Model Sketching by Abstraction Refinement for Lifted Model Checking

Aleksandar S. Dimovski
Faculty of Informatics, Mother Teresa University
Skopje, North Macedonia
aleksandar.dimovski@unt.edu.mk

ABSTRACT

In this work, we show how the use of verification and analysis techniques for model families (software product lines) with numerical features provides an interesting technique to synthesize complete models from sketches (i.e., partial models with holes). In particular, we present an approach for synthesizing PROMELA model sketches using variability-specific abstraction refinement for lifted (family-based) model checking.

CCS CONCEPTS

• Theory of computation → Logic; Models of computation; • Software and its engineering → Software notations and tools.

KEYWORDS

Model sketching, Product-line (lifted) model checking

1 INTRODUCTION

This paper presents a novel synthesis framework for reactive models that adhere to a given set of properties. The input is a sketch [18], i.e., a partial model with holes, where each hole is a placeholder that can be replaced with one of finitely many options; and a set of properties that the model needs to fulfill. Model sketches are represented in the PROMELA modelling language [13] and properties are expressed in LTL [2]. The synthesizer aims to generate as output a sketch realization, i.e., a complete model instantiation, which satisfies the given properties by suitably filling the holes.

In this work, we frame the model sketching problem as a verification/analysis problem for model families (a.k.a. Software Product Lines – SPLs) [3], and then formulate an abstraction refinement algorithm that operates on model families to efficiently solve it. SPL methods and architectures allow building a family of similar models, known as variants (family members), from a common code base. A custom variant is specified in terms of suitable features selected for that particular variant at compile-time.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:
Aleksandar S. Dimovski. 2022. Model Sketching by Abstraction Refinement for Lifted Model Checking. In The 37th ACM/SIGAPP Symposium on Applied Computing (SAC ’22), April 25–29, 2022, Virtual Event, ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3477314.3507170

All possible model sketch realizations constitute a model family, where each hole is represented by a numerical feature with the same domain. In contrast to Boolean features that have only two values, numerical features can have a range of numbers as explicit values. Hence, the model sketching problem reduces to selecting correct variants (family members) from the resulting model family [8]. The automated analysis of such families for finding a correct variant is challenging since in addition to the state-space explosion affecting each family member, the family size (i.e., the number of variants) typically grows exponentially in the number of features. A naive brute force enumerative solution is to check each individual variant of the model family by applying an off-the-shelf model checker. This is shown to be very inefficient for large families [3, 16].

This paper applies an abstraction refinement procedure over the compact, all-in-one, representation of model families, called featured transition system (FTS) [3, 6, 7], to solve the model sketching problem. More specifically, we first devise variability abstractions tailored for model families that contain numerical features. Variability abstractions represent a configuration-space reduction technique that compresses the entire model family (with many configurations and variants) into an abstract model (with a single abstract configuration and variant), so that the result of model checking a set of LTL properties in the abstract model is preserved in all variants of the model family. The procedure is first applied on an abstract model that represents the entire model family, and then is repeated on refined abstract models that represent suitable sub-families of the original model family. Hence, the abstraction refinement approach [4, 6, 7, 10, 11] starts from considering all possible variants, and successively splits the entire family into indecisive and incorrect sub-families with respect to the given set of properties. The approach is sound and complete: either a correct complete model (variant) does exist and it is computed, or no such model exists and the procedure reports this. Because of its special structure and possibilities for sharing of equivalent execution behaviours and model checking results for many variants, this algorithm is often able to converge to a solution very fast after a handful of iterations even for sketches with large search spaces.

We have implemented our prototype model synthesizer, called PROMELASKETCHER. It uses variability-specific abstraction refinement for lifted model checking of model families with numerical features, and calls the SPIN model checker [13] to verify the generated abstract models. The abstraction and refinement are done in an efficient manner as source-to-source transformations of PROMELA code, which makes our procedure easy to implement/maintain as a simple meta-algorithm script. We illustrate this approach for automatic completion of various PROMELA model sketches. We also compare its performance with the brute-force approach.
2 MODEL FAMILIES

Featured transition system. Let \( F = \{A_1, \ldots, A_k\} \) be a finite and totally ordered set of numerical features available in a model family. Let \( \text{dom}(A) \subseteq \mathbb{Z} \) denote the set of possible values that can be assigned to feature \( A \). A valid combination of feature's values represents a configuration \( k \), which specifies one variant of a model family. It is given as a valuation function \( k : F \rightarrow \mathbb{Z} \), which assigns a value from \( \text{dom}(A) \) to each feature \( A \). We assume that only a subset \( \Xi \) of all possible configurations are valid. Each configuration \( k \in \Xi \) can be given by a formula: \((A_1 = k(A_1)) \land \cdots \land (A_k = k(A_k))\).

A transition system [2] is a tuple \( T = (S, I, \text{trans}, AP, L) \), which is used to describe behaviours of single systems. We write \( s_1 \rightarrow s_2 \) whenever \((s_1, s_2) \in \text{trans}\). A path of a TS \( T \) is an infinite sequence \( \rho = s_0 s_1 s_2 \ldots \) with \( s_0 \in I \) s.t. \( s_i \rightarrow s_{i+1} \) for all \( i \geq 0 \). The semantics of a TS \( T \), denoted \( [T]_{TS} \), is the set of its paths.

A featured transition system (FTS) represents a compact model, which describes the behaviour of a whole family of systems in a single monolithic description. The set of feature expressions, \( \text{FeatExp}(F) \), are propositional logic formulas over constraints of \( F: \psi ::= \text{true} | A \rightarrow n \land \neg n \land \psi \lor \psi, \text{where } A \in F, n \in \mathbb{Z}, \rightleftharpoons \in \{=, <, \leq \} \). We write \( [\psi] \) for the set of configurations that satisfy \( \psi \), i.e., \( k \in [\psi] \) iff \( k \models \psi \). A featured transition system (FTS) is \( FTS = (S, \text{trans}, AP, L, F, \Xi, \delta) \), where \((S, \text{trans}, AP, L, F, \Xi, \delta) \) form a TS; \( F \) is a set of available features; \( \Xi \) is a set of valid configurations; and \( \delta : trans \rightarrow \text{FeatExp}(F) \) is a total function decorating transitions with presence conditions (feature expressions). The projection of an FTS \( FTS \) to a configuration \( k \in \Xi \), denoted as \( \pi_k(F) \), is the TS \((S, \text{trans}', AP, L) \), where \( \text{trans}' = \{ t \in \text{trans} \mid k \models \delta(t) \} \). We lift the definition of projection to sets of configurations \( \Xi' \subseteq \Xi \), denoted as \( \pi_{\Xi'}(F) \), by keeping transitions admitted by at least one of configurations in \( \Xi' \). That is, \( \pi_{\Xi'}(F) \), is the FTS \((S, \text{trans}', AP, L, \Xi', \delta') \), where \( \text{trans}' = \{ t \in \text{trans} \mid \exists k \in \Xi', k \models \delta'(t) \} \) and \( \delta' \) is the restriction of \( \delta \) to \( \text{trans}' \). The semantics of an FTS \( FTS \), denoted as \( [FTS]_{FTS} \), is the union of paths of the projections on all valid variants \( k \in \Xi \), i.e., \( [FTS]_{FTS} = \cup_{k \in \Xi} \pi_k(F) \).

Abstraction. We start working with Galois connections between Boolean complete lattices of feature expressions, and then induce a notion of abstraction of FTSs. The Boolean complete lattice of feature expressions is: \((\text{FeatExp}(F), \sqsubseteq, \lor, \land, \text{true}, \text{false}, \neg)\).

The join abstraction, \( \text{join}^F : \text{FeatExp}(F) \rightarrow \text{FeatExp}(\emptyset) \), replaces each feature expression \( \psi \) in an FTS with true if there exists at least one configuration from \( \Xi \) that satisfies \( \psi \). The abstract sets of features and configurations are: \( \text{join}^F(\emptyset) = \emptyset \) and \( \text{join}^F(\Xi) = \{\text{true}\} \). The abstraction and concretization functions between \( \text{FeatExp}(F) \) and \( \text{FeatExp}(\emptyset) \), which form a Galois connection [7], are:

\[
\text{join}^F(\psi) = \begin{cases} \text{true} & \text{if } \exists k \in \Xi, k \models \psi, \\ \text{false} & \text{otherwise} \end{cases}
\]

\[
\text{join}^F(\text{true}) = \text{true}, \text{join}^F(\text{false}) = \emptyset
\]

Given the FTS \( FTS = (S, I, \text{trans}, AP, L, F, \Xi, \delta) \), we will define a TS \( \text{join}^F(FTS) = (S, I, \text{trans}', AP, L) \) to be its abstraction, where \( \text{trans}' = \{ t \in \text{trans} \mid \text{join}^F(\delta(t)) = \text{true} \} \). Note that transitions in the abstract TS \( \text{join}^F(FTS) \) describe the behaviour that is possible in some variants of the concrete FTS \( FTS \), but not need be realized in the other variants. The information about which transitions are associated with which variants is lost, thus causing a precision loss in the abstract model.

Algorithm 1: ARP \((F, \Xi, \phi)\)

Input: An FTS \( F \), a configuration set \( \Xi \), and an LTL formula \( \phi \).

Output: Correct variants \( k \in \Xi \), s.t. \( \pi_k(F) \models \phi \).

Global: end = false

1. \( c = (\text{join}^F(F) \models \phi) \);
2. if (\(c = \text{null}\)) then \( \text{end} = \text{true}; \text{return } \Xi \);
3. \( \psi = \text{FeatExp}(c) \);
4. if \((\text{sat}(\psi \land (\lor_{k \in \Xi} k))) \) then \[ (\psi_1, \ldots, \psi_n) = \text{Split}([[\neg\phi] \land \Xi]) \];
5. if (\(\text{end}\)) then return \( \emptyset \);
6. \( \text{ARP}(\pi_1(\psi_1)), [\psi_1], \phi) \); \( \ldots ; \text{ARP}(\pi_n(\psi_n)), [\psi_n], \phi \)
else \[ 9. \psi' = \text{CraigInterpolation}(\psi, \Xi) ; \]
10. if (\(\text{end}\)) then return \( \emptyset \);
11. \( \text{ARP}(\pi_1(\psi')), [\psi'], \phi) \); \( \text{ARP}(\pi_n(\neg \psi'), [\neg \psi'], \phi) \)

This way, \( [[\text{join}^F(F)]_{TS}] \supseteq \cup_{k \in \Xi} \pi_k(F) \).

The problem \( F \models \phi \) can be reduced to a number of smaller problems by partitioning set \( \Xi \). Let the subsets \( \Xi_1, \Xi_2, \ldots, \Xi_n \) form a partition of \( \Xi \). Then, \( F \models \phi \) iff \( \pi_{\Xi_i}(F) \models \phi \) for all \( i = 1, \ldots, n \).

Abstraction Refinement Framework. The abstraction refinement procedure ARP for checking \( F \models \phi \) is illustrated by Algorithm 1. We first construct an initial abstract model \( \text{join}^F(F) \), and check \( \text{join}^F(F) \models \phi \) (Line 1). If the abstract model satisfies the given property (i.e., the counterexample \( c \) is \( \text{null} \)), then all variants from \( \Xi \) satisfy it and we stop. In this case, the global variable end is also set to true making all other recursive calls to ARP to end (Lines 2, 6, 10). Otherwise, a non-null counterexample \( c \) is found. Let \( \psi \) be the feature expression computed by conjointing feature expressions labelling all transitions that belong to path \( c \) when \( c \) is simulated in \( FTS \) (Line 3). There are two cases to consider.

First, if \( \psi \land (\lor_{k \in \Xi} k) \) is satisfiable (i.e., \( \Xi \land [\psi] \neq \emptyset \)), then the found counterexample \( c \) is genuine for variants in \( \Xi \land [\psi] \). For the other variants from \( \Xi \land [\neg \psi] \), the found counterexample cannot be executed (Lines 5, 6, 7). We call \text{Split} to split the space \( \Xi \land [\neg \psi] \) in sub-families \( [\psi_1], \ldots, [\psi_n] \), such that all atomic constraints in \( [\psi_i] \) are of the form: \( A \rightarrow n \), where \( A \in F \) and \( n \in \text{dom}(A) \). In particular, the \text{Split} function takes as input a set of configurations and returns a list of sets of configurations. For example, assume that we have two numerical features \( \text{Min} \leq A \leq \text{Max} \) and \( \text{Min} \leq B \leq \text{Max} \). If \( \psi \land (A = 3) \), then \( \text{Split}([[\neg \psi]]) \) is \((\text{Min} \leq A \leq 2) \land (\text{Min} \leq B \leq \text{Max}) \) and \((4 \leq A \leq \text{Max}) \land (\text{Min} \leq B \leq \text{Max}) \). Finally, we call ARP to verify the sub-families: \( \pi_{\psi_1}(FTS), \ldots, \pi_{\psi_n}(FTS) \). Note that if \( \Xi \land [\neg \psi] = \emptyset \), \text{Split} updates variable end to true.

Second, if \( \psi \land (\lor_{k \in \Xi} k) \) is unsatisfiable (i.e., \( \Xi \land [\psi] = \emptyset \)), then the found counterexample \( c \) is spurious for all variants in \( \Xi \) (due to
incompatible feature expressions) (Lines 9, 10, 11). A feature expression
\( \psi' \) used for constructing refined sub-families is determined
by means of Craig interpolation \([15]\) from \( \psi \) and \( \mathbb{X} \). First, we find
the minimal unsatisfiable core \( \psi' = X \land Y = \text{false} \) of \( \psi' \land \bigvee_{k \in K} k \).
Next, the interpolant \( \psi' \) is computed, such that \( \psi' \) summarizes and
translates why \( X \) is inconsistent with \( Y \) in their shared language.
Finally, we call the ARP to check \( \pi_1[\psi'/\psi](F) \models \phi \) and \( \pi_1[\psi'/\psi](F) \models \phi \).
By construction, it is guaranteed that the spurious counterexample \( c \) does not occur in both \( \pi_1[\psi'/\psi](F) \) and \( \pi_1[\psi'/\psi](F) \).

Note that abstract models we obtain are ordinary TSs where all feature expressions are replaced with true. Therefore, the verifica-
tion step \( \alpha_{\text{join}}(F) \models \phi ? \) (Line 1) can be performed using a single-system model checker such as SPIN. Also note that we call
ARP until we find a correct variant (variable \( m \) is set to true) or the updated set of configurations \( \mathbb{X} \) becomes empty. Therefore,
\( \alpha_{\text{join}}(F, \mathbb{X}, \phi) \) terminates and is correct.

3 SYNTACTIC TRANSFORMATIONS

We now present the high-level modelling language Promela for
writing sketches and model families. Then, we describe several
transformations of Promela sketches and model families.

Syntax of Promela. Promela \([13]\) is a non-deterministic mod-
eling language designed for describing systems composed of con-
current processes that communicate asynchronously. The basic
statements of processes are given by:

\[
\begin{align*}
stm & ::= \text{skip} | \text{break} | x := \text{expr} | c x = \text{expr} | \text{stm} 1 ; \text{stm} 2 \\
& \quad | \text{if} : g 1 \rightarrow \text{stm} 1 \ldots \text{gn} \rightarrow \text{stm} n \text{ f 1 } | \text{do} : g 1 \rightarrow \text{stm} 1 \ldots \text{gn} \rightarrow \text{stm} n \text{ od}
\end{align*}
\]

where \( x \) is a variable, \( \text{expr} \) is an expression, \( c \) is a channel, and \( g i \)
are conditions over variables and contents of channels.

Sketches. To encode sketches, a single sketching construct of
type expression is included: a basic integer hole denoted by \( ? \). Each hole occurrence is assumed to be uniquely labelled as \( ? i \), and it has a bounded integer domain \([n, n']\].

Model Families. To encode multiple variants, a new compile-time
guarded-by-features statement is included:

\[
\begin{align*}
stm & ::= \ldots \# \text{if} : \psi 1 \rightarrow \text{stm} 1 \ldots \psi n \rightarrow \text{stm} n \text{ endif}
\end{align*}
\]

where \( \psi 1, \ldots , \psi n \) are feature expressions defined over \( \mathbb{F} \). The \#if statement
contains feature expressions \( \psi i \in \text{FeatExp}(\mathbb{F}) \) as presence conditions (guards). If presence condition \( \psi i \) is satisfied by
a configuration \( k \in \mathbb{X} \), the statement \( \text{stm} i \) will be included in the
variant corresponding to \( k \). Hence, \#if plays the same role as
\#ifdef directives in C preprocessor CPE \([9, 14]\). The semantics of
Promela models and Promela model families are given in \([7, 13]\).

Syntactic Transformations. Our aim is to transform an input
sketch \( \tilde{P} \) with a set of \( m \) holes \( ? i [n, n'] , \ldots , ? m [n, n'] \), into an
output model family \( P \) with a set of numerical features \( A 1, \ldots , A m \)
with domains \([n, n'] , \ldots , [n, n'] \). The set of configurations \( \mathbb{X} \) includes all possible combinations of feature’s values. The rewrite
rule for eliminating holes ?? from a model sketch is of the form:

\[
\begin{align*}
\text{stm}[? i [n, n']] \sim \# \text{if} : (A = n) \rightarrow \text{stm}[n] \ldots : (A = n') \rightarrow \text{stm}[n'] \text{ endif}
\end{align*}
\]

where \( \text{stm}[\cdot] \) is a (non-compound) basic statement with a single
expression – in it, \( ? i [n, n'] \) is an occurrence of a hole with domain
\([n, n'] \) , and \( A \) is a fresh numerical feature with domain \([n, n'] \).
The meaning of the rule (R-1) is that if the current sketch being
transformed matches the abstract syntax tree node of the shape
\( \text{stm}[? i [n, n']] \) then replace \( \text{stm}[? i [n, n']] \) according to the rule (R-1).

We write \( \text{Rewrite}(\tilde{P}) \) to be the final model family obtained by re-
peatedly applying the rule (R-1) on sketch \( \tilde{P} \) and on its transformed
versions until we reach a point where it can not be applied.

We now present the syntactic transformations of model families
\( P = \text{Rewrite}(\tilde{P}) \) obtained from PROMELA sketches \( \tilde{P} \). We consider
two transformations: projection \( \pi[\psi/\psi](P) \) and variability abstrac-
tion \( \alpha_{\text{join}}(P) \). Let \( P \) represent a model family.

The projection \( \pi[\psi/\psi](P) \) is obtained by defining a translation
recursively over the structure of \( \psi \). Let \( \psi \) be of the form \((A < m)\).
The rewrite rule is of the form:

\[
\begin{align*}
\# \text{if} : (A = n) \rightarrow \text{stm}[n] \ldots : (A = m) \rightarrow \text{stm}[m] \ldots : (A = n') \rightarrow \text{stm}[n'] \# \text{endif}
\end{align*}
\]

where \( n < m \leq n' \). That is, all guards that do not satisfy \((A < m)\)
are replaced with false. Let \( \psi \) be a feature expression of the form
\( \neg \psi' \). We first transform \( P \) by applying the projection \( \psi' \), then in all
\#if-s obtained from the projection \( \psi' \) we change the guards: those
guards of the form \((A = m')\) become false, and false guards are
returned to the form \((A = m')\) by looking at a special memo list
where we keep record of them. Let \( \psi \) be of the form \( \psi 1 \land \psi 2 \). Then,
we apply projections \( \psi 1 \) and \( \psi 2 \) one after the other.

The abstract model \( \alpha_{\text{join}}(P) \) is obtained by appropriately resolv-
ing all \#ifdef-s. The rewrite rule is:

\[
\begin{align*}
\# \text{if} : (\psi 1) \rightarrow \text{stm} 1 \ldots : (\psi n) \rightarrow \text{stm} n \# \text{endif} \sim \\
\# \text{if} : \alpha_{\text{join}}(\psi 1) \rightarrow \text{stm} 1 \ldots : \alpha_{\text{join}}(\psi n) \rightarrow \text{stm} n \text{ f i}
\end{align*}
\]

where all guards in the new \#if are set to true or false depending
whether there is some valid configurations that satisfies that guard.

The correctness of these transformations are formally proved by
structural induction on \( \tilde{P} \) and \( P \) (see [5, App A]).

4 SYNTHESIS ALGORITHM

We can now encode the sketch synthesis problem as a lifted model
checking problem. In particular, we delegate the effort of conducting
an effective search of all possible sketch realizations to an efficient
abstraction refinement for lifted model checking. Once the lifted
model checking of the corresponding model family is performed,
we can see for which variants the given property is correct. Those
variants represent the correct sketch realizations.

The synthesis algorithm SYNTHESIZE(\( \tilde{P}, \phi \)) for solving a sketch
\( \tilde{P} \) is the following. The sketch \( \tilde{P} \) is first encoded as a model family
\( P = \text{Rewrite}(\tilde{P}) \). Then, we call function ARP(\( P, \mathbb{X}, \phi \)), which takes
as input the model family \( P \), its configuration set \( \mathbb{X} \), and the property
to verify \( \phi \), and returns as solution a set of variants \( K' \subseteq \mathbb{X} \) that
satisfy \( \phi \) obtained after performing the ARP. The correctness and
termination of SYNTHESIZE(\( \tilde{P}, \phi \)) are shown in [5, App A].
5 EVALUATION

Implementation. We have developed a prototype model synthesizer, called PromelaSketcher, for resolving Promela sketches. It uses the ANTLR parser generator [17] for processing Promela code, while projections and abstractions of #if-enriched Promela code are implemented using source-to-source transformations. It calls the SPIN [13] to verify the generated Promela models. If a counterexample trace is returned, the tool inspects the trace by using SPIN’s simulation mode, and generates refined abstractions. Our tool is written in JAVA and consists of around 2K LOC.

Experiment setup and Benchmarks. All experiments are executed on a 64-bit Intel® Core™ i5 CPU, Lubuntu VM, with 8 GB memory. The implementation, benchmarks, and all obtained results are available from: https://github.com/aleksdimovski/Promela_sketcher. We compare our approach with the Brute force enumeration approach that generates all possible sketch realizations and verifies them using SPIN one by one. For each experiment, we report: Time which is the total time to resolve a sketch in seconds; and Calls which is the number of times SPIN is called. We show performances for three different sizes of holes: 3-, 4-, and 8-bits. We only measure the model checking SPIN times to generate a process analyser (pan) and to execute it. We do not count the time for compiling pan, as it is due to a design decision in SPIN rather than its verification algorithm. The evaluation is performed on several suitably adjusted Promela sketches collected from the Sketch project [18], SyGuSComp [1], and SPIN [13] (see benchmarks in [5, App B]).

Performance Results. Table 1 shows the results of synthesizing our benchmarks. We can see from Table 1 that PromelaSketcher significantly outperforms Brute force. On our benchmarks, it translates to speed ups that range from 1.2× to 3.5× for 4-bits holes, and from 11.5× to 51.4× for 8-bits holes. This is due to the fact that the number of calls to SPIN and the number of partitionings of $\mathcal{K}$ that share the same counterexamples or correct traces in PromelaSketcher are much less than the configuration space $\mathcal{K}$ that is inspected one by one using SPIN in the Brute force.

6 CONCLUSION

In this paper, we employ techniques from product-line lifted model checking for automatically resolving of model sketches. By means of an implementation and a number of experiments, we confirm that our technique is effective and works well on a variety of Promela benchmarks and LTL properties.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Junwal, Milo M. K. Martin, Mukund Raghunathan, Sanjit A. Seshia, Rashish Singh, Armando Solar-Lezama, Emin Torlak, and Abhishek Udga. 2013. Syntax-guided synthesis. In Formal Methods in Computer-Aided Design, FMCAD 2013 IEEE 1–8. http://ieeexplore.ieee.org/document/6679398/

[2] Christel Baier and Joost-P. Katoen. 2008. Principles of model checking. MIT Press.

[3] Andreas Cassidy, Maxime Corde, Pierre-Y. Schobbens, Patrick Heymans, Axel Legay, and Jean-F. Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. IEEE Trans. Software Eng. 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[4] Aleksandar S. Dimovski. 2020. CTF* family-based model checking using variability abstractions and modal transition systems. Int. J. Softw. Tools Technol. Transf. 22, 1 (2020), 35–55. https://doi.org/10.1007/s10009-019-00528-0

[5] Aleksandar S. Dimovski. 2021. Model Sketching by Abstraction Refinement for Lifted Model Checking (Extended Version). CoRR abs/2112.11546 (2021). arXiv:2112.11546. https://arxiv.org/abs/2112.11546

[6] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. 2015. Family-Based Model Checking Without a Family-Based Model Checker. In 22nd International Symposium, SPIN 2015, Proceedings (LNCS, Vol. 9232). Springer, 282–299. https://doi.org/10.1007/978-3-319-23404-5_18

[7] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. 2017. Efficient family-based model checking via variability abstractions. STTT 19, 5 (2017), 585–603. https://doi.org/10.1007/s10009-016-0425-2

[8] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2021. Program Sketching using Lifted Analysis for Numerical Program Families. In NASA Formal Methods - 13th Int. Symposium, NFM 2021, Proceedings (LNCS, Vol. 12673). Springer, 95–112. https://doi.org/10.1007/978-3-030-76384-8_7

[9] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2022. Several lifted abstract domains for static analysis of numerical program families. Sci. Comput. Program. 213 (2022), 102725. https://doi.org/10.1016/j.scico.2021.102725

[10] Aleksandar S. Dimovski, Axel Legay, and Andrzej Wasowski. 2019. Variety Abstraction and Refinement for Game-Based Lifted Model Checking of Full CTL. In Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Proceedings (LNCS, Vol. 11424). Springer, 192–209. https://doi.org/10.1007/978-3-030-16722-6_11

[11] Aleksandar S. Dimovski, Axel Legay, and Andrzej Wasowski. 2020. Generalized abstraction-refinement for game-based CTL lifted model checking. Theor. Comput. Sci. 837 (2020), 181–206. https://doi.org/10.1016/j.tcs.2020.06.014

[12] Aleksandar S. Dimovski and Andrzej Wasowski. 2017. Variety-specific Abstraction Refinement for Family-Based Model Checking. In 20th Int. Conference, FASE 2017, Proceedings (LNCS, Vol. 10205). 406–423. https://doi.org/10.1007/978-3-662-54494-5_24

[13] Gerard J. Holzmann. 2004. The SPIN Model Checker - primer and reference manual. Addison-Wesley.

[14] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In Proceedings of the 26th Annual ACM SIGPLAN Conference on OOPSLA 2011. 805–824. https://doi.org/10.1145/2048066.2048128

[15] Kenneth L. McMillan. 2005. Applications of Craig Interpolants in Model Checking. In 11th International Conference, TACAS 2005, Proceedings (LNCS, Vol. 3440). Springer, 1–12. https://doi.org/10.1007/978-3-540-31890-1_1

[16] Jan Midtgård, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2015. Systematic derivation of correct variability-aware program analyses. Sci. Comput. Program. 105 (2015), 145–170. https://doi.org/10.1016/j.scico.2015.04.005

[17] Terence Parr. 2013. The Definitive ANTLR 4 Reference. Theor. Comput. Sci. 387 (2012), 2–4. https://doi.org/10.1016/j.tcs.2012.08.028

[18] Armando Solar-Lezama. 2013. Program sketching. STTT 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7