QUANTICOL

A Quantitative Approach to Management and Design of Collective and Adaptive Behaviours

quantiCOL

http://www.quanticol.eu

TR-QC-3-2013

Model Checking Value-Passing Modal Specifications

Revision: 1; Nov 15, 2013

Author(s): Maurice H. ter Beek (CNR–ISTI), Stefania Gnesi (CNR–ISTI), and Franco Mazzanti (CNR–ISTI)

Publication date: Dec 20, 2013

Funding Scheme: Small or medium scale focused research project (STREP)
Topic: ICT-2011 9.10: FET-Proactive 'Fundamentals of Collective Adaptive Systems' (FOCAS)
Project number: 600708

Coordinator: Jane Hillston (UEDIN)
e-mail: Jane.Hillston@ed.ac.uk
Fax: +44 131 651 1426

Part. no.	Participant organisation name	Acronym	Country
1 (Coord.)	University of Edinburgh	UEDIN	UK
2	Consiglio Nazionale delle Ricerche – Istituto di Scienza e Tecnologie della Informazione "A. Faedo"	CNR	Italy
3	Ludwig-Maximilians-Universität München	LMU	Germany
4	Ecole Polytechnique Fédérale de Lausanne	EPFL	Switzerland
5	IMT Lucca	IMT	Italy
6	University of Southampton	SOTON	UK

COOPERATION

Contents

1 Introduction **1**

2 Background: Modal Transition Systems **2**

3 Modelling and Verification Environment **3**
3.1 A Modal Process Algebra 3
3.2 A Logic to Express Variability 4
3.3 The Variability Model Checker 6

4 Dealing with Data **6**
4.1 Case Study: Bike-Sharing Systems 6

5 Value-Passing Modelling and Verification Environment **7**
5.1 A Value-Passing Modal Process Algebra 7
5.2 Value-Passing v-ACTL 8
5.3 Model Checking Value-Passing Modal Specifications 9

6 Modelling and Analyzing the Case Study **9**

7 Conclusions and Future Work **11**

Abstract

Formal modelling and verification of variability concepts in product families has been the subject of extensive study in the literature on Software Product Lines. In recent years, we have laid the basis for the use of modal specifications and branching-time temporal logics for the specification and analysis of behavioural variability in product family definitions. A critical point in this formalization is the lack of a possibility to model an adequate representation of the data that may need to be described when considering real systems. To this aim, we now extend the modelling and verification environment that we have developed for specifications interpreted over Modal Transition Systems, by adding the possibility to include data in the specifications. In concert with this, we also extend the variability-specific modal logic and the associated special-purpose model checker VMC. As a result, it offers the possibility to efficiently verify formulas over possibly infinite-state systems by using the on-the-fly bounded model-checking algorithms implemented in the model checker.

1 Introduction

Product Line Engineering (PLE) is a paradigm for the development of a variety of products from a common product platform. Its aim is to lower the production costs of individual products by letting them share an overall reference model of a product family, while allowing them to differ with respect to specific features to serve, e.g., different markets. Software Product Line Engineering (SPLE) has translated this paradigm into a software engineering approach aimed at the development, in a cost-effective way, of a variety of software-intensive products that share an overall reference model, i.e., that together form a product family [24]. Usually, the commonality and variability of a product family are defined in terms of features, and managing variability is about identifying variation points in a common family design to encode exactly those combinations of features that lead to valid products. The actual configuration of the products during application engineering then boils down to selecting desired options in the variability model.

Since many software-intensive systems are embedded, distributed and safety-critical, there is a strong need for rigour and for formal modelling and verification (tools). Our contribution to making the development of product families more rigorous consists of an ongoing research effort to elaborate a suitable formal modelling structure to describe behavioural product variability and a suitable temporal

logic that can be interpreted over that structure [13, 3, 4, 6, 8]. We opted for Modal Transition Systems (MTSs) [1], which were recognized in [15, 22, 23] as a useful formal method to describe in a compact way the possible operational behaviour of all products of a product family. We also defined an action-based branching-time CTL-like temporal modal logic over MTSs and moreover developed efficient algorithms to derive valid products from families and to model check properties over products and families alike. Finally, we implemented these algorithms in an experimental tool: the Variability Model Checker (VMC) [6, 8].

A critical point in the formalization by means of MTSs is the lack of a possibility to model an adequate representation of the data that may need to be described when considering real systems. To this aim, in this paper we extend the modelling and verification environment we developed so far by adding the possibility to include data in the specifications. In concert with this, we also extend the logic and the tool. As a result, VMC offers the possibility to efficiently verify properties over possibly infinite-state systems by means of explicit-state on-the-fly bounded model checking. We illustrate our approach by means of a simple yet intuitive example: a bike-sharing system.

2 Background: Modal Transition Systems

Before defining MTSs, we define their underlying Labelled Transition Systems.

Definition 1. *A* Labelled Transition System *(LTS) is a 4-tuple* $(Q, A, \overline{q}, \delta)$*, with set* Q *of states, set* A *of actions, initial state* $\overline{q} \in Q$*, and transition relation* $\delta \subseteq Q \times A \times Q$*; we may write* $q \xrightarrow{a} q'$ *if* $(q, a, q') \in \delta$*.*

An MTS is an LTS which distinguishes between *may* and *must* transitions.

Definition 2. *A* Modal Transition System *(MTS) is a 5-tuple* $(Q, A, \overline{q}, \delta^{\Diamond}, \delta^{\Box})$ *such that* $(Q, A, \overline{q}, \delta^{\Diamond} \cup \delta^{\Box})$ *is an LTS and* $\delta^{\Box} \subseteq \delta^{\Diamond}$*. An MTS distinguishes the* may *transition relation* $\delta^{\Diamond}$*, expressing* admissible *transitions, and the* must *transition relation* $\delta^{\Box}$*, expressing* necessary *transitions; we may write* $q \xrightarrow{a}_{\Diamond} q'$ *for* $(q, a, q') \in \delta^{\Diamond}$ *and* $q \xrightarrow{a}_{\Box} q'$ *for* $(q, a, q') \in \delta^{\Box}$*.*

The inclusion $\delta^{\Box} \subseteq \delta^{\Diamond}$ formalizes that necessary transitions are also admissible. Graphically, an MTS is a directed edge-labelled graph where nodes model states and action-labelled edges model transitions: solid edges are necessary ones (i.e., $\delta^{\Box}$) and dotted edges are admissible but not necessary ones (i.e., $\delta^{\Diamond} \setminus \delta^{\Box}$).

A *full path* is a path that cannot be extended further, i.e., it is infinite or it ends in a state without outgoing transitions. A *must path* is a full path that consists of only must transitions, i.e., it consists of only solid edges.

An MTS can provide an abstract description of the set of (valid) products of a product family, defining both the behaviour that is common to all products and the behaviour that varies among different products. This requires an interpretation of the requirements of a product family and its constraints with respect to certain features as may and must transitions labelled with actions, and a temporal ordering among these transitions. The idea is that the family's products are the ordinary LTSs that can be obtained by resolving the variability modelled through admissible (may) but not necessary (must) transitions (i.e., the aforementioned dotted edges). Resolving variability thus boils down to deciding for each particular optional behaviour whether it is to be included in a specific product LTS, whereas all mandatory behaviour is included by definition.

Definition 3. *Let* $\mathcal{F} = (Q, A, \overline{q}, \delta^{\Diamond}, \delta^{\Box})$ *be an MTS. The set* $\{\mathcal{P}_i = (Q_i, A, \overline{q}, \delta_i) \mid i > 0\}$ *of* derived product LTSs *of* $\mathcal{F}$ *is obtained from* $\mathcal{F}$ *by considering each pair of* $Q_i \subseteq Q$ *and* $\delta_i \subseteq \delta^{\Diamond} \cup \delta^{\Box}$ *to be defined such that:*

1. *every* $q \in Q_i$ *is reachable in* $\mathcal{P}_i$ *from* $\overline{q}$ *via transitions from* δ_i *and*
2. *there exists no* $(q, a, q') \in \delta^{\Box} \setminus \delta_i$ *such that* $q \in Q_i$*.*

3 Modelling and Verification Environment

In this section we present the modelling and verification environment that we have developed recently [3, 4, 6, 8].

3.1 A Modal Process Algebra

MTSs are not suitable for directly specifying the behaviour of a complex system, maybe consisting of several components. In such cases it is better to describe the system in an abstract high-level language that is interpreted over MTSs. We consider a CSP-like process algebra in which the parallel composition operator is parametrized by a set of actions to be synchronized. This is different from the CCS-based approaches in [19, 20, 18, 7]. A system can then be defined inductively by composition, with the additional distinction between may and must actions.

Definition 4. *Let $\mathcal{A}$ be a set of* actions*, let $a \in \mathcal{A}$ and let $L \subseteq \mathcal{A}$.* Processes *are built from terms and actions according to the abstract syntax:*

$$\begin{array}{lcl} N & ::= & [P] \\ P & ::= & T \mid P \,/L/\, P \\ T & ::= & nil \mid K \mid A.T \mid T + T \\ A & ::= & a \mid a(may) \end{array}$$

with K a process identifier from the set of process definitions of the form $K \stackrel{def}{=} T$.

If $L = \varnothing$, then we may also write $P \,//\, P$. The set $\{M, N, \ldots\}$ of *systems* is denoted by $\mathcal{N}$ and the set $\{P, Q, \ldots\}$ of *processes* is denoted by $\mathcal{P}$.

A process can thus be one of the following:

nil : a terminated process that has finished execution;

K : a process identifier that is used for modelling recursive sequential processes;

$A.P$: a process that can execute action A and then behave as P;

$P + Q$: a process that can non-deterministically choose to behave as P or as Q;

$P \,/L/\, Q$: a process formed by the parallel composition of P and Q that can synchronize on actions in L and interleave other actions.

Note that we distinguish between must actions a and may but not must actions $a(may)$. Each action type is treated differently in the rules of the SOS semantics.

Definition 5. *The* operational semantics *of a system $N \in \mathcal{N}$ is given over the MTS $(\mathcal{N}, \mathcal{A}, N, \delta^{\Diamond}, \delta^{\Box})$, where $\delta^{\Diamond}$ and $\delta^{\Box}$ are defined as the least relations that satisfy the set of axioms and transition rules in Figs. 1-2.*

As usual, inference rules are defined in terms of a (possibly empty) set of premises (above the line) and a conclusion (below the line). The reduction relation is defined in SOS style (i.e. by induction on the structure of the terms denoting a process) modulo the structural congruence relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ defined in Fig. 2. Considering terms up to a structural congruence allows identifying different ways of denoting the same process and the expansion of recursive process definitions.

As is common for MTSs, synchronizing $a(may)$ with a results in $a(may)$ [1]. Note, finally, that when restricted to must actions (i.e., LTSs) the rules for non-deterministic choice and parallel composition collapse onto the standard ones.

$$(\text{SYS}_\Box)\quad \frac{P \xrightarrow{a} P'}{[P] \xrightarrow{a} [P']} \qquad (\text{SYS}_\Diamond)\quad \frac{P \overset{a}{\dashrightarrow} P'}{[P] \overset{a}{\dashrightarrow} [P']}$$

$$(\text{ACT}_\Box)\quad \frac{}{a.P \xrightarrow{a} P} \qquad (\text{ACT}_\Diamond)\quad \frac{}{a(may).P \overset{a}{\dashrightarrow} P}$$

$$(\text{OR}_\Box)\quad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad (\text{OR}_\Diamond)\quad \frac{P \overset{a}{\dashrightarrow} P'}{P + Q \overset{a}{\dashrightarrow} P'}$$

$$(\text{INT}_\Box)\quad \frac{P \xrightarrow{\ell} P'}{P\,/L/\,Q \xrightarrow{\ell} P'\,/L/\,Q'}\ \ell \notin L \qquad (\text{INT}_\Diamond)\quad \frac{P \overset{\ell}{\dashrightarrow} P'}{P\,/L/\,Q \overset{\ell}{\dashrightarrow} P'\,/L/\,Q'}\ \ell \notin L$$

$$(\text{PAR}_\Box)\quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P\,/L/\,Q \xrightarrow{a} P'\,/L/\,Q'}\ a \in L \qquad (\text{PAR}_\Diamond)\quad \frac{P \overset{a}{\dashrightarrow} P' \quad Q \overset{a}{\dashrightarrow} Q'}{P\,/L/\,Q \overset{a}{\dashrightarrow} P'\,/L/\,Q'}\ a \in L$$

$$(\text{PAR}_{\boxtimes})\quad \frac{P \xrightarrow{a} P' \quad Q \overset{a}{\dashrightarrow} Q'}{P\,/L/\,Q \overset{a}{\dashrightarrow} P'\,/L/\,Q'}\ a \in L$$

Figure 1: The SOS semantics of the modal process algebra, with $a, \ell \in \mathcal{A}$

$$\begin{array}{rclcrclcrcl} P+Q & \equiv & Q+P & & P+(Q+R) & \equiv & (P+Q)+R & & P+0 & \equiv & P \\ P\,/L/\,Q & \equiv & Q\,/L/\,P & & P\,/L/\,(Q\,/L/\,R) & \equiv & (P\,/L/\,Q)\,/L/\,R & & P\,/L/\,0 & \equiv & P \\ P & \equiv & P[^{Q}/_{K}] & \text{whenever} & K \overset{\text{def}}{=} Q & & & & & & \end{array}$$

Figure 2: Structural congruence relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$

3.2 A Logic to Express Variability

We now present v-ACTL, an action-based branching-time temporal logic for *variability* in the style of the action-based logic ACTL [11], the state-based logic CTL [9], and the Hennessy–Milner Logic (HML) with Until defined in [12, 21]. Next to the standard operators of propositional logic, v-ACTL contains the classical box and, by duality, diamond modal operators from HML, as well as a deontic interpretation of them, the existential and universal path quantifiers from CTL and the (action-based) F and, by duality, G operators from ACTL, both with and without a deontic interpretation. More on *deontic logic* [2] below.

v-ACTL defines action formulas (denoted by ψ), state formulas (denoted by ϕ) and path formulas (denoted by π).

Definition 6. *Action formulas are built over a set $\mathcal{A}$ of* actions*, where $a \in \mathcal{A}$:*

$$\psi \quad ::= \quad \mathit{true} \mid a \mid \neg\psi \mid \psi \wedge \psi$$

Action formulas are thus boolean compositions of actions. As usual, *false* abbreviates $\neg \mathit{true}$, $\psi \vee \psi'$ abbreviates $\neg(\neg\psi \wedge \neg\psi')$ and $\psi \Longrightarrow \psi'$ abbreviates $\neg\psi \vee \psi'$.

Definition 7. *The satisfaction of formula ψ by action a, $a \models \psi$, is defined as:*

$$\begin{array}{l} a \models \mathit{true} \ \ \mathit{always\ holds} \\ a \models b \ \ \mathit{iff}\ \ a = b,\ \mathit{with}\ b \in \mathcal{A} \\ a \models \neg\psi \ \ \mathit{iff}\ \ a \not\models \psi \\ a \models \psi \wedge \psi' \ \ \mathit{iff}\ \ a \models \psi\ \mathit{and}\ a \models \psi' \end{array}$$

Definition 8. *The syntax of v-ACTL is:*

$$\begin{array}{lll} \phi & ::= & \mathit{true} \mid \neg\,\phi \mid \phi \wedge \phi \mid [\psi]\,\phi \mid [\psi]^{\Box}\,\phi \mid E\,\pi \mid A\,\pi \mid \mu\,Y.\phi(Y) \mid \nu\,Y.\phi(Y) \\ \pi & ::= & F\,\phi \mid F^{\Box}\,\phi \mid F\,\{\psi\}\,\phi \mid F^{\Box}\,\{\psi\}\,\phi \end{array}$$

where Y is a propositional variable and $\phi(Y)$ is syntactically monotone in Y.

The least and greatest fixed-point operators μ and ν provide a semantics for recursion, used for "finite looping" and "looping" (or "liveness" and "safety"), respectively. It is well known that the path formulas (such as the Until operator and the derived F and G operators) can be derived from the least and greatest fixed-point operators. We however prefer to represent some of them explicitly to make their understanding simpler. The intuitive interpretation of the remaining nonstandard operators is:

$[\psi]\,\phi$: in all next states reachable by a *may* transition executing an action satisfying ψ, ϕ holds.

$[\psi]^{\Box}\,\phi$: in all next states reachable by a *must* transition executing an action satisfying ψ, ϕ holds.

$E\,\pi$: there exists a full path on which π holds.

$A\,\pi$: on all possible full paths, π holds.

$F\,\phi$: there exists a future state in which ϕ holds.

$F^{\Box}\,\phi$: there exists a future state in which ϕ holds and all transitions until that state are must transitions.

$F\,\{\psi\}\;\phi$: there exists a future state, reached by an action satisfying ψ, in which ϕ holds.

$F^{\Box}\,\{\psi\}\;\phi$: there exists a future state, reached by an action satisfying ψ, in which ϕ holds and all transitions until that state are must transitions.

The formal semantics of v-ACTL is interpreted over MTSs. Let $\mathit{path}(q)$ denote the set of all full paths from a state q. Moreover, for a path $\sigma = q_1 a_1 q_2 a_2 q_3 \cdots$, we denote its ith state (i.e., q_i) by $\sigma(i)$ and its ith action (i.e., a_i) by $\sigma\{i\}$.

Definition 9. *Let $(Q, A, \overline{q}, \delta^{\Diamond}, \delta^{\Box})$ be an MTS, with $q \in Q$ and $\sigma \in \mathit{path}(q)$. The satisfaction relation $\models$ of v-ACTL is:*

$$\begin{array}{l}
q \models \mathit{true} \;\; \mathit{always\ holds} \\
q \models \neg\,\phi \;\; \mathit{iff} \;\; q \not\models \phi \\
q \models \phi \wedge \phi' \;\; \mathit{iff} \;\; q \models \phi \;\mathit{and}\; q \models \phi' \\
q \models [\psi]\,\phi \;\; \mathit{iff} \;\; \forall\, q' \in Q \;\mathit{such\ that}\; q \xrightarrow{a}_{\Diamond} q' \;\mathit{and}\; a \models \psi,\; \mathit{we\ have}\; q' \models \phi \\
q \models [\psi]^{\Box}\,\phi \;\; \mathit{iff} \;\; \forall\, q' \in Q \;\mathit{such\ that}\; q \xrightarrow{a}_{\Box} q' \;\mathit{and}\; a \models \psi,\; \mathit{we\ have}\; q' \models \phi \\
q \models E\,\pi \;\; \mathit{iff} \;\; \exists\, \sigma' \in \mathit{path}(q) \colon \sigma' \models \pi \\
q \models A\,\pi \;\; \mathit{iff} \;\; \forall\, \sigma' \in \mathit{path}(q) \colon \sigma' \models \pi \\
q \models \mu\,Y.\phi(Y) \;\; \mathit{iff} \;\; \bigvee_{i \geq 0} \phi^i(\mathit{false}) \\
q \models \nu\,Y.\phi(Y) \;\; \mathit{iff} \;\; \bigwedge_{i \geq 0} \phi^i(\mathit{true}) \\
q \models F\,\phi \;\; \mathit{iff} \;\; \exists\, j \geq 1 \colon \sigma(j) \models \phi \\
q \models F^{\Box}\,\phi \;\; \mathit{iff} \;\; \exists\, j \geq 1 \colon \sigma(j) \models \phi \;\mathit{and}\; \forall\, 1 \leq i < j \colon (\sigma(i), \sigma\{i\}, \sigma(i+1)) \in \delta^{\Box} \\
q \models F\,\{\psi\}\;\phi \;\; \mathit{iff} \;\; \exists\, j \geq 1 \colon \sigma\{j\} \models \psi \;\mathit{and}\; \sigma(j+1) \models \phi \\
q \models F^{\Box}\,\{\psi\}\;\phi \;\; \mathit{iff} \;\; \exists\, j \geq 1 \colon \sigma\{j\} \models \psi \;\mathit{and}\; \sigma(j+1) \models \phi, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{and}\; \forall\, 1 \leq i \leq j \colon (\sigma(i), \sigma\{i\}, \sigma(i+1)) \in \delta^{\Box}
\end{array}$$

Some further operators can be derived as usual. $\langle\psi\rangle\,\phi$ abbreviates $\neg\,[\psi]\,\neg\,\phi$: a next state exists, reachable by a *may* transition executing an action satisfying ψ, in which ϕ holds; $\langle\psi\rangle^{\Box}\,\phi$ abbreviates $\neg\,[\psi]^{\Box}\,\neg\,\phi$: a next state exists, reachable by a *must* transition executing an action satisfying ψ, in which ϕ holds.

$G\,\phi$ abbreviates $\neg F\,\neg\phi$: the path is a full path on which ϕ holds in all states. $AG\,\phi$ abbreviates $\neg EF\,\neg\phi$: in all states on all paths, ϕ holds;

v-ACTL thus interprets some classical modal and temporal operators in a *deontic* way by considering the modalities of the transitions of an MTS. Deontic logic formalises notions like violation, obligation, permission, and prohibition [2]. v-ACTL implicitly incorporates two of the most classic deontic modalities, as $\langle\psi\rangle^{\Box}$ represents O ("obligation") whereas $[\psi]^{\Box}$ represents P ("permission").

3.3 The Variability Model Checker

The modelling and verification environment described in the previous sections has been implemented in the Variability Model Checker (VMC) [6, 8], which is freely usable online (`http://fmt.isti.cnr.it/vmc/v5.5`). VMC accepts as input a model specified in the modal process algebra presented in § 3.1 and it allows to verify properties expressed in the v-ACTL logic presented in § 3.2.

VMC is the most recent product of a family of model checkers that have been developed at ISTI–CNR over the past two decades, including FMC [17], UMC [5] and CMC [14]. Each of these allows the efficient verification by means of explicit-state on-the-fly model checking of functional properties expressed in a specific action- and state-based branching-time temporal logic derived from the family of logics based on CTL [9], including ACTL [11]. The on-the-fly nature of this family of model checkers means that in general not the whole state space needs to be generated and explored. This feature improves performance and allows to deal with infinite-state systems.

In the case of infinite-state systems, a bounded model-checking approach is adopted, i.e., the evaluation is started by assuming a certain value as a maximum depth of the evaluation. If the evaluation of a formula reaches a result within the requested depth, then the result holds for the whole system; otherwise the maximum depth is increased and the evaluation is retried (preserving all useful partial results already found). This approach, initially introduced in UMC to address infinite state spaces, happens to be quite useful also for another reason: by setting a small initial maximum depth and a small automatic increment of this bound at each re-evaluation failure, once a result is finally found then we also have a reasonable (almost minimal) explanation for it.

On the basis of the algorithms presented in [5, 17], on-the-fly model checking of v-ACTL formulas over MTSs can be achieved in a complexity that is linear with respect to the size of the state space. It is beyond the scope of this paper to present detailed descriptions of the model-checking algorithms and architecture underlying this family of model checkers, but we refer the interested reader to [5].

4 Dealing with Data

In recent years, we have laid the basis for the use of modal specifications and branching-time temporal logics for the specification and analysis of behavioural variability in SPLE, by developing the environment presented in the previous section. A critical point in this approach is the lack of a possibility to model an adequate representation of the data that may need to be described when considering realistic systems. We now present a case study that makes this clear.

4.1 Case Study: Bike-Sharing Systems

An increasing number of cities worldwide are adopting fully automated public bike-sharing systems (BSS) as a green urban mode of transportation [10]. The concept is simple and their benefits multiple, including the reduction of vehicular traffic (congestion), pollution, and energy consumption. A BSS

consists of parking stations distributed over a city, typically in close proximity to other public transportation hubs such as subway and tram stations. (Subscribed) users may rent an available bike from one of the stations, use it for a while and then drop it off at any (other) station. BSS offer a number of challenging run-time optimization problems aimed at improving the efficiency and user satisfaction. A primary example is balancing the load between the different stations, e.g., by using incentive (reward) schemes that may change the behaviour of users but also by efficient (dynamic) redistribution of bikes between stations.

A side-study of the EU FP7 project QUANTICOL (`http://www.quanticol.eu`) concerns the quantitative analysis of BSS seen as so-called Collective Adaptive Systems (CAS). The design of CAS must be supported by a powerful and well-founded framework for quantitative modelling and analysis. CAS consist of a large number of spatially distributed entities, which may be competing for shared resources even when collaborating to reach common goals. The nature of CAS, together with the importance of the societal goals they address, mean that it is imperative to carry out thorough analyses of their design to investigate all aspects of their behaviour before they are put into operation. In the context of QUANTICOL, we have started to collaborate with "PisaMo azienda per la mobilità s.p.a.", an in-house public mobility company of the city of Pisa's administration. They recently introduced the public BSS *CicloPi* in the city of Pisa, which currently consists of some 150 bikes and 15 stations and thus forms a perfect test case for our research and an interesting benchmark for the QUANTICOL project.

Inspired by [16], we consider a BSS with N stations and a fleet of M bikes. Each station i has a capacity K_i. The dynamic behaviour of the system is then:

1. Users arrive at station i.
2. If a user arrives at a station and there is no available bike, then (s)he leaves the system.
3. Otherwise, (s)he takes a bike and chooses station j to return the bike.
4. When (s)he arrives at station j, if there are less than K_j bikes in this station, (s)he returns the bike and leaves the system.
5. If the station is full the user chooses another station, say k, and goes there.
6. A redistribution activity of bikes *may* be asked and *may* possibly be satisfied.
7. The user rides like this again until (s)he can return the bike.

This list contains a mix of a kind of static constraints defining the differences in configuration (features), such as the optional possibility to have a redistribution mechanism in our BSS, between products as well as more operational constraints defining the behaviour of products through admitted sequences (temporal orderings) of actions or operations implementing features according to certain values.

5 Value-Passing Modelling and Verification Environment

We now extend the modelling and verification environment of § 3 to handle data.

5.1 A Value-Passing Modal Process Algebra

First, we extend the modal process algebra of § 3.1 with values and parameters.

Definition 10. *Let $\mathcal{A}$ be a set of* actions*, let $a \in \mathcal{A}$ and let $L \subseteq \mathcal{A}$.* Processes *are built from terms and actions according to the abstract syntax:*

$$\begin{array}{rcl} N & ::= & [P] \\ P & ::= & T(e) \mid P \,/L/\, P \\ T(v) & ::= & \mathit{nil} \mid K(e) \mid A.T(e) \mid T(e) + T(e) \mid [e \bowtie e]\, T(e) \\ A & ::= & a(e) \mid a(\mathit{may}, e) \mid a(?v) \mid a(\mathit{may}, ?v) \\ e & ::= & v \mid \boldsymbol{id} \mid \boldsymbol{int} \mid e \pm e \end{array}$$

*where $K(e)$ is a process identifier from the set of process definitions of the form $K(e) \stackrel{def}{=} T(e)$, $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ is a comparison relation, v is a variable, **id** is a constant, **int** is an integer and $\pm \in \{+, -, *, /\}$ is an arithmetic operation.*

Also the semantics of this value-passing modal process algebra is given over MTSs, but here we only provide the SOS rules for the must actions (in Fig. 3). The other ones follow in a straightforward manner from the rules given in Fig. 1.

$$(\textsc{sys}) \quad \frac{P \xrightarrow{a(e)} P'}{[P] \xrightarrow{a(e)} [P']} \qquad (\textsc{act}_\Box) \quad \frac{}{\alpha.P \xrightarrow{\alpha} P}\ {\scriptstyle \alpha \in \{a(e), a(?v)\}}$$

$$(\textsc{or}_\Box) \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\ {\scriptstyle \alpha \in \{a(e), a(?v)\}} \qquad (\textsc{guard}) \quad \frac{}{[e_1 \bowtie e_2]\, P(e_3) \longrightarrow P(e_3)}\ {\scriptstyle e_1 \bowtie e_2}$$

$$(\textsc{par}_\Box) \quad \frac{P \xrightarrow{a(e_1)} P' \quad Q \xrightarrow{a(e_2)} Q'}{P \,/L/\, Q \xrightarrow{a} P' \,/L/\, Q'}\ {\scriptstyle a \in L, e_1 = e_2} \qquad (\textsc{par}_\Box) \quad \frac{P \xrightarrow{a(?v)} P' \quad Q \xrightarrow{a(e)} Q'}{P \,/L/\, Q \xrightarrow{a} P'[{}^{e}/_{v}] \,/L/\, Q'}\ {\scriptstyle a \in L}$$

Figure 3: The SOS semantics of the value-passing modal process algebra, with $a, \ell \in \mathcal{A}$

Note that the SYS rule implies that we assume a closed-world semantics, i.e., a system cannot evolve on input actions of the form $a(?v)$.

5.2 Value-Passing v-ACTL

To handle action values in v-ACTL, we only need to extend the definition of action formulas and consequently their satisfaction relation.

Definition 11. *Action formulas are built over a set $\mathcal{A}$ of* actions*, where $a \in \mathcal{A}$:*

$$\psi \quad ::= \quad \mathit{true} \mid a \mid a(e) \mid \neg\psi \mid \psi \wedge \psi$$

The satisfaction relation of action formulas given in Def. 7 is extended with:

Definition 12. *Let $a, b \in \mathcal{A}$.*

$$\begin{array}{l} a(e) \models \mathit{true} \quad \mathit{always\ holds} \\ a(e) \models b \quad \mathit{iff} \quad a = b \\ a(e) \models b(*) \quad \mathit{iff} \quad a = b \\ a(e) \models b(e') \quad \mathit{iff} \quad a = b \ \mathit{and}\ e = e' \\ a(e) \models \neg\psi \quad \mathit{iff} \quad a(e) \not\models \psi \\ a(e) \models \psi \wedge \psi' \quad \mathit{iff} \quad a(e) \models \psi \ \mathit{and}\ a(e) \models \psi' \end{array}$$

5.3 Model Checking Value-Passing Modal Specifications

The extended modelling and verification environment described in the previous sections has been implemented in VMC v6.0 (`http://fmt.isti.cnr.it/vmc/v6.0`), which now thus accepts models specified in the value-passing modal process algebra presented in § 5.1 and allows model checking properties expressed in value-passing v-ACTL presented in § 5.2.

6 Modelling and Analyzing the Case Study

We first specify the behaviour of a family of bike-sharing stations in the value-passing modal process algebra, taking into account the possibility of having a dynamic redistribution scheme as an optional feature of the BSS. Without loss of generality, we assume a bike-sharing station with 2 as its maximum capacity:

```
Station(X) = request.StationBikeRequested(X)
StationBikeRequested(Y) =
  [Y<1] ( nobike.Station(Y) +
          redistribute(may).Station(Y+2) ) +
  [Y>0] givebike.Station(Y-1)

net BSS = Station(2)
```

From this specification of a family of bike-sharing stations, VMC generates the MTS depicted in Fig. 4(a) and its possible products depicted in Figs. 4(b)-4(c).

If we want to consider also the behaviour of a user, we might specify the following family of BSS:

```
User = request.(givebike.User + nobike.User + redistribute.User)

net BSS = Station(2) /request,givebike,nobike,redistribute/ User
```

Due to the synchronous parallel composition, this specification of course results in the same family MTS and products LTSs depicted in Fig. 4.

To illustrate what kind of variability analyses can be performed with the extended value-passing modelling and verification environment introduced in § 5, we now present a few properties and the result of model checking them with VMC against the above example BSS:[1]

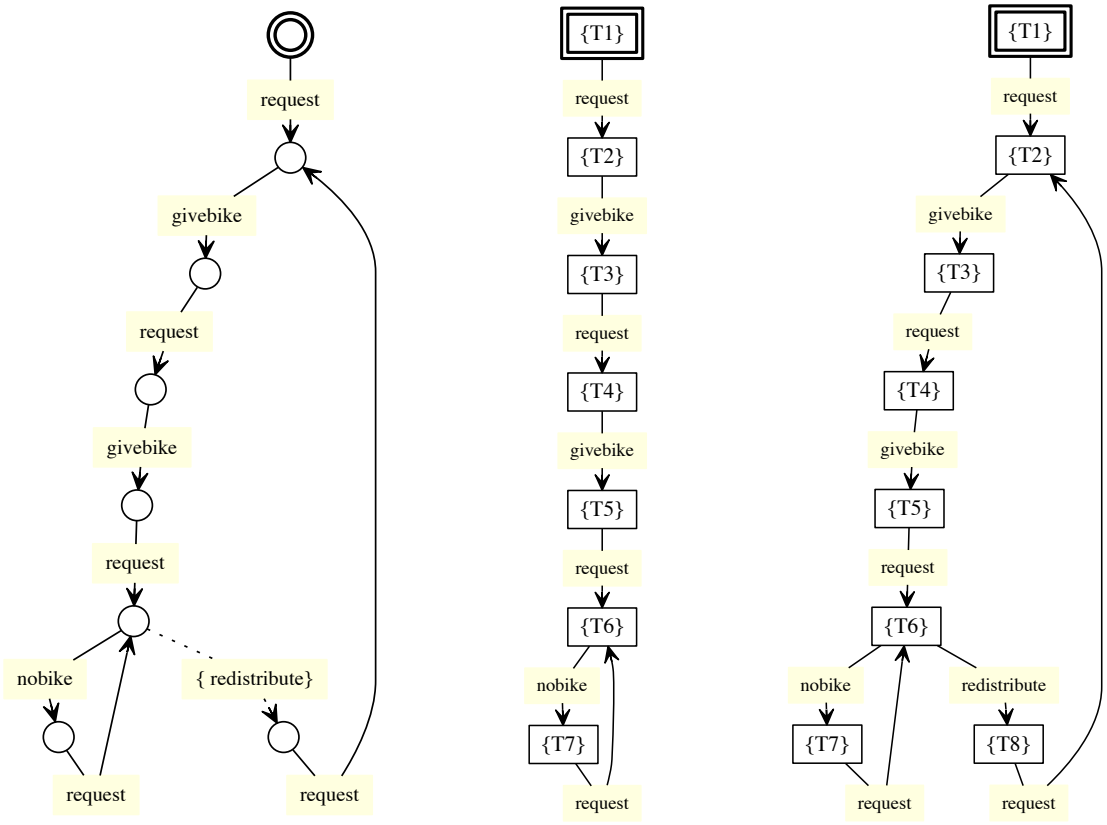
Eventually it must occur that no more bike is available:
$EF^{\Box}\,\{nobike\}\ true$.
This formula obviously is true.

It is always the case that eventually it must occur that no bike is available:
$AG\,EF^{\Box}\,\{nobike\}\ true$.
Also this formula is obviously true.

It is possible for a user to request and receive a bike for three times in a row:
$\langle request\rangle\,\langle givebike\rangle\,\langle request\rangle\,\langle givebike\rangle\,\langle request\rangle\,\langle givebike\rangle\ true$.
This formula is of course false.

Formulas without negation and only composed from *false*, *true* and the operators $\wedge$, $\vee$, $\langle\rangle^{\Box}$, $[]$, μ, ν, $EF^{\Box}$, $EF^{\Box}\{\}$, $AF^{\Box}$, $AF^{\Box}\{\}$ and AG that are valid for a family MTS are valid for all its product LTSs [4].

[1] In VMC, $[]^{\Box}$, μ, ν and $F^{\Box}$ need to be written as `[]#`, `min`, `max` and `F#`, respectively.



(a) Family MTS (b) Product LTS (c) Product LTS

Figure 4: (a)-(c) A family MTS and its product LTSs generated by VMC

Dually, formulas without negation and only composed from *false*, *true* and the operators $\wedge$, $\vee$, $\langle\,\rangle$, μ, ν, EF and $EF\{\}$ that are false for a family MTS are false for all its product LTSs.

As a final example, we model a possibly infinite number of users that take a bike from station I to station J. Initially, station I has N bikes, which it delivers (when available) to a requesting user or accepts from a returning user. If the station receives more than M bikes, the exceeding $N - M$ bikes are distributed to station J. Station I must accept all bikes distributed by other stations or returned by a user (possibly for redistribution). It could easily be extended to N stations and K groups of users that take a bike from one station to another.

```
Station(I,N,J,M) =
  request(I).
    ( [N=0] nobike(I).Station(I,N,J,M) +
      [N>0] givebike(I).Station(I,N-1,J,M) ) +
  deliver(I).Station(I,N+1,J,M) +
  redistribute(may,?FROM,?TO,?K).
    ( [TO = I] Station(I,N+K,J,M) +
      [TO /= I] Station(I,N,J,M) ) +
  [N > M] redistribute(may,I,J,N-M).Station(I,M,J,M)
```

```
-- two stations:
net STATIONS = Station(s1,2,s2,2) /redistribute/ Station(s2,2,s1,2)

Users(I,J) =
  request(I).
    ( givebike(I).deliver(J).Users(I,J) +
      nobike(I).Users(I,J) )

--  one or two groups of users
net USERS = Users(s1,s2) -- // Users(s2,s1)

net BSS = STATIONS /request,givebike,nobike,deliver/ USERS
```

From this specification of a family of bike-sharing stations, VMC generates the MTS with 18 states depicted in Fig. 5 in case of a BSS with only one user group; in case of a BSS with two user groups the MTS has 224 states.

Also for this family of BSS, we present a few properties and the result of model checking them with VMC against the above example BSS:

Eventually it must occur that station 1 has no bikes:
$EF^{\Box}\,\{nobike(s1)\}\ true$.
This formula is of course true.

Eventually it may occur that station 2 has no more bikes:
$EF\,\{nobike(s2)\}\ true$.
This formula however is false.

It is always the case that eventually station 1 must give a bike, possibly after it has first received bikes after redistribution:
$AG(\,(EF^{\Box}\,\{givebike(s1)\}\ true) \vee (EF^{\Box}\,[redistribute(*,s1,*)]\ EF^{\Box}\,\{givebike(s1)\}\ true)\,)$.
This formula is true.

7 Conclusions and Future Work

In this paper we present the most recent developments concerning our ongoing research effort to elaborate a rigorous modelling and verification environment for behavioural variability analyses of product families. Until recently, a major limitation for applying our approach to realistic case studies from industry was the lack of a possibility to model an adequate representation of the data that may need to be described.

This paper contributes to removing this limitation as it defines an extension of the environment that can deal with data. In particular, VMC v6.0 now accepts models specified in a value-passing modal process algebra and allows explicit-state on-the-fly model checking of properties expressed in a value-passing action-based branching-time modal temporal logic.

We illustrate the new features of VMC v6.0 by means of simple yet intuitive examples from a case study on bike-sharing systems.

In the future, we intend to further investigate the application of our new modelling and verification environment to the behavioural analysis of product families, such as the preservation of properties from families to their products, in particular in the presence of the complex constraints that usually exist between the various features that are available in a product family. We also intend to address the scalability of our approach, which is of utmost importance for any variability analysis technique to be succesful in SPLE, since a product family's variability is exponential in the number of available features.

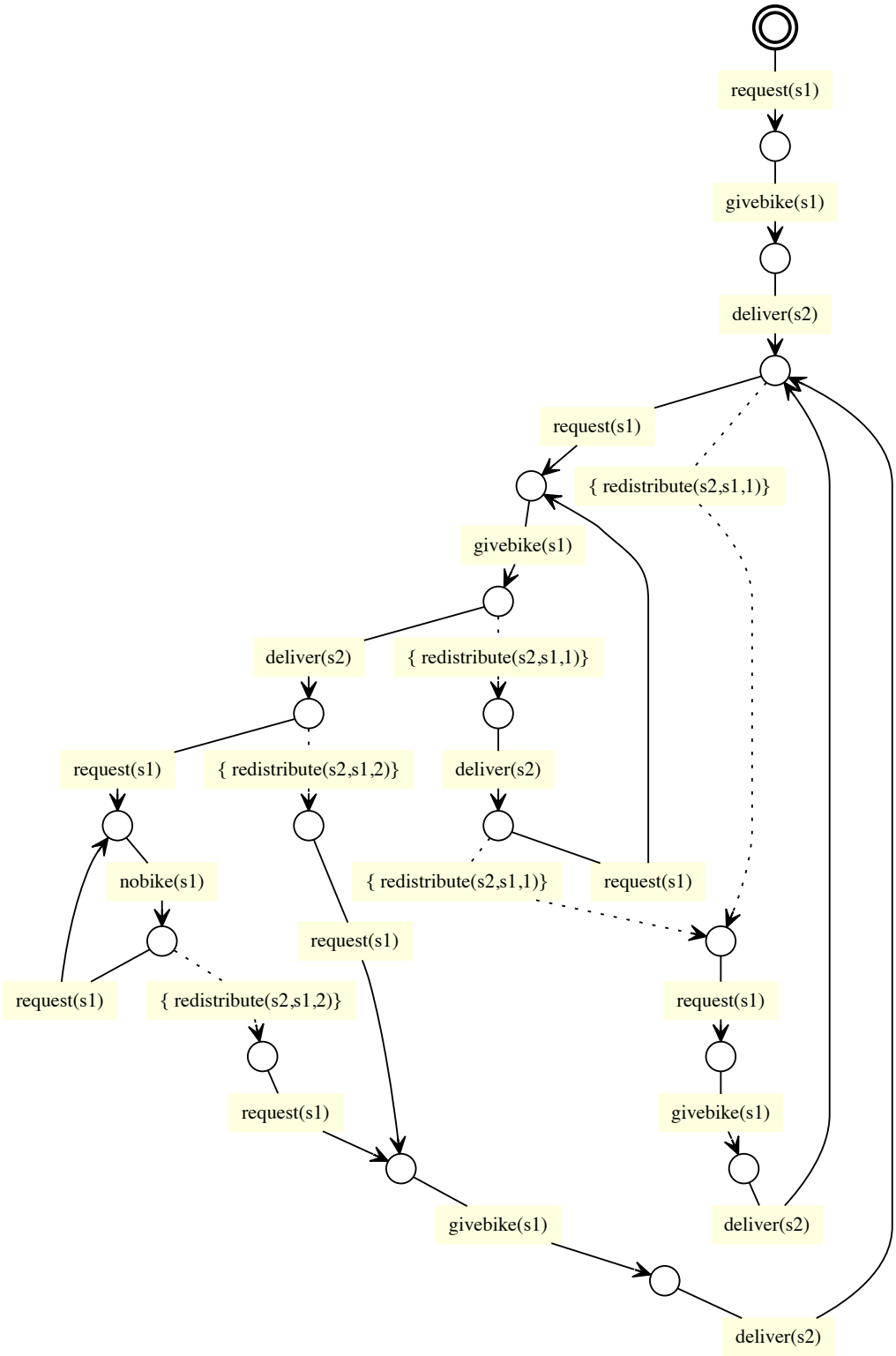


Figure 5: A family MTS of a BSS with 2 stations and 1 group of users generated by VMC

Acknowledgements

Research partly supported by the EU FP7-ICT FET-Proactive project QUANTICOL (600708) and by the Italian MIUR project CINA (PRIN 2010LHT4KM).

The authors would like to thank Marco Bertini from PisaMo s.p.a. for kindly sharing his expertise on the public BSS *CicloPi* with us.

References

[1] A. Antonik, M. Huth, K.G. Larsen, U. Nyman, and A. Wąsowski. 20 Years of Modal and Mixed Specifications. *Bulletin of the EATCS* 95 (2008), 94–129.

[2] L. Åqvist. Deontic Logic. In *Handbook of Philosophical Logic (2nd edition)*, vol. 8 (D. Gabbay and F. Guenthner, eds.). Kluwer, 2002, 147–264.

[3] P. Asirelli, M.H. ter Beek, A. Fantechi, and S. Gnesi. A Logical Framework to Deal with Variability. In *Proceedings 8th International Conference on Integrated Formal Methods (IFM'10)* (D. Méry and S. Merz, eds.). *LNCS* 6396, Springer, 2010, 43–58.

[4] P. Asirelli, M.H. ter Beek, A. Fantechi, and S. Gnesi. Formal Description of Variability in Product Families. In *Proceedings 15th International Software Product Line Conference (SPLC'11)* (E.S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, eds.). IEEE, 2011, 130–139.

[5] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming* 76, 2 (2011), 119–135.

[6] M.H. ter Beek, S. Gnesi, and F. Mazzanti. Demonstration of a model checker for the analysis of product variability. In *Proceedings 16th International Software Product Line Conference (SPLC'12)*. ACM, 2012, 242–245.

[7] M.H. ter Beek, A. Lluch Lafuente, and M. Petrocchi. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families In *Proceedings 17th International Software Product Line Conference (SPLC'13), Volume 2*. ACM, 2013, 10–17.

[8] M.H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A Tool for Product Variability Analysis. In *Proceedings 18th International Symposium on Formal Methods (FM'12)* (D. Giannakopoulou and D. Méry, eds.). *LNCS* 7436, Springer, 2012, 450–454.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (1986), 244–263.

[10] P. DeMaio. Bike-sharing: History, Impacts, Models of Provision, and Future. *Journal of Public Transportation* 12, 4 (2009), 41–56.

[11] R. De Nicola and F.W. Vaandrager. Actions versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes—Proceedings LITP Spring School on Theoretical Computer Science* (I. Guessarian, ed.), *LNCS* 469, Springer, 1990, 407–419.

[12] R. De Nicola and F.W. Vaandrager. Three Logics for Branching Bisimulation. *Journal of the ACM* 42, 2 (1995), 458–487.

[13] A. Fantechi and S. Gnesi. Formal Modelling for Product Families Engineering. In *Proceedings 12th Software Product Lines Conference (SPLC'08)*. IEEE, 2008, 193–202.

[14] A. Fantechi, A. Lapadula, R. Pugliese, F. Tiezzi, S. Gnesi, F. Mazzanti. A Logical Verification Methodology for Service-Oriented Computing. *ACM Transactions on Software Engineering and Methodology* 21, 3, Article 16 (2012), 1–46.

[15] D. Fischbein, S. Uchitel, and V.A. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proceedings ISSTA'06 Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA'06)* (R.M. Hierons and H. Muccini, eds.). ACM, 2006, 39–48.

[16] C. Fricker and N. Gast. Incentives and Redistribution in Bike-Sharing Systems with Stations of Finite Capacity. arXiv:1201.1178v3 [nlin.AO], September 2013.

[17] S. Gnesi and F. Mazzanti. On the Fly Verification of Networks of Automata. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. CSREA, 1999, 1040–1046.

[18] S. Gnesi and M. Petrocchi. Towards an Executable Algebra for Product Lines. In *Proceedings 16th International Software Product Line Conference (SPLC'12), Volume 2*. ACM, 2012, 66–73.

[19] A. Gruler, M. Leucker, and K.D. Scheidemann. Modeling and Model Checking Software Product Lines. In *Proceedings 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)* (G. Barthe and F.S. de Boer, eds.). *LNCS* 5051, Springer, 2008, 113–131.

[20] A. Gruler, M. Leucker, and K.D. Scheidemann. Calculating and Modeling Common Parts of Software Product Lines. In *Proceedings 12th International Software Product Line Conference (SPLC'08)*. IEEE, 2008, 203–212.

[21] K.G. Larsen. Proof systems for satisfiability in Hennessy-Milner Logic with recursion. *Theoretical Computer Science* 72, 2–3 (1990), 265–288.

[22] K.G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In *Proceedings 16th European Symposium on Programming (ESOP'07)* (R. De Nicola, ed.). *LNCS* 4421, Springer, 2007, 64–79.

[23] K. Lauenroth, K. Pohl, and S. Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings 24th International Conference on Automated Software Engineering (ASE'09)*. IEEE, 2009, 269–280

[24] K. Pohl, G. Böckle, and F.J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.