Model Checking Value-Passing Modal Specifications*

Maurice H. ter Beek, Stefania Gnesi, and Franco Mazzanti

ISTI–CNR
Via G. Moruzzi 1, Pisa, Italy
{terbeek,gnesi,mazzanti}@isti.cnr.it

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. We illustrate our approach by means of a simple yet intuitive example: a bike-sharing system.

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 [34]. 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.

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

Since many software-intensive systems are embedded, distributed and safety-critical, there is a need for rigour and formal modelling and verification (tools). Our contribution to make 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, together with a temporal logic that can be interpreted over that structure [3, 4]. We opted for Modal Transition Systems (MTSs) [29, 1], which were recognized in [22, 28, 30] as a useful formal method to describe in a compact way the possible operational behaviour of all products of a product family and in [26] to generate component-level MTSs from system level specifications. The most closely related approach is based on Featured Transition Systems (FTSs) [16], where actions are labelled with features and an associated feature model expresses feature constraints. A detailed comparison is given in [4]. We moreover defined an action-based branching-time CTL-like temporal modal logic over MTSs and we developed efficient algorithms to derive valid products from families and to model check properties over products and families alike. We implemented these algorithms in an experimental tool: the Variability Model Checker (VMC) [6, 8, 9]. Our approach thus differs from the more widespread use of LTL model checking MTSs [20, 13].

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

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 then 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.[1] This thus differs from the usual notion of MTS refinement [1, 22, 35].

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 $q \in Q_i$.*

2.1 A Modal Process Algebra

Rather than directly specifying the behaviour of a complex system in an MTS, it is often convenient to describe it in an abstract high-level language interpreted over MTSs. We consider a process algebra in which the parallel composition operator is parametrized by a set of actions to be synchronized, which contrasts the recent approaches in [31, 24, 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:*

$$N ::= [P] \qquad\qquad T ::= nil \mid K \mid A.T \mid T + T$$
$$P ::= K \mid P /L/ P \qquad\qquad A ::= a \mid a(may)$$

where $[P]$ denotes the complete system and K is a process identifier from the set of process definitions of the form $K \overset{def}{=} T$.

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:

[1] Actually, each product moreover needs to satisfy assumptions of *coherence* and *consistency* and *variability constraints* of the form alternative, excludes, and requires [9].

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.

Note that when restricted to must actions (i.e., LTSs) the rules for non-deterministic choice and parallel composition collapse onto the standard ones [33]. As is common for MTSs, synchronizing $a(may)$ with a results in $a(may)$ [35, 1].

$$(\textsc{sys}_\Box)\ \frac{P \xrightarrow{a} P'}{[P] \xrightarrow{a} [P']} \qquad (\textsc{sys}_\Diamond)\ \frac{P \overset{a}{\dashrightarrow} P'}{[P] \overset{a}{\dashrightarrow} [P']}$$

$$(\textsc{act}_\Box)\ \frac{}{a.P \xrightarrow{a} P} \qquad (\textsc{act}_\Diamond)\ \frac{}{a(may).P \overset{a}{\dashrightarrow} P}$$

$$(\textsc{or}_\Box)\ \frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'} \qquad (\textsc{or}_\Diamond)\ \frac{P \overset{a}{\dashrightarrow} P'}{P+Q \overset{a}{\dashrightarrow} P'}$$

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

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

$$(\textsc{par}_{\boxdot})\ \frac{P \xrightarrow{a} P' \quad Q \overset{a}{\dashrightarrow} Q'}{P\,/L/\,Q \overset{a}{\dashrightarrow} P'\,/L/\,Q'}\ a \in L$$

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

$$P+Q \equiv Q+P \qquad P+(Q+R) \equiv (P+Q)+R \qquad P \equiv P+0$$
$$P\,/L/\,Q \equiv Q\,/L/\,P \qquad P\,/L/\,(Q\,/L/\,R) \equiv (P\,/L/\,Q)\,/L/\,R \qquad P \equiv P[{}^{Q}/_{K}] \text{ iff } K \overset{\text{def}}{=} Q$$

Fig. 2. Structural congruence relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$

3 Dealing with Data

A critical point in the approach presented so far 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 present a case study to makes this clear.

3.1 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 [17]. 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 collaborate with "PisaMo S.p.A. azienda per la mobilità pisana", an in-house public mobility company of the Municipality of Pisa. 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 [23], 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), like 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.

4 Value-Passing Modelling and Verification Environment

We now extend the modelling and verification environment of § 2 to handle data. First, we extend the modal process algebra of § 2.1 with values and parameters.

4.1 A Value-Passing Modal Process Algebra

Definition 6. *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:*

$$\begin{aligned} N &::= [P] \\ P &::= K(e) \mid \; P\,/L/\,P \end{aligned}$$

where $[P]$ denotes a closed system and $K(e)$ is a process identifier from the set of process definitions of the form $K(v) \stackrel{def}{=} T$, and

$$\begin{aligned} T &::= nil \;\mid\; K(e) \;\mid\; A.T \;\mid\; T + T \;\mid\; [e \bowtie e]\,T \\ A &::= a(e) \;\mid\; a(may, e) \;\mid\; a(?v) \;\mid\; a(may, ?v) \\ e &::= v \;\mid\; \textbf{\textit{int}} \;\mid\; e \pm e \end{aligned}$$

*where $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ is a comparison relation, v is a variable, **int** is an integer, and $\pm \in \{+, -, \times, \div\}$ is an arithmetic operation.*

Also the semantics of this value-passing modal process algebra is given over MTSs, but we only provide the SOS rules for the must actions (in Fig. 3); the others follow straightforwardly from those in Fig. 1. In the structural congruence relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ defined in Fig. 2, the addition of value passing is reflected by replacing $P \equiv P[{}^{Q}/_{K}]$ iff $K \stackrel{\text{def}}{=} Q$ with $P \equiv P[{}^{Q[^{e}/v]}/_{K(e)}]$ iff $K(v) \stackrel{\text{def}}{=} Q$.

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 $a(?v)$.

The intuition of parallel composition is that both partners must fully and deterministically agree on the actual parameter values for the synchronization to occur. The rules in Fig. 3 refer to the case of just two parameters. In general, e.g., $a(X, 2).nil$ and $a(3, Y).nil$ can synchronize and perform the action $a(3, 2)$.

4.2 A Value-Passing Logic to Express Variability

We define value-passing v-ACTL, an action-based branching-time temporal logic for *variability* in the style of (action-based) CTL [18, 15] and Hennessy–Milner

$$(\mathrm{SYS})\ \frac{P \xrightarrow{a(e)} P'}{[P] \xrightarrow{a(e)} [P']} \qquad (\mathrm{ACT}_\Box)\ \frac{}{\alpha.P \xrightarrow{\alpha} P}\ \alpha\in\{a(e),a(?v)\}$$

$$(\mathrm{OR}_\Box)\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}\ \alpha\in\{a(e),a(?v)\} \qquad (\mathrm{INT}_\Box)\ \frac{P \xrightarrow{\ell} P'}{P\,/L/\,Q \xrightarrow{\ell} P'\,/L/\,Q}\ \ell\notin L$$

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

$$(\mathrm{GUARD})\ \frac{}{[e_1 \bowtie e_2]\,P(e_3) \longrightarrow P(e_3)}\ e_1 \bowtie e_2$$

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

Logic (HML) with Until defined in [27, 19]. Next to the operators of propositional logic, v-ACTL contains the classical box and, by duality, diamond modal operators from HML, the existential and universal path quantifiers and next operator from CTL and the (action-based) F and, by duality, G operators from ACTL, as well as the (action-based) Until and Weak until operators U and W drawn from those firstly introduced in [18] and elaborated in [32]. For the box, diamond and F operators, v-ACTL also contains a *deontic* interpretation that takes the modality (or 'deonticity') of the transitions (may or must) into account. In the SPLE context, these deontic interpretations allow to suitably capture behavioural properties over MTSs that are inherited by all its product LTSs. More on this and 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 7. *Action formulas are built over a set $\mathcal{A}$ of* actions*, where $a \in \mathcal{A}$:*

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

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 8. *Let $a, b \in \mathcal{A}$. The satisfaction of formula ψ by $a(e)$, denoted by $a(e) \models \psi$, is defined as:*

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

Definition 9. *The syntax of v-ACTL is:*

$$\begin{aligned}
\phi \ &::= \ \mathit{true} \mid \neg\,\phi \mid \phi \wedge \phi \mid [\psi]\,\phi \mid [\psi]^{\Box}\,\phi \mid E\,\pi \mid A\,\pi \mid \mu Y.\phi(Y) \mid \nu Y.\phi(Y) \\
\pi \ &::= \ [\phi\ \{\psi\}\,U\,\{\psi'\}\ \phi'] \mid [\phi\ \{\psi\}\,U\ \phi'] \mid [\phi\ \{\psi\}\,W\,\{\psi'\}\ \phi'] \mid [\phi\ \{\psi\}\,W\ \phi'] \mid \\
&\quad\ X\,\{\psi\}\ \phi \mid F\,\phi \mid F^{\Box}\,\phi \mid F\,\{\psi\}\,\phi \mid F^{\Box}\,\{\psi\}\,\phi
\end{aligned}$$

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 (e.g., the Until and 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.
$X\{\psi\}\,\phi:$ in the next state of the path, reached by an action satisfying ψ, ϕ 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.
$\phi\;\{\psi\}\,U\,\{\psi'\}\;\phi':$ in a future state (reached by an action satisfying ψ'), ϕ' holds, while ϕ holds from the current state until that state is reached and all actions executed in the meantime along the path satisfy ψ.
$\phi\;\{\psi\}\,W\,\{\psi'\}\;\phi':$ either $\phi\;\{\psi\}\,U\,\{\psi'\}\;\phi'$ or ϕ holds from the current state onwards and all actions executed along the path satisfy ψ.

The semantics of v-ACTL is interpreted over MTSs. Let $path(q)$ denote the set of all full paths from a state q. Moreover, for a path $\sigma = q_1 a_1(e_1) q_2 a_2(e_2) q_3 \cdots$, we denote its ith state (i.e., q_i) by $\sigma(i)$ and its ith action (i.e., $a_i(e_i)$) by $\sigma\{i\}$.

Definition 10. *Let $(Q, A, \overline{q}, \delta^{\Diamond}, \delta^{\Box})$ be an MTS, with $q \in Q$ and $\sigma \in path(q)$. The satisfaction relation $\models$ of v-ACTL is defined as:*

$q \models true$ *always holds*
$q \models \neg\,\phi$ *iff* $q \not\models \phi$
$q \models \phi \wedge \phi'$ *iff* $q \models \phi$ *and* $q \models \phi'$
$q \models [\psi]\,\phi$ *iff* $\forall\, q' \in Q$ *such that* $q \xrightarrow{a(e)}_{\Diamond} q'$ *and* $a(e) \models \psi$, *we have* $q' \models \phi$
$q \models [\psi]^{\Box}\,\phi$ *iff* $\forall\, q' \in Q$ *such that* $q \xrightarrow{a(e)}_{\Box} q'$ *and* $a(e) \models \psi$, *we have* $q' \models \phi$
$q \models E\,\pi$ *iff* $\exists\,\sigma' \in path(q) \colon \sigma' \models \pi$
$q \models A\,\pi$ *iff* $\forall\,\sigma' \in path(q) \colon \sigma' \models \pi$
$q \models \mu\,Y.\phi(Y)$ *iff* $\bigvee_{i \geq 0}\ \phi^i(false)$
$q \models \nu\,Y.\phi(Y)$ *iff* $\bigwedge_{i \geq 0}\ \phi^i(true)$
$q \models X\{\psi\}\,\phi$ *iff* $\sigma\{1\} \models \psi$ *and* $\sigma(2) \models \phi$
$q \models F\,\phi$ *iff* $\exists\, j \geq 1 \colon \sigma(j) \models \phi$
$q \models F^{\Box}\,\phi$ *iff* $\exists\, j \geq 1 \colon \sigma(j) \models \phi$ *and* $\forall\, 1 \leq i < j \colon (\sigma(i), \sigma\{i\}, \sigma(i+1)) \in \delta^{\Box}$
$q \models F\,\{\psi\}\;\phi$ *iff* $\exists\, j \geq 1 \colon \sigma\{j\} \models \psi$ *and* $\sigma(j+1) \models \phi$

$$q \models F^{\Box}\,\{\psi\}\ \phi \quad \textit{iff} \ \ \exists\, j \geq 1 \colon \sigma\{j\} \models \psi \ \textit{and} \ \sigma(j+1) \models \phi,$$
$$\textit{and} \ \forall\, 1 \leq i \leq j \colon (\sigma(i), \sigma\{i\}, \sigma(i+1)) \in \delta^{\Box}$$
$$\sigma \models \phi\ \{\psi\}\, U\, \{\psi'\}\ \phi' \quad \textit{iff} \ \ \exists\, j \geq 1 \colon \sigma(j) \models \phi',\ \sigma\{j\} \models \psi', \textit{and} \ \sigma(j+1) \models \phi',$$
$$\textit{and} \ \forall\, 1 \leq i < j \colon \sigma(i) \models \phi \ \textit{and} \ \sigma\{i\} \models \psi$$
$$\sigma \models \phi\,\{\psi\}\, W\, \{\psi'\}\, \phi' \ \textit{iff} \ \sigma \models \phi\,\{\psi\}\, U\, \{\psi'\}\, \phi' \ \textit{or} \ \forall j \geq 1 \colon \sigma(j) \models \phi \ \textit{and} \ \sigma\{j\} \models \psi$$

$\langle\psi\rangle\, \phi$ 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].

4.3 Model Checking Value-Passing Modal Specifications

The modelling and verification environment described so far has been implemented in the Variability Model Checker (VMC) [6, 8, 9], which is freely usable online (http://fmt.isti.cnr.it/vmc/). VMC accepts as input a model specified in the value-passing modal process algebra presented in § 4.1 and it allows to verify properties expressed in the value-passing v-ACTL logic presented in § 4.2.

We are unaware of other model-checking tools for MTSs that support value passing. MTSA [20] is a prototype, built on top of the LTS Analyser LTSA, for the analysis of MTSs specified in an extension of the process algebra FSP (Finite State Processes). MTSA allows 3-valued FLTL (Fluent LTL) model checking of MTSs by reducing the verification to two FLTL model-checking runs on LTSs.

VMC is the most recent product of a family of model checkers we developed at ISTI–CNR over the past two decades, including UMC [5] and CMC [21]. Each 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 [15], including ACTL [18]. 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 [5] 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], on-the-fly model checking v-ACTL formulas (without fixed points) over MTSs can be achieved in a complexity that is linear w.r.t. the size of the state space. It is beyond the scope of this paper to give detailed descriptions of the model-checking algorithms and architecture underlying this family of model checkers (for which we refer to [5]).

5 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 also user behaviour, 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 product LTSs depicted in Fig. 4.

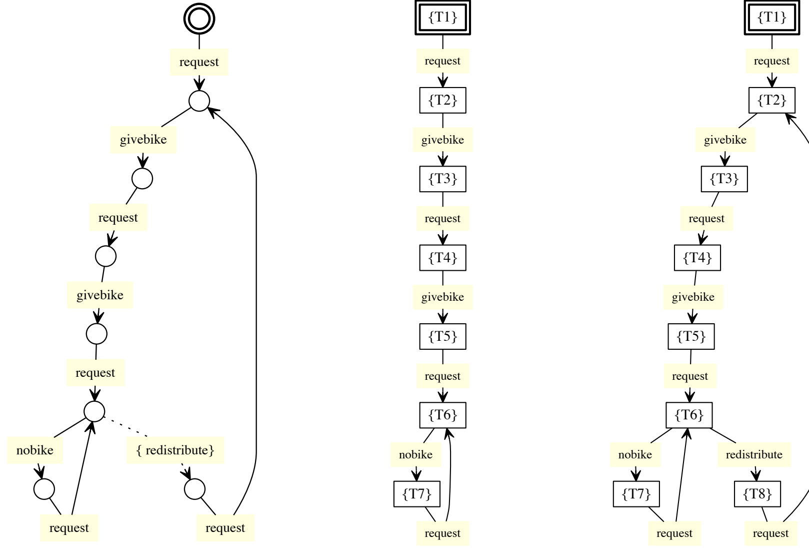
To illustrate what kind of variability analyses can be performed with the extended value-passing modelling and verification environment introduced in § 4, we now present a few properties and the result of model checking them with VMC against the above family of BSS (i.e., on the MTS depicted in Fig. 4(a)):[2]

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.

[2] In VMC, $\neg$, $\vee$, $\wedge$, $[\,]^{\Box}$, μ, ν, and $F^{\Box}$ are written as `not`, `or`, `and`, `[]#`, `min`, `max`, and `F#`, respectively, whereas ‘ * ’ can be used as ‘don’t care’ symbol for parameter values.



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

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

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]. 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 gives (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) ) +
  return(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).return(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,return/ 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 (i.e., `net USERS = Users(s1,s2)`); in case of a BSS with two user groups (i.e., `net USERS = Users(s1,s2) // Users(s2,s1)`) the MTS has 224 states.[3]

For the family of BSS with one user group, we present some properties and the result of model checking them with VMC (i.e., on the MTS depicted in Fig. 5):

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

Eventually it may occur that station 2 has no more bikes: $EF\,\{\mathit{nobike(s2)}\}\ \mathit{true}$. This formula however is false. (Note that it is true in case of two user groups.)

For all products, if redistribution is implemented, then it is always the case that eventually station 1 gives the user a bike: $(\neg\, EF\,\{\mathit{redistribute(*,s1,*)}\}\ \mathit{true}) \vee (AG\,EF\,\{\mathit{givebike(s1)}\}\ \mathit{true})$. This formula is actually true for all products (LTS) of the family (MTS in Fig. 5). However, it does not make much sense to verify this formula over the MTS, since it is not expressed in the specific fragment of v-ACTL that has the characteristic that any formula expressed in it and which is true for the MTS, is also true for all its products (cf. [9]).

6 Conclusions and Future Work

In this paper we have presented some of the recent developments concerning our ongoing research effort to elaborate a rigorous modelling and verification environment for behavioural variability analyses of product families. These developments, which concern the extension of both the input language of VMC and its logic to be able to deal with (integer) value-passing, stem from the fact that we realized that a major limitation for applying our approach to realistic case studies from industry is the lack of a possibility to model an adequate representation of the data that may need to be described.

[3] In VMC, text or code can be commented out by prefixing it with `--`.

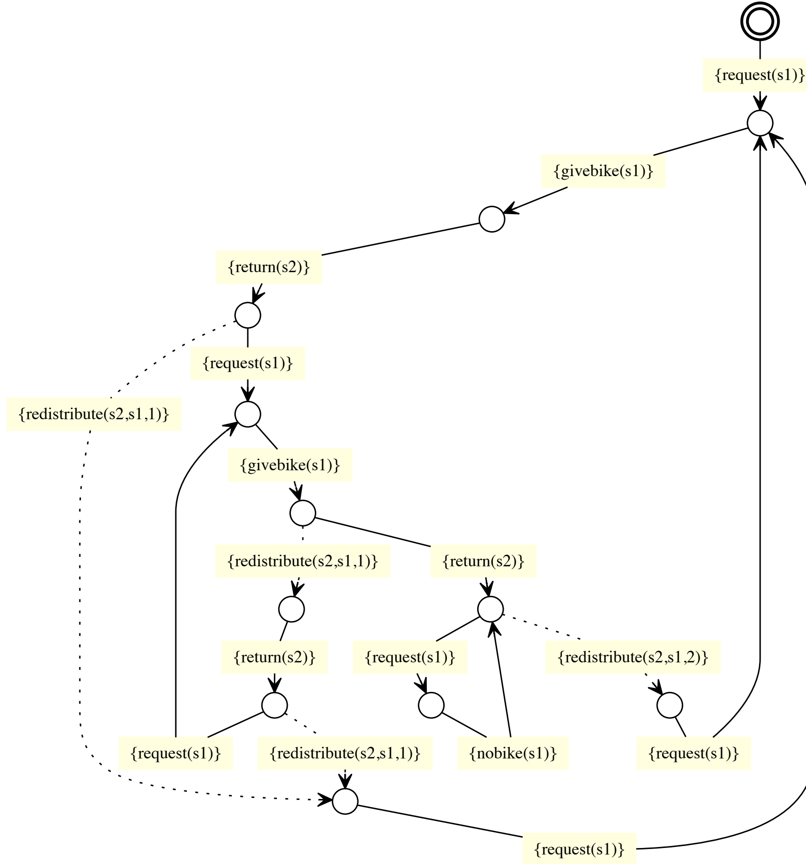


Fig. 5. A family MTS of a BSS with 2 stations and 1 group of users generated by VMC

This paper is only a first contribution to removing this limitation as it defines an extension of the environment that can deal with data in the form of integer value-passing. In particular, VMC 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.

It thus remains to extend the data handling in VMC to more than just integers. to this aim, we might turn to the mCRL2 toolset (http://www.mcrl2.org) for inspiration, since it allows to model actions parametrized with user-defined abstract datatypes and to verify formulas in the modal μ-calculus, thus allowing to quantify over data [25]. Moreover, also mCRL2 is recently being used for product family analysis [10–12].

In this paper we furthermore illustrated the new features of VMC by means of simple yet intuitive examples from a case study on bike-sharing systems originating from the EU FP7 project QUANTICOL (http://www.quanticol.eu).

In the future, we intend to further investigate the application of the modelling and verification environment presented in this paper 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 can be distinguished in a product family. A promising starting point could be the results on generalized model checking [14].

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.

Acknowledgments We thank Marco Bertini from PisaMo S.p.A. for generously sharing with us his knowledge on bike-sharing systems in general and *CicloPi* in particular.

References

1. A. Antonik, M. Huth, K.G. Larsen, U. Nyman, and A. Wąsowski. 20 Years of Modal and Mixed Specifications. *Bulletin EATCS* 95 (2008), 94–129.
2. L. Åqvist. Deontic Logic. In *Handbook of Philosophical Logic (2nd edition)*, vol. 8. (D. Gabbay, 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 *IFM'10*. *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 *SPLC'11*. 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. *Sci. Comput. Program.* 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 *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 *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 *FM'12*. *LNCS* 7436, Springer, 2012, 450–454.
9. M.H. ter Beek and F. Mazzanti. VMC: Recent Advances and Challenges Ahead. In *SPLC'14, Volume 2*. ACM, 2014, 70–77.
10. M.H. ter Beek and E.P. de Vink. Using mCRL2 for the analysis of software product lines. In *FormaliSE'14*. IEEE, 2014.
11. M.H. ter Beek and E.P. de Vink. Software Product Line Analysis with mCRL2. In *SPLC'14, Volume 2*. ACM, 2014, 78–85.
12. M.H. ter Beek and E.P. de Vink. Towards Modular Verification of Software Product Lines with mCRL2. In *ISoLA'14*. *LNCS*, Springer, 2014.
13. N. Benes, I. Cerná, and J. Kretínský. Modal Transition Systems: Composition and LTL Model Checking. In *ATVA'11*. *LNCS* 6996, Springer, 2011, 228–242.

14. G. Bruns and P. Godefroid. Generalized Model Checking: Reasoning about Partial State Spaces. In *CONCUR'00*. *LNCS* 1877, Springer, 2000, 168–182.
15. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS* 8, 2 (1986), 244–263.
16. A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking. *IEEE TSE* 39, 8 (2013), 1069–1089.
17. P. DeMaio. Bike-sharing: History, Impacts, Models of Provision, and Future. *J. Public Transportation* 12, 4 (2009), 41–56.
18. R. De Nicola and F.W. Vaandrager. Actions versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes*. *LNCS* 469, Springer, 1990, 407–419.
19. R. De Nicola and F.W. Vaandrager. Three Logics for Branching Bisimulation. *J. ACM* 42, 2 (1995), 458–487.
20. N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. MTSA: The Modal Transition System Analyser. In *ASE'08*, IEEE, 2008, 475–476.
21. A. Fantechi, A. Lapadula, R. Pugliese, F. Tiezzi, S. Gnesi, and F. Mazzanti. A Logical Verification Methodology for Service-Oriented Computing. *ACM TOSEM* 21, 3, Article 16 (2012), 1–46.
22. D. Fischbein, S. Uchitel, and V.A. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *ROSATEA'06*. ACM, 2006, 39–48.
23. C. Fricker and N. Gast. Incentives and Redistribution in Bike-Sharing Systems with Stations of Finite Capacity. arXiv:1201.1178v3 [nlin.AO], September 2013.
24. S. Gnesi and M. Petrocchi. Towards an Executable Algebra for Product Lines. In *SPLC'12, Volume 2*. ACM, 2012, 66–73.
25. J.F. Groote and R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In *AMAST'98*. *LNCS* 1548, Springer, 1999, 74–90.
26. I. Krka, G. Edwards, Y. Brun, and N. Medvidovic. From system specifications to component behavioral models. In *ICSE'09*. IEEE, 2009, 315–318.
27. K.G. Larsen. Proof systems for satisfiability in Hennessy-Milner Logic with recursion. *Theoret. Comput. Sci.* 72, 2–3 (1990), 265–288.
28. K.G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In *ESOP'07*. *LNCS* 4421, Springer, 2007, 64–79.
29. K.G. Larsen and B. Thomsen. A Modal Process Logic. In *LICS'88*. IEEE, 1988, 203–210.
30. K. Lauenroth, K. Pohl, and S. Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *ASE'09*. IEEE, 2009, 269–280.
31. M. Leucker and D. Thoma. A Formal Approach to Software Product Families. In *ISoLA'12*. *LNCS* 7609, Springer, 2012, 131–145.
32. R. Meolic, T. Kapus, and Z. Brezocnik. ACTLW: An action-based computation tree logic with unless operator. *Inform. Sci.* 178, 6 (2008), 1542–1557.
33. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
34. K. Pohl, G. Böckle, and F.J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
35. G.E. Sibay, S. Uchitel, V.A. Braberman, and J. Kramer. Distribution of Modal Transition Systems. In *FM'12*. *LNCS* 7436, Springer, 2012, 403–417.