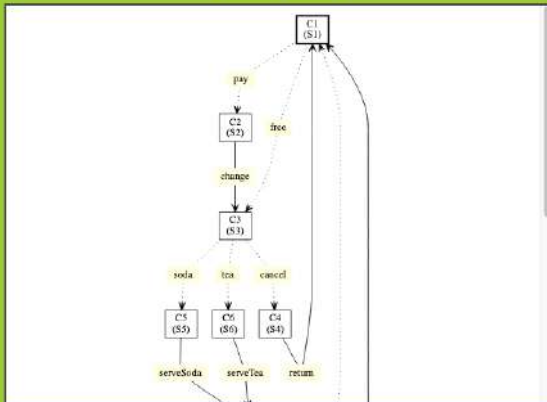


Kandinsky 1919

Abstract Model Evolutions Chart

Zoom Out

Zoom In

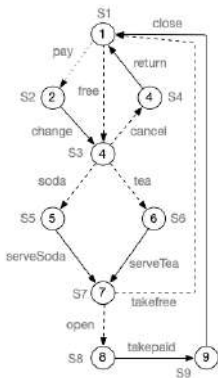


Click over a node to view the details of the corresponding system configuration.
View the graph in [DOT](#) format or as a [PDF](#) pdf picture or as plain [SVG](#) data.

Family-based model checking with VMC

Which properties on the MTS_v are preserved for all LTS products of the family?

Those holding for **all** possible executions, and making use of a **live** fragment of the MTS_v (recall: live implies no hidden deadlocks)



- all executions execute a loop that returns in the initial state

$AG AF S1$

- each execution loop contains either a serveSoda, a serveTea or a cancel action

$AF (\text{serveSoda or serveTea or cancel})$

- each pay action is always eventually followed by a takepaid action

$AG [\text{pay}] AF [\text{takepaid}]$

FALSE for the MTS, and actually FALSE for some products

- for each cycle, if an action free is performed, no action takepaid can be performed

$[\text{free}] A [(\text{not takepaid}) U S1]$

TRUE for all products, but FALSE for the MTS

Vending Machine: confirmation of validity

VMC notifies whenever preservation of an analysis result applies

The screenshot displays the VMC V6.5 (2019) interface. On the left is a green sidebar with navigation buttons: Edit Model, View Current Model, Explore the MTS, Draw Family MTS, Generate Products, Welcome, and Quit. Below these is a painting by Kandinsky from 1908. The main area shows the formula $AF \{serveSoda \text{ or } (serveTea \text{ or } cancel)\} \text{ true}$ is TRUE. A green box highlights the result: "The formula holds for ALL the MTS variants". Below this, it states "(evaluation time= 0.062 sec.)". Further down, it provides performance metrics: "(total states generated= 8, computations fragments generated= 12, total evaluation time= 0.062 sec.)". At the bottom, the formula $AF \{serveSoda \text{ or } serveTea \text{ or } cancel\}$ is shown in a white box, with buttons for "Check The Formula" and "Explain the Result".

VMC V6.5
(2019)

Edit Model

View Current Model

Explore the MTS

Draw Family MTS

Generate Products

Welcome

Quit

Kandinsky 1908

The Formula: $AF \{serveSoda \text{ or } (serveTea \text{ or } cancel)\} \text{ true}$
is TRUE

The formula holds for ALL the MTS variants

(evaluation time= 0.062 sec.)

(total states generated= 8, computations fragments generated= 12, total evaluation time= 0.062 sec.)

ACTL-UCTL-SocL-vACTL

$AF \{serveSoda \text{ or } serveTea \text{ or } cancel\}$

Check The Formula

Explain the Result

Vending Machine: explanation of results

VMC explanation for the formula $AG [pay] AF_{takepaid} \top$

VMC V6.5
(2019)

● ● ● ● ●

Edit Model

View Current Model


Explore the MTS

Draw Family MTS

Generate Products

Welcome

Quit



Kindinky 1908

☒ **The formula:**
 $AG [pay] AF \{takepaid\} true$
is **FOUND_FALSE** in State C1

This happens because the subformula:
 $[pay] AF \{takepaid\} true$
is **already Not Satisfied** in State C1

☒ **The formula:**
 $[pay] AF \{takepaid\} true$
is **FOUND_FALSE** in State C1

This happens because
C1 \rightarrow C2 {may, pay}
the transition label satisfies the action expression pay
but in State C2 the subformula:
 $AF \{takepaid\} true$ **Is Not Satisfied**.

☒ **The formula:**
 $AF \{takepaid\} true$
is **FOUND_FALSE** in State C2

This happens because there is at least one maximal path from C2 in which all transitions DO NOT satisfy the action expression takepaid.

For example:

- C2 \rightarrow C3 (change)
- C3 \rightarrow C4 (cancel, may)
- C4 \rightarrow C1 (return)
- C1 \rightarrow C2 (may, pay) (C2 closes a loop)

is one of the above mentioned failing paths.


Vending Machine: product-based model checking

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 "[free] A[not takepaid] U S1" on all family products

product+cancel+free+soda+takefree+tea_12	Formula evaluates	TRUE
product+cancel+free+soda+takefree_08	Formula evaluates	TRUE
product+cancel+free+takefree+tea_10	Formula evaluates	TRUE
product+cancel+open+pay+soda+tea_06	Formula evaluates	TRUE
product+cancel+open+pay+soda_02	Formula evaluates	TRUE
product+cancel+open+pay+tea_04	Formula evaluates	TRUE
product+free+soda+takefree+tea_11	Formula evaluates	TRUE
product+free+soda+takefree_07	Formula evaluates	TRUE
product+free+takefree+tea_09	Formula evaluates	TRUE
product+open+pay+soda+tea_05	Formula evaluates	TRUE
product+open+pay+soda_01	Formula evaluates	TRUE
product+open+pay+tea_03	Formula evaluates	TRUE

Logic Formula for all Products

[free] A[not takepaid] U S1

Check The Formula Explain the Result

Family-based model checking FTSs

Part I

- Behavioural variability modelling and analysis
 - 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: benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- FTS4VMC toolchain [demo]

Family-based model checking (1/2): v -ACTLive[□]

Note

1. Any FTS \mathcal{F} can trivially be transformed into an MTS \mathcal{F}_{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): v-ACTLive[□]

Note

1. Any FTS \mathcal{F} can trivially be transformed into an MTS \mathcal{F}_{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 MTS_vs to unambiguous FTSs:

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

Family-based model checking (1/2): v -ACTLive $^{\square}$

Note

1. Any FTS \mathcal{F} can trivially be transformed into an MTS \mathcal{F}_{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 MTS $_v$ s to unambiguous FTSs:

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

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

FTS4VMC: front-end for VMC

Part I

- Behavioural variability modelling and analysis
 - 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: benchmark experiments

Part II

- Modal Transition System with variability constraints (MTS_v)
- Model checking principles, temporal logic: LTL, (A)CTL, v-CTL
- VMC: family-based model checking of MTSs [demo]
- Family-based model checking of FTSs
- **FTS4VMC toolchain [demo]**

What is it?

A research tool developed in Python to model check properties on FTSs expressed in v-ACTLive[□] in a family-based manner with VMC

What is it?

A research tool developed in Python to model check properties on FTSs expressed in v-ACTLive[□] in a family-based manner with VMC

Where I can get it?

Source code is available on [GitHub](#)



Preconfigured Docker image on [DockerHub](#)



What it does?

What it does?

1. Provide a simple UI to work with the proposed toolchain

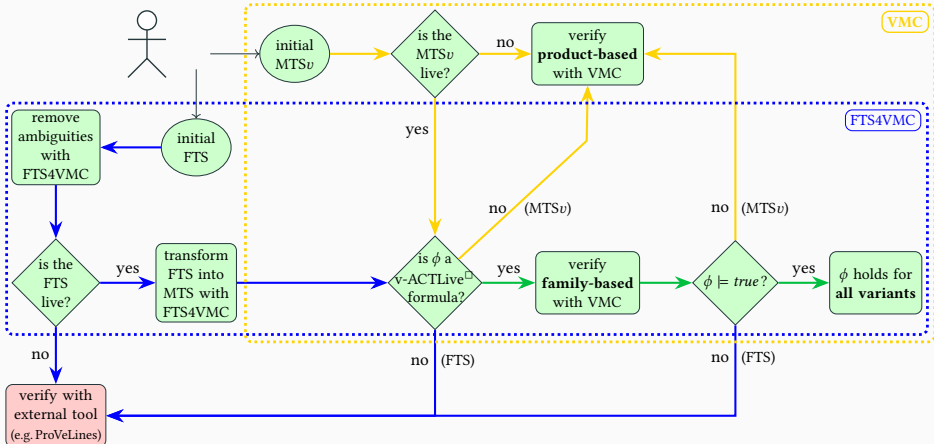
What it does?

1. Provide a simple UI to work with the proposed toolchain
2. Implements automatic ambiguities removal using the previously shown procedure

What it does?

1. Provide a simple UI to work with the proposed toolchain
2. Implements automatic ambiguities removal using the previously shown procedure
3. Convert unambiguous FTSs into MTSs compatible with VMC

Toolchain



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

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

FTS4VMC: static analyser integration (2/2)

Ambiguities found are stored into dictionaries

For ambiguities inside a transition:
{src, dst, label, constraint}

For ambiguities about a state: {id}

This conversion process enables to easily create JSON objects that can be manipulated using JavaScript

Upon analysis, FTS4VMC returns a JSON object called ambiguities that contains 3 arrays, one for each type of ambiguity



The screenshot displays the FTS4VMC static analyser interface. On the left is a vertical navigation menu with buttons for: Search FTS states, Remove result, Download clustered result, Find ambiguities analysis, Cleaners analysis, Remove all ambiguities, Remove FTS4VMC system transitions, Remove dead transitions and hidden deadlock states, Hide transitions, Hide model transition system, and Refresh properties. The main content area is titled "Analysis of test_paper.dot" and shows the following information:

- FTS
- Buttons: Canvase, Source, Graph, Summary, Customizable graph
- Number of states: 3
- Number of transitions: 4
- Ambiguities found**
- Hidden deadlock states**
 - C1
 - C2
- Dead transitions**
- Table with columns: Source state, Destination state, Label, Feature expression
- Buttons: Show Ambiguities

FTS4VMC: VMC integration (1/5)

VMC is used inside FTS4VMC
to verify properties

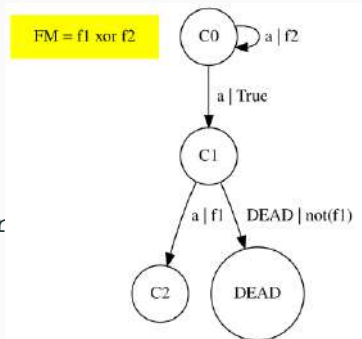
Since VMC is developed in
Ada, the current
implementation spawns a
sub-process that executes VMC
on an MTS model and property
stored into a file and then
parses the results to extract
the elements used inside the UI

The screenshot displays the FTS4VMC interface. On the left is a vertical sidebar with green buttons for various actions: 'Select FTS Model', 'Unload model', 'Save model', 'Refresh displayed model', 'Full pathfinder analysis', 'Current analysis', 'Stop processing', 'Remove all properties', 'Remove false optional transitions', 'Remove dead transitions and false states', 'Apply transformation', 'View featured transition system', and 'Verify property'. Below these buttons is a text area containing 'AF (name) AF (res)' and a 'Show explanation' button. The main panel on the right is titled 'Analysis of vendingnew.dot' and 'MTS'. It includes navigation links for 'Console', 'Source', 'Graph', 'Summary', and 'Counterexample graph'. Under the 'Results' section, it states: 'The formula AF (open) AF (close) is TRUE and holds for all the MTS models'.

FTS4VMC: VMC integration (2/5)

VMC requires an MTS to work on, thus FTS4VMC converts in the background FTSs into MTSs

```
digraph TEST {  
  FM="f1 xor f2";  
  name="TEST";  
  rankdir=TB;  
  node [shape=circle];  
  FeatureModel [label="FM = f1 xor f1 xor f2"];  
  C0 [initial=True];  
  C0 -> C0 [label="a | f2"];  
  C0 -> C1 [label="a | True"];  
  C1 -> C2 [label="a | f1"];  
  C1 -> DEAD [label="DEAD | not(f1)"];  
}
```



FTS4VMC: VMC integration (3/5)

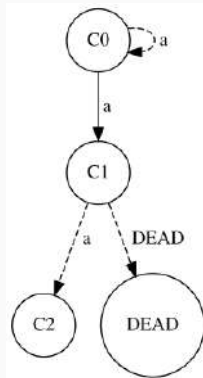
$C0 = a(\text{may}).C0 + a(\text{must}).C1$

$C1 = a(\text{may}).\text{nil} + \text{DEAD}(\text{may}).\text{nil}$

$\text{SYS} = C0$

Constraints { LIVE }

Both of C1's outgoing transitions reach nil since the states C2 and DEAD have no outgoing transitions



FTS4VMC: VMC integration (4/5)

After converting the FTS into an MTS, FTS4VMC executes the following command:

```
vmc mts-model -z prop.txt
```

where `prop.txt` is a file that contains a property expressed in v-ACTLive[□] and `mts-model` is a file that contains an MTS in the previously shown format

The screenshot displays the FTS4VMC web interface. On the left is a vertical sidebar with green buttons for various actions: 'Select FTS Model', 'Upload model', 'Delete model', 'Refresh displayed model', 'Full optimization analysis', 'Current analysis', 'Stop processing', 'Remove all optimizations', 'Remove false optional transitions', 'Remove dead transitions and hidden final states', 'Apply transformations', 'View featured transition systems', and 'Verify property'. Below these buttons is a text input field labeled 'AP name: AP (frees)' and a 'Show explanation' button.

The main content area is titled 'Analysis of vendingnew.dot' and 'MTS'. It includes navigation links for 'Console', 'Source', 'Graph', 'Summary', and 'Counterexample graph'. The 'Results' section shows a green message: 'The formula: AP (open) AP (close) is TRUE and holds for all the MTS models'.

FTS4VMC: VMC integration (5/5)

FTS4VMC parses VMC output and it can render the counterexample graph for false properties

Analysis of vendingnew.dot

MTS

Console Source Graph Summary Counterexample graph

Results

The formula: $\text{AG}(\text{pay}) \wedge \text{AF}(\text{correct})$ is **FALSE**

Even if the formula is FALSE for the MTS, its validity is not necessarily preserved by the MTS actions

Buttons: Select PTS Model, Edit model, Delete model, Download displayed model, Full antiequiv. analysis, Linear analysis, Stop processing, Remove all annotations, Remove false optional transitions, Remove dead transitions and false deadlock states, Apply transformations, View featured transition system, Verify property, Show explanation

Analysis of vendingnew.dot

MTS

Console Source Graph Summary Counterexample graph

Here's an example where the original property is false:

```
graph TD; C1((C1)) -- pay --> C2((C2)); C2 -- change --> C3((C3)); C3 -- take --> C1; C3 -- insertCoin --> C5((C5)); C5 -- insertCoin --> C7((C7));
```

Buttons: Select PTS Model, Edit model, Delete model, Download displayed model, Full antiequiv. analysis, Linear analysis, Stop processing, Remove all annotations, Remove false optional transitions, Remove dead transitions and false deadlock states, Apply transformations, View featured transition system, Verify property, Show explanation

Conclusion and Outlook

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:
 - publicly available 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:
 - publicly available front-end tool FTS4VMC
4. Ongoing work:
 - integrating FTS2PROMELA transformation

Family-based model checking (1/2): v -ACTLive[□] [recall]

Note

1. Any FTS \mathcal{F} can trivially be transformed into an MTS \mathcal{F}_{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 MTS_vs to unambiguous FTSs:

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

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

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 . . .

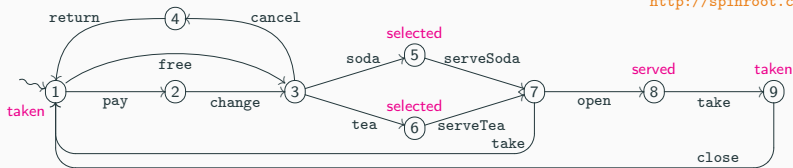
Example LTL formulas that can thus be verified with SPIN (for instance):

<http://spinroot.com/>

Family-based model checking with SPIN ...

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

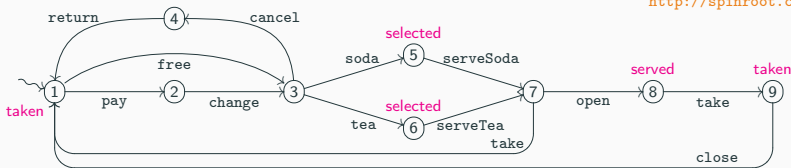
<http://spinroot.com/>



Family-based model checking with SPIN ...

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

<http://spinroot.com/>

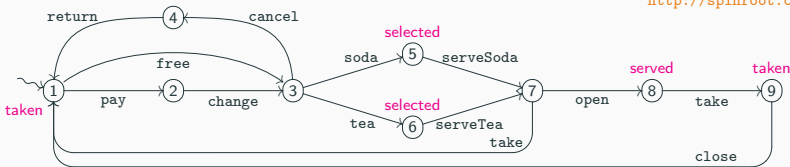


1. $G(\text{selected} \Rightarrow F \text{ served})$: after a beverage has been *selected*, the vending machine will always eventually have *served* a beverage
2. $G(\text{served} \Rightarrow F \text{ taken})$: after a beverage has been *served*, a customer will always eventually have *taken* the beverage

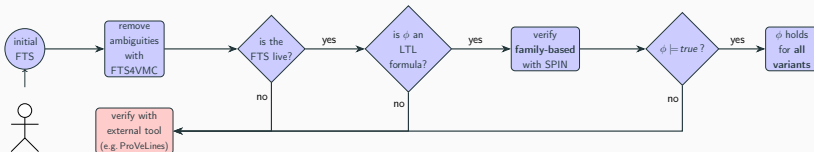
Family-based model checking with SPIN ...

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

<http://spinroot.com/>



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



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, New York, 2019, 39–51 (**best paper**)
<https://doi.org/10.1145/3336294.3336295>
- 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**)
<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 of the 25th International Systems and Software Product Line Conference (SPLC'21), Leicester, UK, ACM, New York, 2021 (**tool demo**)
<https://doi.org/10.1145/3461002.3473071>

Thanks for your attention!

We hope to meet many of you at **VaMoS 2022** 'here' in Florence



Technical and Variability-in-Practice papers deadline: November 6

PC chairs: Paolo Arcaini, National Institute of Informatics, Japan
and Xavier Devroey, University of Namur, Belgium

Organiser: Alessandro Fantechi (University of Florence, Italy) et al.

<https://vamos2022.isti.cnr.it/>