VMC: Recent Advances and Challenges Ahead

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
terbeek@isti.cnr.it

Franco Mazzanti
ISTI–CNR, Pisa, Italy
mazzanti@isti.cnr.it

ABSTRACT

The variability model checker VMC accepts a product family specified as a Modal Transition System (MTS) with additional variability constraints. Consequently, it offers behavioral variability analyses over both the family and its valid product behavior. This ranges from product derivation and simulation to efficient on-the-fly model checking of logical properties expressed in a variability-aware version of action-based CTL. In this paper, we first explain the reasons and assumptions underlying the choice for a modeling and analysis framework based on MTSs. Subsequently, we present recent advances on proving inheritance of behavioral analysis properties from a product family to its valid products. Finally, we illustrate challenges remaining for the future.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

General Terms

Design, Experimentation, Verification

Keywords

Product families, Behavioral variability, Model checking

1. INTRODUCTION

Software Product Line Engineering (SPLE) is now an established software-intensive system development technique which propagates the systematic reuse of assets or features (i.e. user-visible product characteristics or aspects of which some are common to all family members while others are only shared by some) in an attempt to lower production costs and time-to-market but increase overall efficiency. Guaranteeing the correctness of software (components) intended for systematic reuse and overall correctness of the developed product family are thus important for SPLE. This means adapting traditional quality assurance techniques, such as model checking, to cope with variability. We contributed to this by developing the variability model checker VMC. VMC accepts a product family modeled as a modal process algebra plus variability constraints. It offers behavioral variability analyses over both the family and its valid product behavior, ranging from product derivation and simulation to efficient on-the-fly model checking of logical properties expressed in v-ACTL, a variability-aware version of action-based CTL.

The recent advances of VMC concern an extension of its input language to a modal process algebra sustaining deontic synchronization operators, value-passing communication and n-ary variability constraints, and a thoroughly revised version of the supported v-ACTL logic with user notifications in case a family-based analysis result is guaranteed to be preserved by all products. The most important challenge ahead of us is to experiment VMC on a realistic, industrial-size case study. To this aim, we need to enrich the process algebra with advanced data types, while we might want to implement some product line-specific features into VMC's model-checking algorithms. Finally, it might be interesting to define explicit behavioral variability constraints, possibly allowing parameter values in constraints. More on this later.

After briefly describing the family of model checkers developed the last two decades at ISTI–CNR in §2, we explain the reasons and assumptions underlying the choice for a SPL modeling and analysis framework based on MTSs in §3. Subsequently, in §4 and §5, we describe in more detail how the SPL analysis tool VMC implements MTSs and how it can perform SPL analyses by means of a variability-aware logic interpreted over MTSs. In §6, we discuss related tools, after which we provide an example in §7. Finally, in §8 we conclude the paper and list some challenges left for the future.

2. A FAMILY OF MODEL CHECKERS

Our experiments with on-the-fly model checking began with the FMC model checker for ACTL extended with fixed-point operators [24]. Its computational model was a network of automata built from terms of a process algebra derived from the value-passing CCS/CSP-like calculus. On-the-fly verification means that in general not the whole state space needs to be generated and explored, improving performance and allowing to partly deal also with infinite-state systems.

The same model-checking approach was later applied to a computational model based on UML state machines. This required switching to an action- and state-based logic, which allows one to express in a natural way not only properties of

SPLC '14, September 15 - 19 2014, Florence, Italy

http://dx.doi.org/10.1145/2647908.2655969

evolution steps (i.e. related to the executed actions) but also internal properties of states (e.g. values of object attributes). This resulted in the UMC model checker for UCTL logic [6].

A third application of our approach concerned the process algebra COWS [27], which is a specification language specifically developed for the design and orchestration of service-oriented systems. Verifying such systems requires properties to be able to express the correlation between dynamically generated values appearing inside actions at different times. This led to the the CMC model checker for SocL logic [22].

These three approaches recently were integrated into a common framework [25] whose behavioral semantic model is an abstract Doubly-Labeled Transition System (L^2TS) [18]. An L^2TS is an extension of an ordinary Labeled Transition System (LTS) in which not only edges but also states can be associated with sets of (parametric) labels. This allows us to define and implement the on-the-fly logical verification engine on the abstract L^2TS in a way that is completely independent from the details of the underlying computational model (either based on UML state machines or a process calculus). It is the task of the specification language and computational model to define what kind of information is to be mapped from the ground internal structure of the computational model onto the abstract form of labels in the L^2TS.

VMC is the most recent extension of this framework. VMC was specifically introduced a few years ago to support behavioral variability analysis of product families [7,10]. It accepts the specification of a Modal Transition System (MTS) [1,30] in process-algebraic terms plus an optional set of variability constraints. An MTS is an extension of an LTS in which two distinct types of transitions (admissible and necessary) are distinguished (cf. §3). VMC allows to perform two kinds of behavioral variability analyses on a given family of products.

1. The actual set of valid product behavior can explicitly be generated and the resulting LTSs can be verified against the same logic property (expressed in ACTL).

2. A logic property (expressed in v-ACTL) can directly be verified against the MTS modeling the product family behavior, relying on the fact that under certain syntactic conditions validity over the MTS guarantees validity of the same property for all the family's products.

As for all model checkers of our family, the core of VMC is constituted by a command-line version of the tool written in Ada, which can be easily compiled for the Windows, Linux, Solaris and Mac OS X platforms. These core executables are wrapped with CGI scripts handled by a web server, facilitating an html-oriented GUI and integration with graph drawing tools. Its development is still ongoing, but the current version of the tool is freely usable online (`fmt.isti.cnr.it/vmc/`). It is beyond the scope of this paper to give detailed descriptions of the model-checking algorithms and architecture underlying our family of model checkers. The interested reader can consult [6, 22, 24].

3. MODELING SPLs WITH MTSs

An MTS [1,30] is an LTS distinguishing 'admissible' may from 'necessary' must transitions. By definition, every must transition is a may transition. Graphically, an MTS is drawn (cf. Fig. 1(left)) as a directed edge-labeled graph where nodes model states (with a black initial state) and edges model transitions (labeled with actions). Solid edges model necessary transitions while dotted edges model transitions that are admissible but not necessary. A path in the graph models a sequence of state changes as the result of executing actions; if all edges are must transitions, it is a must path.

In [23], MTSs were first recognized as a compact model for describing the possible operational behavior of products in a product family. Over the years many variants and extensions of MTSs for SPLE were proposed (e.g. [2, 4, 20, 21, 29, 31]).

Implementations of an MTS are LTSs that capture the idea of refining a partial description into a more detailed one, reflecting increased knowledge on admissible (but not necessary) behavior. This fits the SPLE notion that each product of a product family is a refinement of that family, based on the understanding that an LTS (product behavior) conforms to the MTS (family behavior). The relation which binds an MTS to its associated set of products (implementations) is defined by a refinement relation between the MTS and the product LTSs. The classical definition of this refinement relation allows an infinite set of implementations to be associated as possible products to a given MTS. This explosion comes from the unbounded expansion of loops in the MTS, which may lead to an infinite set of potentially not bisimilar LTSs. In fact, in [21, 23] it was noted that refinement need not preserve an MTS's branching structure.

Figure 1 shows an MTS (on the left) and two of its implementations (following the classical definition [1, 30]). Note that the LTSs (on the right) are just two elements of the infinite set of possible implementations which may result from unfolding the a-labeled transition and from the possible instantiation, at each step, of the optional b-transition.

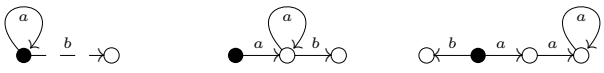

Figure 1: An MTS and two of its implementations

In our SPLE context, instead, we believe it to be more useful to have a simpler notion of refinement which preserves the exact original branching structure of the MTS in the products, corresponding to the modelers' intuition, and which gives to the implementations the choice of just turning dotted edges into solid edges or removing them altogether. This stricter notion of refinement has the immediate advantage of leading to a limited set of product behaviors. For instance, the only acceptable products of the MTS in Fig. 2(left) are the two LTSs depicted on the right, i.e. the dotted edge in the MTS is either turned into a solid edge or removed.

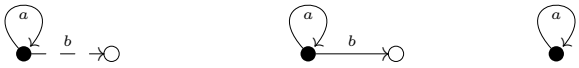

Figure 2: The MTS and its only two valid products

Following [2–5, 12–15, 20, 21, 23, 29, 31], an action models a piece of functionality (a feature if you like). We therefore assume that no solid edge is labeled with the same action as a (different) dotted edge, i.e. no necessary a-transition exists if the set of admissible but not necessary transitions contains an a-transition. This is the assumption of *coherence*.

We moreover assume behavioral variability to be resolved consistently: A decision to 'implement' an admissible but not necessary a-transition in a product must be *consistent*

throughout the product (recall that a dotted edge of an MTS modeling family behavior can be preserved or removed in an LTS modeling product behavior). This assumption reflects the fact that functionality (or a feature) either is or is not present in a product, independent of its behavioral context. The only consistent products of the MTS in Fig. 3(left) are thus the two LTSs on the right, i.e. an LTS with only one b-transition does not model acceptable product behavior.

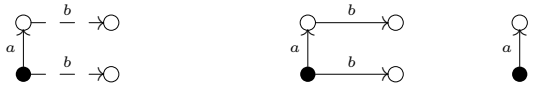

Figure 3: An MTS and its only two valid products

While an MTS thus naturally caters for optional behavior, it cannot capture all common variability notions (e.g. alternative, requires and excludes constraints). In [2, 4] we therefore opted for the addition of a set of *variability constraints* to be taken into account when deriving products to guarantee only LTSs representing valid product behavior.

If the MTSs depicted in Figs. 2-3 were accompanied by the alternative constraint a ALT b, then only the rightmost LTSs in these figures would represent valid product behavior (i.e. the optional b-transition would need to be removed to satisfy the constraint stating that a and b are alternatives).

Possible variability constraints over a set $\{ a_i \mid 1 \leq i \leq n \}$ of actions and their ACTL (cf. Appendix B) semantics are:[1]

a_1 ALT a_2 ALT $\cdots$ ALT a_n :
$\bigvee_{1 \leq i \leq n}((EF\,\{a_i\}\,true) \bigwedge_{1 \leq j \neq i \leq n}(\neg\, EF\,\{a_j\}\,true))$

a_1 OR a_2 OR $\cdots$ OR a_n : $\bigvee_{1 \leq i \leq n}(EF\,\{a_i\}\,true)$

a_j EXC a_k : $(EF\,\{a_j\}\,true) \implies (\neg\, EF\,\{a_k\}\,true) \wedge$
$(EF\,\{a_k\}\,true) \implies (\neg\, EF\,\{a_j\}\,true)$

a_j REQ a_k : $(EF\,\{a_j\}\,true) \implies (EF\,\{a_k\}\,true)$

Note that syntactic constraints on actions (i.e. defined in terms of their *presence* in a product) are thus checked semantically (i.e. defined in terms of their *reachability* in the LTS).

Summarizing, we have described a modeling framework in which an MTS, together with an additional set of variability constraints, specifies the behavior of a product family in a compact way. Derivation of all valid products leads to a limited set of LTSs specifying the possible product behavior. In the next sections we describe how VMC can be used to perform behavioral analysis over families and products alike.

4. MODELING MTSs WITH L^{2}TSs

The FMC/UMC/CMC model checking framework is based on the notion of an L^{2}TS as underlying abstract semantic computational model. An MTS differs from an LTS only by distinguishing two possible kinds of transitions, which can be necessary or admissible. This aspect can easily be encoded in an L^{2}TS by extending the labels on the transitions with information about the modality of the edge. We recall that in the L^{2}TSs that we consider, edges can be associated with *sets* of labels, so it is quite straightforward to add also this additional information. Moreover, the information on the modality of the transitions must in some way be specified also in the syntax of the process calculus used to specify the MTS. Our choice has been to model it as a special additional parameter associated to the basic actions of the calculus.

In Fig. 4 we show an example MTS we want to encode (and a formula over it) on the left, its syntax in terms of the process algebra in the centre, and its encoding in terms of an L^{2}TS (and the corresponding formula) on the right.

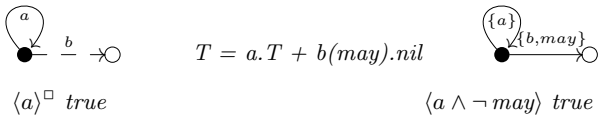

Figure 4: From MTS + v-ACTL$^\Box$ to L^{2}TS + ACTL

The logical verification engine of the model checking framework has no problems in analyzing the L^{2}TS derived in this way from an MTS and additional variability-aware logical operators can easily be defined through translations into the classical ACTL operators (cf. Appendix B). In Fig. 4, e.g., we show the way a v-ACTL$^\Box$ formula over an MTS is interpreted over an L^{2}TS. This deontic ('boxed') version of the classical diamond operator from Hennessy-Milner logic requires that a next state exists, reachable by a necessary (must) transition labeled with action a, in which *true* holds.

If the MTS contains no optional transitions, then the encoding of the MTS in VMC becomes precisely the standard encoding of an ordinary LTS (with no need for state labels).

More details on the syntax and semantics of the value-passing modal process algebra accepted by VMC are given in Appendix A, while more on the syntax and semantics of the variability-aware ACTL-like logic accepted by VMC for the specification of properties of the MTS and its products is given in Appendix B. Full details of both are given in [8].

5. SPL ANALYSIS STRATEGIES

It is by now common to classify SPL analysis techniques into *product-based* and *family-based* analyses [35]. The former operate on individually generated products (or at most a subset), whereas the latter operate on an entire product family at once using variability knowledge about valid feature configurations to deduce results for products.

5.1 Verifying Properties over the MTS

The most interesting and efficient way to verify properties of a product family relies on the feasibility of performing the verification directly on the MTS modeling the family (i.e. to perform a family-based analysis). Indeed, in Appendix B we define a thoroughly revised version of the logic v-ACTL$^\Box$ which guarantees (cf. [4, 8]) that whenever a formula holds for the MTS, then it also holds for all products of the family. Moreover, VMC now automatically notifies the user in case a formula was verified to which such a preservation applies.

A first limit of this approach is that only the validity of a formula (with respect to an MTS) is preserved by the products. If a formula does not hold for a family, nothing can be said on its validity for the products.

A second limit is that this kind of verification does not take the additional constraints associated to the MTS into account. Therefore if a property holds for the MTS, then it is guaranteed that it holds for all (valid or not) products. Conversely, any property which holds for just the valid products (i.e. there is at least one not valid product for which it

[1]Support for n-ary constraints (and more, cf. §8) is recent.

does not hold), then it cannot hold for the MTS. The logic formulations of the variability constraints in §3 are simple examples of properties that hold for just the valid products, but which do not hold for the MTS.

A third limit of this approach is the logic's expressive power, which does not allow to express certain interesting properties over the family of products. It would, e.g., be nice to express that a certain property φ holds for all products that eventually execute an optional action b (i.e. support a certain feature) but this kind of property is not among those that can be expressed and evaluated in the context of MTSs.

5.2 Verifying Properties over the LTSs

Another possibility offered by VMC, which exploits the assumptions of coherence, consistency and product validity, is to generate all valid products of the family (i.e. the finite set of consistent products satisfying all variability constraints) and to directly verify a property on each and every one of them (i.e. to perform a product-based analysis). In this case the logic does not even need to consider variability (since the formula is just evaluated on the products, i.e. LTSs) so we can resort to the full action-based fragment of our verification framework's logic (i.e. ACTL as defined in Appendix B).

The generation of the valid products of an MTS needs to evaluate for each of them the reachability of the actions occurring in the variability constraints. This task is performed by VMC in an incremental way, analyzing the MTS structure in a breadth-first fashion, generating new templates for groups of sub-products each time an optional transition is encountered and removing invalid templates as soon as their structure results incompatible with the required constraints.

This approach has proved to be rather effective and so far the generation of all valid product has never proved to be a bottleneck for verification, but we must admit that the scalability to realistic problems of industrial size has so far never been investigated nor attempted. This is future work.

6. VMC AND RELATED TOOLS

Several of the behavioral variability models mentioned in this paper come with an SPL tool for product-/family-based analysis with verification techniques like model checking.

SNIP [12] is a model checker for product families modeled as Featured Transition Systems (FTSs) specified in a language based on that of SPIN. The feature diagram is coded in the textual variability language TVL to be consulted by the explicit-state on-the-fly model-checking algorithm of SNIP to verify properties expressed in a feature-aware version of LTL interpreted over FTSs (e.g. to verify a property over only a subset of the valid products). Symbolic FTS model checking [13] was implemented as an extension of NuSMV, with a fully symbolic algorithm for a feature-aware version of CTL. In SNIP, special-purpose exhaustive model checking algorithms (continuing a search also after a violation was found) allow the user to verify all products of a product line at once and to output counterexamples for all products that violate a property (in contrast with the NuSMV extension that only produces a counterexample for the first violating product found). SNIP has recently been re-engineered into the tool suite ProVeLines [16].

In VMC there is no explicit reference to features and feature models, which is one of the more obvious differences[2] with these successful approaches based on FTSs, in which transitions are labeled with actions *and* features and a feature model encoding *is* included (basically, the set of features and the set of valid products in terms of their features).

In [34], Finite State Machines (FSMs) are extended with variability by means of guards over variables on transitions and a global predicate defining the valid configurations by value assignments. For each product line feature, two FSMs are built, one for the requirements and one for the design level, and it is specified how to check their conformance. This check is implemented by a never claim in SPIN. The prototypical tool SPLEnD transforms pairs of XML files for the FSMs into a file that can be fed to SPIN, which either returns the conformance mappings or declares nonconformance, after which the behavior of the product line can be checked by SAT solving.

We conjecture that there exists a trade-off between brute-force product-based analysis with model checkers optimized for single product engineering, like SPIN (`spinroot.com`), NuSMV (`nusmv.fbk.eu`), mCRL2 (`www.mcrl2.org`), and—to a much lesser degree—VMC, and innovative family-based analysis with model checkers that were developed specifically for SPLE, like SNIP [12] and the NuSMV extension of [13]. To a certain extent, VMC offers the full spectrum of analyses, but—contrary to the special-purpose FTS model-checking algorithms of SNIP—when a formula is verified over an entire product family, then a negative result does not actually list the specific products in which the property fails to hold. However, the full list of violating products can be obtained by means of a product-based analysis.

We are not aware of any other model-checking tool for MTSs that also supports value passing. MTSA [19] 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.

7. AN EXAMPLE IN VMC

Consider the behavior of a family of bike-sharing systems specified in the value-passing modal process algebra accepted by VMC, in which processes can pass and receive integer parameter values (and store them in a variable preceded by a '`?`'), actions can be optional (i.e. typed `may`), nondeterministic choice can be guarded by a comparison of values, parallel composition is parametrized by the actions `/.../` to be synchronized,[3] and a system definition must be complemented with a top term of the form `net SYSTEM = P`, where `P` is the initial process (or composition of processes):

```
Station(I,N,J,M) =
  request(I).
    ( [N = 0] nobike(I).Station(I,N,J,M) +
      [N > 0] bike(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)

Users(I,J) =
  request(I).
    ( bike(I).return(J).Users(I,J) +
      nobike(I).Users(I,J) )
```

[2]Their commonalities and differences are discussed in [4].

[3]In MTSs, $a(may)$ synchronized with a results in $a(may)$ [1].

```
net STATIONS =
  Station(s1,1,s2,1) /redistribute/ Station(s2,0,s1,0)
net USERS = Users(s1,s2) // Users(s2,s1)
net BSS = STATIONS /request,bike,nobike,return/ USERS
```

It models a possibly infinite number of users that can take a bike from docking station I, ride it for a while (not specifically modeled), and deliver it to docking station J. Initially, docking station I has N bikes, which it gives (when available) to a requesting user or accepts from a returning user. If docking station I receives more than M bikes, the exceeding $N - M$ bikes are distributed to docking station J. Docking station I must accept all bikes distributed by other docking stations or returned by a user (possibly for redistribution).

The specific case of a bike-sharing family of two docking stations with two user groups and one bike is depicted in Fig. 5. Given that redistribution is optional, this family has two valid products: The behavior of the one without redistribution is modeled by the LTS depicted in Fig. 5, while that of the one with distribution is of course the LTS that is obtained from the MTS in Fig. 5 by simply turning the dotted redistribution edge into a solid one.

Now suppose that we want to verify the property that for both products, i.e. bike-sharing systems, it is always the case that eventually docking station 1 must give the user a bike. A candidate v-ACTL$^{\Box}$ formula to express this property is:[4]

$$AG\,EF^{\Box}\,\{bike(s1)\}\;true$$

This formula is true for all products (LTSs) of the family, and it makes sense to verify it directly over the family since it is a formula expressed in v-ACTL$^{\Box}$. In fact, the above formula holds over the MTS and therefore for all its products. Clearly, in this case of only two products, a direct evaluation over the products themselves would not constitute a problem either. More on this example can be found in [8].

8. CONCLUSIONS AND FUTURE WORK

VMC is an SPL modeling and analysis framework that accepts an MTS model of the behavior of a product family specified in a value-passing process algebra, enriched with an optional set of variability constraints. It offers product- and family-based analyses over the behavior of product families, implemented by efficient on-the-fly model checking of logical properties expressed in v-ACTL. Recent additions include a calculus deontic synchronization operators, value-passing communication and n-ary variability constraints, and a revised version of v-ACTL with user notifications whenever family-based analysis results are preserved by all products.

There are several directions into which the analysis techniques should to be improved before VMC can be experimented with on a case study from industry, which is the most important goal for the future. To this aim, a richer process algebra is required, with more advanced data types (not just integers, but tuples, sets, lists, etc.), which calls for a more complex underlying computational model.

More advanced MTS model-checking techniques might explicitly take into consideration the variability constraints that are used to define product validity and/or more operators beyond those currently supported by VMC.

VMC actually allows more complex variability constraints than the ones in §3, e.g. compositions like a REQ (b ALT c). It might be interesting to consider also explicit behavioral variability constraints like $X\{a\}$ ALT $X\{b\}$, which is based on the next operator $X\{a\}$ that intuitively says that the next state of a path can be reached by an action a. If both actions a and b were possible in a state of the MTS, then a valid product would only include one of the two. It would be interesting to see how this could work, in particular as it might interfere with the consistency assumption. Finally, it might be possible to allow parameter values in constraints.

[4]In VMC, $\neg$ | $\vee$ | $\wedge$ | $F^{\Box}$ are written as `not` | `or` | `and` | `F#`.

9. ACKNOWLEDGEMENTS

Maurice ter Beek is supported by the Italian Ministry of Instruction, University and Research project CINA (PRIN 2010LHT4KM) and by the European projects QUANTICOL (FP7 600708) and LearnPAd (FP7 619583).

The authors would like to thank Alessandro Fantechi and Stefania Gnesi for having involved us in their endeavor to develop a SPL modeling and verification framework based on MTSs, and for numerous inspiring discussions on VMC.

10. REFERENCES

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

[2] 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)*, volume 6396 of *LNCS*, pages 43–58. Springer, 2010.

[3] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A Model-Checking Tool for Families of Services. In *Proceedings 13th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'11)*, volume 6722 of *LNCS*, pages 44–58. Springer, 2011.

[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 Lines Conference (SPLC'11)*, pages 130–139. IEEE, 2011.

[5] M. H. ter Beek and E. P. de Vink. Using mCRL2 for the analysis of software product lines. In *Proceedings 2nd FME Workshop on Formal Methods in Software Engineering (FormaliSE'14)*, pages 31–37. IEEE, 2014.

[6] 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):119–135, 2011.

[7] 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)*, volume 2, pages 242–245. ACM, 2012.

[8] M. H. ter Beek, S. Gnesi, and F. Mazzanti. Model Checking Value-Passing Modal Specifications. In *Perspectives of Systems Informatics*, LNCS. Springer, 2014. To appear.

[9] 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, pages 10–17. ACM, 2013.

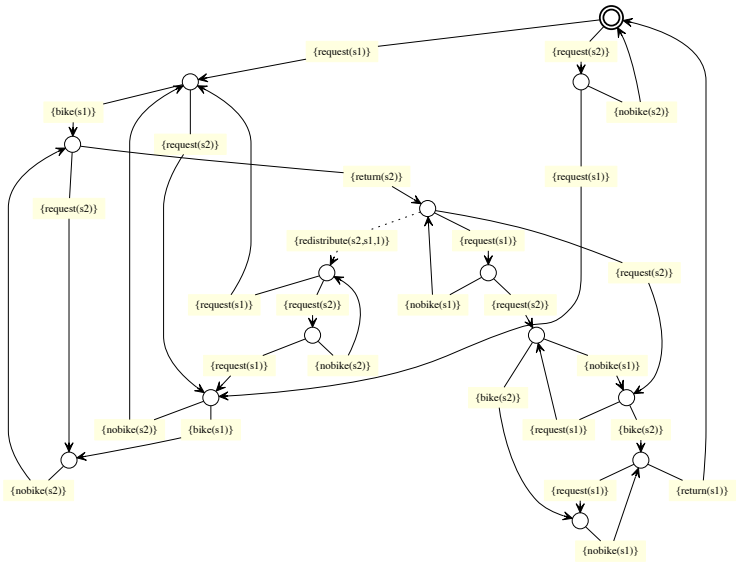

Figure 5: MTS of a bike-sharing family of two docking stations with two users and one bike

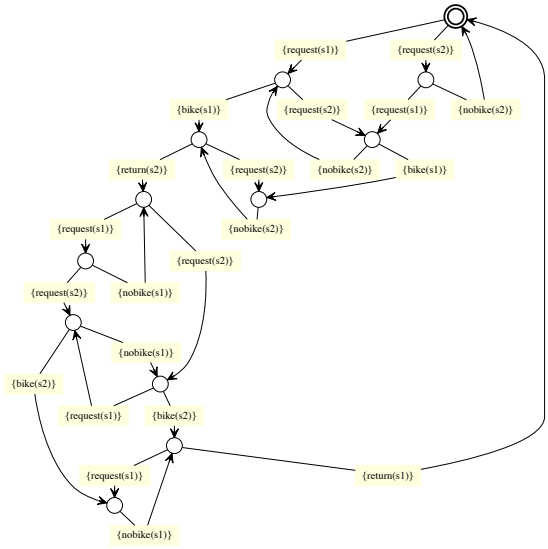

Figure 6: LTS of a product of the bike-sharing family of docking systems without redistribution

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

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

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

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

Figure 7: Fragment of SOS semantics of the value-passing modal process algebra, with $a, \ell \in \mathcal{A}$

[10] 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)*, volume 7436 of *LNCS*, pages 450–454. Springer, 2012.

[11] 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):244–263, 1986.

[12] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.

[13] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.*, 80:416–439, 2014.

[14] 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):1069–1089, 2013.

[15] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings 32nd International Conference on Software Engineering (ICSE)*, volume 1, pages 335–344. ACM, 2010.

[16] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: a product line of verifiers for software product lines. In *Proceedings 17th International Software Product Line Conference (SPLC'13)*, volume 2, pages 141–146. ACM, 2013.

[17] R. De Nicola and F. W. Vaandrager. Action versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.

[18] R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[19] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. MTSA: The Modal Transition System Analyser. In *Proceedings 23rd International Conference on Automated Software Engineering (ASE'08)*, pages 475–476. IEEE, 2008.

[20] A. Fantechi and S. Gnesi. A behavioural model for product families. In *Proceedings 6th Joint European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE'07)*, pages 521–524. ACM, 2007.

[21] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *Proceedings 12th International Conference on Software Product Line Engineering (SPLC'12)*, pages 193–202. IEEE, 2008.

[22] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A logical verification methodology for service-oriented computing. *ACM TOSEM*, 21(3):16, 2012.

[23] D. Fischbein, S. Uchitel, and V. A. Braberman. A foundation for behavioural conformance in software product line architectures. In *Proceedings Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06)*, pages 39–48. ACM, 2006.

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

[25] S. Gnesi and F. Mazzanti. An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems. In *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *LNCS*, pages 390–407. Springer, 2011.

[26] S. Gnesi and M. Petrocchi. Towards an executable algebra for product lines. In *Proceedings 16th International Software Product Line Conference (SPLC'12)*, volume 2, pages 66–73. ACM, 2012.

[27] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proceedings 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[28] K. G. Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theoret. Comput. Sci.*, 72(2–3):265–288, 1990.

[29] 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)*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[30] K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Proceedings 3rd Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE, 1988.

[31] 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)*, pages 269–280, 2009.

[32] M. Leucker and D. Thoma. A Formal Approach to Software Product Families. In *Proceedings 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12)*, volume 7609 of *LNCS*, pages 131–145. Springer, 2012.

[33] R. Meolic, T. Kapus, and Z. Brezocnik. ACTLW: An action-based computation tree logic with unless operator. *Inform. Sciences*, 178(6):1542–1557, 2008.

[34] J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane. Compositional Verification of Software Product Lines. In *Proceedings 10th International Conference on Integrated Formal Methods (IFM'13)*, volume 7940 of *LNCS*, pages 109–123. Springer, 2013.

[35] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surv.*, 47(1):6:1–6:45, 2014.

APPENDIX

A. VALUE-PASSING MODAL PROCESSES

The latest version of VMC accepts an MTS (modeling the behavior of a product family) specified in a *value-passing* modal process algebra in which the deontic parallel composition operator is parametrized by the actions to be synchronized [8] (i.e. contrasting recent approaches in [9, 26, 32]). A product family can be defined inductively by composition,

with the additional distinction between necessary actions of type a and admissible but not necessary ones of type $a(may)$.

Definition 1. *Consider action set $\mathcal{A}$ with $a \in \mathcal{A}$ and $L \subseteq \mathcal{A}$. Processes are built from terms and actions of this syntax:*

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

where $[P]$ denotes a closed system, i.e. a process that cannot evolve on input actions $a(?v)$, and $K(e)$ is a process identifier from the set of process definitions of the form $K(v) \stackrel{def}{=} T$,

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

*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.*

This value-passing modal process algebra is interpreted over MTSs, but for reasons of space we only provide the semantics of the necessary actions in Fig. 7 (cf. [8] for the full syntax).

The intuition of parallel composition is that both partners must fully and deterministically agree on the parameter values for the synchronization to occur. The rules in Fig. 7 refer to a case of two parameters. In general, e.g., $a(X, 2).nil$ and $a(3, Y).nil$ can synchronize and execute action $a(3, 2)$.

B. v-ACTL$^{\Box}$: VARIABILITY-AWARE ACTL

We present the latest, revised version of the *variability-aware* action-based branching-time temporal modal logic v-ACTL. It was originally developed in [2–4] in the style of (action-based) CTL [11, 17] and the Hennessy–Milner Logic (HML) with until [18, 28], starting from the ACTL logic accepted by VMC. Next to the operators of propositional logic, ACTL contains the 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 ('eventually') and, by duality, G ('globally') operators from action-based CTL as well as the (action-based) until and weak until operators U and W drawn from those firstly introduced in [17] and later elaborated in [33]. Finally, ACTL contains the least and greatest fixed-point operators μ and ν, which provide a semantics for recursion used for "finite looping" and "looping" (or "liveness" and "safety").

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

Definition 2. *Action formulae are built as follows over a set $\mathcal{A}$ of atomic actions $\{a, b, \ldots\}$:*

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

Action formulae are thus simply Boolean compositions of actions. As usual, *false* abbreviates $\neg\, true$, $\psi \vee \psi'$ abbreviates $\neg(\neg\psi \wedge \neg\psi')$ and $\psi \Longrightarrow \psi'$ abbreviates $\neg\psi \vee \psi'$.

For completeness we now give the full syntax of ACTL, as accepted by VMC. Details of (v-)ACTL can be found in [8].

Definition 3. *The syntax of ACTL is as follows:*

$$\begin{array}{rcl} \phi & ::= & true \mid \neg\phi \mid \phi \wedge \phi \mid [\psi]\,\phi \mid \langle\psi\rangle\,\phi \mid E\,\pi \mid A\,\pi \mid \\ & & \mu\, Y.\phi(Y) \mid \nu\, Y.\phi(Y) \\ \pi & ::= & X\,\{\psi\}\ \phi \mid [\phi\ \{\psi\}\,U\,\{\psi'\}\ \phi'] \mid [\phi\ \{\psi\}\,W\,\{\psi'\}\ \phi'] \mid \\ & & [\phi\ \{\psi\}\,U\ \phi'] \mid [\phi\ \{\psi\}\,W\ \phi'] \mid F\,\phi \mid F\,\{\psi\}\ \phi \mid G\,\phi \end{array}$$

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

In VMC, the least and greatest fixed-point operators μ and ν are written as `min` and `max`, respectively, and recall that $\neg$, $\vee$, $\wedge$, and $F^{\Box}$ are written as `not`, `or`, `and`, and `F#`, respectively.

Intuitively, the action-based until operators $[\phi\ \{\psi\}\,U\ \phi']$ ($[\phi\ \{\psi\}\,U\,\{\psi'\}\ \phi']$) say that ϕ' holds at some future state of the path (reached by a final action satisfying ψ'), while ϕ holds from the current state until that state is reached and all the actions executed meanwhile along the path satisfy ψ. The action-based weak until operators $[\phi\ \{\psi\}\,W\ \phi']$ and $[\phi\ \{\psi\}\,W\,\{\psi'\}\ \phi']$ (also called unless) hold on a path either if the corresponding strong until operator holds or if for all states of the path the formula ϕ holds and all actions executed on the path satisfy ψ.

To make this ACTL logic variability-aware, for the box, diamond and F operators we defined also a *deontic* interpretation that takes the modality (or 'deonticity') of the transitions (may or must) into account, resulting in v-ACTL. The intuitive interpretation of the different variants of these operators is as follows. $[\psi]\,\phi$: 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. $F\,\phi$: 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.

Now v-ACTL$^{\Box}$, finally, is a fragment of v-ACTL that enjoys some convenient properties, elaborated on below, and which moreover suffices for specifying interesting properties for product families in the presence of variability as well as for specifying the additional variability constraints.

Definition 4. *The syntax of v-ACTL$^{\Box}$ is as follows:*

$$\begin{array}{rcl} \phi & ::= & false \mid true \mid \phi \wedge \phi \mid \phi \vee \phi \mid [\psi]\,\phi \mid \langle\psi\rangle^{\Box}\,\phi \mid \\ & & EF^{\Box}\,\phi \mid EF^{\Box}\,\{\psi\}\,\phi \mid AF^{\Box}\,\phi \mid AF^{\Box}\,\{\psi\}\,\phi \mid AG\,\phi \mid \\ & & \neg\,\chi \\ \chi & ::= & false \mid true \mid \chi \wedge \chi \mid \chi \vee \chi \mid \langle\psi\rangle\,\chi \mid \\ & & EF\,\chi \mid EF\{\psi\}\,\chi \mid \neg\,\phi \end{array}$$

Note that v-ACTL$^{\Box}$ consists of two parts. The first fragment is such that any formula expressed in it that is true for the MTS, is also true for all products. The second fragment (which in v-ACTL$^{\Box}$ appears negated) is such that any formula expressed in it that is false for the MTS, is also false for all products. The latest version of VMC notifies the user whenever preservation of a verification result is applicable.

From an implementation point of view, VMC handles v-ACTL$^{\Box}$ formula by means of the following translations into classical ACTL operators.

$$\begin{array}{rcl} \langle\psi\rangle^{\Box}\,\phi & \stackrel{\text{def}}{=} & \langle\psi \wedge \neg\, may\rangle\,\phi \\ EF^{\Box}\,\phi & \stackrel{\text{def}}{=} & E\,[true\ \{\neg\, may\}\ U\ \phi] \\ AF^{\Box}\,\phi & \stackrel{\text{def}}{=} & A\,[true\ \{\neg\, may\}\ U\ \phi] \\ EF^{\Box}\,\{\psi\}\,\phi & \stackrel{\text{def}}{=} & E\,[true\ \{\neg\, may\}\ U\ \{\psi \wedge \neg\, may\}\,\phi] \\ AF^{\Box}\,\{\psi\}\,\phi & \stackrel{\text{def}}{=} & A\,[true\ \{\neg\, may\}\ U\ \{\psi \wedge \neg\, may\}\,\phi] \end{array}$$