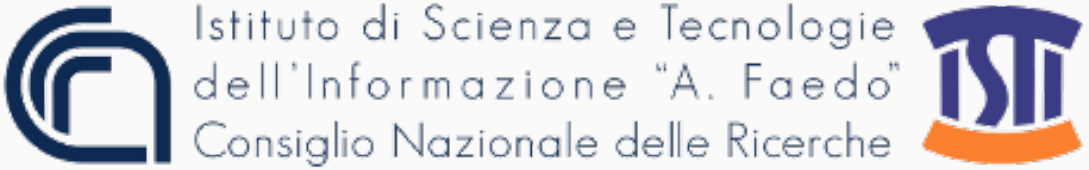
Formal Methods and Tools for Software Product Lines

Maurice ter Beek



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

ICTAC'23, Lima, Peru, December 5^{th}, 2023

Who am I?

- Head of FMT lab @ CNR-ISTI (currently 19 members) in Pisa (IT)
- MSc ('96) and PhD ('03) degrees from Leiden University (NL)
- Positions in HU ('95-'96,'02), BE ('05), IT ('00-'01), NL ('12-'13,'15)

Who am I? What do I work on?

- Head of FMT lab @ CNR-ISTI (currently 19 members) in Pisa (IT)
- MSc ('96) and PhD ('03) degrees from Leiden University (NL)
- Positions in HU ('95-'96,'02), BE ('05), IT ('00-'01), NL ('12-'13,'15)
- Formal methods, model checking, SPL, SOC, team automata, railway systems, ...

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

Software Product Line Engineering

Outline

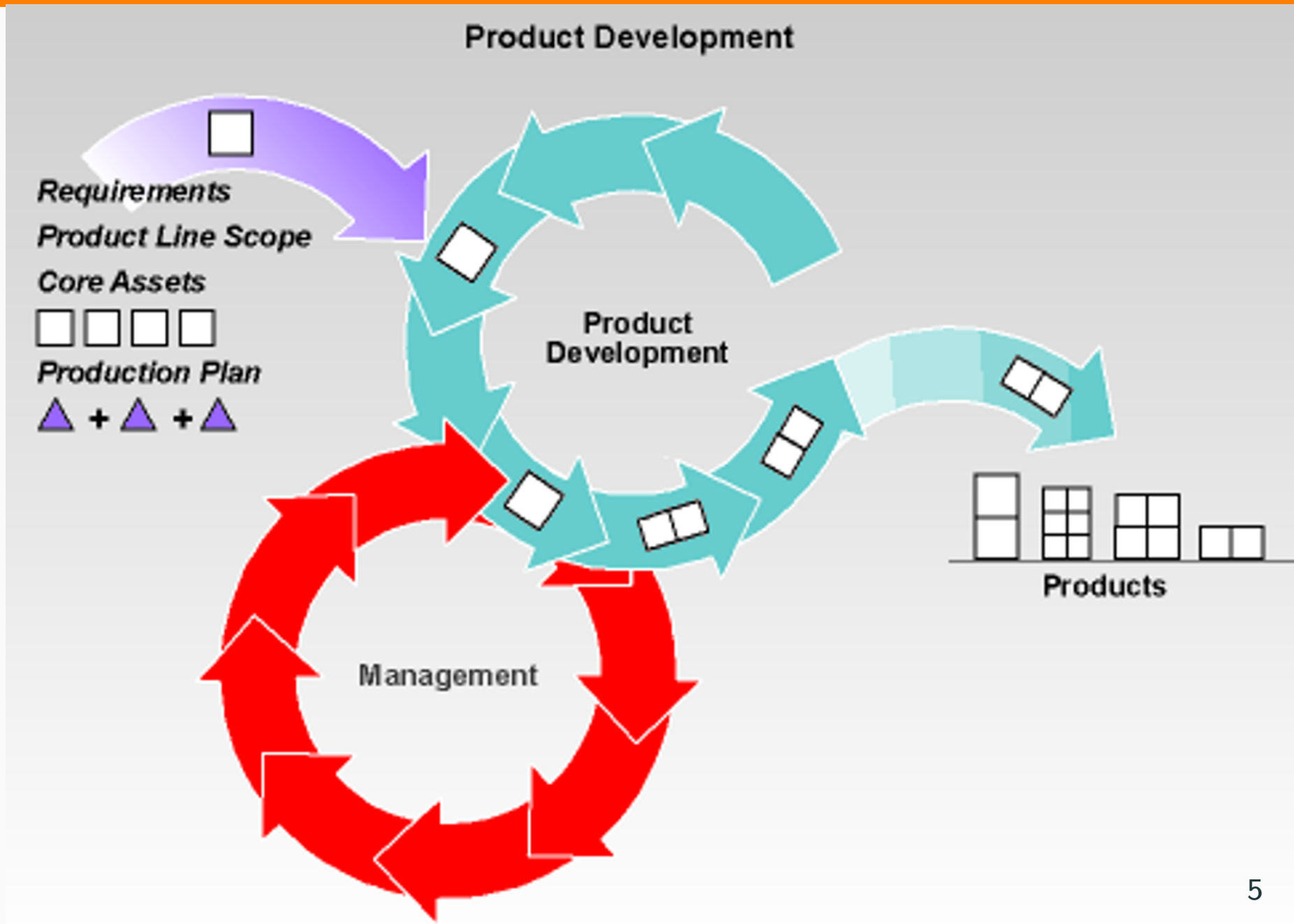
Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

Product Line Engineering in theory...



. . . and in practice

Software Product Line Engineering (SPLE)

Develop a family of products from a common reference model (usually in terms of features) and mass customisation (to serve different markets)

Software Product Line Engineering (SPLE)

Develop a family of products from a common reference model (usually in terms of features) and mass customisation (to serve different markets)

Two main differences with classical software engineering

- Two distinct development processes

- Variability in terms of features

Software Product Line Engineering (SPLE)

Develop a family of products from a common reference model (usually in terms of features) and mass customisation (to serve different markets)

Two main differences with classical software engineering

- Two distinct development processes
  - Domain engineering: develop reusable domain artefacts
  - Application engineering: develop individual products by reusing domain artefacts
- Variability in terms of features

Software Product Line Engineering (SPLE)

Develop a family of products from a common reference model (usually in terms of features) and mass customisation (to serve different markets)

Two main differences with classical software engineering

- Two distinct development processes
  - Domain engineering: develop reusable domain artefacts
  - Application engineering: develop individual products by reusing domain artefacts
- Variability in terms of features
  - Common features that are part of all products
  - Variable features that can be selected per product

Example of variability in requirements

R12: “The navigation system must allow the user to make inputs using a control panel or by voice entry”

Example of variability in requirements

R12: "The navigation system must allow the user to make inputs using a control panel or by voice entry"

- R12 comprises the following three realisations:
  1. A navigation system that allows the user to make inputs only via the control panel
  2. A navigation system that allows the user to make inputs only via voice entry
  3. A navigation system that allows the user to make inputs only via the control panel and by voice entry

Example of variability in requirements

R12: "The navigation system must allow the user to make inputs using a control panel or by voice entry"

- R12 comprises the following three realisations:
  1. A navigation system that allows the user to make inputs only via the control panel
  2. A navigation system that allows the user to make inputs only via voice entry
  3. A navigation system that allows the user to make inputs only via the control panel and by voice entry
- Conjunction is a logical 'or' or an exclusive 'or'?

Example of variability in requirements

R12: "The navigation system must allow the user to make inputs using a control panel or by voice entry"

- R12 comprises the following three realisations:
  1. A navigation system that allows the user to make inputs only via the control panel
  2. A navigation system that allows the user to make inputs only via voice entry
  3. A navigation system that allows the user to make inputs only via the control panel and by voice entry
- Conjunction is a logical 'or' or an exclusive 'or'?
- Is only one system asked for, or two or three different systems?

Variation points and variants

- A variation point represents an aspect of a product family that varies among the different products

  R12: 'input modality of the user interface'

Variation points and variants

- A variation point represents an aspect of a product family that varies among the different products

  R12: 'input modality of the user interface'

- A variant represents a specific configuration (i.e., an incarnation) of a variable aspect that a product in a product family can have

  R12: 'input via control panel', 'input via voice entry'

Variability: prerequisite for success

- Domain engineering: explicit documentation of variability supports the identification of possible variable aspects and fosters explicit decisions about which aspects shall be variable in the product family (variation points) and which options shall exist for each variable aspect (variants);

Variability: prerequisite for success

- Domain engineering: explicit documentation of variability supports the identification of possible variable aspects and fosters explicit decisions about which aspects shall be variable in the product family (variation points) and which options shall exist for each variable aspect (variants);

  it also supports engineers, architects, designers and testers in realising the defined variation points and variants

Variability: prerequisite for success

- Domain engineering: explicit documentation of variability supports the identification of possible variable aspects and fosters explicit decisions about which aspects shall be variable in the product family (variation points) and which options shall exist for each variable aspect (variants);

  it also supports engineers, architects, designers and testers in realising the defined variation points and variants

- Application engineering: explicit documentation of variability in terms of variation points and variants supports system development by making explicit the necessary decisions and decision options

What's in a *feature*?

Reference	Definition
Kang *et al.* [3]	*"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"*
Kang *et al.* [8]	*"distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained"*
Eisenecker and Czarnecki [6]	*"anything users or client programs might want to control about a concept"*
Bosch *et al.* [9]	*"A logical unit of behaviour specified by a set of functional and non-functional requirements."*
Chen *et al.* [10]	*"a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"*
Batory [11]	*"an elaboration or augmentation of an entity(s) that introduces a new service, capability or relationship"*
Batory *et al.* [12]	*"an increment in product functionality"*
Apel *et al.* [13]	*"a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option."*

Classen *et al.*, What's in a Feature: A Requirements Engineering Perspective @ FASE'08

Features

- What is a feature?

Features

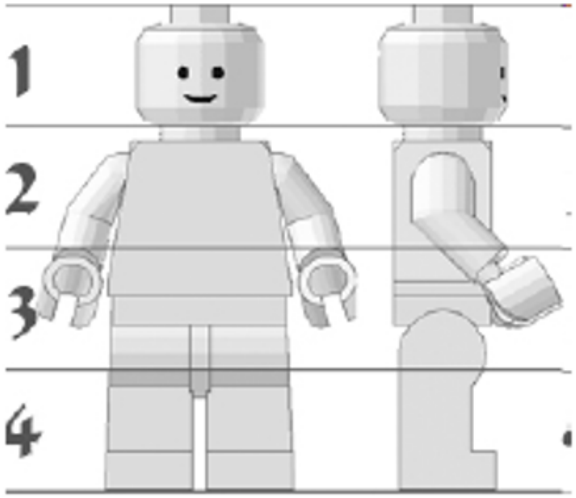
- What is a feature?
  - End-user visible behaviour or property of a system...

Features

- What is a feature?
  - End-user visible behaviour or property of a system...
  - ... that may be optional and/or may have alternatives

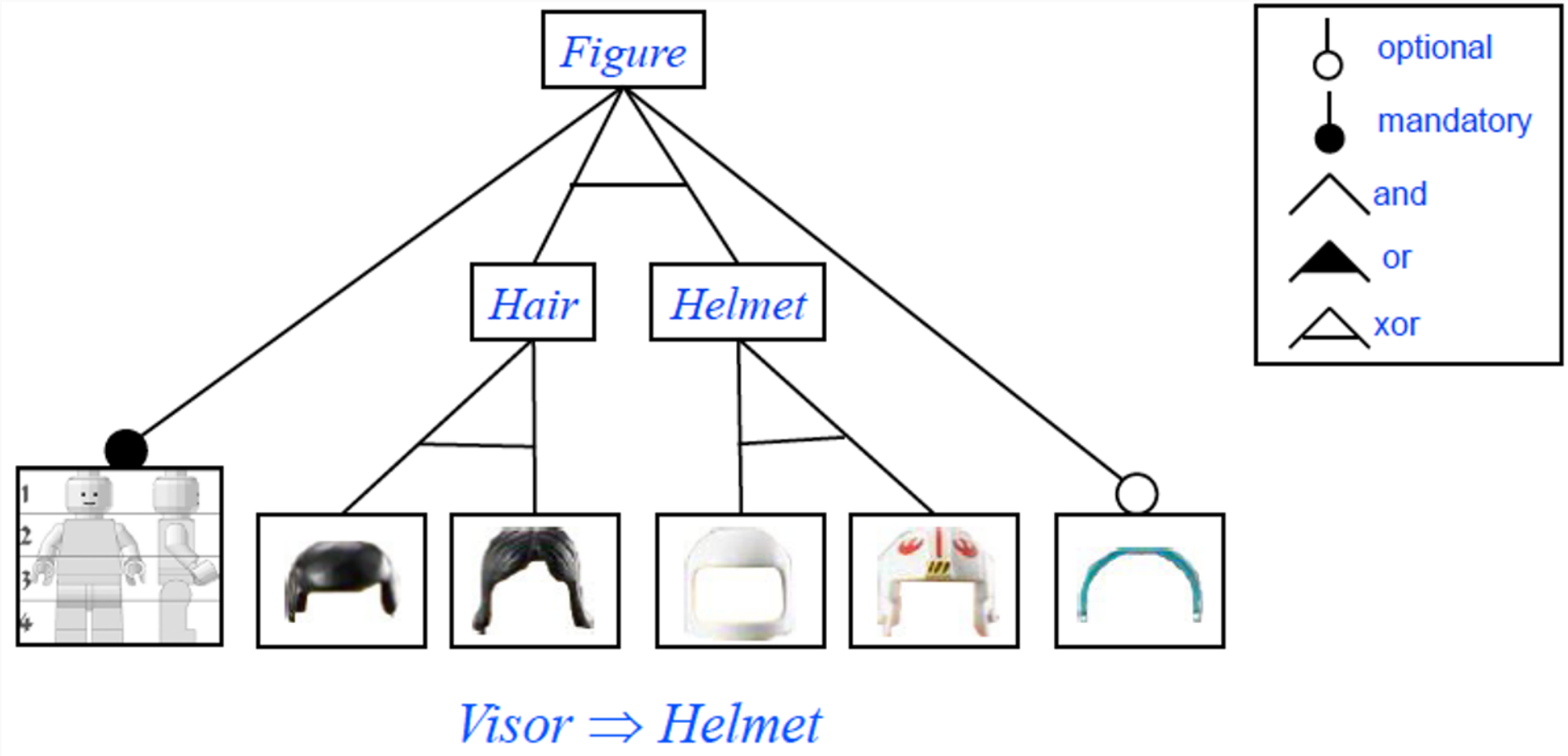
Features

- What is a feature?
  - End-user visible behaviour or property of a system...
  - ... that may be optional and/or may have alternatives
- Features represent commonalities and variabilities of (software) systems



Feature selection

Typically, only a subset of feature combinations is valid



Products and product lines

Product $\Leftrightarrow$ valid feature combination (configuration)

Products and product lines

Product $\Leftrightarrow$ valid feature combination (configuration)

Product line $\Leftrightarrow$ set of valid feature combinations of a domain

Features and software

Main concept in SPLE

- Easy to use in informal models
- Easily converts into business: product sales
- Easily converts into product design: variability
- Enables reuse of features

Features and software

Main concept in SPLE

- Easy to use in informal models
- Easily converts into business: product sales
- Easily converts into product design: variability
- Enables reuse of features

In telecommunication, features became popular in the 1960s with the advent of computer-controlled telephone switches; software for telecommunication has been conceived in terms of features since

- 1992–2009 International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI)

Variability models: explicit decisions

- Compact representations of all products of a product family in terms of their features

Variability models: explicit decisions

- Compact representations of all products of a product family in terms of their features
- Feature diagram/model: hierarchical tree structure, with the family as its root, features as its nodes and possibly further cross-tree constraints

Kang *et al.*, Feature-Oriented Domain Analysis (FODA) Feasibility Study, SEI@CMU'90

Variability models: explicit decisions

- Compact representations of all products of a product family in terms of their features
- Feature diagram/model: hierarchical tree structure, with the family as its root, features as its nodes and possibly further cross-tree constraints

  Kang *et al.*, Feature-Oriented Domain Analysis (FODA) Feasibility Study, SEI@CMU'90

- Orthogonal Variability Model — Pohl, Requirements Engineering, 2010, etc.

Variability models: explicit decisions

- Compact representations of all products of a product family in terms of their features
- Feature diagram/model: hierarchical tree structure, with the family as its root, features as its nodes and possibly further cross-tree constraints

  Kang *et al.*, Feature-Oriented Domain Analysis (FODA) Feasibility Study, SEI@CMU'90

- Orthogonal Variability Model — Pohl, Requirements Engineering, 2010, etc.
- Common Variability Language (CVL): OMG's (failed) effort to standardise variability modelling as a separate and generic language to define variability on base models

Variability models: explicit decisions

- Compact representations of all products of a product family in terms of their features
- Feature diagram/model: hierarchical tree structure, with the family as its root, features as its nodes and possibly further cross-tree constraints

  Kang *et al.*, Feature-Oriented Domain Analysis (FODA) Feasibility Study, SEI@CMU'90
- Orthogonal Variability Model — Pohl, Requirements Engineering, 2010, etc.
- Common Variability Language (CVL): OMG's (failed) effort to standardise variability modelling as a separate and generic language to define variability on base models
- Universal Variability Language (UVL): latest community effort (6th International Workshop on Languages for Modelling Variability: MODEVAR@VaMoS'24)

ter Beek *et al.*, Textual Variability Modeling Languages: An Overview and Considerations @ SPLC'19

Advantages of dedicated conceptual model

- Improved communication with different stakeholders (e.g., communicate to customers which variants can be selected at which variation points)

Advantages of dedicated conceptual model

- Improved communication with different stakeholders (e.g., communicate to customers which variants can be selected at which variation points)
- Transparent decisions, i.e., the originator of a variation point is forced to state the rationale for introducing variability in a specific domain artefact

Advantages of dedicated conceptual model

- Improved communication with different stakeholders (e.g., communicate to customers which variants can be selected at which variation points)
- Transparent decisions, i.e., the originator of a variation point is forced to state the rationale for introducing variability in a specific domain artefact
- Relationships between requirements and variants become traceable (e.g., stakeholders can document which requirements, design, implementation and test artefacts are influenced by a variant)

Variability dependencies (1/2)

Allowed choices of variants at a specific variation point

Variability dependencies (1/2)

Allowed choices of variants at a specific variation point

- Mandatory variant: if the variation point is selected, this variant *must* always be selected

Variability dependencies (1/2)

Allowed choices of variants at a specific variation point

- Mandatory variant: if the variation point is selected, this variant *must* always be selected
- Optional variant: if the variation point is selected, this variant *may* be selected but it does not have to be

Variability dependencies (1/2)

Allowed choices of variants at a specific variation point

- Mandatory variant: if the variation point is selected, this variant *must* always be selected
- Optional variant: if the variation point is selected, this variant *may* be selected but it does not have to be
- Alternative choice, i.e., a collection of at least two optional variants, possibly together with a [min . . . max] notation to indicate the permissible number of variants to be selected: if the variation point is selected, at least "min" variants *must* be selected while at most "max" variants *may* be selected

Variability dependencies (2/2)

Cross-tree constraints

Variability dependencies (2/2)

Cross-tree constraints

- Requires: indicates that the presence of one feature requires the presence of another feature

Variability dependencies (2/2)

Cross-tree constraints

- Requires: indicates that the presence of one feature requires the presence of another feature
- Excludes: indicates that the presence of two features is mutually exclusive

Variability dependencies (2/2)

Cross-tree constraints

- Requires: indicates that the presence of one feature requires the presence of another feature
- Excludes: indicates that the presence of two features is mutually exclusive

Quantitative constraints

Variability dependencies (2/2)

Cross-tree constraints

- Requires: indicates that the presence of one feature requires the presence of another feature
- Excludes: indicates that the presence of two features is mutually exclusive

Quantitative constraints

- Feature attributes (non-functional): $cost\,(\text{feature}) = 7$, etc.

Variability dependencies (2/2)

Cross-tree constraints

- Requires: indicates that the presence of one feature requires the presence of another feature
- Excludes: indicates that the presence of two features is mutually exclusive

Quantitative constraints

- Feature attributes (non-functional): $cost\,(\text{feature}) = 7$, etc.
- $cost\,(\text{product}) = \sum\{cost\,(\text{feature}) \mid \text{feature} \in \text{product}\}$

Example requirements of a coffee machine

1. Initially, a coin must be inserted: either a €, exclusively in case of coffee machines for the European market, or a \$, exclusively in case of coffee machines for the Canadian market (alternative features)

Example requirements of a coffee machine

1. Initially, a coin must be inserted: either a €, exclusively in case of coffee machines for the European market, or a \$, exclusively in case of coffee machines for the Canadian market (alternative features)
2. After having inserted a coin, the user has to be offered the option to choose whether or not (s)he wants sugar in her/his beverage, after which (s)he has to be offered to select a beverage (mandatory features)

Example requirements of a coffee machine

1. Initially, a coin must be inserted: either a €, exclusively in case of coffee machines for the European market, or a \$, exclusively in case of coffee machines for the Canadian market (alternative features)
2. After having inserted a coin, the user has to be offered the option to choose whether or not (s)he wants sugar in her/his beverage, after which (s)he has to be offered to select a beverage (mandatory features)
3. The choice of beverages offered by a coffee machine may vary (the options being cappuccino, coffee and tea), but every coffee machine must offer at least one beverage (or relation among features);

Example requirements of a coffee machine

1. Initially, a coin must be inserted: either a €, exclusively in case of coffee machines for the European market, or a $, exclusively in case of coffee machines for the Canadian market (alternative features)
2. After having inserted a coin, the user has to be offered the option to choose whether or not (s)he wants sugar in her/his beverage, after which (s)he has to be offered to select a beverage (mandatory features)
3. The choice of beverages offered by a coffee machine may vary (the options being cappuccino, coffee and tea), but every coffee machine must offer at least one beverage (or relation among features);

   furthermore, whenever a coffee machine offers cappuccino, then it must offer coffee as well, while tea may only be offered by coffee machines for the European market (requires and excludes relations among features)

Example requirements of a coffee machine

1. Initially, a coin must be inserted: either a €, exclusively in case of coffee machines for the European market, or a $, exclusively in case of coffee machines for the Canadian market (alternative features)
2. After having inserted a coin, the user has to be offered the option to choose whether or not (s)he wants sugar in her/his beverage, after which (s)he has to be offered to select a beverage (mandatory features)
3. The choice of beverages offered by a coffee machine may vary (the options being cappuccino, coffee and tea), but every coffee machine must offer at least one beverage (or relation among features);

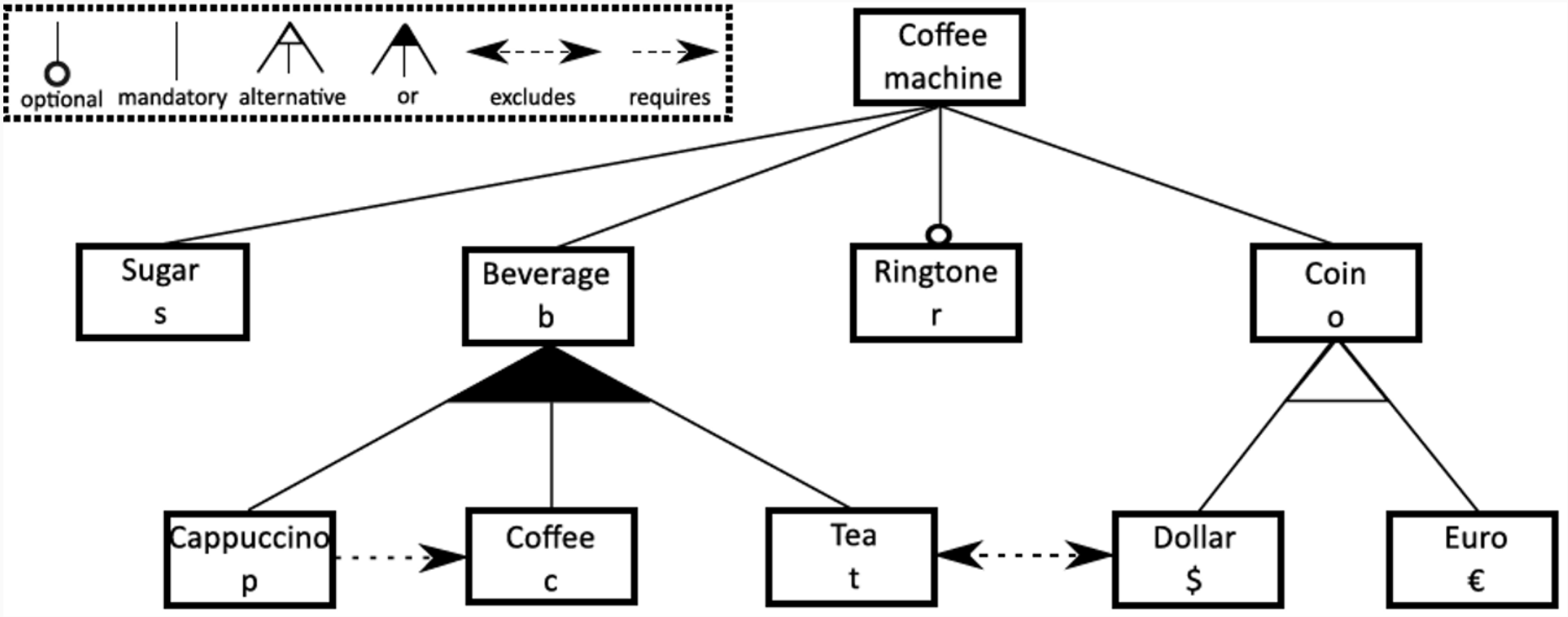
   furthermore, whenever a coffee machine offers cappuccino, then it must offer coffee as well, while tea may only be offered by coffee machines for the European market (requires and excludes relations among features)
4. Optionally, a ringtone may be rung after the coffee machine has delivered the chosen beverage (optional feature)

Example requirements of a coffee machine

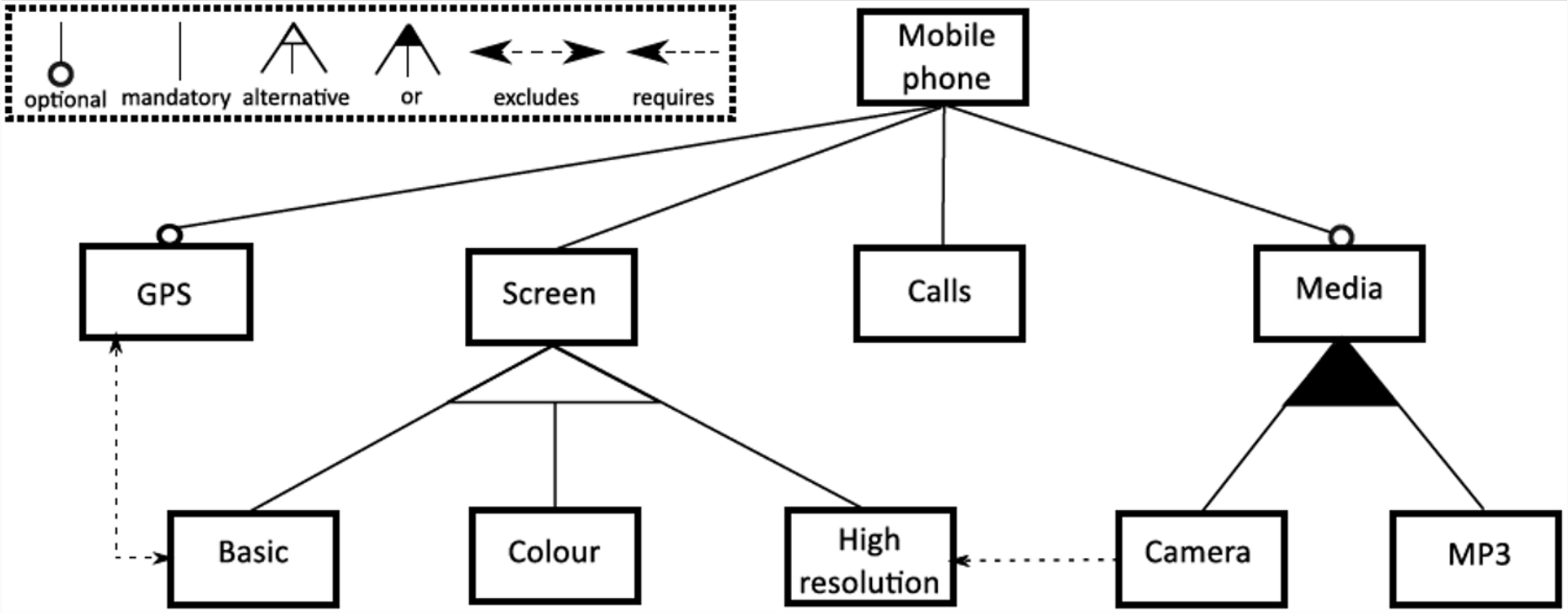
1. Initially, a coin must be inserted: either a €, exclusively in case of coffee machines for the European market, or a $, exclusively in case of coffee machines for the Canadian market (alternative features)
2. After having inserted a coin, the user has to be offered the option to choose whether or not (s)he wants sugar in her/his beverage, after which (s)he has to be offered to select a beverage (mandatory features)
3. The choice of beverages offered by a coffee machine may vary (the options being cappuccino, coffee and tea), but every coffee machine must offer at least one beverage (or relation among features);

   furthermore, whenever a coffee machine offers cappuccino, then it must offer coffee as well, while tea may only be offered by coffee machines for the European market (requires and excludes relations among features)
4. Optionally, a ringtone may be rung after the coffee machine has delivered the chosen beverage (optional feature)
5. As soon as the user has taken her/his beverage, the coffee machine must return in its idle state

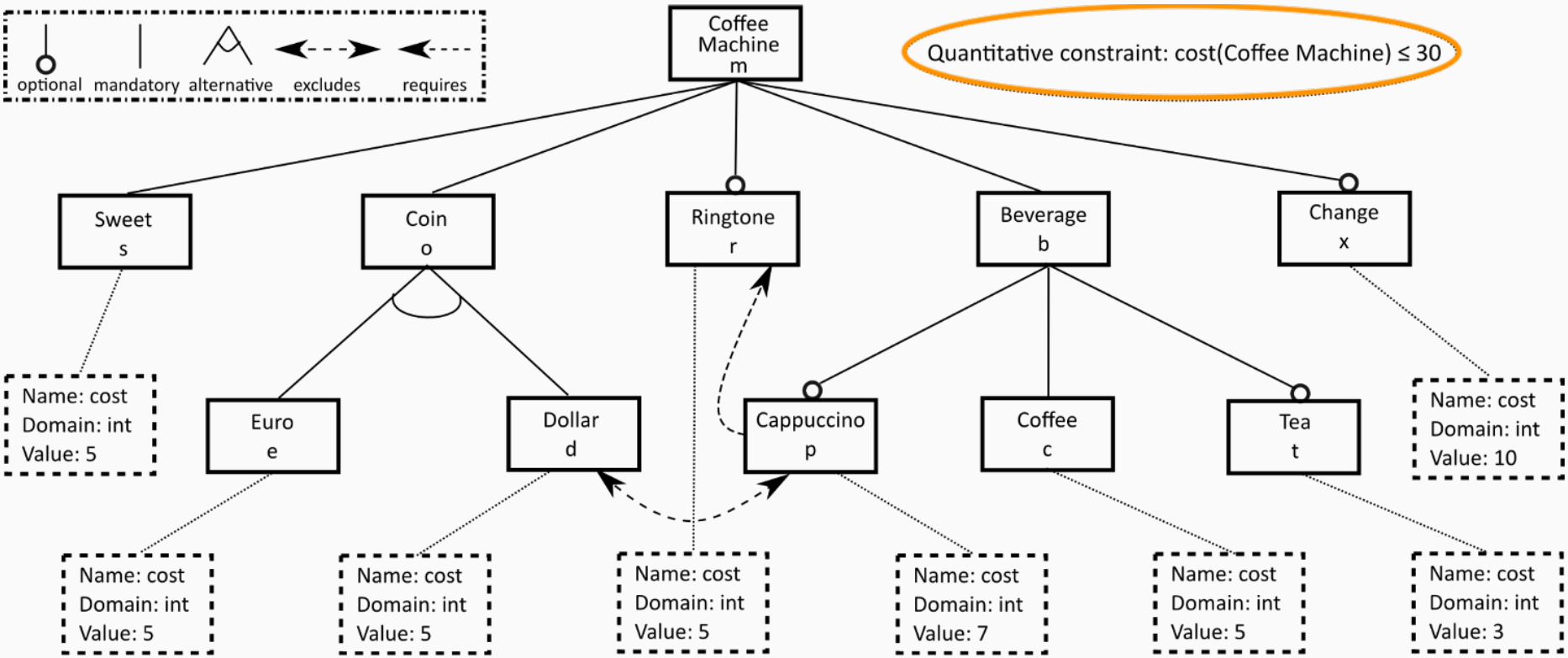
Example: coffee machine feature model



Example: mobile phone feature model



Example: coffee machine attributed feature model



Applications of feature models

- Communicate with stakeholders

Applications of feature models

- Communicate with stakeholders
- Identify objects for reuse

Applications of feature models

- Communicate with stakeholders
- Identify objects for reuse
- Identify objects for sales opportunities

Applications of feature models

- Communicate with stakeholders
- Identify objects for reuse
- Identify objects for sales opportunities
- Identify cross-cutting concerns

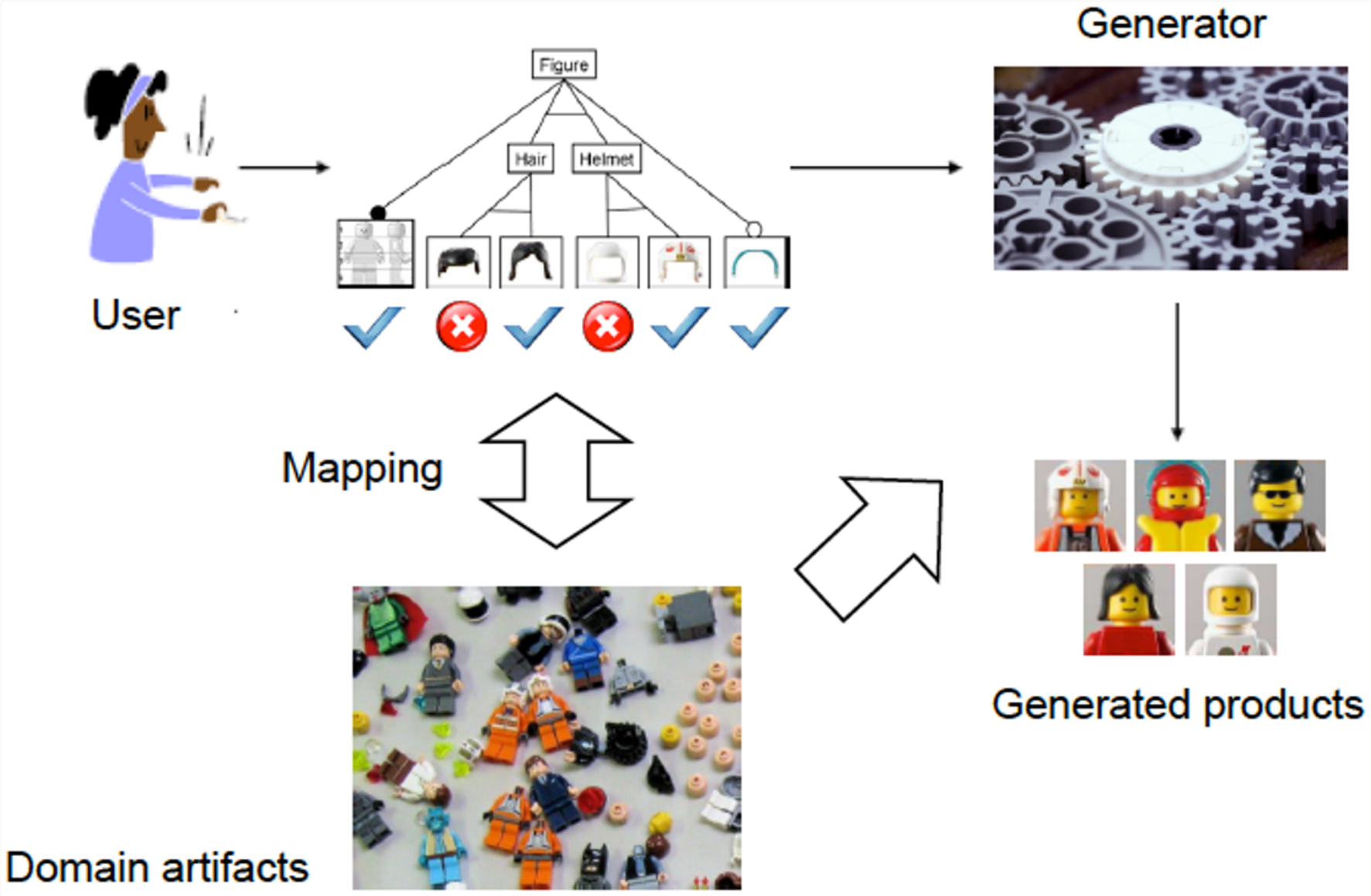
Applications of feature models

- Communicate with stakeholders
- Identify objects for reuse
- Identify objects for sales opportunities
- Identify cross-cutting concerns
- Software composition and deployment

Applications of feature models

- Communicate with stakeholders
- Identify objects for reuse
- Identify objects for sales opportunities
- Identify cross-cutting concerns
- Software composition and deployment
- etc.

Automated product derivation

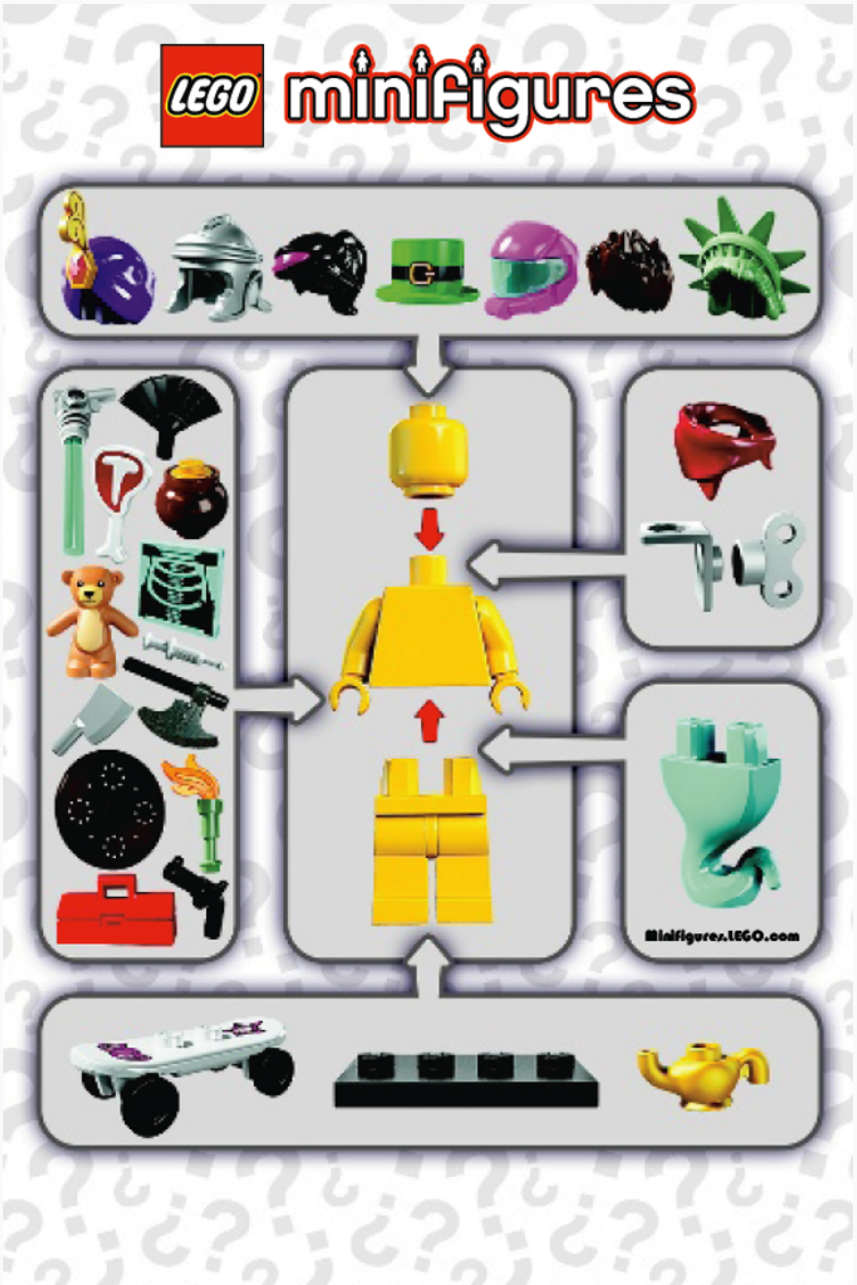


Czarnecki & Eisenecker, Generative Programming: Methods, Tools and Applications, 2000

Real-life example: a product line of Lego minifigures

(16 valid feature combinations)

‘Feature model’ for configuring the 16 valid products



(allowing more. . .)

Product lines are everywhere

Configure your 11-inch MacBook Air

Hardware | Service and Support | Accessories | Printers

Hardware

Processor

Enjoy incredible performance from fourth-generation Intel Core processors. Choose the speed and processor you want.

Learn more

- (•) 1.3GHz Dual-Core Intel Core i5, Turbo Boost up to 2.6GHz
- ( ) 1.7GHz Dual-Core Intel Core i7, Turbo Boost up to 3.3GHz [+ £130.00]

Memory

More memory (RAM) increases overall performance and enables your computer to run more applications at the same time.

Learn more

- (•) 4GB 1600MHz LPDDR3 SDRAM
- ( ) 8GB 1600MHz LPDDR3 SDRAM [+ £80.00]

Storage

Your MacBook Air comes as standard with flash storage. Flash storage has no moving parts and provides faster responsiveness and enhanced durability.

Learn more

- (•) 256GB Flash Storage
- ( ) 512GB Flash Storage [+ £240.00]

Summary

£1,029.00 incl. VAT

Special 0% financing

Estimate Payments

Dispatched:
Within 24 hours
Free Delivery

Add to Basket

Gift package available

Contact Us

0800 048 0408

Live Chat

Specifications

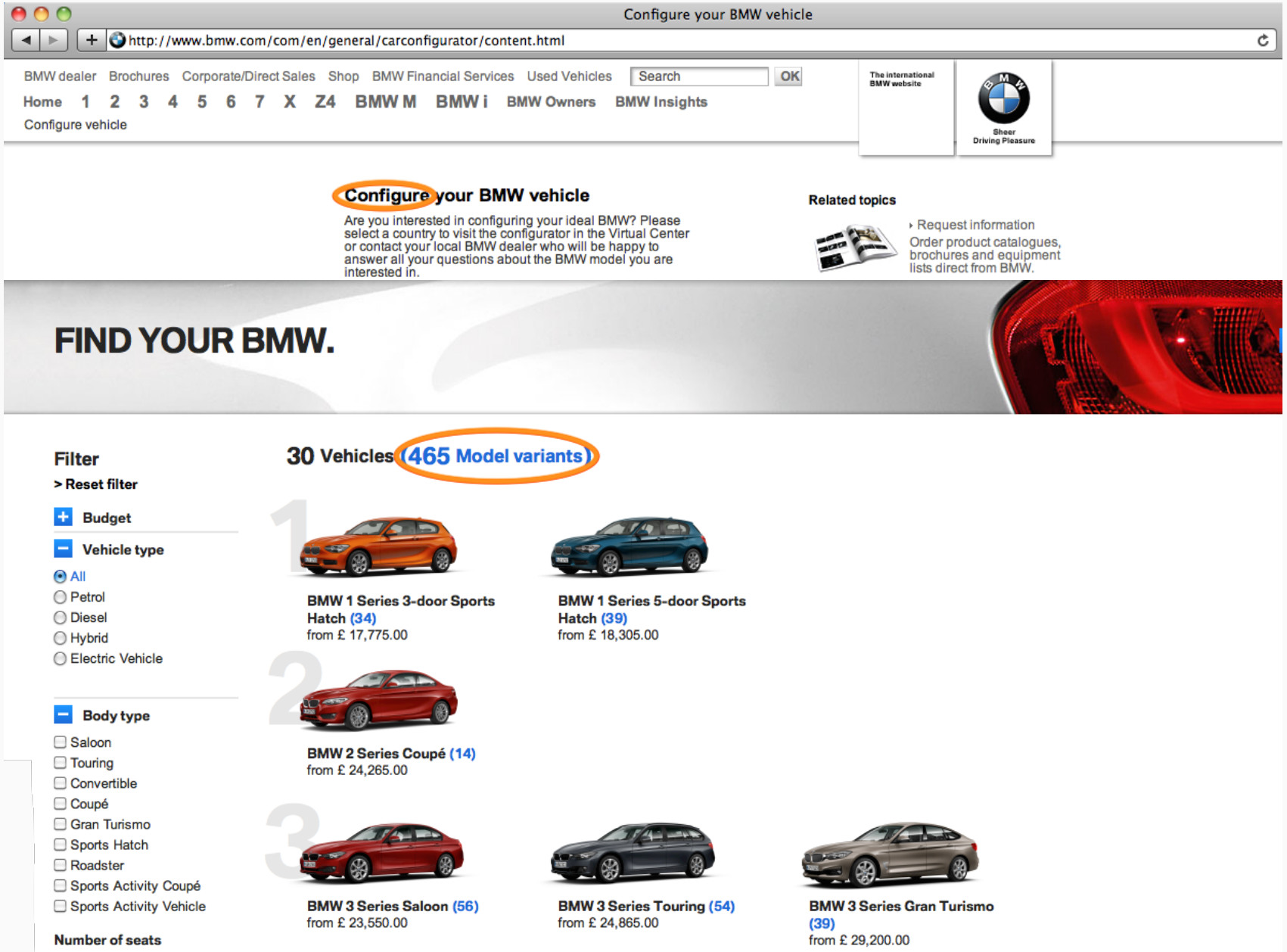
1.3GHz Dual-Core Intel Core i5, Turbo Boost up to 2.6GHz

4GB 1600MHz LPDDR3 SDRAM

256GB Flash Storage

Backlit Keyboard (British) & User's Guide (English)

Product lines are everywhere



Feature binding times

- The moment a choice is made for a variable feature

Feature binding times

- The moment a choice is made for a variable feature
- For a software product:
  - Compile time
  - Release time
  - Deployment time
  - Start-up time
  - Runtime

Feature binding times

- The moment a choice is made for a variable feature
- For a software product:
  - Compile time
  - Release time
  - Deployment time
  - Start-up time
  - Runtime
- For a car:
  - Spray time
  - Drive-out-of-the-factory time
  - ...

Feature binding times

- The moment a choice is made for a variable feature
- For a software product:
  - Compile time
  - Release time
  - Deployment time
  - Start-up time
  - Runtime
- For a car:
  - Spray time
  - Drive-out-of-the-factory time
  - ...
- For lego:
  - Playing time 

Domain Requirements Engineering (RE)

During domain RE, the requirements for the entire SPL are defined

Domain Requirements Engineering (RE)

During domain RE, the requirements for the entire SPL are defined

Basis for:

1. Developing the entire SPL family
2. Defining the requirements of each product

Domain Requirements Engineering (RE)

During domain RE, the requirements for the entire SPL are defined

Basis for:

1. Developing the entire SPL family
2. Defining the requirements of each product

Comprises the same core and cross-sectional RE activities as RE for single systems;

Domain Requirements Engineering (RE)

During domain RE, the requirements for the entire SPL are defined

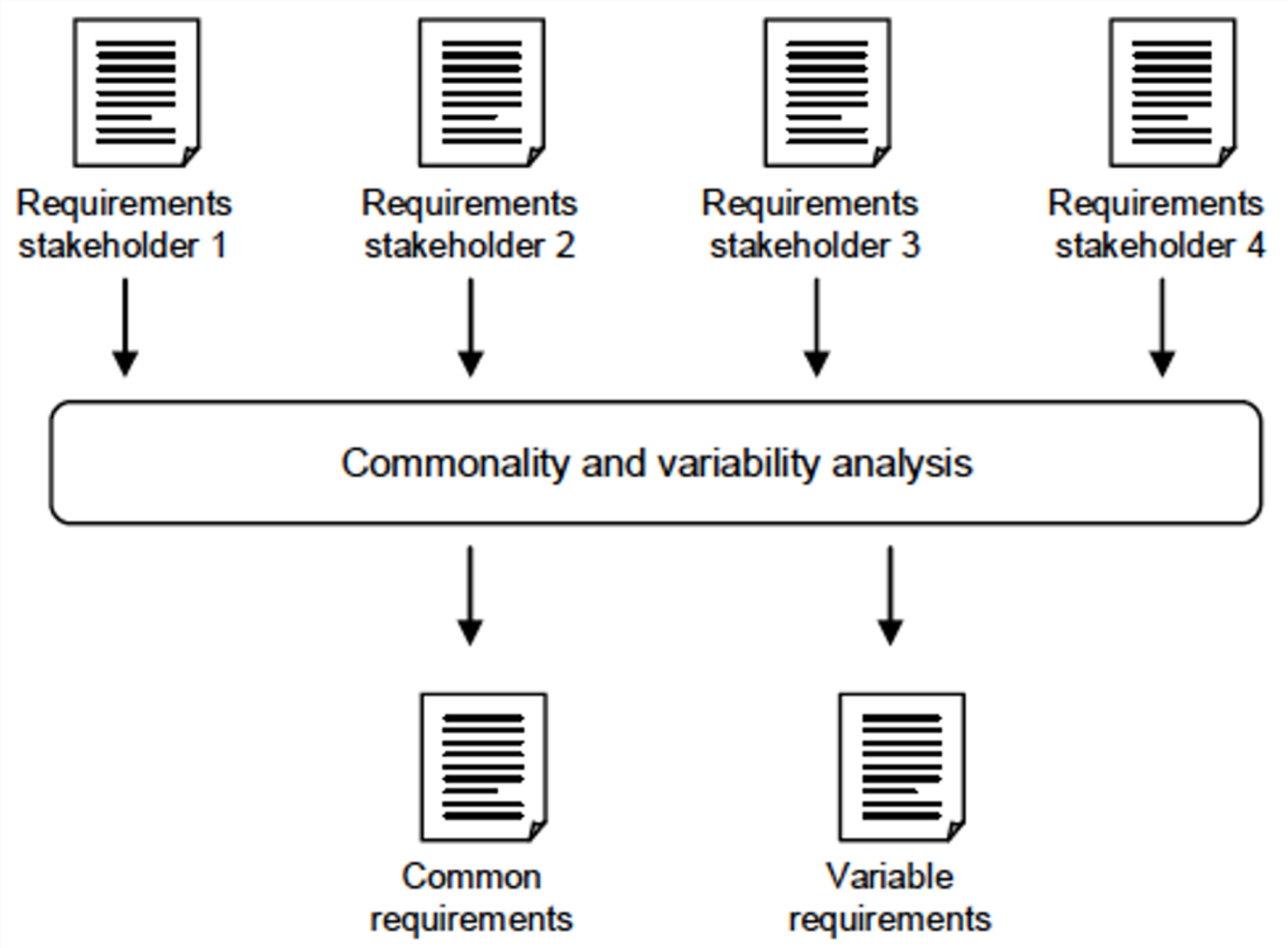
Basis for:

1. Developing the entire SPL family
2. Defining the requirements of each product

Comprises the same core and cross-sectional RE activities as RE for single systems;

these activities have the same goals plus—additionally—the goal to define the SPL variability

Elicitation of requirements variability



Application Requirements Engineering (RE)

Formal Methods and Tools Lab

During application RE, the requirements for a specific application of the SPL are defined by exploiting the domain requirements artefacts (incl. defined variation points and variants)

Application Requirements Engineering (RE)

During application RE, the requirements for a specific application of the SPL are defined by exploiting the domain requirements artefacts (incl. defined variation points and variants)

Compared with RE for single systems, two additional tasks must be accomplished:

1. Binding the defined variability
2. Documenting the variability binding

So why have variability?

Flexibility to deliver

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)
- Onto different plug-in products, etc., etc., etc.

Is is an advantage then? Yes, but...

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)
- Onto different plug-in products, etc., etc., etc.

Is is an advantage then? Yes, but...

- Requirements engineering becomes more complex

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)
- Onto different plug-in products, etc., etc., etc.

Is is an advantage then? Yes, but...

- Requirements engineering becomes more complex
- Deployment becomes more complex

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)
- Onto different plug-in products, etc., etc., etc.

Is is an advantage then? Yes, but...

- Requirements engineering becomes more complex
- Deployment becomes more complex
- Sales becomes more complex

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)
- Onto different plug-in products, etc., etc., etc.

Is is an advantage then? Yes, but...

- Requirements engineering becomes more complex
- Deployment becomes more complex
- Sales becomes more complex
- Updates become WAY more complex

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)
- Onto different plug-in products, etc., etc., etc.

Is is an advantage then? Yes, but...

- Requirements engineering becomes more complex
- Deployment becomes more complex
- Sales becomes more complex
- Updates become WAY more complex
- Testing becomes more complex (all configurations need to be tested)

So why have variability?

Flexibility to deliver

- Different versions of products based on the same trunk of code (Windows XP professional, Windows XP Home edition, etc.)
- Onto different platforms (a Linux, a Mac and a Windows version)
- With different components (Windows Media Player, iTunes, etc.)
- Onto different plug-in products, etc., etc., etc.

Is is an advantage then? Yes, but...

- Requirements engineering becomes more complex
- Deployment becomes more complex
- Sales becomes more complex
- Updates become WAY more complex
- Testing becomes more complex (all configurations need to be tested)

"We always have 126,000,000 different bicycles in store!
(but only the parts for 1,000...)"

Weakness: feature interactions

- Features need to be combined, but have restrictions on each other

Weakness: feature interactions

- Features need to be combined, but have restrictions on each other
- Consider, e.g., a phone switching system with the following features
  1. Call forwarding
  2. Do not disturb
  3. 3-way calling
  4. No interaction between the features

Weakness: feature interactions

- Features need to be combined, but have restrictions on each other
- Consider, e.g., a phone switching system with the following features
  1. Call forwarding
  2. Do not disturb
  3. 3-way calling
  4. No interaction between the features
- Scenario 1: Bob forwards his calls to Alice; Alice sets "Do not disturb"; Bob receives a call, which is forwarded to Alice; Alice's phone rings!

Weakness: feature interactions

- Features need to be combined, but have restrictions on each other
- Consider, e.g., a phone switching system with the following features
  1. Call forwarding
  2. Do not disturb
  3. 3-way calling
  4. No interaction between the features
- Scenario 1: Bob forwards his calls to Alice; Alice sets "Do not disturb"; Bob receives a call, which is forwarded to Alice; Alice's phone rings!
- Scenario 2: Bob still forwards his calls to Alice; Bob invites Alice for a 3-way call; Carolina calls Bob to become part of the 3-way call; either:
  1. Carolina is forwarded to Alice, who cannot accept any other calls; or
  2. Carolina is accepted into the 3-way call

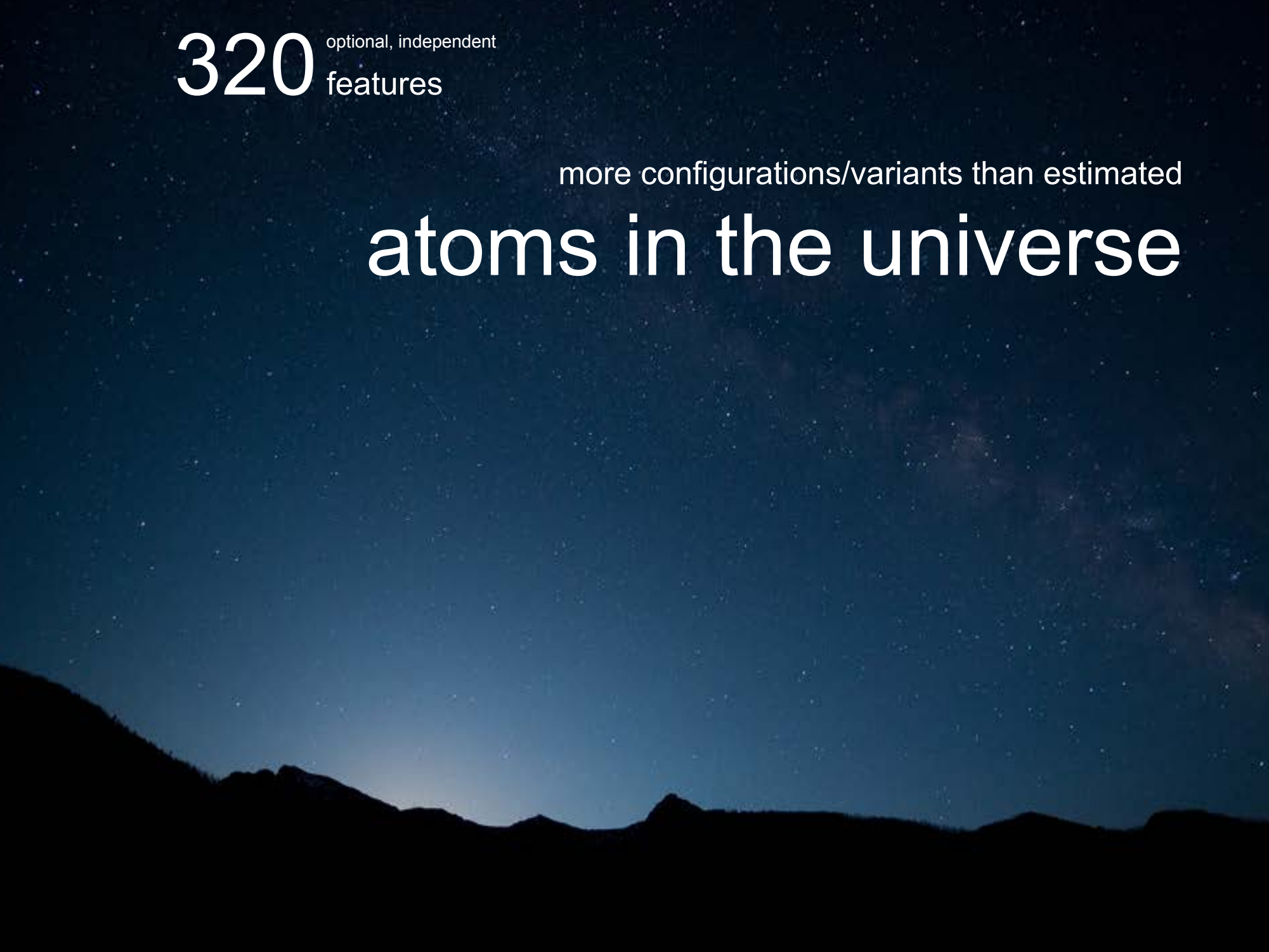
Challenge: scalability is an issue

33 optional, independent features

a unique configuration/variant for every

person on this planet

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


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

Behavioural variability modelling and analysis

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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

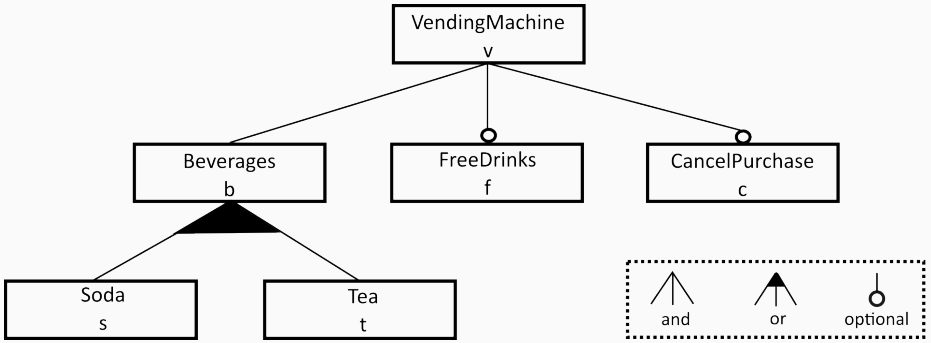
- Modal Transition System with variability constraints (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

Software Product Lines (SPL) behaviour

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

Feature models

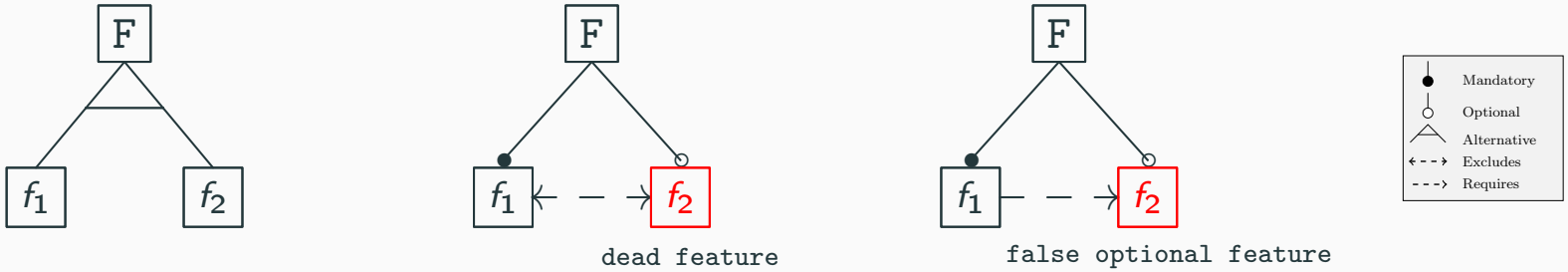
- Configurable (software) system whose variants (products) differ by the provided features, i.e. the functionality that is relevant for an end-user



Feature Model (diagram) of a beverage vending machine

Feature models analysis

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

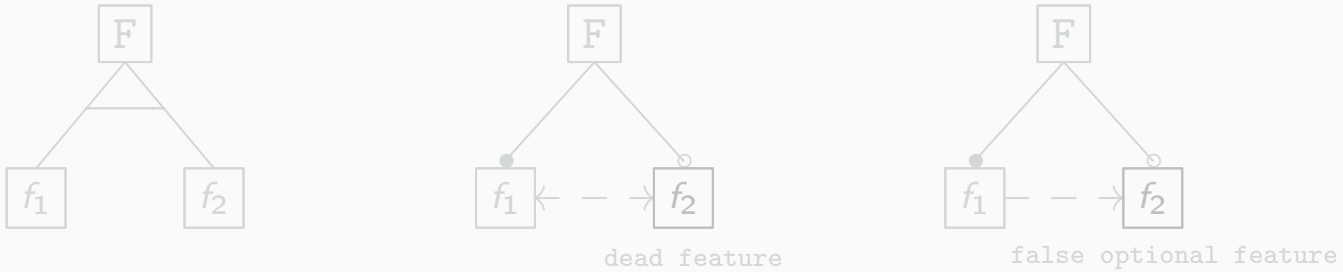


Benavides *et al.*, Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.*, 2010

Meinicke, Thüm *et al.*, Mastering Software Variability with FeatureIDE, 2017

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

Meinicke, Thüm *et al.*, Mastering Software Variability with FeatureIDE, 2017

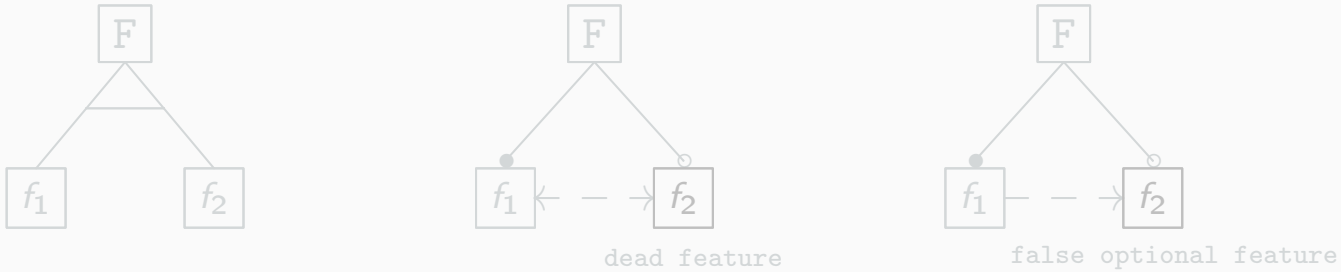
- Popular in embedded / critical systems domain: formal modelling and analysis techniques to prove SPL behaviour correct are widely studied

  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

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

Meinicke, Thüm *et al.*, Mastering Software Variability with FeatureIDE, 2017

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

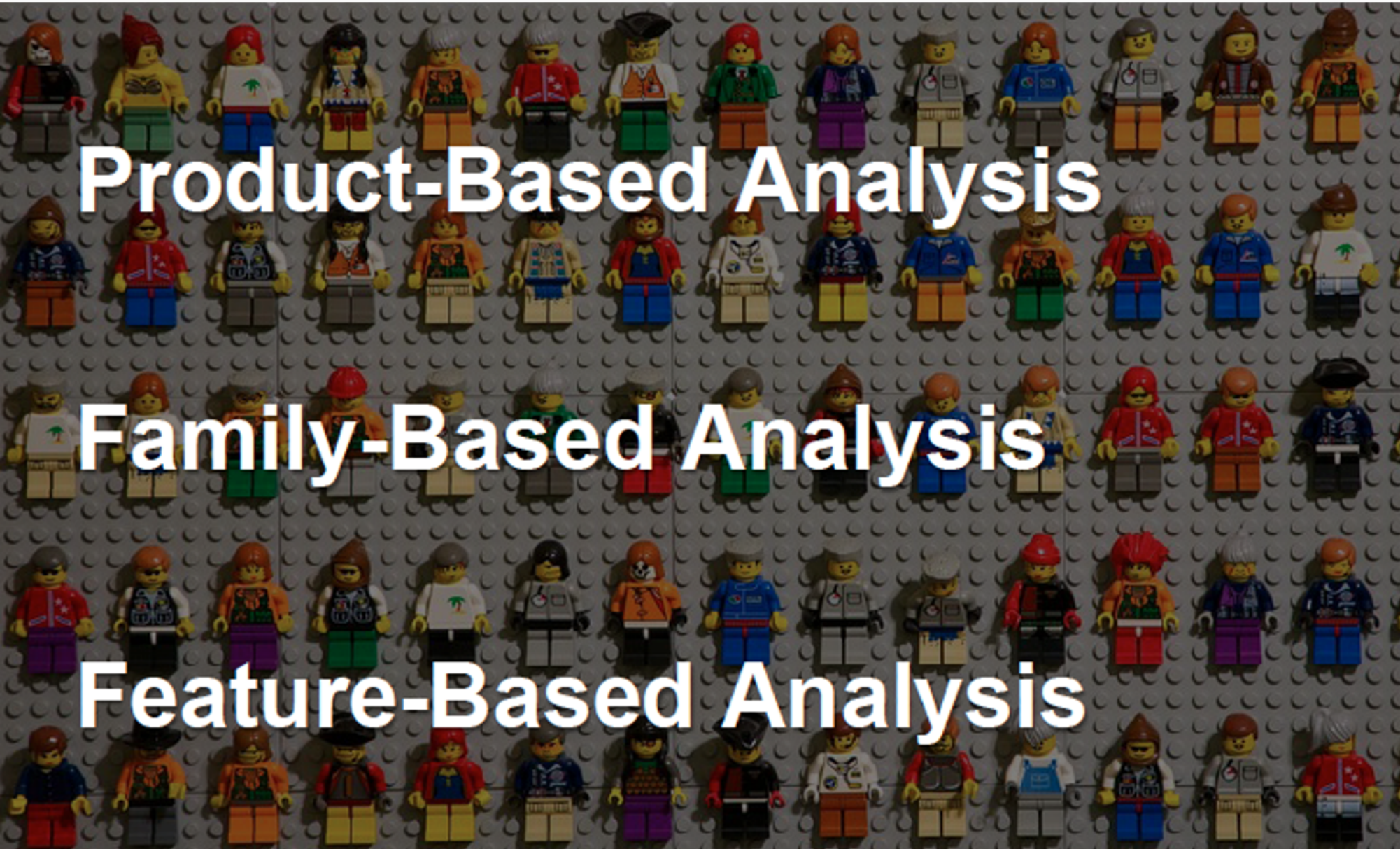
Thüm *et al.*, A classification and survey of analysis strategies for SPLs. *ACM Comput. Surv.*, 2014

- 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

Variability analysis strategies

(type checking, static analysis, model checking, theorem proving, testing, etc.)



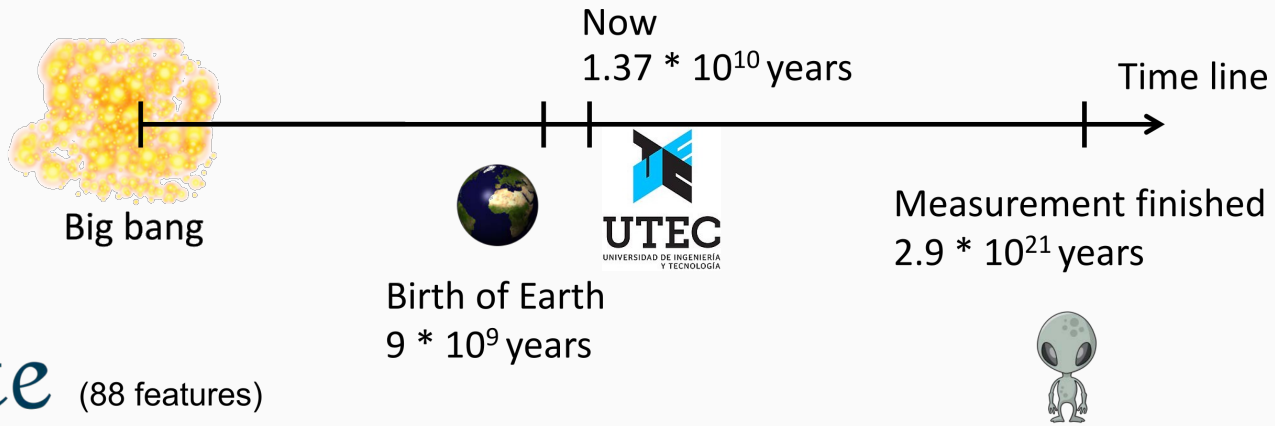
Product-based analysis:

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

Product-based analysis: inefficient, if not infeasible

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

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



SQLite (88 features)

(example by S. Apel, Saarland University, Germany)

Solution: family-based analysis

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

Solution: family-based analysis

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

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

Solution: family-based analysis


Beneficial in case of many products with substantial similarities


Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it


More complex analysis tasks


Requires (compact) family models (superimposed, *150% models*)


Dedicated variability-based models, logics, and model checkers

(e.g. FTS, MTSυ, Feature Net, PL-CCS, fLTL, fCTL, v-ACTL, QFLan, SNIP, VMC)

Solution: family-based analysis


Beneficial in case of many products with substantial similarities


Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it


More complex analysis tasks


Requires (compact) family models (superimposed, *150% models*)


Dedicated variability-based models, logics, and model checkers

(e.g. FTS, MTSυ, Feature Net, PL-CCS, fLTL, fCTL, v-ACTL, QFLan, SNIP, VMC)


Dedicated model checkers need to be maintained and optimised

Solution: family-based analysis


Beneficial in case of many products with substantial similarities


Same piece of behaviour (or code) is verified only once, regardless of how many variants can produce it


More complex analysis tasks


Requires (compact) family models (superimposed, *150% models*)


Dedicated variability-based models, logics, and model checkers

(e.g. FTS, MTSυ, Feature Net, PL-CCS, fLTL, fCTL, v-ACTL, QFLan, SNIP, VMC)


Dedicated model checkers need to be maintained and optimised

Dimovski et al., Family-based model checking without a family-based model checker @ SPIN'15

Chrszon *et al.*, Family-based modeling and analysis for probabilistic systems: featuring ProFeat @ FASE'16

ter Beek *et al.*, Family-Based Model Checking with mCRL2 @ FASE'17

Dimovski *et al.*, Variability-specific Abstraction Refinement for Family-based Model Checking @ FASE'17

Dimovski, Abstract Family-Based Model Checking Using Modal Featured Transition Systems @ FASE'18

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

Featured Transition Systems (FTSs)

Outline

Part I

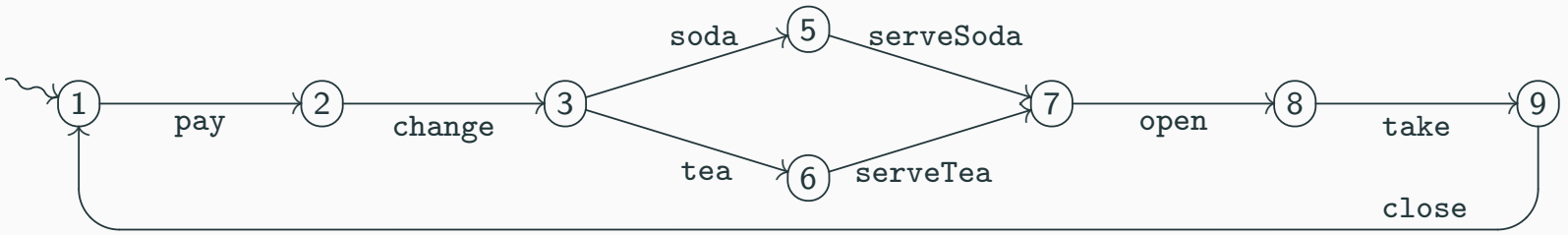
- Software Product Line Engineering (SPLE): variability, features
- Behavioural variability modelling and analysis
  - Product- vs family-based analysis
  - Featured Transition System (FTS)
  - Ambiguities: dead/false optional transitions, hidden deadlocks
- Automated static analysis of FTSs
  - Detecting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

Part II

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

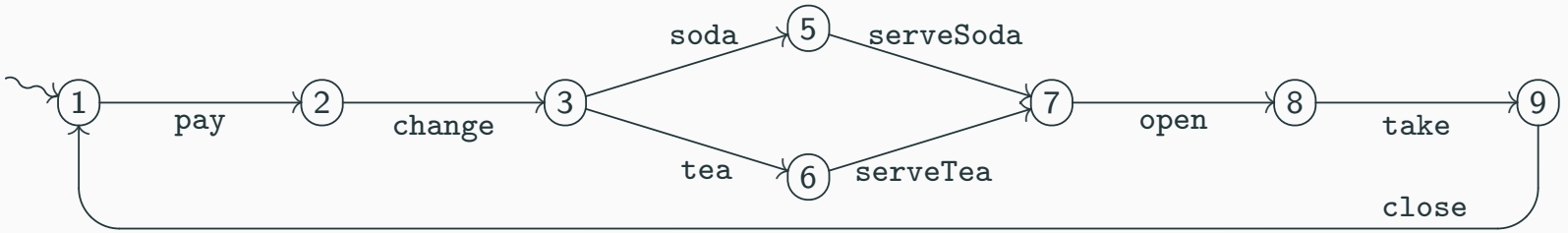
Labelled Transition System (LTS)

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 $\quad$ 1 2 3 ··· 7 8 9

Σ $\quad$ pay change ··· open take close

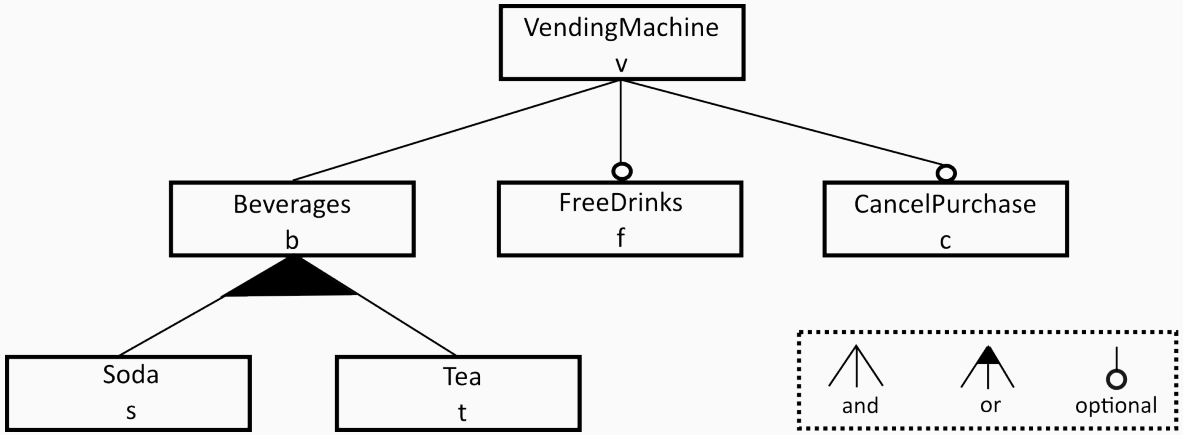
s_0 $\quad$ 1

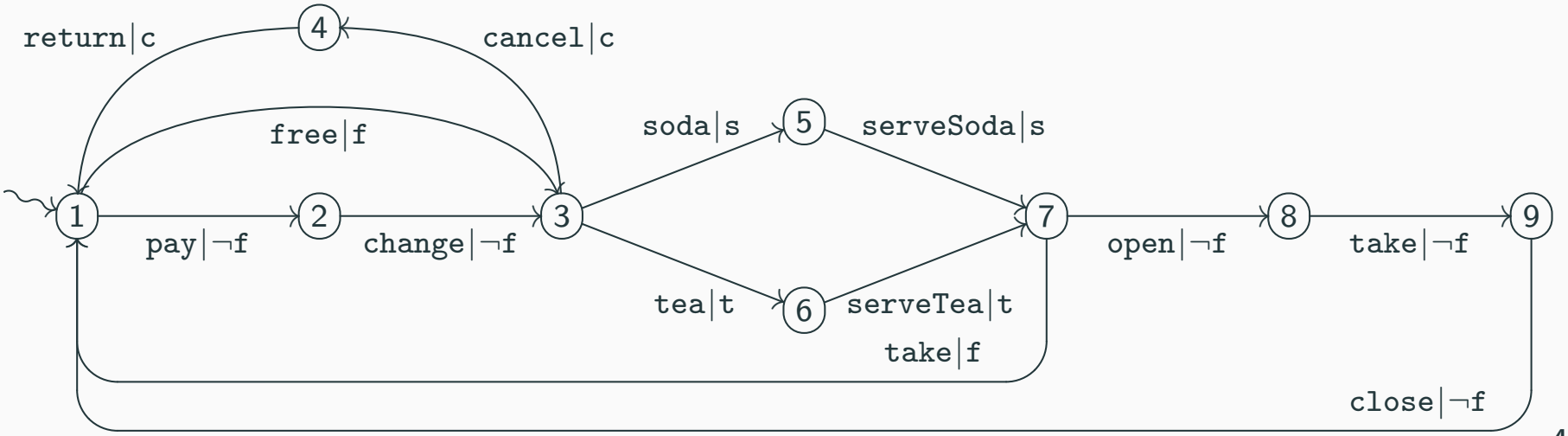
δ $\quad$ 1 —pay→ 2 $\quad$ 2 —change→ 3 $\quad$ ··· $\quad$ 7 —open→ 8 $\quad$ 8 —take→ 9 $\quad$ 9 —close→ 1

From LTS to FTS

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

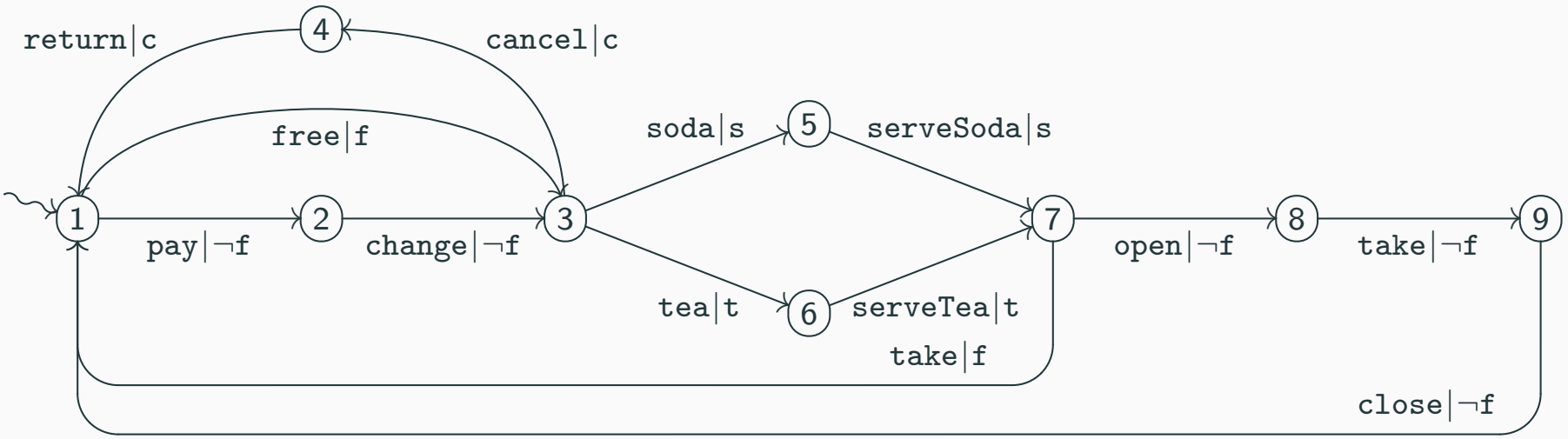
Classen *et al.*, Model checking lots of systems @ ICSE'10, FTSs. *IEEE TSE*, 2013





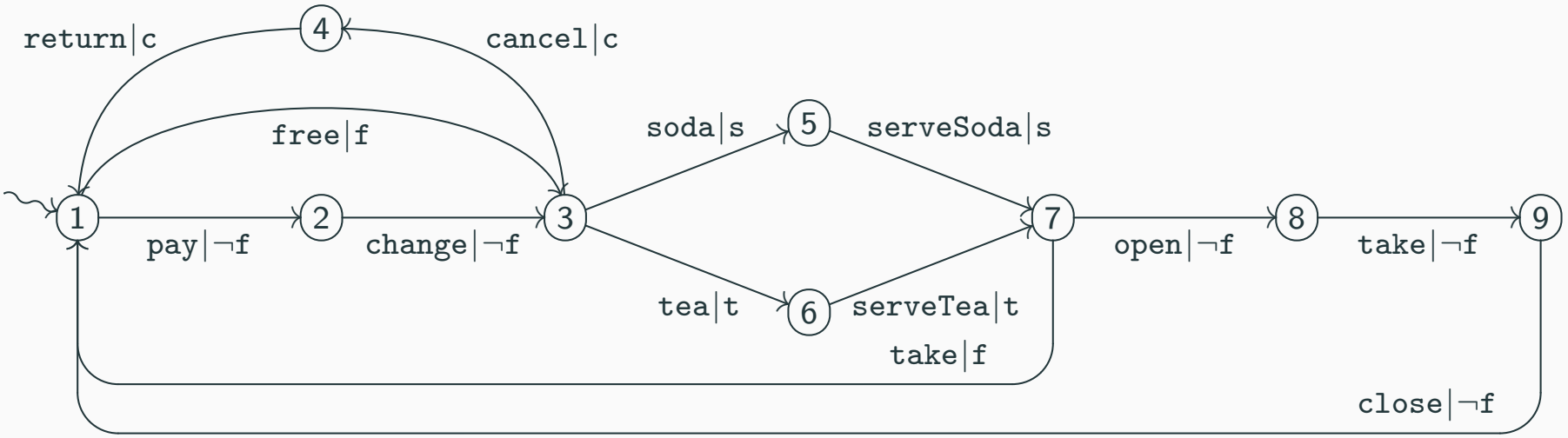
Featured Transition System (FTS)

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



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 $\Lambda \subseteq \{\, \lambda : F \to \mathbb{B} \,\}$

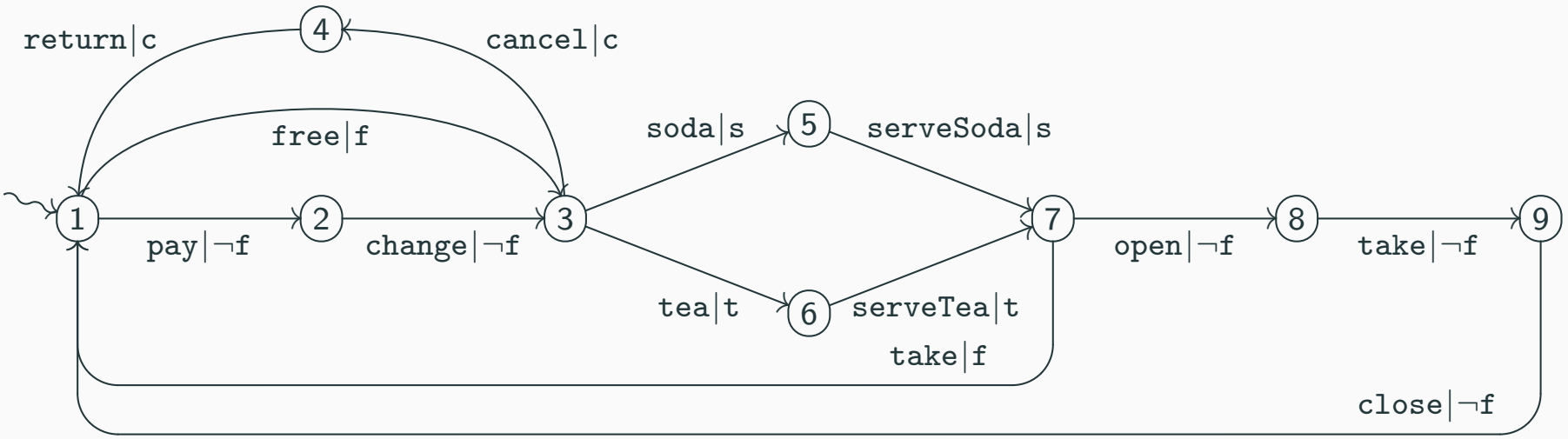


δ 1 —pay|¬f→ 2 1 —free|f→ 3 ···

F	v	b	f	c	s	t
Λ	{v,b,s,t}	{v,b,s,c}	···			

Featured Transition System (FTS)

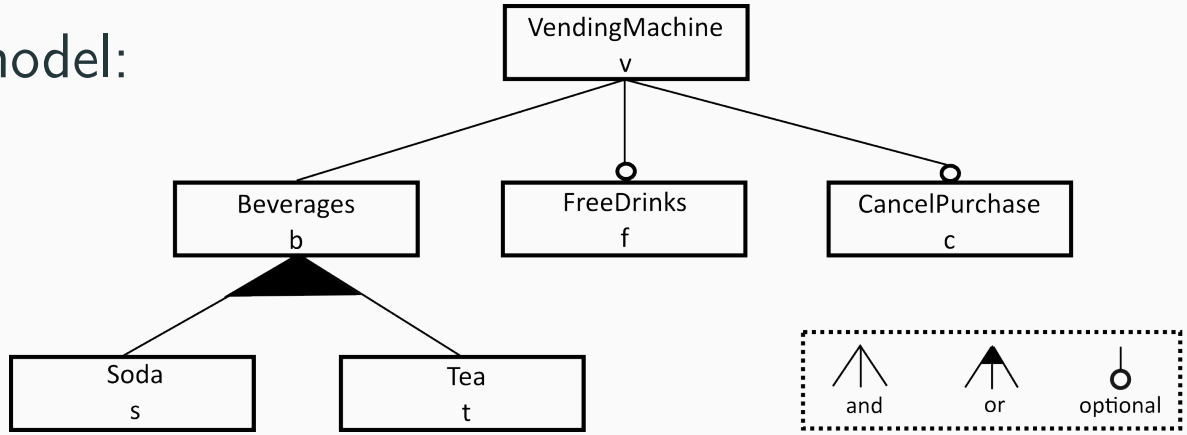
A Featured Transition System (FTS) is a sextuple $(S, \Sigma, s_0, \delta, F, \Lambda)$ with states S, actions Σ, initial state s_0, (featured) transitions $\delta \subseteq S \times \Sigma \times \mathbb{B}(F) \times S$, with Boolean (feature) expressions $\mathbb{B}(F)$ over features F, and (product)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



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

FTS of example SPL: a vending machine

Feature model:

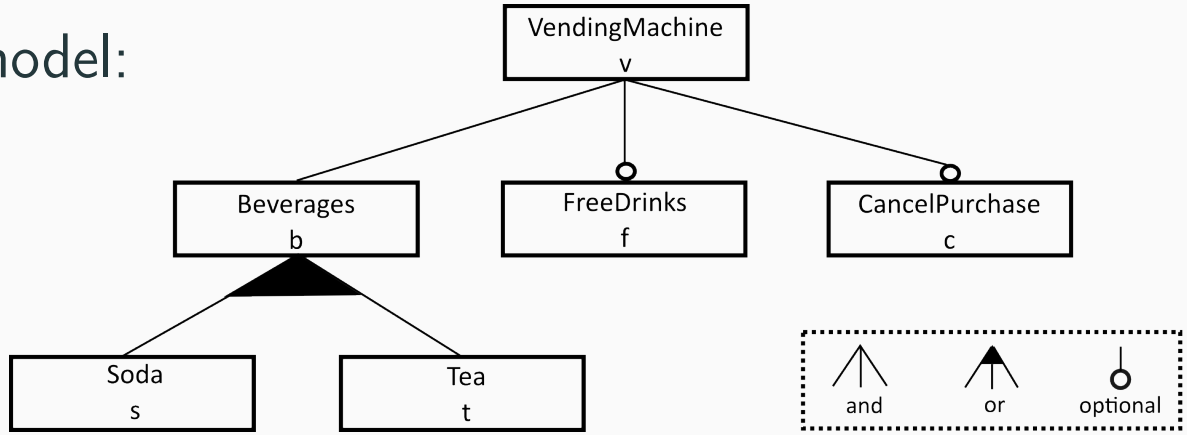


12 valid products

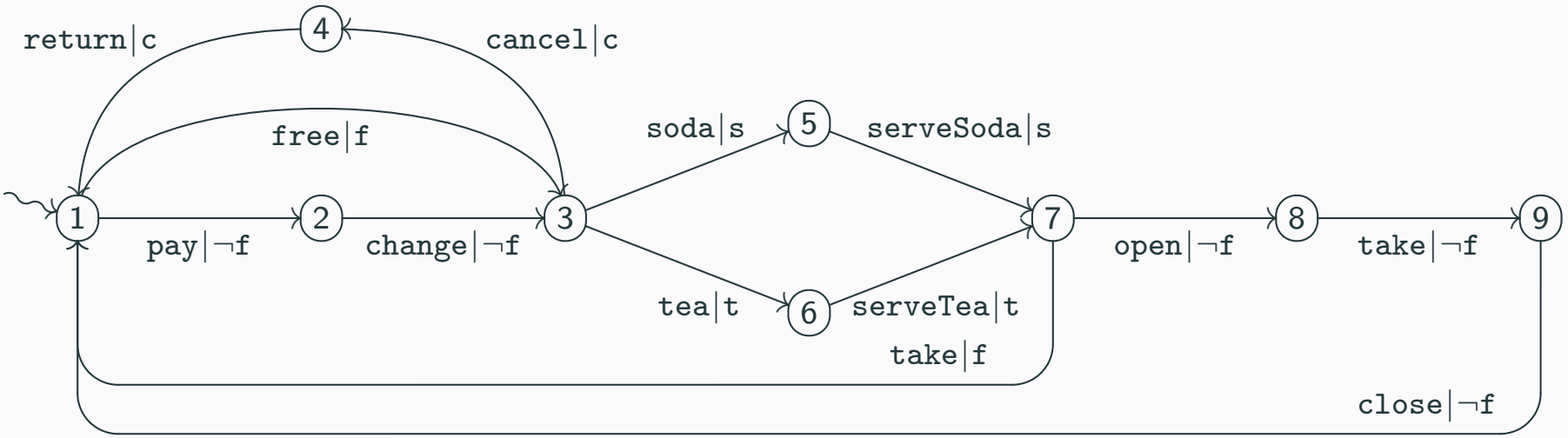
e.g., $\{\mathtt{v},\mathtt{b},\mathtt{s},\mathtt{t}\}$, $\{\mathtt{v},\mathtt{b},\mathtt{s},\mathtt{c}\}$

FTS of example SPL: a vending machine

Feature model:

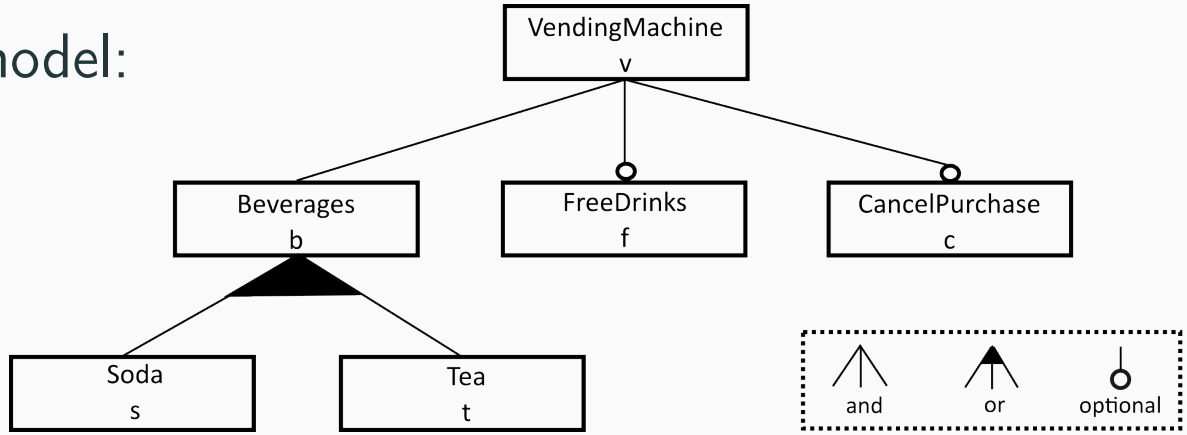


FTS of 12 valid products (LTSs) e.g., {v,b,s,t}, {v,b,s,c}

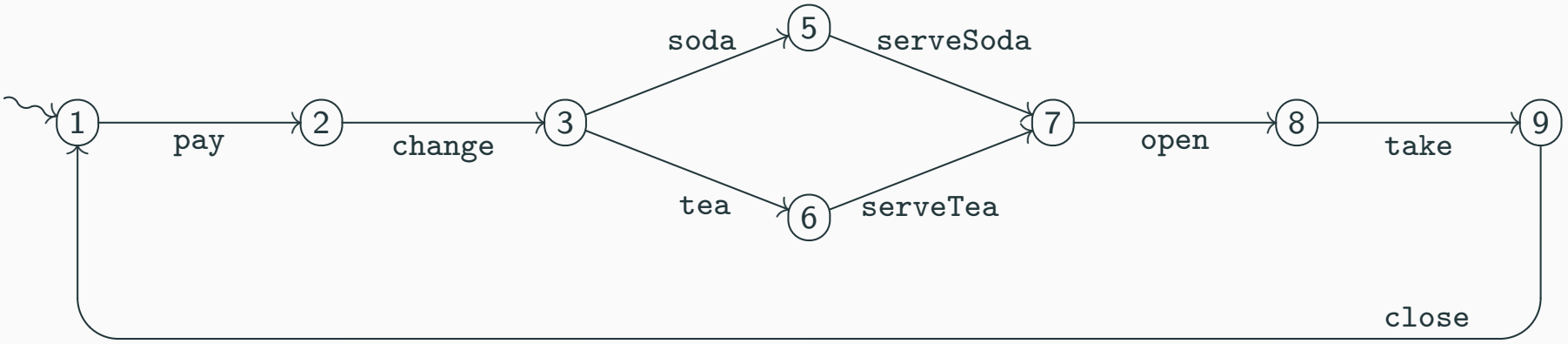


FTS of example SPL: a vending machine

Feature model:

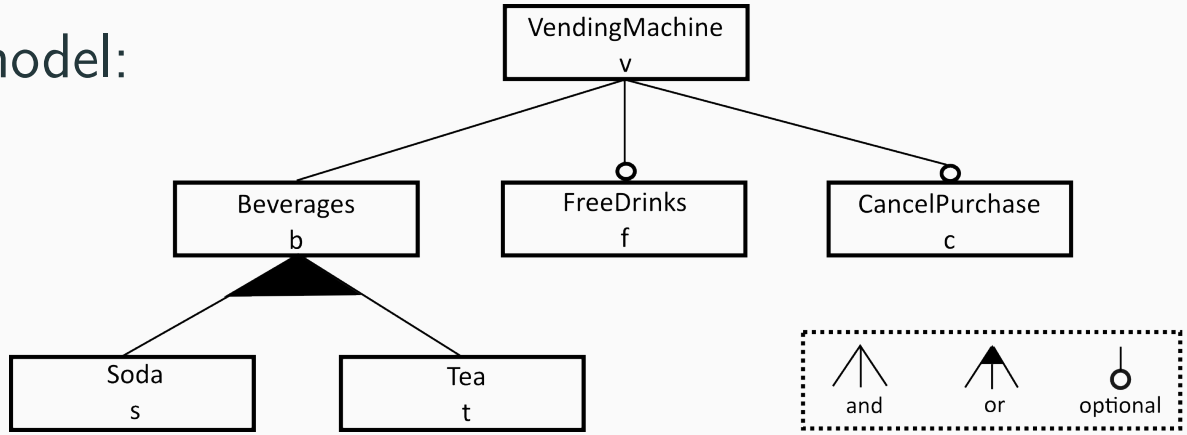


12 valid products (LTSs) e.g., {v,b,s,t}, {v,b,s,c}

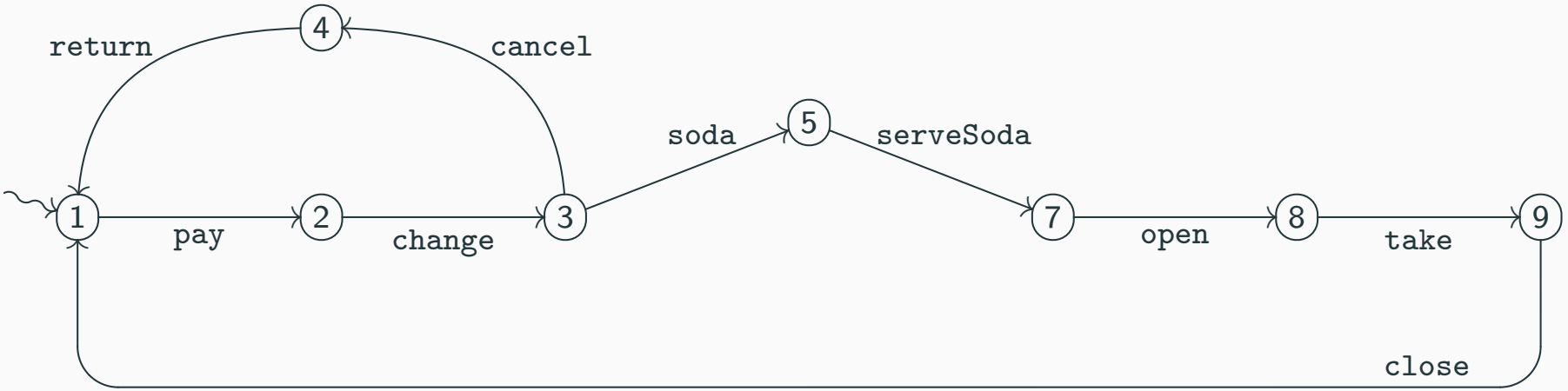


FTS of example SPL: a vending machine

Feature model:



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



Ambiguities in behavioural models

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

Key idea and aim of our SPLC'19 paper

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

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

Key idea and aim of our SPLC'19 paper

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

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

Catch and offer means to remove possible ambiguities in FTSs:

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

Ambiguous FTS

dead transition

an FTS transition not reachable in any product (LTS)

Ambiguous FTS

dead transition

an FTS transition not reachable in any product (LTS)

false optional transition a featured FTS transition which is

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

Ambiguous FTS

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

false optional transition a featured FTS transition which is

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

hidden deadlock state an FTS state which is

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

---

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

Ambiguous FTS $\longrightarrow$ unambiguous FTS

Formal Methods and Tools Lab

Transformation:

1. remove dead transitions

Ambiguous FTS $\longrightarrow$ unambiguous FTS

Transformation:

1. remove dead transitions

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

Ambiguous FTS $\longrightarrow$ 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 $\longrightarrow$ 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 $\longrightarrow$ 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

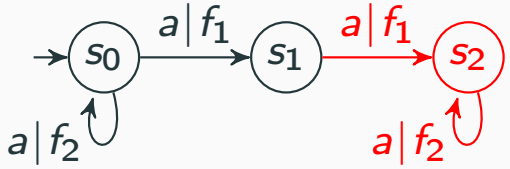
---

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)

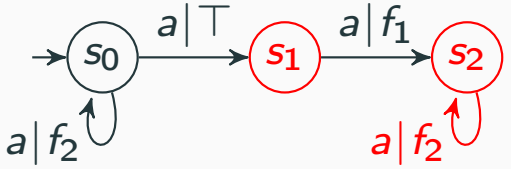
Example transformations

Feature Model: $f_1 \oplus f_2$

$\mathcal{F}$



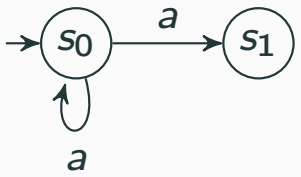
$\mathcal{F}'$



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

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

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

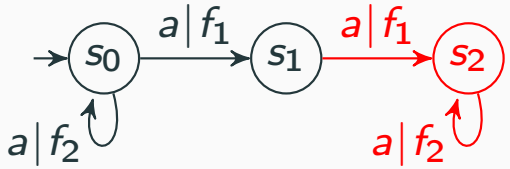


products $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$

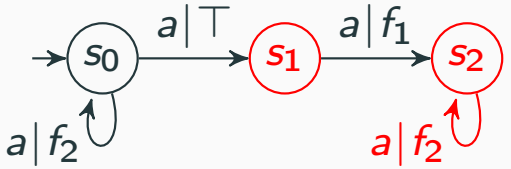
Example transformations

Feature Model: $f_1 \oplus f_2$

$\mathcal{F}$



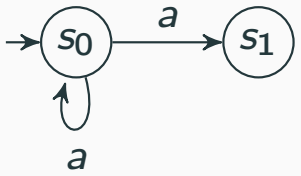
$\mathcal{F}'$



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

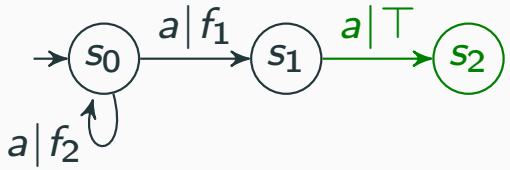
$\mathcal{F}|_{\lambda_2}$

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

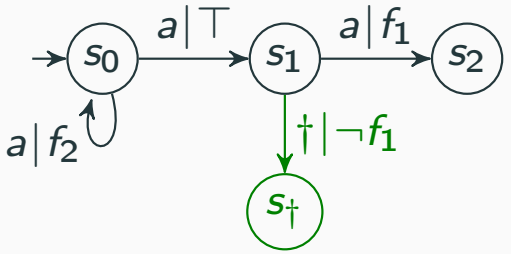


products $\lambda_1 = \{f_1\}$ and $\lambda_2 = \{f_2\}$

$\mathcal{F}_*$



$\mathcal{F}'_*$



Efficient static analysis of FTSs

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

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)

SAT solving

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

SAT solving

1. SAT solving is an expensive task, since each additional propositional atom added to the considered formula doubles 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 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 $\mathcal{I} : (F \cup S \cup T) \longrightarrow \{\top, \bot\}$

- 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` $\setminus$ {`fts.initial`};
  An initial path is a path that starts from the initial state

Useful formulas

- $\phi_{\texttt{initial}}$ is the formula `fts.initial`

Useful formulas

- $\phi_{\texttt{initial}}$ is the formula $\texttt{fts.initial}$
- $\phi_{\texttt{inner}}$ is $\bigwedge_{s \in \texttt{inner\_states}} (s \Rightarrow \texttt{atLeastOneTransitionOf}(s.\texttt{in\_trs}))$, where $\texttt{atLeastOneTransitionOf}(X)$ is a placeholder for

$$\bigvee_{t \in X} (t.\texttt{bx} \wedge t \wedge t.\texttt{source})$$

Useful formulas

- $\phi_{\texttt{initial}}$ is the formula $\texttt{fts.initial}$
- $\phi_{\texttt{inner}}$ is $\bigwedge_{s \in \texttt{inner\_states}} (s \Rightarrow \texttt{atLeastOneTransitionOf}(s.\texttt{in\_trs}))$, where $\texttt{atLeastOneTransitionOf}(X)$ is a placeholder for $\bigvee_{t \in X}(t.\texttt{bx} \wedge t \wedge t.\texttt{source})$
- $\phi_{\texttt{single}}$ is the formula $\bigwedge_{s \in \texttt{fts.states}} \texttt{atMostOneOf}(s.\texttt{out\_trs})$, where $\texttt{atMostOneOf(X)}$ is a placeholder for $\bigwedge_{t \in X} t \Rightarrow (\bigwedge_{t \in X \setminus \{t\}} \neg t)$

Useful formulas

- $\phi_{\texttt{initial}}$ is the formula $\texttt{fts.initial}$
- $\phi_{\texttt{inner}}$ is $\bigwedge_{s \in \texttt{inner\_states}} (s \Rightarrow \texttt{atLeastOneTransitionOf}(s.\texttt{in\_trs}))$, where $\texttt{atLeastOneTransitionOf}(X)$ is a placeholder for $\bigvee_{t \in X}(t.\texttt{bx} \wedge t \wedge t.\texttt{source})$
- $\phi_{\texttt{single}}$ is the formula $\bigwedge_{s \in \texttt{fts.states}} \texttt{atMostOneOf}(s.\texttt{out\_trs})$, where $\texttt{atMostOneOf(X)}$ is a placeholder for $\bigwedge_{t \in X} t \Rightarrow (\bigwedge_{t \in X \setminus \{t\}} \neg t)$
- $\texttt{end}(s)$ is the formula $s \wedge (\bigwedge_{t \in s.\texttt{out\_trs}} \neg t)$

Useful formulas

Lemma

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

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

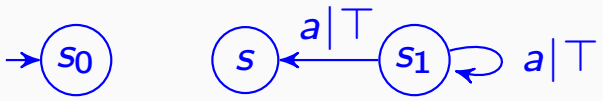
Formula is_useful_state(s)

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

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, $\texttt{fts.fm} = \top$

The formulas $\texttt{is\_useful\_state}(s)$ and $\texttt{is\_useful\_state}(s_1)$ 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 $\texttt{is\_useful\_state}(s)$

- $\texttt{exists\_deadlock}(s)$ is the formula

$$\texttt{is\_useful\_state}(s) \wedge \bigwedge_{t \in s.\texttt{out\_trs}} \neg t.\texttt{bx}$$

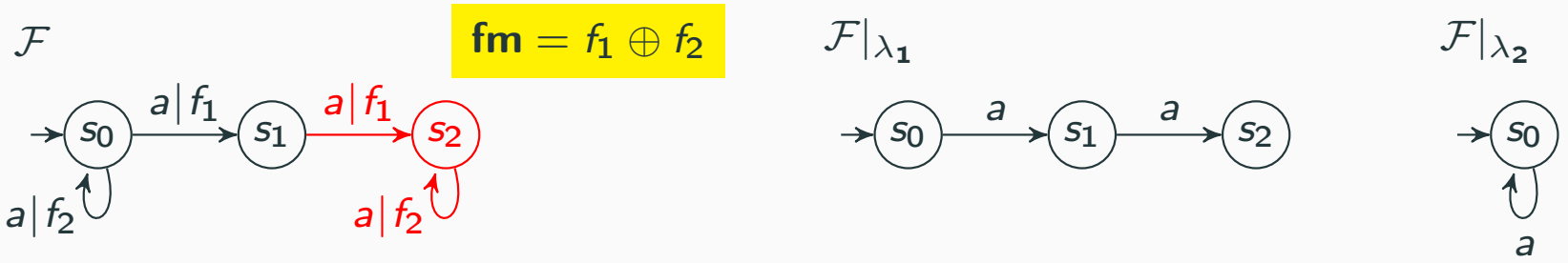
- $\texttt{is\_not\_dead\_transition}(t)$ is the formula

$$\texttt{is\_useful\_state}(t.\texttt{source}) \wedge t.\texttt{bx}$$

- $\texttt{may\_be\_opt\_transition}(t)$ is the formula

$$\texttt{is\_useful\_state}(t.\texttt{source}) \wedge \neg t.\texttt{bx}$$

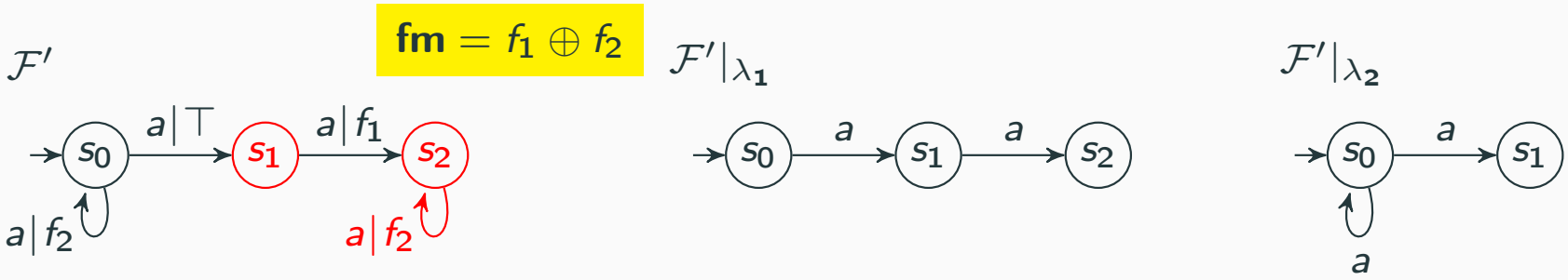
Ambiguities encoding (example 1)



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

1. $\texttt{exists\_deadlock}(s)$ *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 $\mathtt{fts}$ *represent the FTS* $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ *and* s *be a state:*

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

2. $\mathtt{is\_not\_dead\_transition}(t)$ *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 $\mathtt{fts}$ *represent the FTS* $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$ *and* s *be a state:*

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

2. $\mathtt{is\_not\_dead\_transition}(t)$ *is satisfiable iff there is* $\lambda \in \Lambda$ *s.t. the LTS transition corresponding to transition* t *is reachable in* $\mathcal{F}|_\lambda$

3. $\mathtt{may\_be\_opt\_transition}(t)$ *is satisfiable iff there is a* $\lambda \in \Lambda$ *s.t. the state* $t.\mathtt{source}$ *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 = ∅):
3          s.hdead ←False
4      else:
5          s.hdead ←check(exist_deadlock(s))
6
```

Ambiguities discovery algorithm

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

FTS ambiguity detection: NP-hard

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

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

is such that its (unique) transition is:

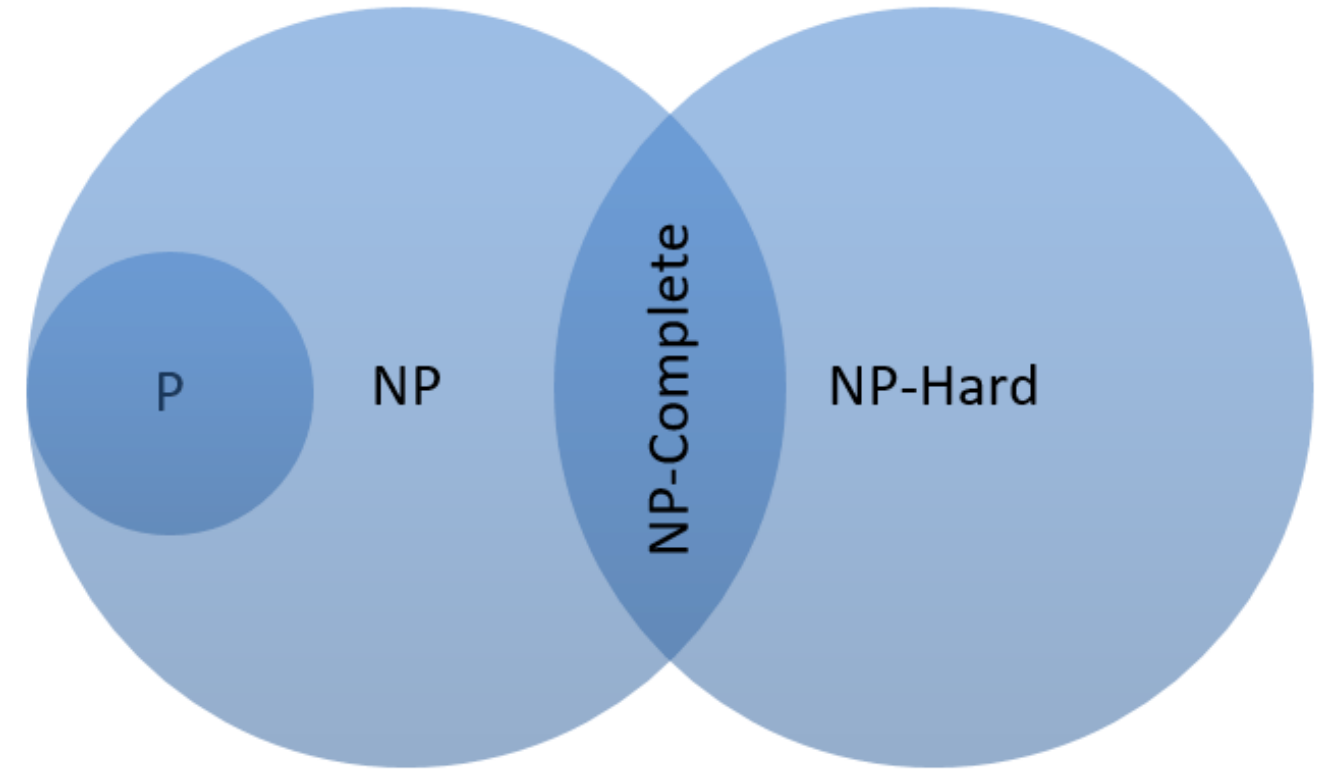
(i) dead iff ϕ is not satisfiable (i.e. $\neg\phi$ is valid)

(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 = ∅):
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)
```

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 artefact received the ACM reusable badge:

[SPLC'19]

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

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

Examples and benchmark experiments

Outline

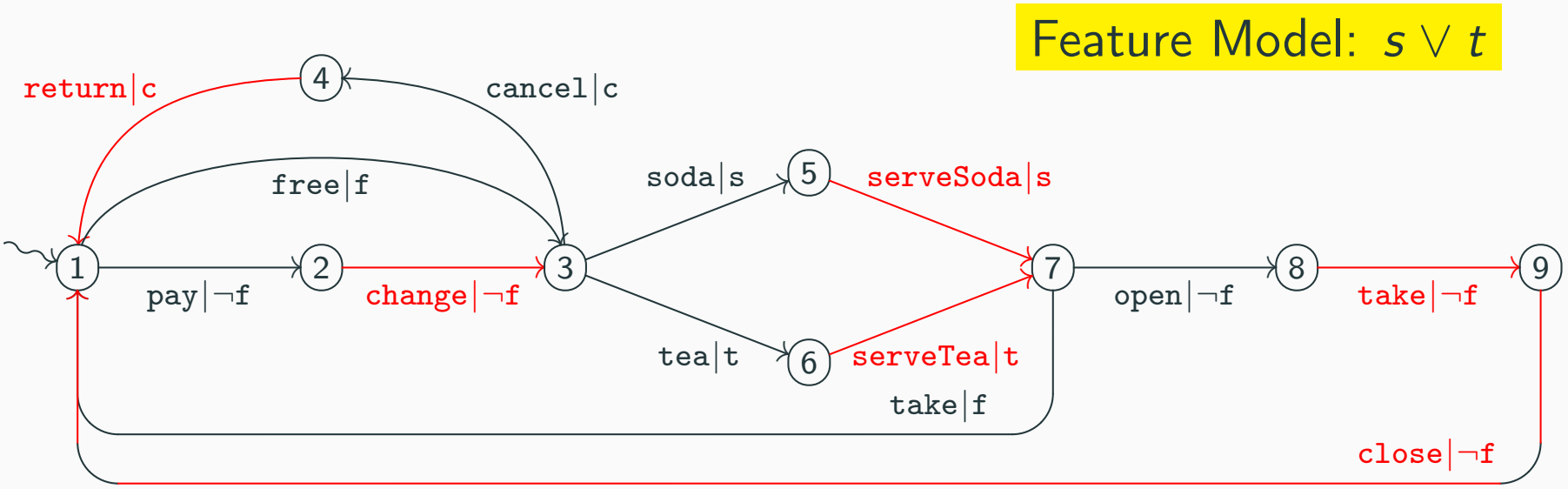
Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

Example static analysis: vending machine

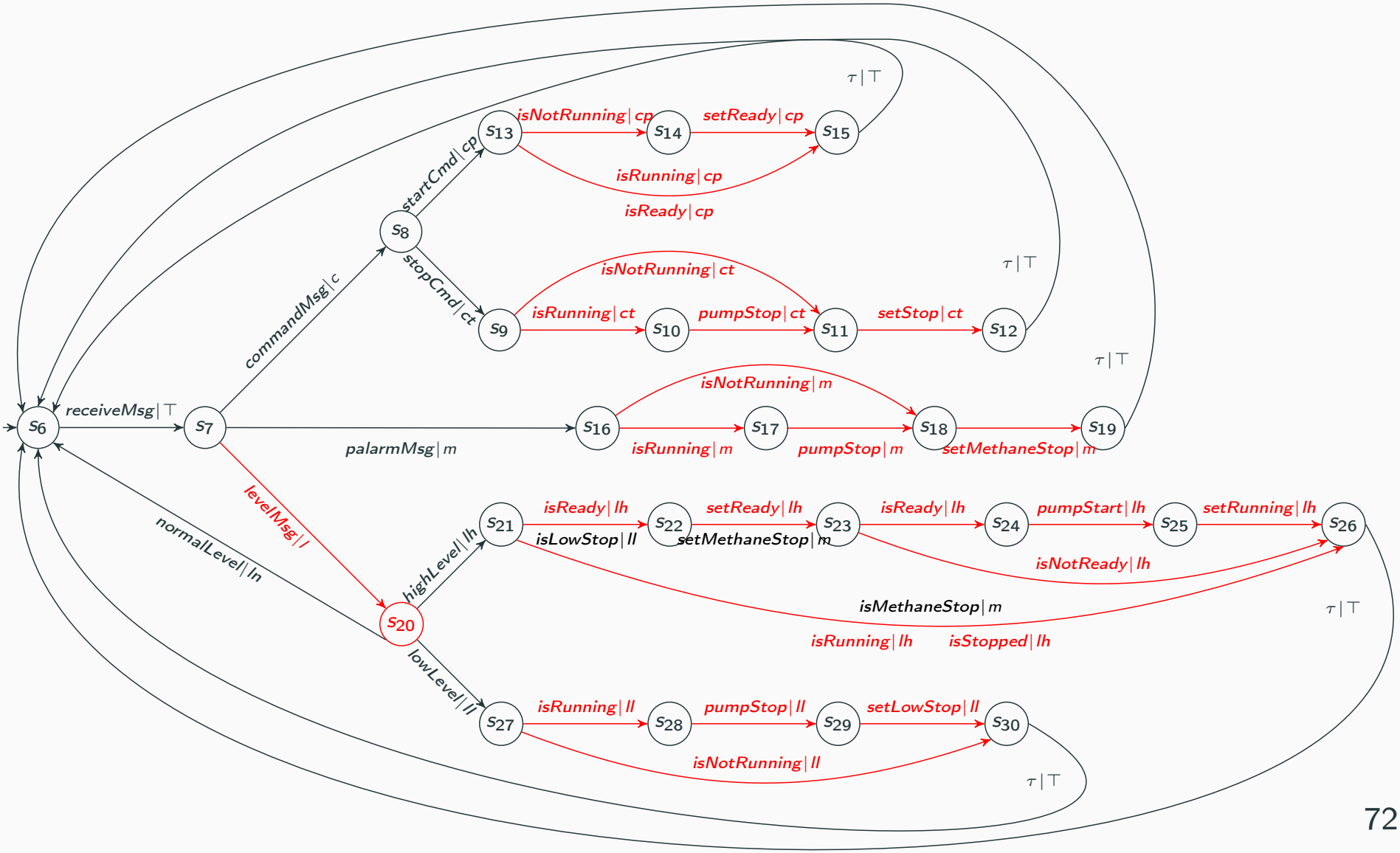


Result of static analysis on FTS

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

Example static analysis: mine pump (1/2)

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



Example static analysis: mine pump (2/2)

Result of static analysis on FTS

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

Benchmark experiments (1/3) [EMSE22]

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

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

Benchmark experiments (2/3) [SPLC19] vs. [EMSE22]

FTS	characteristics			computational effort				results
				implementation [SPLC19]		implementation [EMSE22]		
Model	$\|S\|$	$\|\delta\|$	$\|\Sigma\|$	runtime (s)	memory usage (Mb)	runtime (s)	memory usage (Mb)	runtime speedup
Vending machine Classen, PhD thesis, 2011	9	13	12	0.92	38.230	0.26	29.765	3.54x
Coffee machine Asirelli *et al.* @ SPLC'11	14	23	15	2.822	40.140	0.29	30.305	9.72x
Soup component Belder *et al.* @ FMSPLE'15	13	28	18	2.544	40.870	0.316	30.85	8.05x
Mine pump (system) Classen, PhD thesis, 2011	25	41	22	2.192	41.899	0.344	31.704	6.37x
Mine pump (controller) Classen, PhD thesis, 2011	77	104	22	8.12	49.091	0.548	36.295	14.82x
Coffee/Soup machine Belder *et al.* @ FMSPLE'15	182	691	33	timeout	–	37.766	119.427	>7200.00x
Mine pump (complete) Classen, PhD thesis, 2011	418	1,255	26	timeout	–	98.994	119.127	>7200.00x
Claroline Devroey *et al.* @ VaMoS'14	107	11,236	106	timeout	–	2413.8	2010.229	>7200.00x

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

Benchmark experiments (3/3) [liveness only]

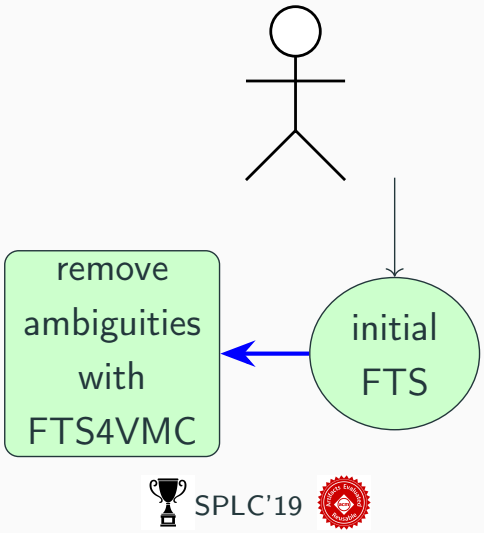
FTS	characteristics			computational effort				results
				full-fledged implementation		specialised implementation		
Model	$\|S\|$	$\|\delta\|$	$\|\Sigma\|$	runtime (s)	memory usage (Mb)	runtime (s)	memory usage (Mb)	runtime fraction
Coffee/Soup machine Belder *et al.* @ FMSPLE'15	182	691	33	37.766	119.427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

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

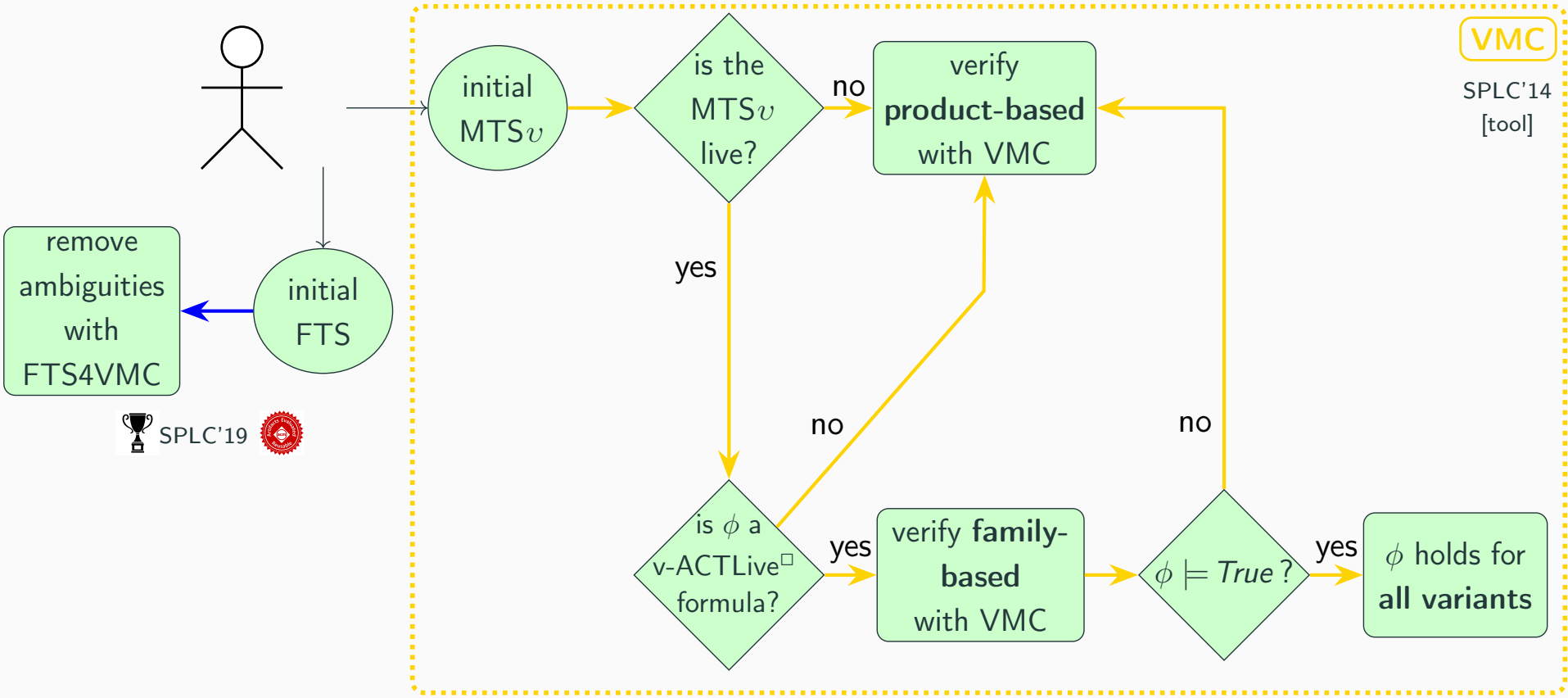
Wrap up of Part I. . .

. . . and outlook on Part II

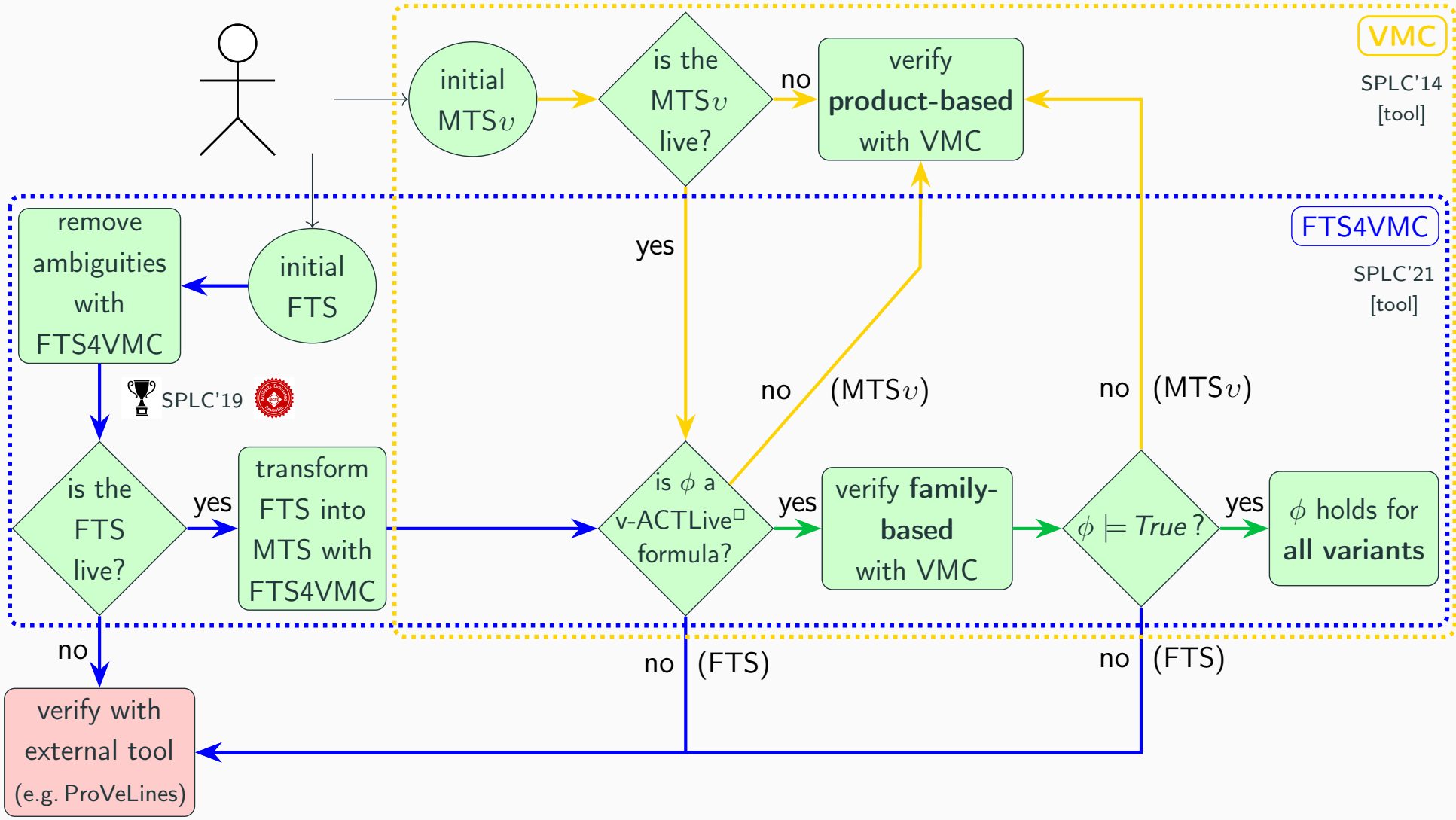
Detect and remove ambiguities



VMC: model checking MTSυ



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



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

Part II

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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

Outline

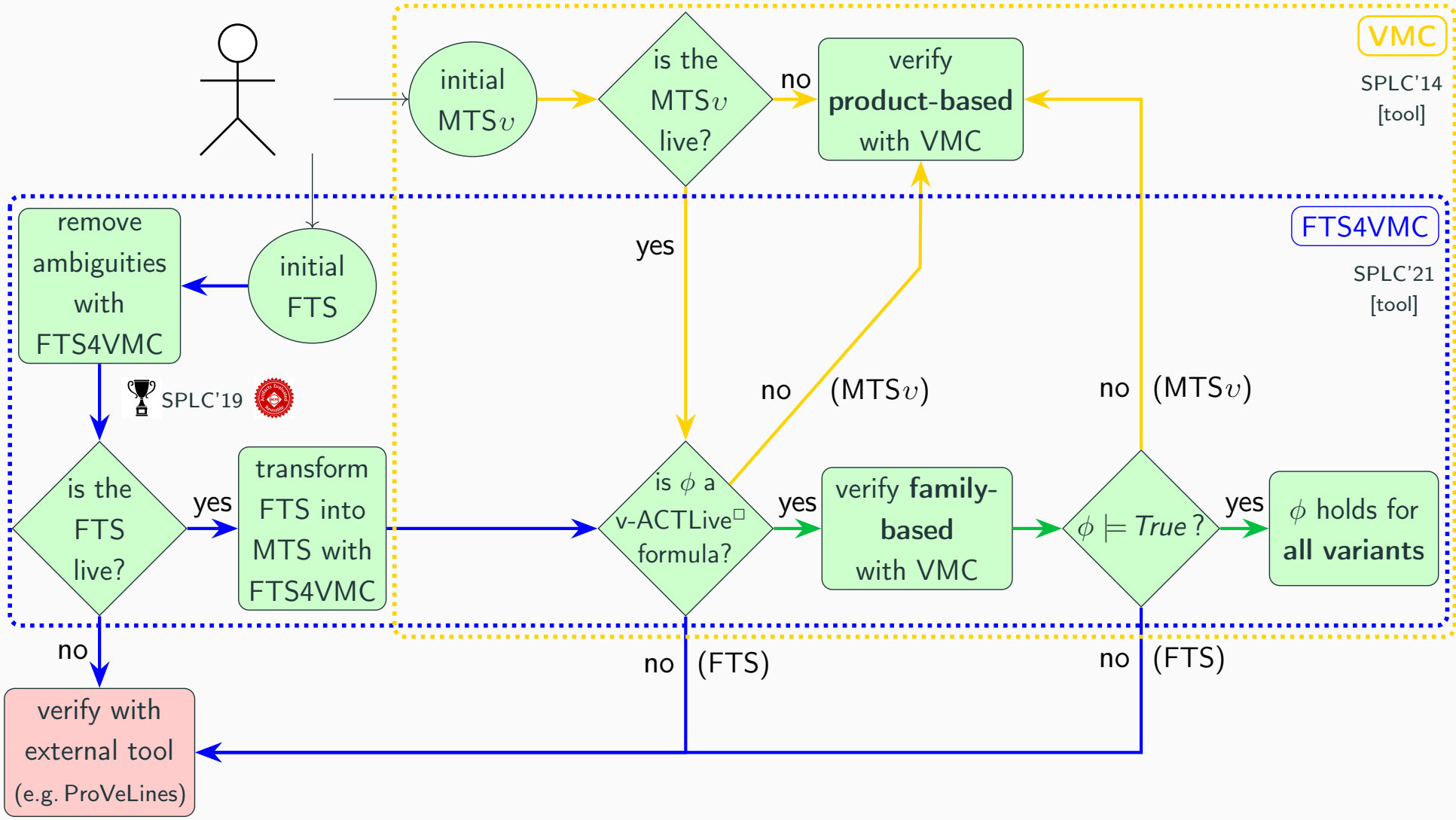
Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

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



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

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

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

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

MTSυ: MTS with...

Recall:

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

A Modal Transition System with variability constraints (MTSυ) is a sextuple $(S, \Sigma, s_0, \rightarrow_\Box, \rightarrow_\Diamond, \Upsilon)$ such that $(S, \Sigma, s_0, \rightarrow_\Box \cup \rightarrow_\Diamond)$ is an LTS

MTSυ: MTS with...

Recall:

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

A Modal Transition System with variability constraints (MTSυ) is a sextuple $(S, \Sigma, s_0, \rightarrow_\Box, \rightarrow_\Diamond, \Upsilon)$ such that $(S, \Sigma, s_0, \rightarrow_\Box \cup \rightarrow_\Diamond)$ 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 $\rightarrow_\Box \subseteq S \times \Sigma \times S$: necessary transitions

By definition, any necessary transition is also admissible: $\rightarrow_\Box \subseteq \rightarrow_\Diamond$

We denote the transitions $\dashrightarrow \equiv \rightarrow_\Diamond \setminus \rightarrow_\Box$ as optional transitions

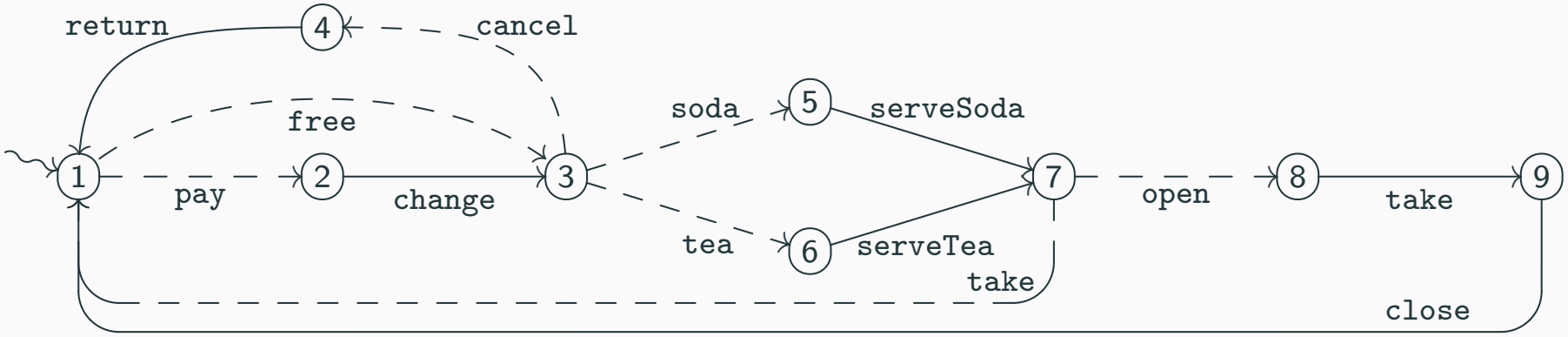
MTSυ: MTS with...

Recall:

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

A Modal Transition System with variability constraints (MTSυ) is a sextuple $(S, \Sigma, s_0, \rightarrow_\Box, \rightarrow_\Diamond, \Upsilon)$ such that $(S, \Sigma, s_0, \rightarrow_\Box \cup \rightarrow_\Diamond)$ is an LTS

Practically:



plus an additional set of variability constraints Υ, defined next

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

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

b_1 OR $\cdots$ OR b_n, **where b_i is either a_i or $\neg\, a_i$** : at least one among the conditions on $n \geq 2$ actions $b_1, ..., b_n$ 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 $\mathcal{L}$)

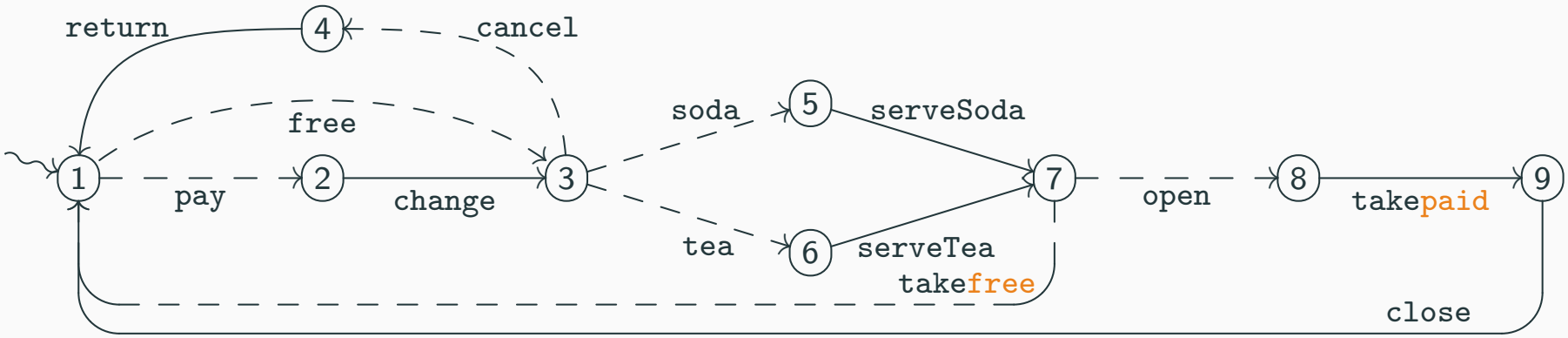
a_1 REQ a_2 : action a_2 is reachable whenever a_1 is reachable (in $\mathcal{L}$)

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

a_1 REQ (a_2 OR $\cdots$ OR a_n) : at least one among the $n \geq 2$ actions $a_2, ..., a_n$ 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)

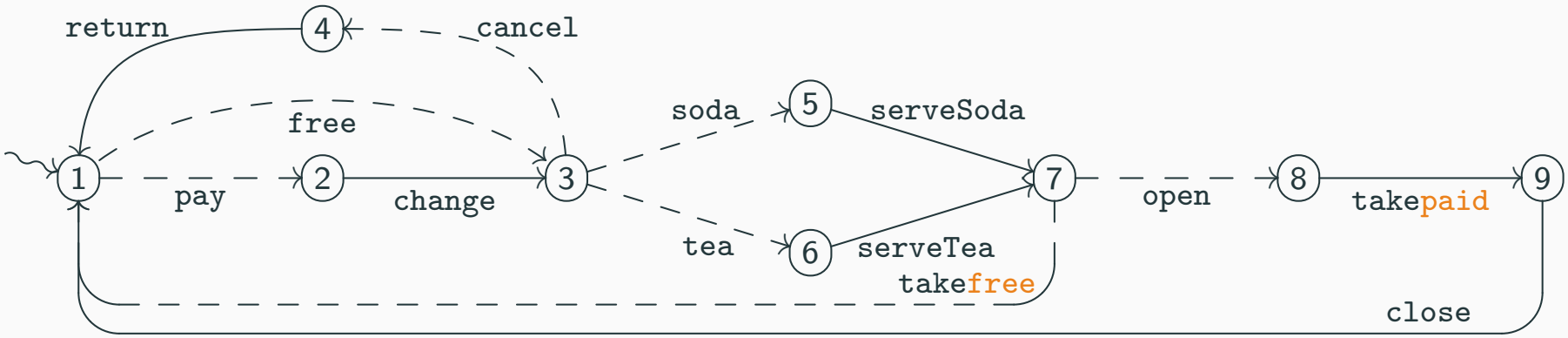
MTSv of example SPL (example from VMC)




```
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
```

MTSv of example SPL (example from VMC)




```
...

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
}
```

Derive products (implemented in VMC)

Intuitively, a product LTS is obtained from a family MTS υ 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

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

Derive products (implemented in VMC)

Intuitively, a product LTS is obtained from a family MTS υ 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

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

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

Derive products (implemented in VMC)

Intuitively, a product LTS is obtained from a family MTS υ 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

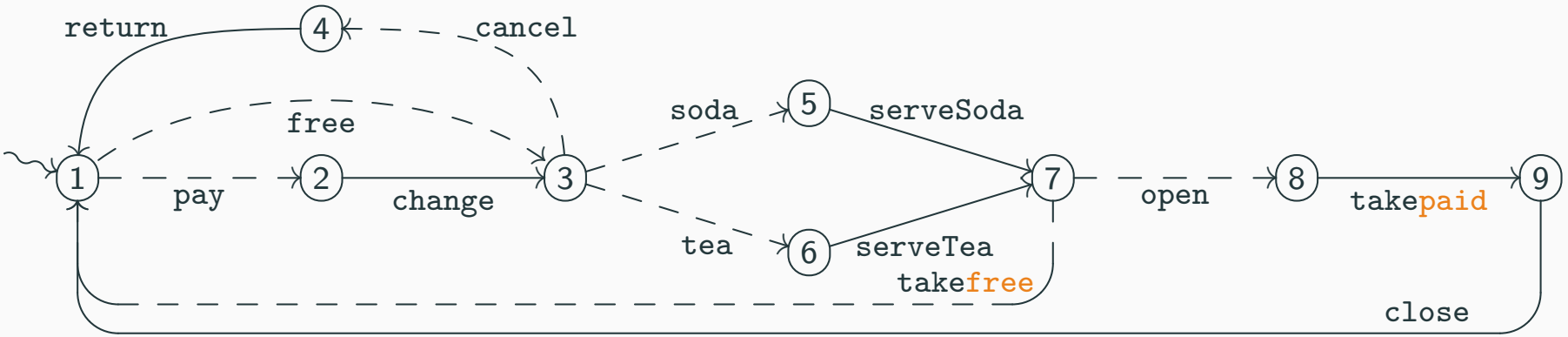
$\Rightarrow$ Each selection gives rise to a different variant

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

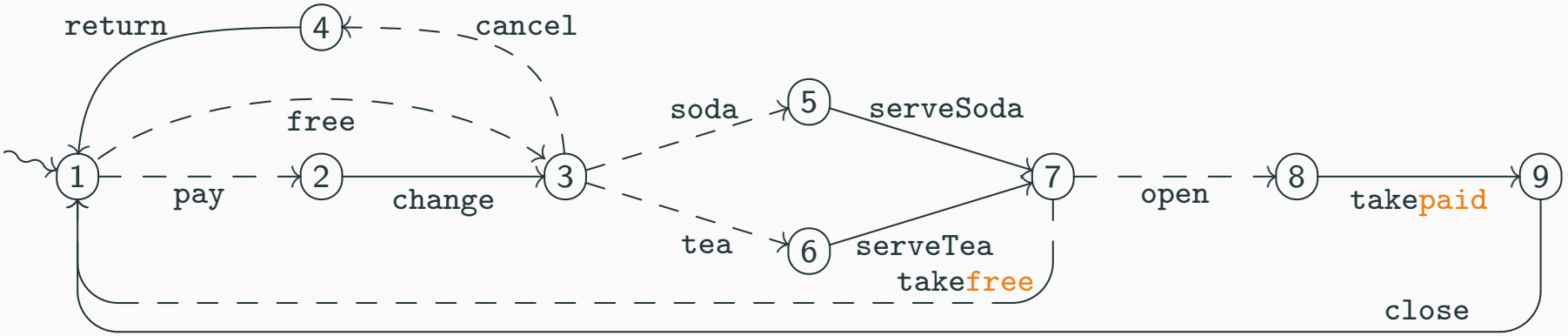
Then the set $\{\mathcal{P}_i = (S_i, \Sigma, s_0, \delta_i) \mid i > 0\}$ of product LTSs is obtained by considering each pair of $S_i \subseteq S$ and $\delta_i \subseteq \delta^{\Diamond} \cup \delta^{\Box}$ to be defined s.t.

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

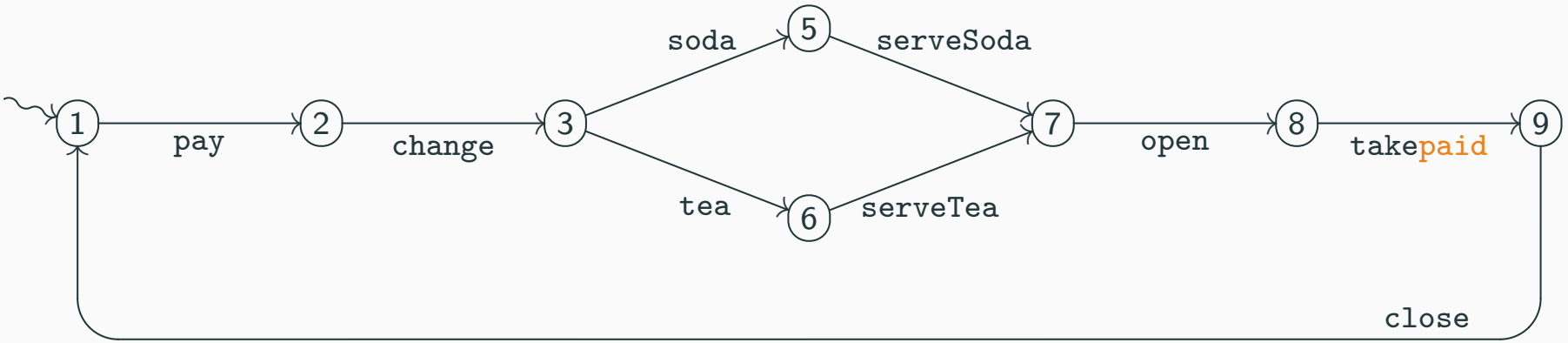
Product LTSs of MTSυ of example SPL



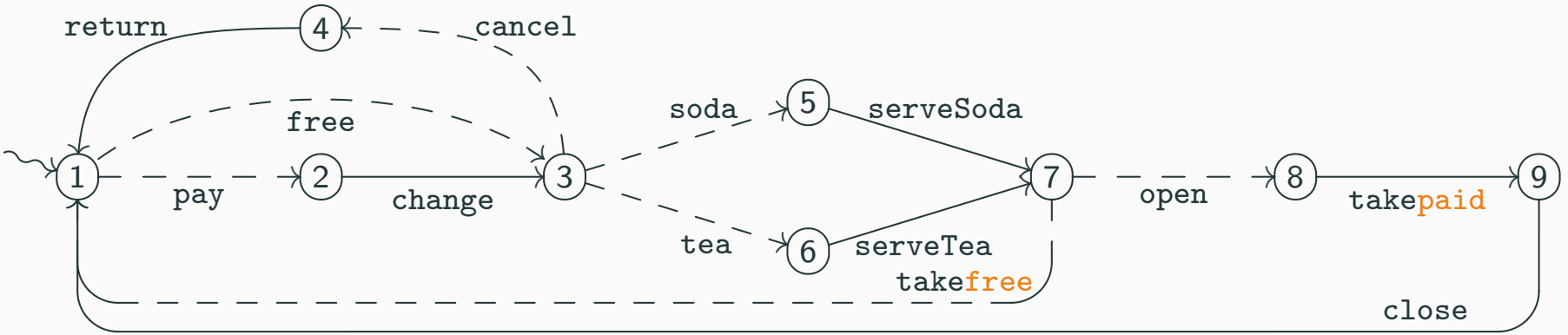
Product LTSs of MTSυ of example SPL

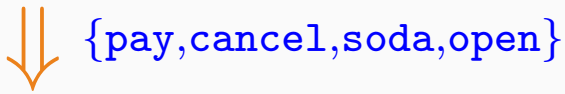


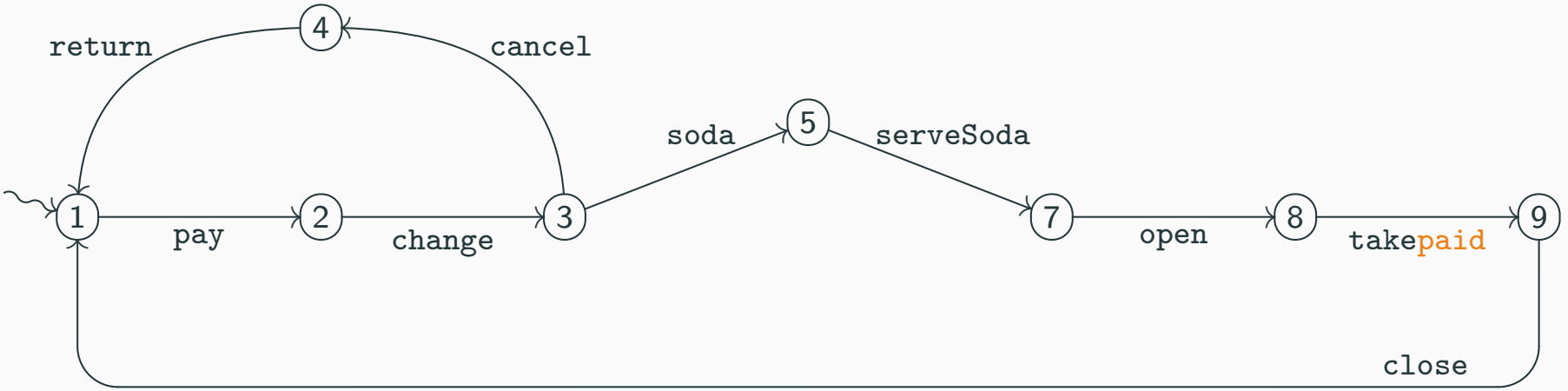
{pay,soda,tea,open}



Product LTSs of MTSυ of example SPL







MTSυs are equally expressive as FTSs

Theorem (FTS2MTSυ transformation sound and complete)

Let $\mathcal{F}$ be an FTS and let $\mathcal{M}$ be the MTSυ generated from $\mathcal{F}$ according to the FTS2MTSυ model transformation algorithm.

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

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

MTSυs are equally expressive as FTSs

Theorem (FTS2MTSυ transformation sound and complete)

Let $\mathcal{F}$ be an FTS and let $\mathcal{M}$ be the MTSυ generated from $\mathcal{F}$ according to the FTS2MTSυ model transformation algorithm.

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

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

Theorem (MTSυ2FTS transformation sound and complete)

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

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

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

Model checking principles

Outline

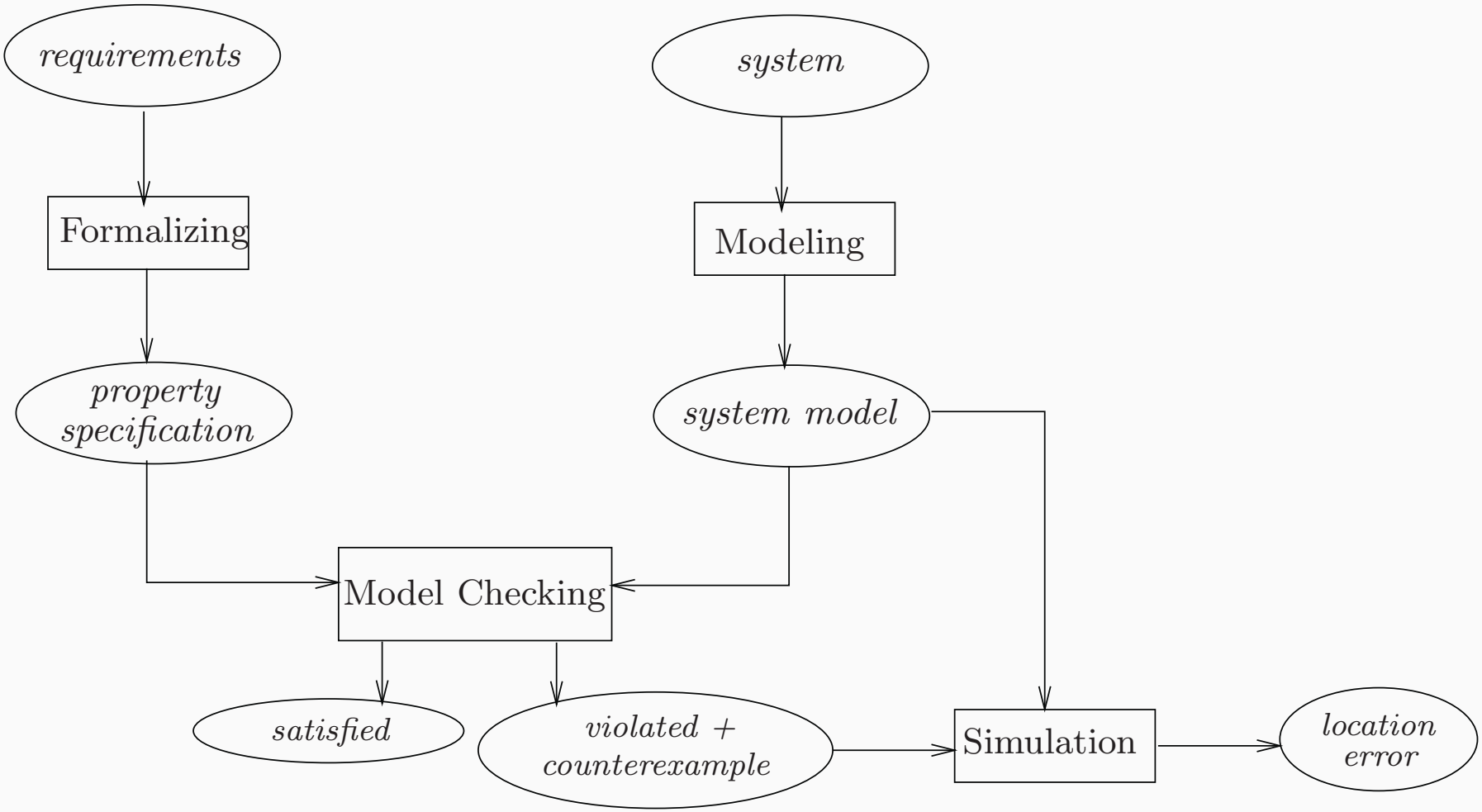
Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

What is model checking?



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

Model checking process

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

Strengths and weaknesses

- 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

- 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

- 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

- 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-ACTL

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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

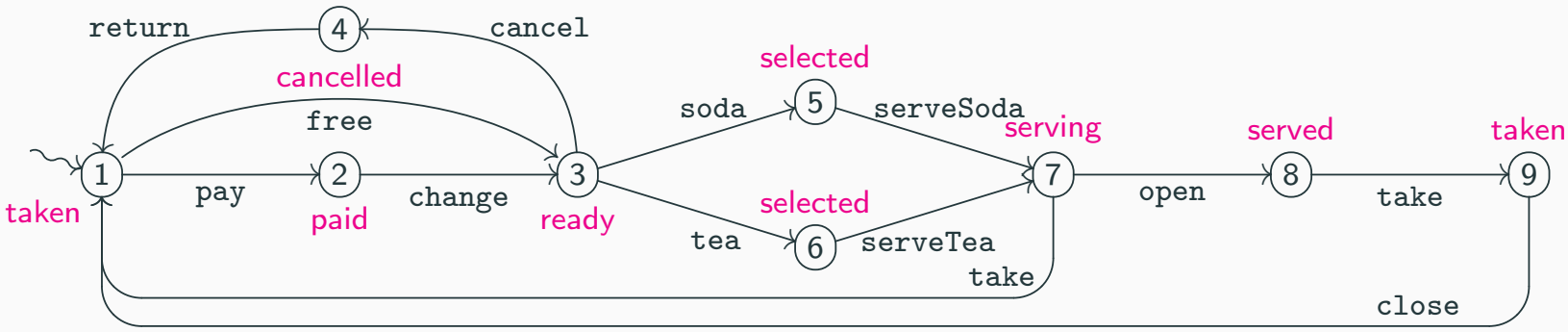
Part II

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

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 $L : S \to 2^{AP}$ is a function labelling each state with a set of AP (here singleton)

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

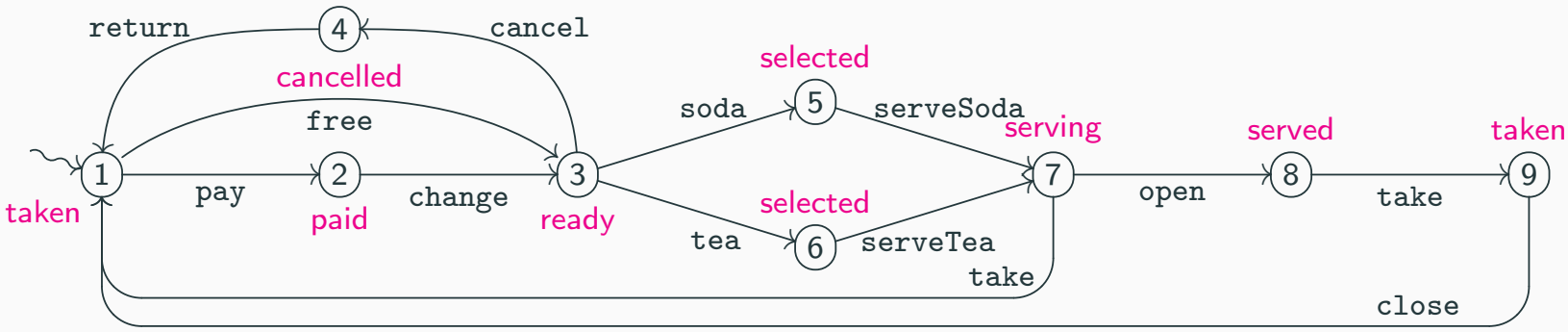


Kripke structure: only state labelling, no transition labelling

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 $L : S \to 2^{AP}$ is a function labelling each state with a set of AP (here singleton)

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

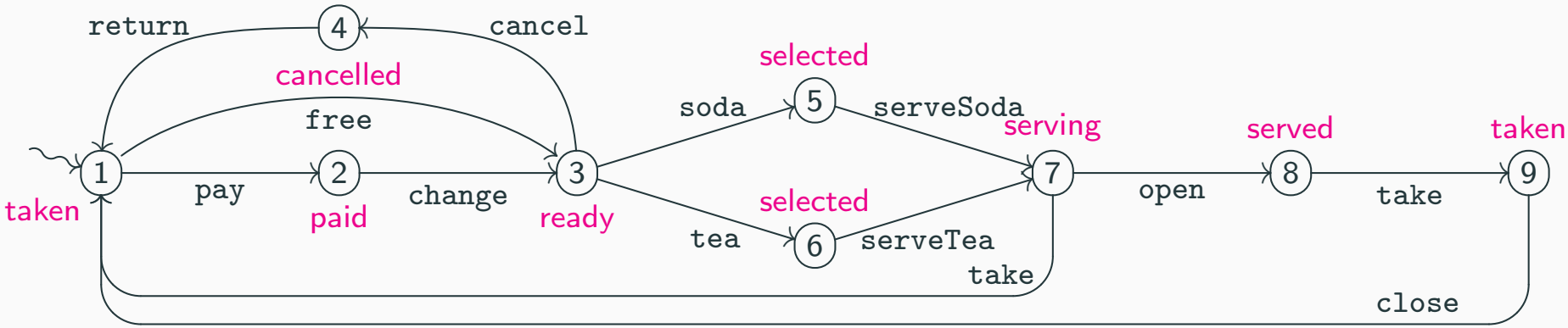


AP	paid	ready	...	served	taken
L	2	3	...	8	9
	paid	ready		served	taken

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:

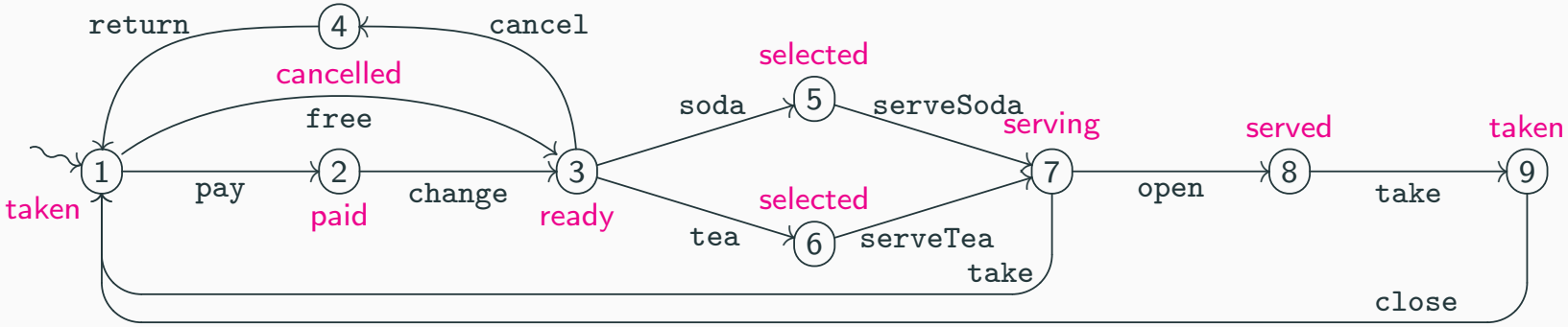
- Independently, on all elements of the set of possible execution sequences, without considering the branching structure of the system evolutions
- Or considering the actual branching structure of the system evolutions

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

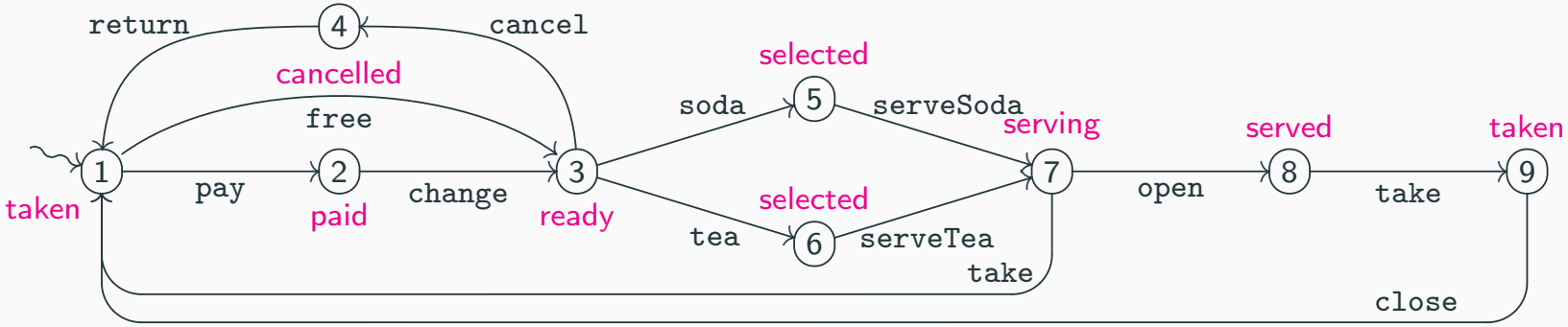


$G\ (serving \Rightarrow (F\ taken))$: for all paths, it is always (globally) true that a state satisfying serving is eventually followed by a (future) state satisfying taken

Temporal logic operators: linear-time examples

Linear Temporal Logic (LTL)

Pnueli, The temporal logic of programs @ FOCS'77



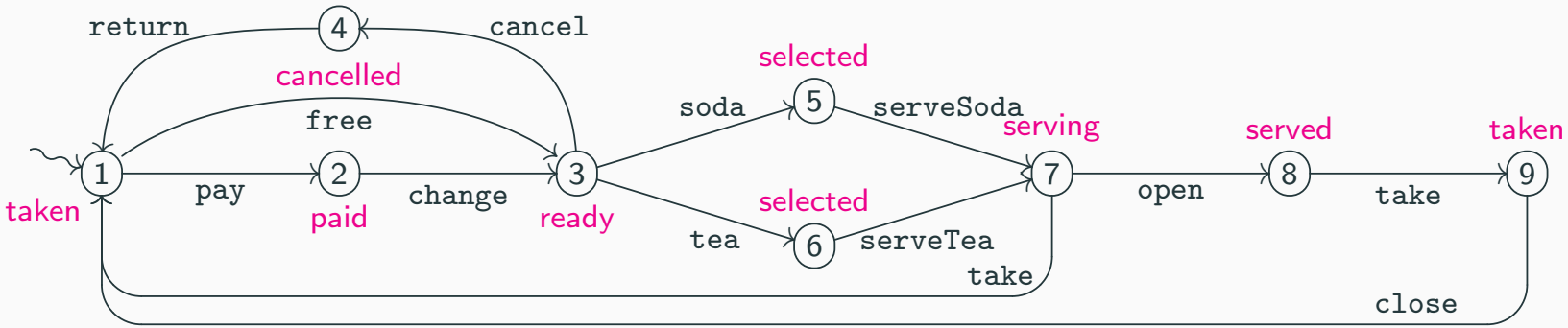
$G\ (serving \Rightarrow (F\ taken))$: for all paths, it is always (globally) true that a state satisfying serving is eventually followed by a (future) state satisfying taken

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

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

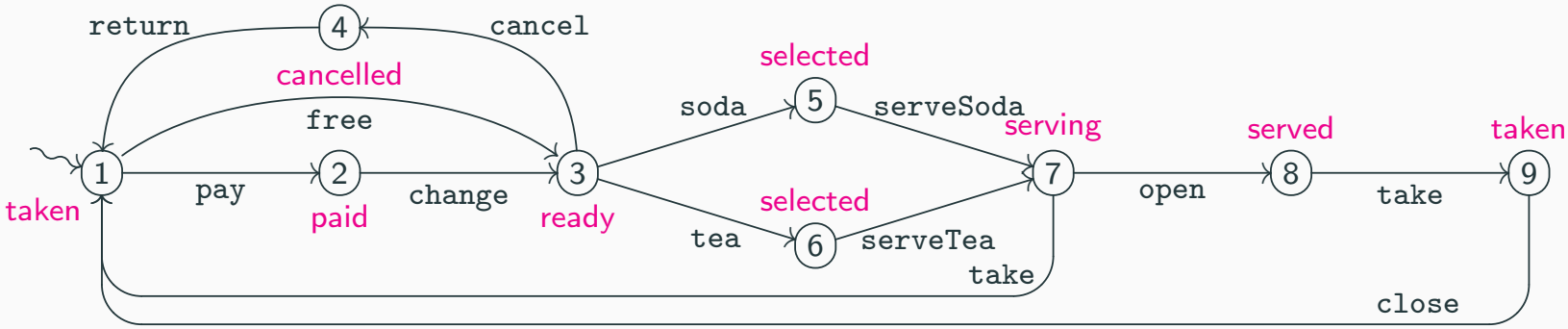


AG EF taken: for all (A) paths it is globally (G) true that there exists (E) a possibility to eventually (F) reach a (future) state satisfying taken

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 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

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

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

---

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

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.:

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

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

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

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

χ denotes a Boolean composition of actions (transition labels)

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.:

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

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

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

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

v-ACTLive$^{\Box}$ is a specific subset of v-ACTL with the following property: any formula that is true for a live MTS υ, is also true for all product LTSs

---

χ denotes a Boolean composition of actions (transition labels)

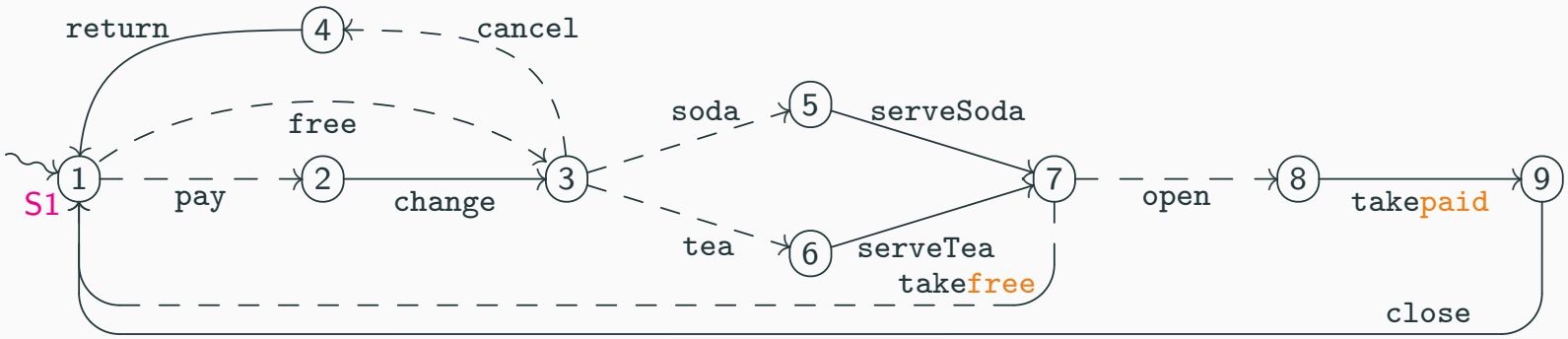
v-ACTLive$^{\Box}$:

1. Universal (A) properties that hold for all execution paths (thus also for the subset of MTSυ 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$^{\Box}$: family-based model checking with VMC

1. Universal (A) properties that hold for all execution paths (thus also for the subset of MTSυ 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$^{\Box}$ 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. $[pay]\ EX^{\Box}_{change}$ (also $[pay]\ AX_{change}$, but not $[pay]\ EX_{change}$ seen before)
3. $AG\ [tea]\ \neg E\ [\ _{cancel}U\ S1]$

VMC

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

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

- 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 CNR–ISTI (`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

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 MTSυs:

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

. . . with a linear complexity

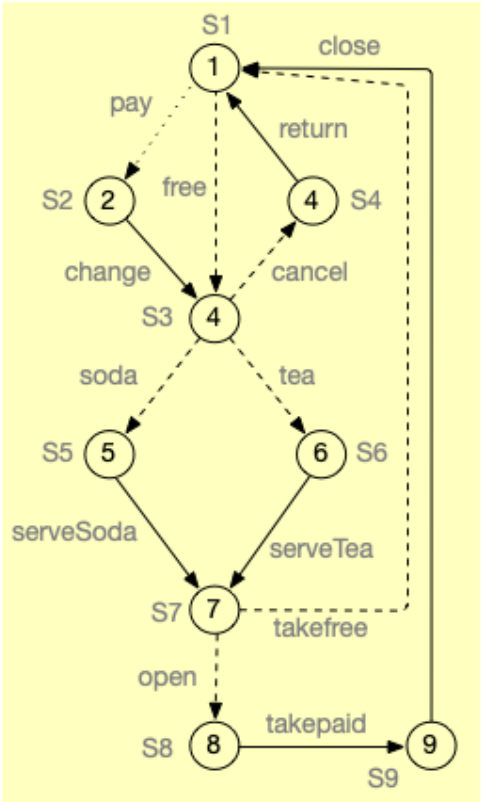
Textual model encoding as process definitions [recall]

VMC V6.5
(2019)

New Model ...
Load Current Model
Welcome
Quit


Kandinsky 1908

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



Graphical model displayed as MTSv

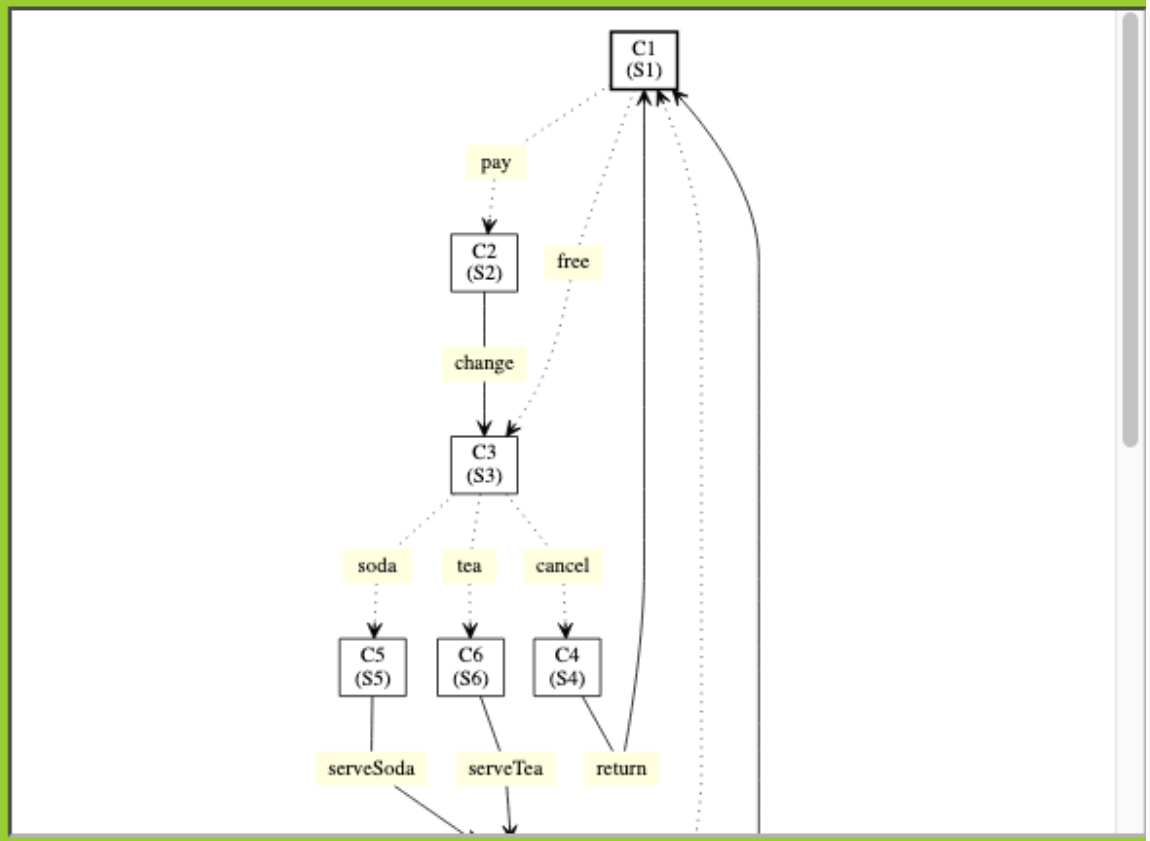
VMC V6.5
(2019)

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

Kandinsky 1919

Abstract Model Evolutions Chart

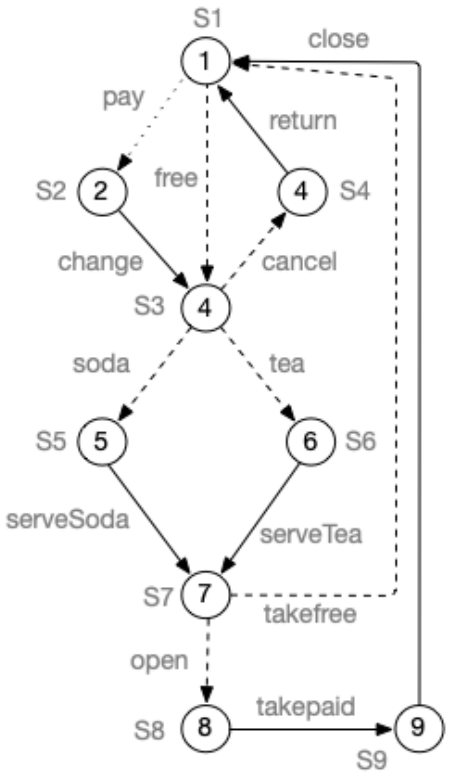
Zoom Out | Zoom In



Click over a node to view the detaials of the corresponding system configuration.
View the graph in DOT format or as a PDF pdf picture or as plain SVG data.

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

Those holding for all possible executions—and making use of a live fragment of the MTSυ (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 {serveSoda or serveTea or cancel}
- each pay action is always eventually followed by a takepaid action
  AG [pay] AF {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
  [free] A [{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

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 or (serveTea or cancel)} 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 or serveTea 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


Kandinsky 1908

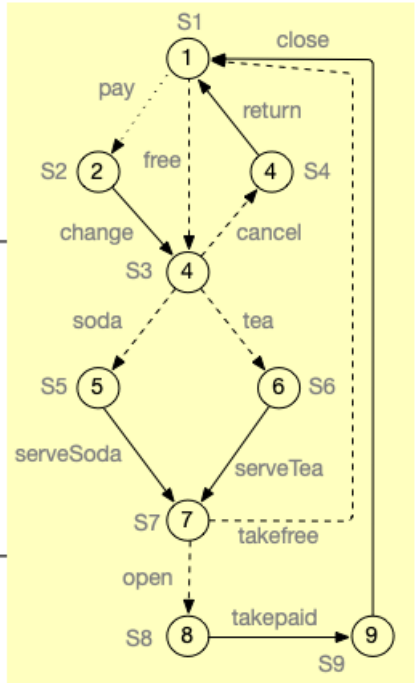
☒ **The formula:**
AG [pay] AF {takepaid} true
is FOUND_FALSE in State C1

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 --> 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



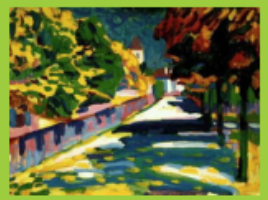
This happens because there is at least one maximal path from C2 in which all transitions DO NOT satisfy the action expression *takepaid*.
For example:
C2 --> C3 {change}
C3 --> C4 {cancel, may}
C4 --> C1 {return}
C1 --> C2 {may, pay} (C2 closes a loop)
is one of the above mentioned failing paths.

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 LTS

VMC V6.5
(2019)

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



Kandinsky 1908

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

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

Logic Formula for all Products

[pay] AF {takePaid} true

Check The Formula | Explain the Result

Family-based model checking FTSs

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

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

Note

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

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

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

Note

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

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

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

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

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

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

Note

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

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

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

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

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

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

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

Outline

Part I

- Software Product Line Engineering (SPLE): variability, features
- 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 (MTSυ)
- Model checking principles, temporal logic: LTL, (A)CTL, v-ACTL
- VMC: family-based model checking of MTSs
- Family-based model checking of FTSs
- FTS4VMC toolchain

What is it?

A research tool developed in Python to model check properties on FTSs expressed in v-ACTLive$^{\square}$ 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$^{\square}$ in a family-based manner with VMC

Where can I get it?

Source code is available on GitHub

Preconfigured Docker image on DockerHub

What does it do?

What does it do?

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

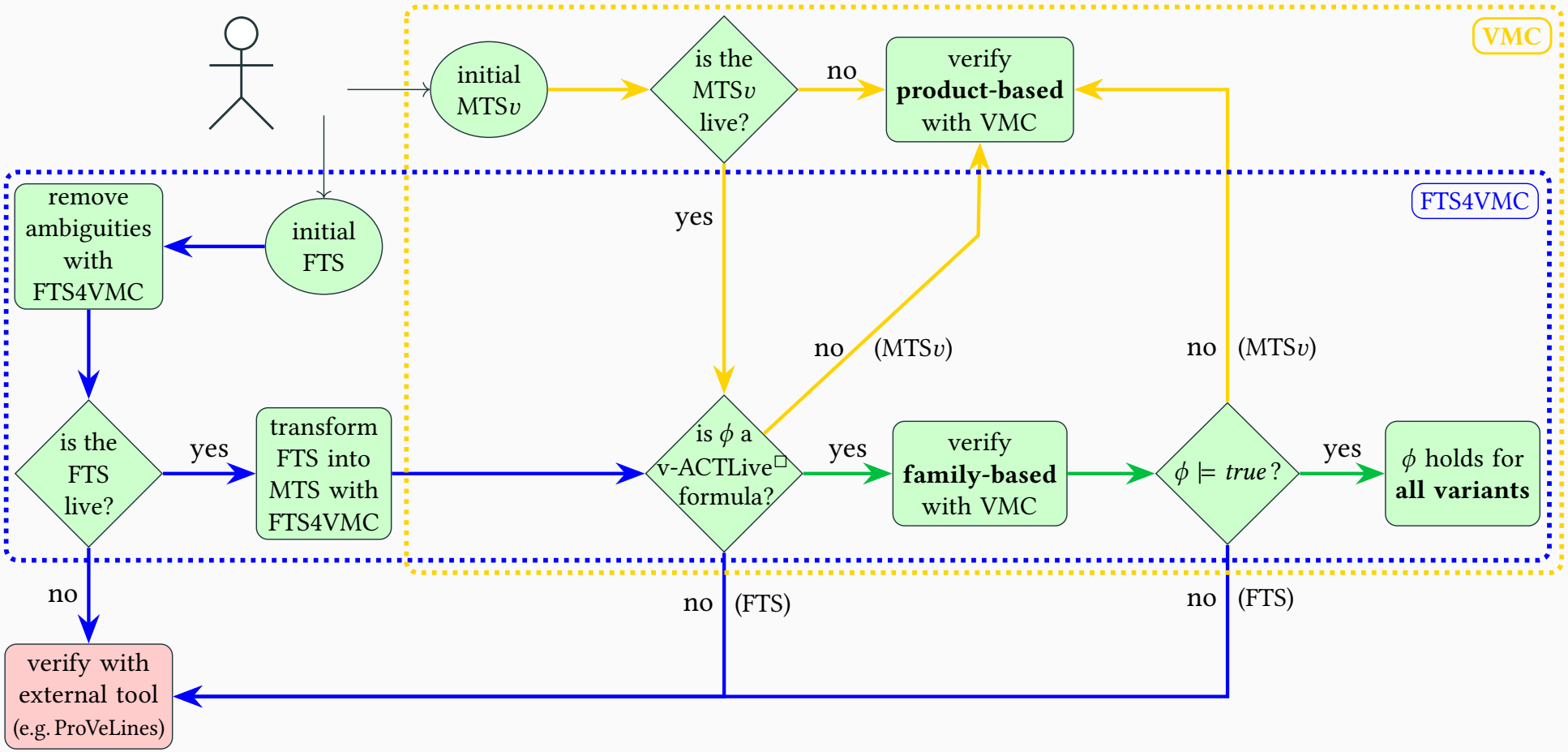
What does it do?

1. Provide a simple UI to work with the proposed toolchain
2. Implements automatic ambiguities removal as shown before

What does it do?

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

Toolchain: family-based model checking of MTSυs

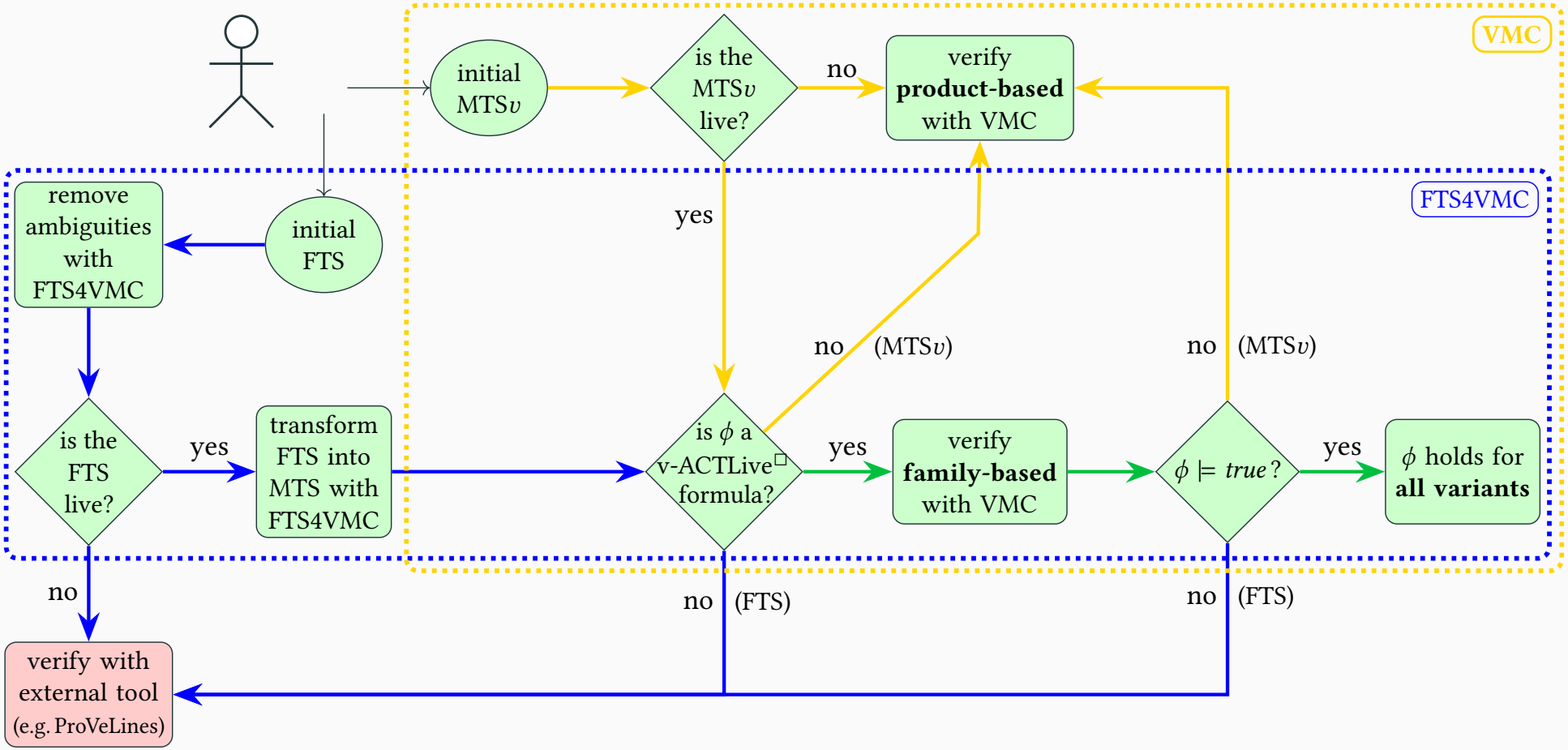


The green blocks are automated by the toolchain (FTS4VMC+VMC)

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

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

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



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

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

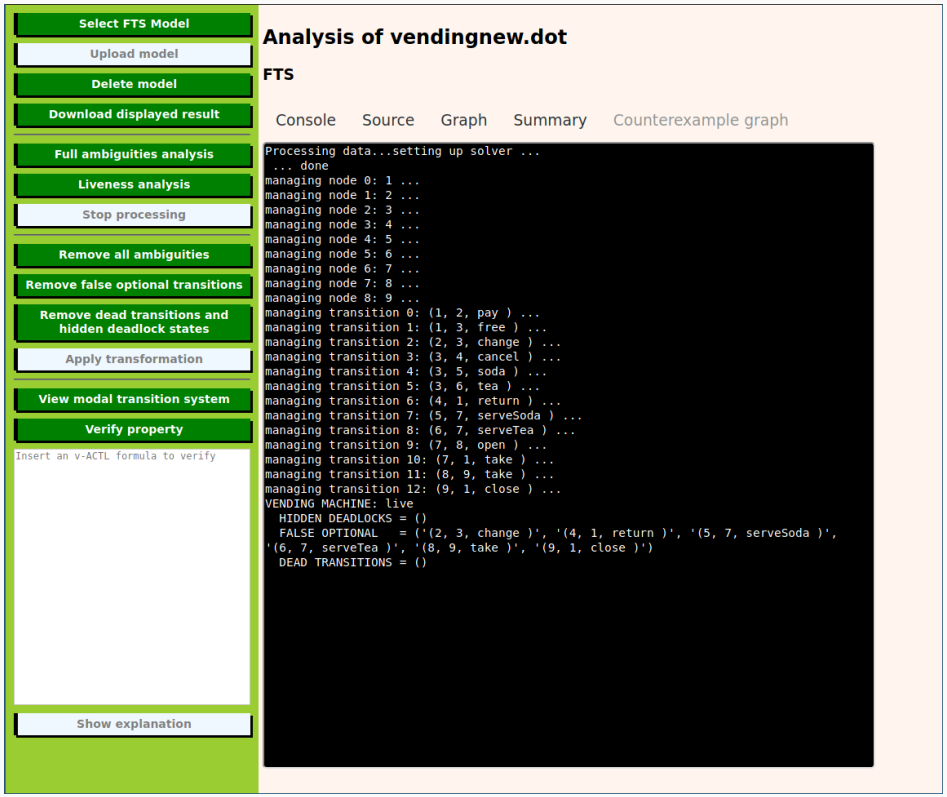
The FTS4VMC implementation is freely available online: https://github.com/fts4vmc/FTS4VMC

FTS4VMC: static analyser integration (1/2)

The static analyser is used inside FTS4VMC to detect the 3 types of ambiguities that may appear inside an FTS

After clicking on Full ambiguities analysis or Liveness analysis, the analyser's output is shown in the console tab

The analyser's `load_dot` method is used to determine wether the uploaded dot file contains an FTS



FTS4VMC: static analyser integration (2/2)

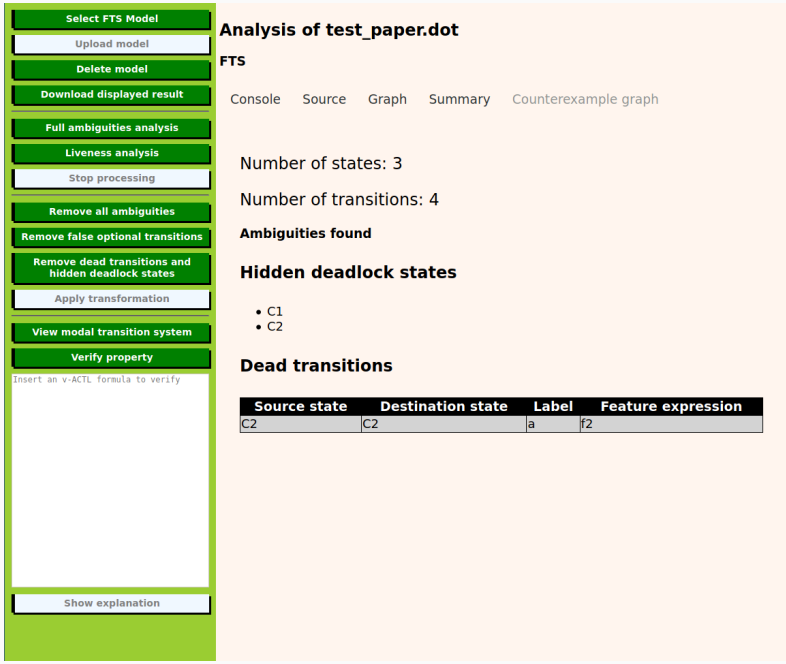
Ambiguities found are stored into dictionaries

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

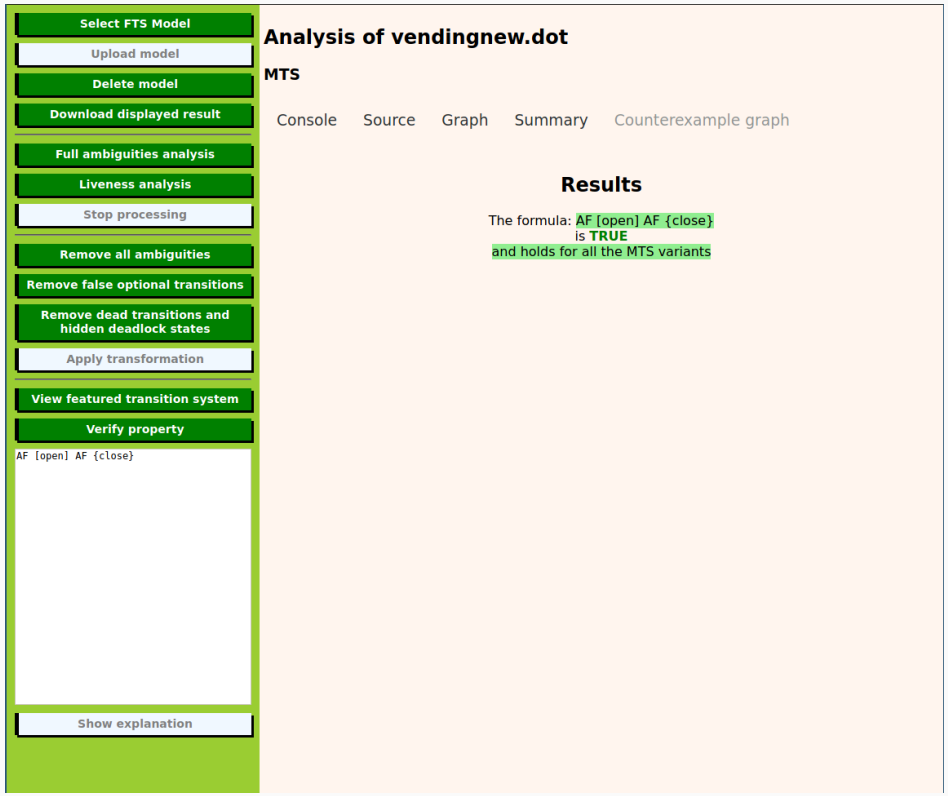
Upon analysis, FTS4VMC returns a JSON object called ambiguities that contains 3 arrays, one for each type of ambiguity



FTS4VMC: VMC integration (1/5)

VMC is used inside FTS4VMC to verify properties

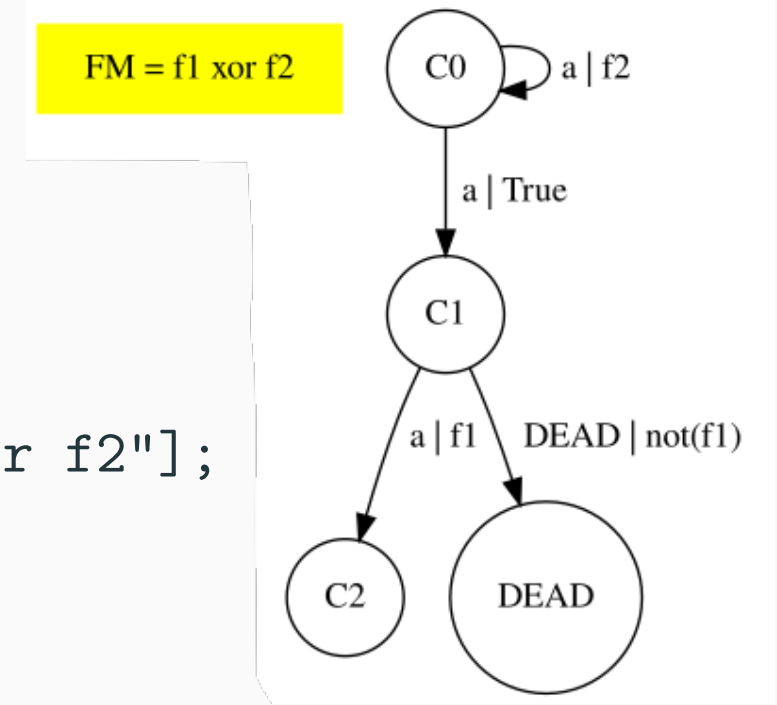
Since VMC is developed in Ada, the current implementation spawns a sub-process that executes VMC on an MTS model and property stored into a file and then parses the results to extract the elements used inside the UI



FTS4VMC: VMC integration (2/5)

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

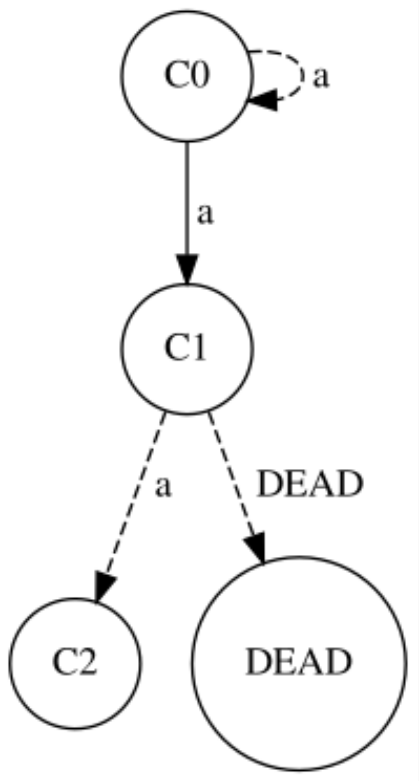
```
digraph TEST {
  FM="f1 xor f2";
  name="TEST";
  rankdir=TB;
  node [shape=circle];
  FeatureModel [label="FM = f1 xor f2"];
  C0 [initial=True];
  C0 -> C0  [label="a | f2"];
  C0 -> C1  [label="a | True"];
  C1 -> C2  [label="a | f1"];
  C1 -> DEAD  [label="DEAD | not(f1)"];
}
```



FTS4VMC: VMC integration (3/5)

```
C0 = a(may).C0 + a(must).C1
C1 = a(may).nil + DEAD(may).nil
SYS = C0
Constraints { LIVE }
```

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

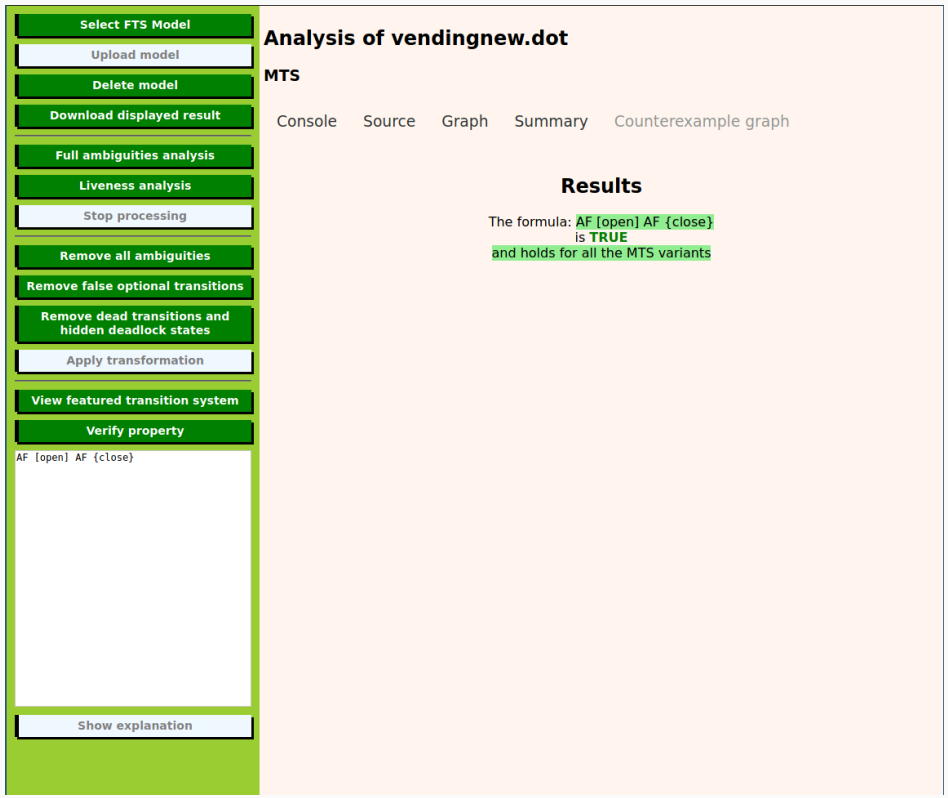


FTS4VMC: VMC integration (4/5)

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

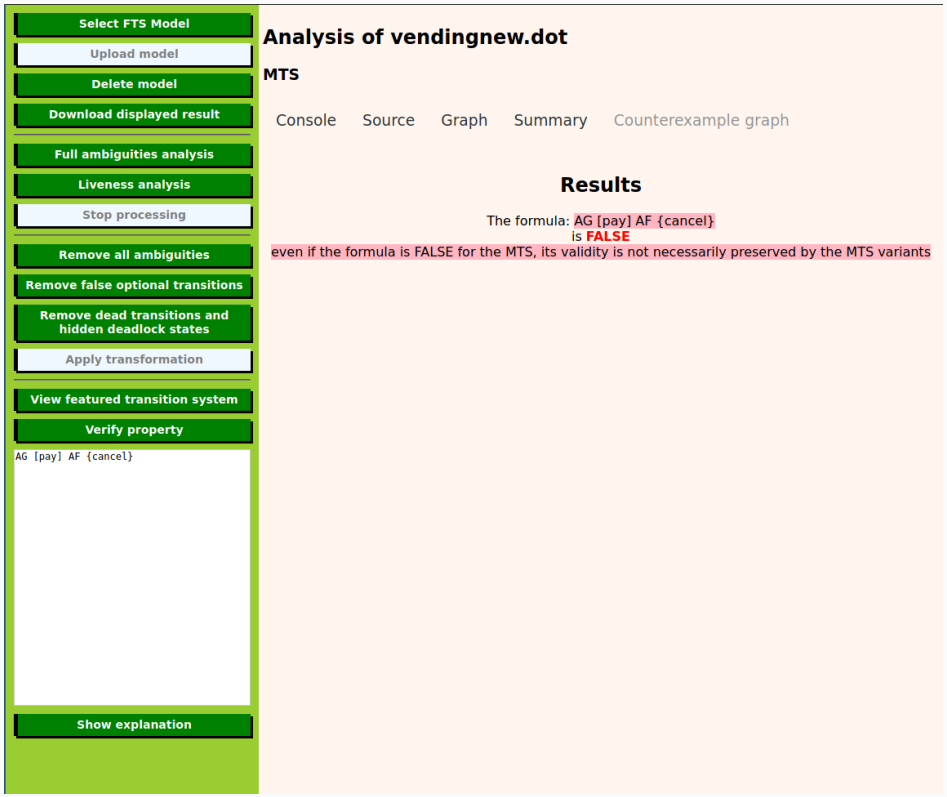
```
vmc mts-model -z prop.txt
```

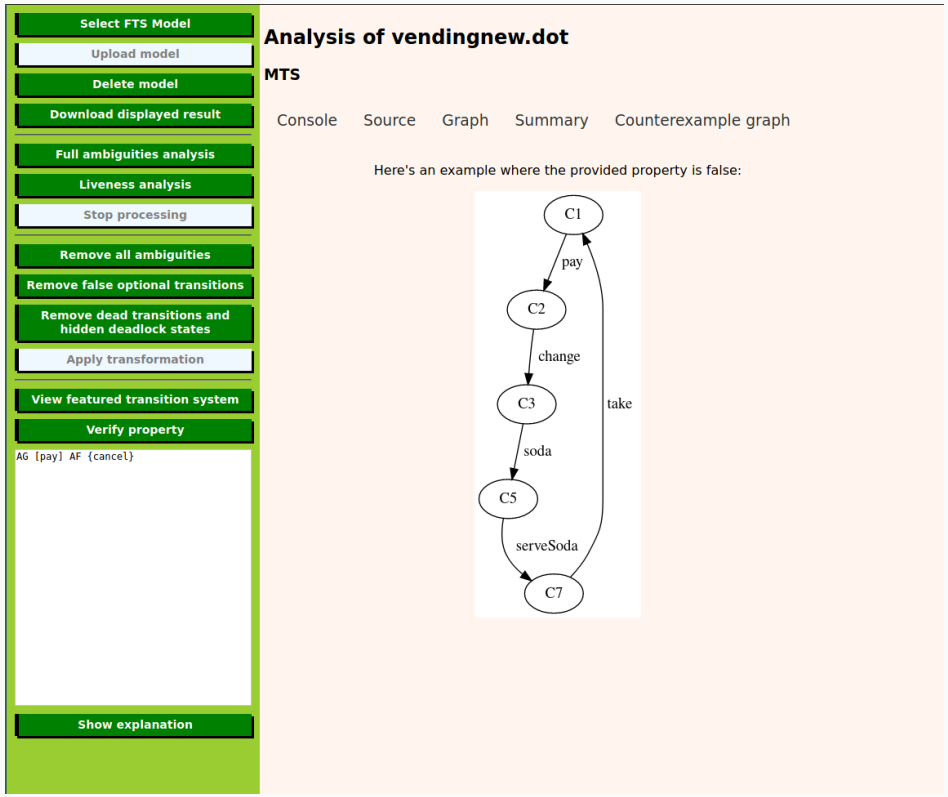
where `prop.txt` is a file that contains a property expressed in v-ACTLive$^{\square}$ and `mts-model` is a file that contains an MTS in the previously shown format



FTS4VMC: VMC integration (5/5)

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





Conclusion and outlook

Conclusion and outlook

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

Conclusion and outlook

1. Efficient static analysis of FTSs:
   - scalable algorithm
   - proof of correctness [EMSE22]
   - 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 [EMSE22]
   - 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 [EMSE22]
   - 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$^{\Box}$ [recall]

Note

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

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

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

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

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

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

Family-based model checking (2/2): LTL

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

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

Family-based model checking (2/2): LTL

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

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

Note

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

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

Family-based model checking (2/2): LTL

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

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

Note

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

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

Thus:

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

Family-based model checking with SPIN ...

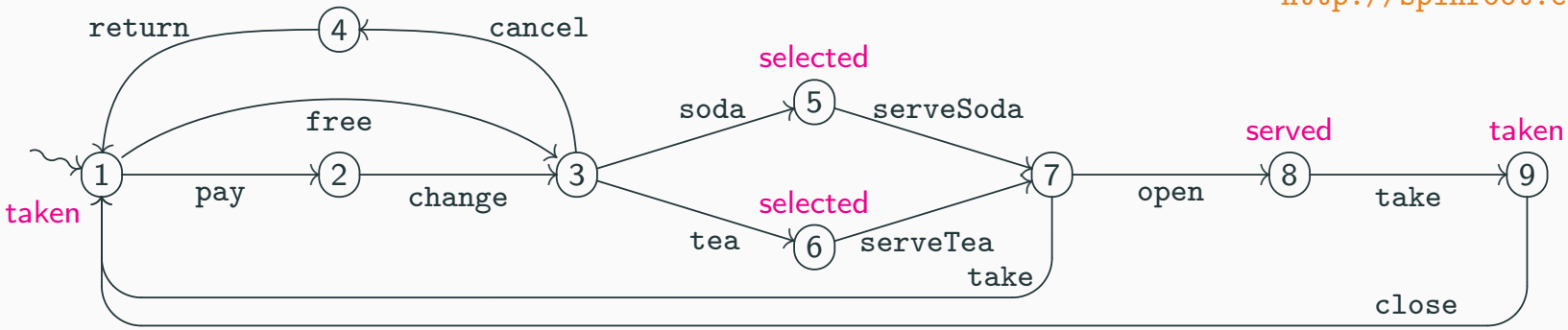
Example LTL formulas that can thus be verified with SPIN (for instance):

http://spinroot.com/

Family-based model checking with SPIN ...

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

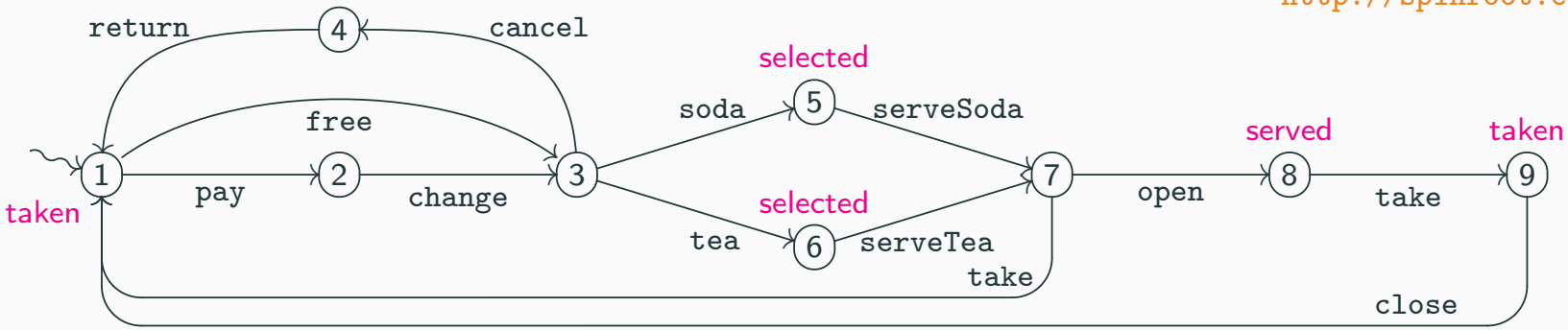
http://spinroot.com/



Family-based model checking with SPIN ...

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

http://spinroot.com/

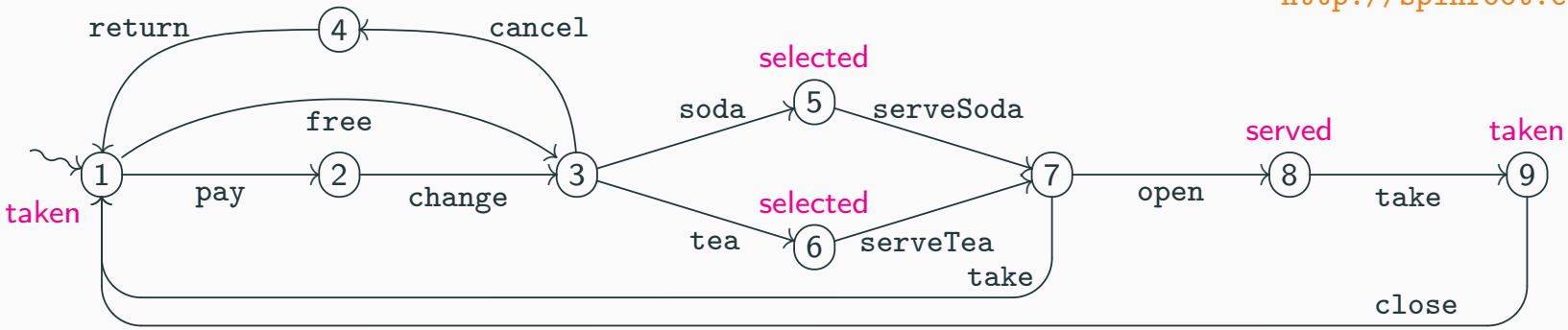


1. $G\,(selected \Rightarrow F\ served)$: *after a beverage has been selected, the vending machine will always eventually have served a beverage*
2. $G\,(served \Rightarrow F\ taken)$: *after a beverage has been served, a customer will always eventually have taken the beverage*

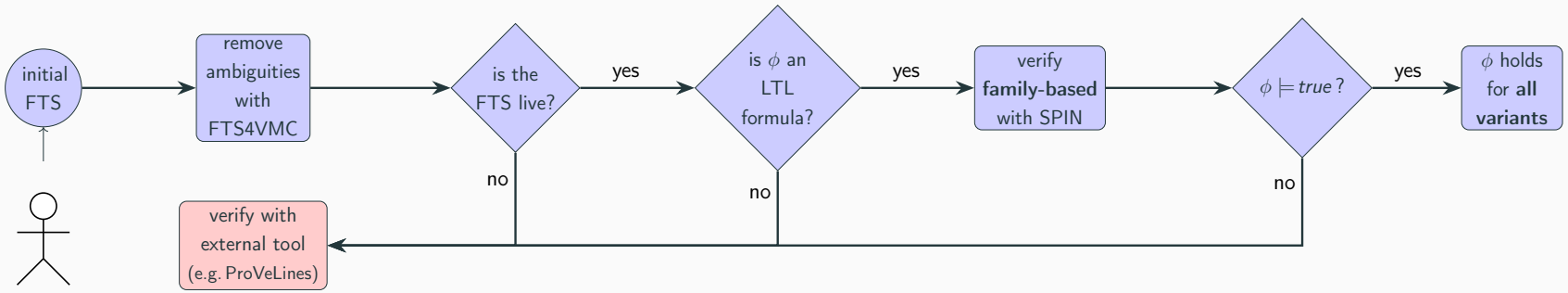
Family-based model checking with SPIN ...

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

http://spinroot.com/



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



References

SPLC19 M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, and L. Paolini, Static Analysis of Featured Transition Systems. In Proceedings 23rd International Systems and Software Product Line Conference (SPLC'19), ACM, 2019, 39–51 (best paper) https://doi.org/10.1145/3336294.3336295

EMSE22 M.H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, and L. Paolini, Efficient Static Analysis and Verification of Featured Transition Systems. *Empirical Software Engineering* 22, 1 (2022), 10:1–10:43 https://doi.org/10.1007/s10664-020-09930-8

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

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

Thanks for your attention!

It would be great to meet some of you at FM 2024:

- Submission: April 19th
- Conference: September 9th–13th
- Co-located: FACS, FMICS, LOPSTR/PPDP, TAP

https://www.fm24.polimi.it/