Synthesizing Promela Model Sketches Using Abstract Lifted Model Checking

Aleksandar S. Dimovski[1*]

[1*]Faculty of Informatics, Mother Teresa University, Mirche Acev no. 4, Skopje, 1000, North Macedonia .

Corresponding author(s). E-mail(s): aleksandar.dimovski@unt.edu.mk;

Abstract

We present a novel approach to synthesize complete models from Promela model sketches by using of lifted (family-based) verification and analysis techniques for model families (a.k.a software product lines - SPLs). The input is a Promela model sketch, which represents a partial model with missing numerical holes. The goal is to automatically synthesize values for the holes, such that the resulting complete model satisfies a given Linear Temporal Logic (LTL) specification. First, we encode a model sketch as a model family, such that all possible sketch realizations correspond to possible variants in the model family. Then, we preform a *lifted (family-based) model checking* of the resulting model family using variability-specific abstraction refinement, so that only those variants (family members) that satisfy the given LTL properties represent "correct" realizations of the given model sketch. We have implemented a prototype model synthesizer for resolving Promela sketches. It calls the SPIN model checker for verifying Promela models. We illustrate the practicality of this approach for synthesizing several Promela models.

Keywords: Model sketching, SPL (lifted) model checking, Abstract Interpretation

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* [24], 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 [20] and properties are expressed in Linear Temporal Logic (LTL) [1]. 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. This is so-called *model sketching problem*.

In this work, we frame the model sketching problem as a verification problem [18, 23] for model families (a.k.a. software product lines – SPLs) [4]. SPL Engineering represents an efficient method to achieve customization of software systems by using *features* (statically configured options) to organize the variable functionality. Family members, called *variants*, are specified in terms of features selected for that particular variant at compile-time. We consider model families implemented using `#if` directives from the C preprocessor `CPP` [4, 10].

All model sketch realizations can be encoded as a model family, where each numerical hole is represented by a numerical feature with the

same domain. Hence, the model sketching problem reduces to selecting correct variants (family members) from the resulting model family that satisfy the given LTL properties. The automated analysis [19] 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. This is particulary apparent in the case of model families that contain numerical features with big domains, thus admitting astronomic family sizes. This also affects the model sketching problem. A naive *brute force enumerative* solution is to check each individual variant of the model family by applying an off-the-shelf model checker to each variant. This is shown to be very inefficient for large families [3, 5] because the same execution behaviour is checked multiple times, whenever it is shared by many variants. An alternative is to model the entire family by a single compact model, called featured transition system (FTS), and then apply specialized *lifted (family-based) model checking* algorithms [3, 5] on it. Each execution behaviour in an FTS is associated with the set of variants able to produce it. This way, an execution behaviour is checked only once regardless of how many variants include it.

This paper applies an abstraction refinement procedure over the compact, all-in-one, representation of model families (FTS) [13, 14] to solve the model sketching problem. We use *variability abstractions* [13, 14] to construct a divide-and-conquer meta-algorithm for verifying FTSs that relies on using an existing single-system model checker SPIN. More specifically, we first devise variability abstractions tailored for model families that contain numerical features. This way, we extend the previous variability abstractions [13, 14] that were designed for model families with only Boolean features. Variability abstractions represent a configuration-space reduction technique that compresses the entire model family (with many configurations and variants) into an abstract model [22] (with a single abstract configuration and variant), so that the result of model checking a set of properties in the abstract model is preserved in all variants of the model family. The variability abstractions forget in which variant the abstract model operates. In particular, the abstraction aggregates multiple variants in a single abstract model, which is then fed to an off-the-shelf model checker. If no counterexample is found in the abstract model, then all variants satisfy the given property and any of them represents a solution to the sketching problem. Otherwise, the counterexamples are analysed and classified as either *genuine*, which correspond to execution behaviours of some concrete variants, or *spurious*, which are introduced due to the abstraction. If a genuine counterexample exist, the corresponding variants do not satisfy the given properties and are rejected. Otherwise, the abstraction is too coarse and a spurious counterexample is used to refine the current abstract model. The procedure is first applied on an abstract model that represents the entire model family, and then is repeated on the refined abstract models representing suitable sub-families of the original model family for which no conclusive results have been found so far. 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. This way, we also avoid the need for intermediate storage in memory of the concrete full-blown models. Experiments show that this approach often drastically reduces the average time and search steps required for finding one correct variant, compared to the brute-force approach and specialized lifted model checkers.

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 [20] to verify the generated abstract models. We illustrate this approach for automatic completion of various PROMELA model sketches with very large realization spaces. We also compare its performance with the brute-force enumerative approach and the lifted model checking approach.

The contributions of this paper are:

- We use *an abstraction-refinement verification procedure* for lifted model checking to resolve PROMELA model sketches with respect to LTL properties.
- We give *a proof of correctness* of our procedure.
- We *implement our procedure in the* PROMELASKETCHER *tool*, which calls the SPIN model checker to verify abstractions of model families.

- We perform *experimental evaluation* to assess the effectiveness of our technique by comparing its performances with the brute-force enumerative approach and the specialized lifted model checking tool ProVeLine.

This work is an extended and revised version of [7].

2 Motivating Example

Let us consider the following PROMELA code SIMPLE, which is the "Hello World" example of sketching [24]:

```
init {
  byte x;  int y;
  do :: break :: x++ od;
  y := ??*x;
  assert (y ≤ x+x) }
```

This sketch contains one integer hole, denoted by `??`, which should be replace with a constant from $\mathbb{Z}$, such that the synthesized model satisfies the given assertion for all possible traces in it. We use '`do :: break :: x++ od`' to non-deterministically initialize variable `x` of type `byte` to any integer value from its domain $[0, 255]$.

As observed before in the literature [2, 17], a sketch can be represented as a model family, such that numerical holes correspond to numerical features and all possible sketch realizations correspond to possible variants in the model family. For example, the SIMPLE sketch can be encoded as a model family that contains one numerical feature `A` with domain $[\text{Min}, \text{Max}] \subseteq \mathbb{Z}$ [1], such that the occurrence of the hole `??` is replaced with a constant value that `A` can take. The obtained model family SIMPLE is shown in Fig. 1a, where an `#if` directive from the `CPP` preprocessor [10] is used to encode multiple variants in the model family. There is one variant for each possible configuration, i.e. for each possible value that `A` can take from its domain $[\text{Min}, \text{Max}]$ at compile-time. Thus, there are $\text{Max} - \text{Min} + 1$ variants that can be derived from this family. The SIMPLE model family is represented by a featured transition system, which describes in an aggregated form behaviors of its variants. Each transition corresponding to `#if`-statements is labeled by a *feature expression* specifying for which variants the transition is to be included. For instance, from state (`x=1`,`y=0`) there are transitions labeled by: `A=`Min to state (`x=1`,`y=`Min), ..., `A=`Max to state (`x=1`,`y=`Max).

In the first iteration of our abstraction refinement procedure for the SIMPLE model family, we apply the join abstraction $\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}$ on the configuration space $\mathbb{K}$, thus obtaining an abstract model with a single abstract configuration, where the `#if` directive is replaced with an ordinary `if` statement where all guards corresponding to various variants are enabled (i.e., set to `true`). The abstract model of the SIMPLE family is shown in Fig. 1b. We can verify the obtained abstract model using the SPIN model checker, which finds that the assertion fails and reports a counterexample (trace). In this example, it reports the trace corresponding to the case when $(\texttt{A} = 3)$, i.e., when `y := 3*x`. Hence, in the second iteration, we divide the configuration space $\mathbb{K} = (\text{Min} \leq \texttt{A} \leq \text{Max})$ into two parts $(\text{Min} \leq \texttt{A} \leq 2)$ and $(4 \leq \texttt{A} \leq \text{Max})$, in order to eliminate the "erroneous" variant. We use the projection operator π to generate the corresponding partitions $\pi_{(\text{Min} \leq \texttt{A} \leq 2)}(\text{SIMPLE})$ and $\pi_{(4 \leq \texttt{A} \leq \text{Max})}(\text{SIMPLE})$. We construct single abstract models for each sub-family (partition), and repeat the verification task on each of them using SPIN. Fig. 1c shows the abstract model corresponding to the sub-family $(\text{Min} \leq \texttt{A} \leq 2)$. SPIN reports that the abstract model of $\pi_{(\text{Min} \leq \texttt{A} \leq 2)}(\text{SIMPLE})$ satisfies the given assertion. Hence, we can correctly replace the hole `??` with an integer from $[\text{Min}, 2]$.

3 Background: Model Families

In this section, we present the definitions of featured transition systems and LTL formulae.

3.1 Featured transition system

Let $\mathbb{F} = \{A_1, \ldots, A_k\}$ be a finite set of *numerical features* available in a model family. For each feature $A \in \mathbb{F}$, let $\text{dom}(A) \subseteq \mathbb{Z}$ denote the set of possible values of A. A valid combination of feature's values represents a *configuration* k, which specifies one *variant* of a program family. It is given as a *valuation function* $k : \mathbb{F} \to \mathbb{Z}$, which is

[1] Feature `A` is represented as a global variable of type `int`, and Min, Max represent minimal and maximal representable integers.

s

```
init {
  byte x;  int y;
  do :: break :: x++ od;
  #if :: (A=Min) → y := Min *x;
      ...
    :: (A=Max) → y := Max *x;
  #endif
  assert (y ≤ x+x) }
```

(a) SIMPLE.

```
init {
  byte x;  int y;
  do :: break :: x++ od;
  if :: true → y := Min *x;
       ...
     :: true → y := 3*x;
       ...
     :: true → y := Max *x;
  fi
  assert (y ≤ x+x) }
```

(b) $\alpha^{\text{join}}_{\mathbb{K}}$(SIMPLE).

```
init {
  byte x;  int y;
  do :: break :: x++ od;
  if :: true → y := Min *x;
       ...
     :: true → y := 2*x;
     :: false → y := 3*x;
       ...
     :: false → y := Max *x;
  fi
  assert (y ≤ x+x) }
```

(c) $\alpha^{\text{join}}_{\mathbb{K}}(\pi_{(\text{Min}\leq A\leq 2)}(\text{SIMPLE}))$.

Fig. 1: The SIMPLE model family and its abstractions.

a mapping that assigns a value from $\text{dom}(A)$ to each feature A. We assume that only a subset $\mathbb{K}$ of all possible configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be given by a formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_k = k(A_k))$. We have $\mathbb{K} \equiv \vee_{k\in\mathbb{K}} k$.

We use *transition systems* (TS) to describe behaviours of single systems. A *transition system* is a tuple $\mathcal{T} = (S, I, \mathit{trans}, AP, L)$, where S is a set of states; $I \subseteq S$ is a set of initial states; $\mathit{trans} \subseteq S \times S$ is a transition relation; AP is a set of atomic propositions; and $L : S \to 2^{AP}$ is a labelling function specifying which atomic propositions hold in a state. We write $s_1 \longrightarrow s_2$ whenever $(s_1, s_2) \in \mathit{trans}$. A *path* of a TS $\mathcal{T}$ is an *infinite* sequence $\rho = s_0 s_1 s_2 \ldots$ with $s_0 \in I$ s.t. $s_i \longrightarrow s_{i+1}$ for all $i \geq 0$. The *semantics* of $\mathcal{T}$, $[\![\mathcal{T}]\!]_{TS}$, 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. Their transitions are guarded by a *feature expression* that identifies the variants they belong to. The feature expressions, $\mathit{FeatExp}(\mathbb{F})$, are generated by the grammar:

$$\psi ::= \mathsf{true} \mid A \bowtie n \mid \neg\psi \mid \psi \wedge \psi$$

where $A \in \mathbb{F}$, $n \in \mathbb{Z}$, and $\bowtie \in \{=, <\}$. The atomic feature expressions $(A \bowtie n)$ are relationships between numerical features and constants. This is enough for the aim of this paper, so we do not consider more complex constraints. We write $[\![\psi]\!]$ for the set of configurations that satisfy ψ, i.e. $k \in [\![\psi]\!]$ iff $k \models \psi$.

A *featured transition system* (FTS) is a tuple $\mathcal{F} = (S, I, \mathit{trans}, AP, L, \mathbb{F}, \mathbb{K}, \delta)$, where $(S, I, \mathit{trans}, AP, L)$ form a TS; $\mathbb{F}$ is a set of available features; $\mathbb{K}$ is a set of valid configurations; and $\delta : \mathit{trans} \to \mathit{FeatExp}(\mathbb{F})$ is a total function decorating transitions with feature expressions.

Variant derivation is a simple projection of an FTS onto a single configuration producing a regular TS. Formally, the *projection* of an FTS $\mathcal{F}$ to a configuration $k \in \mathbb{K}$, denoted as $\pi_k(\mathcal{F})$, is the TS $(S, I, \mathit{trans}', AP, L)$, where $\mathit{trans}' = \{t \in \mathit{trans} \mid k \models \delta(t)\}$. We lift the definition of *projection* to sets of configurations $\mathbb{K}' \subseteq \mathbb{K}$, denoted as $\pi_{\mathbb{K}'}(\mathcal{F})$, by keeping the transitions admitted by at least one of the configurations in $\mathbb{K}'$. That is, $\pi_{\mathbb{K}'}(\mathcal{F})$, is the FTS $(S, I, \mathit{trans}', AP, L, \mathbb{F}, \mathbb{K}', \delta')$, where $\mathit{trans}' = \{t \in \mathit{trans} \mid \exists k \in \mathbb{K}'.k \models \delta(t)\}$ and $\delta' = \delta\,|_{\mathit{trans}'}$ is the restriction of δ to trans'. The *semantics* of an FTS $\mathcal{F}$, denoted as $[\![\mathcal{F}]\!]_{FTS}$, is the union of paths of the projections on all valid variants $k \in \mathbb{K}$, i.e. $[\![\mathcal{F}]\!]_{FTS} = \cup_{k\in\mathbb{K}} [\![\pi_k(\mathcal{F})]\!]_{TS}$.

Example 1 Figure 2 shows the FTS $\mathcal{F}'$ of a beverage vending machine [13, 14], which we will use as a running example . The FTS $\mathcal{F}'$ has two features: `CancelPurchase` (c, in brown), for canceling a purchase after a coin is entered that is a Boolean feature (we abbreviate $(c = 1)$ with c and $(c = 0)$ with $\neg c$); `ChooseDrink` (d, in red) for choosing drinks with domain $\{0 = \text{none}, 1 = \text{coffee}, 2 = \text{tea}, 3 = \text{capuccino}\}$. Each transition is labeled by a *feature expression* specifying for which variants the transition is included. For instance, the transition $s_1 \xrightarrow{d=2} s_3$ is included in variants where feature d is set to 2.

For clarity, we omit writing presence condition true in transitions. There is an atomic proposition $a \in L(s_6)$, which holds in state s_6.

By combining various features, a number of variants of $\mathcal{F}'$ can be obtained. The set of valid configurations is: $\mathbb{K}' = \{c \wedge (d=0), c \wedge (d=1), c \wedge (d=2), c \wedge (d=3), \neg c \wedge (d=0), \neg c \wedge (d=1), \neg c \wedge (d=2), \neg c \wedge (d=3)\}$. The basic version of $\mathcal{F}'$, described by the configuration $\neg c \wedge (d=2)$, works as follows: the machine takes a coin ($s_0 \to s_1$), selects a tea drink ($s_1 \to s_3$), serves it ($s_3 \to s_5$), opens the compartment so that a customer can take the drink ($s_5 \to s_6$), and then closes the compartment and waits for another order again ($s_6 \to s_0$). Note that $[\![d \neq 0]\!] = \{c \wedge (d=1), c \wedge (d=2), c \wedge (d=3), \neg c \wedge (d=1), \neg c \wedge (d=2), \neg c \wedge (d=3)\}$. Thus, $c \wedge (d=1) \models (d \neq 0)$, but $c \wedge (d=0) \not\models (d \neq 0)$.

3.2 LTL Properties

For specifying system properties, we consider the Linear Temporal Logic (LTL) [1] generated by the grammar:

$$\phi ::= \mathsf{true} \mid a \in AP \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \phi_1 \mathsf{U} \phi_2$$

LTL formulae ϕ are interpreted over paths of transition systems. The formula true holds for all paths. The formula a means that the atomic proposition a holds in the current (first) state of a path. The formula $\bigcirc\phi$ means that ϕ holds in the next state of a path, and $\phi_1 \mathsf{U} \phi_2$ means that ϕ_1 holds in all states visited until a state where ϕ_2 holds is reached. Other operators can be defined by means of syntactic sugar, for instance: $\Diamond\phi = \mathsf{true}\,\mathsf{U}\phi$ (ϕ holds eventually) and $\Box\phi = \neg\Diamond\neg\phi$ (ϕ always holds). We say that a TS $\mathcal{T}$ satisfies ϕ, written $\mathcal{T} \models \phi$, iff all paths of $\mathcal{T}$ satisfy formula ϕ: $\forall \rho \in [\![\mathcal{T}]\!]_{\mathrm{TS}}.\, \rho \models \phi$ [1]. We say that an FTS $\mathcal{F}$ satisfies ϕ, written $\mathcal{F} \models \phi$, iff all its variants satisfy ϕ, i.e. $\forall k \in \mathbb{K}.\, \pi_k(\mathcal{F}) \models \phi$.

Example 2 Recall FTS $\mathcal{F}'$ from Fig. 2. Consider the property $\phi' = \Box\Diamond a$, which states that the system will always reach the state where a holds eventually. Note that $\mathcal{F}' \not\models \phi'$. E.g., if feature c is enabled, a counter-example is: $s_0 \to s_1 \to s_0 \to \ldots$. However, there exist variants of $\mathcal{F}'$ satisfying ϕ'. E.g., $\pi_{[\![\neg c \wedge (d=1)]\!]}(\mathcal{F}') \models \phi'$.

4 Abstraction Refinement Framework

We now introduce variability abstractions and the abstraction refinement procedure for verifying FTSs, which are the key components of our model sketching algorithm.

4.1 Abstraction

We start working with Galois connections [14] between Boolean complete lattices of feature expressions, and then induce a notion of abstraction of FTSs. The Boolean complete lattice of feature expressions is: $(FeatExp(\mathbb{F})_{/\equiv}, \models, \vee, \wedge, \mathsf{true}, \mathsf{false}, \neg)$. The *join abstraction*, $\boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}$, replaces each feature expression ψ with true if there exists at least one configuration from $\mathbb{K}$ that satisfies ψ. The abstract set of features is empty: $\boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}(\mathbb{F}) = \emptyset$, and the abstract set of configurations is a singleton: $\boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}(\mathbb{K}) = \{\mathsf{true}\}$. The abstraction and concretization functions between $FeatExp(\mathbb{F})$ and $FeatExp(\emptyset)$, forming a Galois connection [14], are:

$$\boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}(\psi) = \begin{cases} \mathsf{true} & \text{if } \exists k \in \mathbb{K}. k \models \psi \\ \mathsf{false} & \text{otherwise} \end{cases}$$
$$\boldsymbol{\gamma}^{\mathrm{join}}_{\mathbb{K}}(\mathsf{true}) = \mathsf{true}, \quad \boldsymbol{\gamma}^{\mathrm{join}}_{\mathbb{K}}(\mathsf{false}) = \bigvee_{k \in 2^{\mathbb{F}} \setminus \mathbb{K}} k$$

We shall now use the above Galois connections for feature expressions to define abstraction of entire FTSs. This abstraction will basically apply the Galois connection $(\boldsymbol{\alpha}^{\mathrm{join}}, \boldsymbol{\gamma}^{\mathrm{join}})$ to each feature expression. Given the FTS $\mathcal{F} = (S, I, trans, AP, L, \mathbb{F}, \mathbb{K}, \delta)$, we define TS $\boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}(\mathcal{F}) = (S, I, trans', AP, L)$ to be its *abstraction*, where $trans' = \{t \in trans \mid \boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}(\delta(t)) = \mathsf{true}\}$. Note that transitions in the abstract TS $\boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}(\mathcal{F})$ describe the behaviour that is possible in some variants of the concrete FTS $\mathcal{F}$, 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. This way, the abstract model contains all paths from any valid variant of the concrete FTS, and moreover it may contain other paths that are not present in some concrete variant. That is, $[\![\boldsymbol{\alpha}^{\mathrm{join}}_{\mathbb{K}}(\mathcal{F})]\!]_{TS} \supseteq \cup_{k \in \mathbb{K}} [\![\pi_k(\mathcal{F})]\!]_{TS}$. Since a TS satisfies an LTL formula if all its paths satisfy the formula, the following holds.

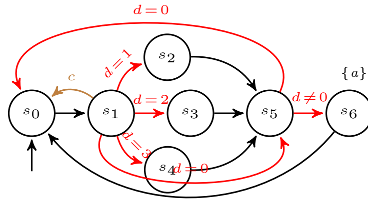

Fig. 2: $\mathcal{F}'$.

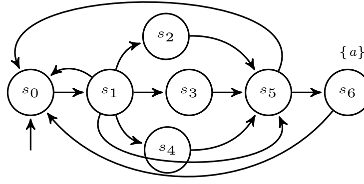

Fig. 3: $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}'}(\mathcal{F}')$.

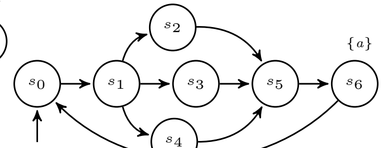

Fig. 4: $\boldsymbol{\alpha}^{\text{join}}_{[\![\neg c\wedge(d\neq0)]\!]}(\pi_{\neg c\wedge(d\neq0)}(\mathcal{F}'))$.

Theorem 1 (Preservation results) *For every $\phi \in LTL$, $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}}(\mathcal{F}) \models \phi \implies \mathcal{F} \models \phi$.*

The problem of evaluating $\mathcal{F} \models \phi$ can be reduced to a number of smaller problems. Let the subsets $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a *partition* of $\mathbb{K}$. Then, $\mathcal{F} \models \Phi$ iff $\pi_{\mathbb{K}_i}(\mathcal{F}) \models \phi$ for all $i = 1,\ldots,n$.

Corollary 2 *Let $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a* partition *of $\mathbb{K}$. If $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}_1}(\pi_{\mathbb{K}_1}(\mathcal{F})) \models \phi \wedge \ldots \wedge \boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}_n}(\pi_{\mathbb{K}_n}(\mathcal{F})) \models \phi$, then $\mathcal{F} \models \phi$. Moreover, if $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}_j}(\pi_{\mathbb{K}_j}(\mathcal{F})) \models \phi$ then $\pi_k(\mathcal{F}) \models \phi$ for all variants $k \in \mathbb{K}_j$.*

Example 3 Recall FTS $\mathcal{F}'$ of Fig. 2. Figure 3 shows its abstract model, the TS $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}'}(\mathcal{F}')$. Consider the property $\phi' = \Box\Diamond a$ introduced in Example 2. We have $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}'}(\mathcal{F}') \not\models \phi'$, since there is a path in $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}'}(\mathcal{F}')$ where s_6 is never reached: $s_0 \to s_1 \to s_0 \to \ldots$. So we cannot conclude whether ϕ' is satisfied or not by $\mathcal{F}'$ using the abstract model $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}'}(\mathcal{F}')$. Hence, refinement is needed to make the abstract model more precise.

4.2 Abstraction Refinement

We now describe an abstraction refinement procedure (`ARP`), which uses spurious counterexamples to iteratively refine partitions of $\mathbb{K}$ and abstract models until either property satisfaction is shown for some variants or a genuine counterexample is found for all variants.

The `ARP` for checking $\mathcal{F} \models \phi$ is illustrated by Algorithm 1. We first construct an initial abstract model $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}}(\mathcal{F})$, and check $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}}(\mathcal{F}) \models \phi$ (Line 2). This verification step can be performed using a single-system model checker such as SPIN. If the abstract model satisfies the given property (i.e., the counterexample c is `null`), then all variants from $\mathbb{K}$ 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 5, 10, 15).

Otherwise, a non-`null` counterexample c is found. Let ψ be the feature expression computed by conjoining feature expressions labelling all transitions that belong to path c when c is simulated in $\mathcal{F}$ (Line 7). This simulation of path c from $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}}(\mathcal{F})$ in $\mathcal{F}$ can be done due to the fact that $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}}(\mathcal{F})$ and $\mathcal{F}$ have the same control structures, except that the transitions in $\mathcal{F}$ are guarded by feature expressions. There are two cases to consider.

Algorithm 1 `ARP`$(\mathcal{F}, \mathbb{K}, \phi)$

Input: An FTS $\mathcal{F}$, a configuration set $\mathbb{K}$, and an LTL formula ϕ
Output: Correct variants $k \in \mathbb{K}$, s.t. $\pi_k(\mathcal{F}) \models \phi$

1: `Global Var:` `end:=false`
2: $c = (\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}}(\mathcal{F}) \models \phi)$
3: **if** c=`null` **then**
4:     `end:=true`
5:     **return** $\mathbb{K}$
6: **end if**
7: ψ`:=FeatExp(c)`
8: **if** `sat`$(\psi \wedge (\bigvee_{k\in\mathbb{K}} k))$ **then**
9:     $(\psi_1, \ldots, \psi_n)$`:= Split`$([\![\neg\psi]\!] \cap \mathbb{K})$
10:     **if** (`end`) **then return** $\emptyset$
11:     **end if**
12:     $ARP(\pi_{[\![\psi_1]\!]}(\mathcal{F}),[\![\psi_1]\!],\phi)$; ...; $ARP(\pi_{[\![\psi_n]\!]}(\mathcal{F}),[\![\psi_n]\!],\phi)$
13: **else**
14:     $\psi' =$ `CraigInterpolation`$(\psi, \mathbb{K})$
15:     **if** (`end`) **then return** $\emptyset$
16:     **end if**
17:     $ARP(\pi_{[\![\psi']\!]}(\mathcal{F}),[\![\psi']\!],\phi)$; $ARP(\pi_{[\![\neg\psi']\!]}(\mathcal{F}),[\![\neg\psi']\!],\phi)$
18: **end if**

First, if $\psi \wedge (\bigvee_{k\in\mathbb{K}} k)$ is satisfiable (i.e. $\mathbb{K} \cap [\![\psi]\!] \neq \emptyset$), then the found counterexample c is *genuine* for variants in $\mathbb{K} \cap [\![\psi]\!]$. For the other variants from $\mathbb{K} \cap [\![\neg\psi]\!]$, the found counterexample cannot be executed (Lines 9,10,11,12). We call `Split` to split the space $\mathbb{K} \cap [\![\neg\psi]\!]$ in sub-families $[\![\psi_1]\!], \ldots, [\![\psi_n]\!]$, such that all atomic constraints

in ψ_i are of the form: $(A \bowtie n)$, where $A \in \mathbb{F}$ and $n \in dom(A)$. For example, assume that we have two numerical features $\text{Min} \leq \texttt{A} \leq \text{Max}$ and $\text{Min} \leq \texttt{B} \leq \text{Max}$. If $\psi = (\texttt{A}=3)$, then $\texttt{Split}([\![\neg\psi]\!])$ is $(\text{Min} \leq \texttt{A} \leq 2) \wedge (\text{Min} \leq \texttt{B} \leq \text{Max})$ and $(4 \leq \texttt{A} \leq \text{Max}) \wedge (\text{Min} \leq \texttt{B} \leq \text{Max})$; if $\psi = (\texttt{A}=3) \wedge (\texttt{B} = 2)$, then $\texttt{Split}([\![\neg\psi]\!])$ is $(\text{Min} \leq \texttt{A} \leq 2) \wedge (\text{Min} \leq \texttt{B} \leq 1)$, $(\text{Min} \leq \texttt{A} \leq 2) \wedge (3 \leq \texttt{B} \leq \text{Max})$, $(4 \leq \texttt{A} \leq \text{Max}) \wedge (\text{Min} \leq \texttt{B} \leq 1)$ and $(4 \leq \texttt{A} \leq \text{Max}) \wedge (3 \leq \texttt{B} \leq \text{Max})$. Finally, we call `ARP` to verify the sub-families: $\pi_{[\![\psi_1]\!]}(\mathcal{F}), \ldots, \pi_{[\![\psi_n]\!]}(\mathcal{F})$. Note that if $\mathbb{K} \cap [\![\neg\psi]\!] = \emptyset$, then `Split` updates the variable `end` to `true` and so no recursive `ARP`-s are called.

Second, if $\psi \wedge (\bigvee_{k \in \mathbb{K}} k)$ is unsatisfiable (i.e. $\mathbb{K} \cap [\![\psi]\!] = \emptyset$), then the found counterexample c is *spurious* for all variants in $\mathbb{K}$ (due to incompatible feature expressions) (Lines 14,15,16,17). Now, we describe how a feature expression ψ' used for constructing refined sub-families is determined by means of Craig interpolation [12] from ψ and $\mathbb{K}$. We find the minimal unsatisfiable core ψ^c of $\psi \wedge (\bigvee_{k \in \mathbb{K}} k)$, which contains a subset of conjuncts in $\psi \wedge (\bigvee_{k \in \mathbb{K}} k)$, such that ψ^c is still unsatisfiable and if we drop any single conjunct in ψ^c then the result becomes satisfiable. We group conjuncts in ψ^c in two non-empty groups X and Y such that $\psi^c = X \wedge Y = \textsf{false}$. Then, the interpolant ψ' is such that: 1) $X \implies \psi'$, 2) $\psi' \wedge Y = \textsf{false}$, 3) ψ' refers only to common variables of X and Y. The interpolant ψ' summarizes and translates why X is inconsistent with Y in their shared language. Finally, we call the `ARP` to check $\pi_{[\![\psi']\!]}(\mathcal{F}) \models \phi$ and $\pi_{[\![\neg\psi']\!]}(\mathcal{F}) \models \phi$. By construction, it is guaranteed that the spurious counterexample c does not occur in both $\pi_{[\![\psi']\!]}(\mathcal{F})$ and $\pi_{[\![\neg\psi']\!]}(\mathcal{F})$ [12].

Theorem 3 *ARP$(\mathcal{F}, \mathbb{K}, \phi)$ terminates and is correct.*

Proof Since `ARP` is always called in the next iteration for strictly smaller partitions $\mathbb{K}' \subset \mathbb{K}$ and the configuration space is finite, the number of iterations is also finite. The correctness of `ARP` follows from Theorem 1 and Corollary 2. □

Example 4 Consider the call $\texttt{ARP}(\mathcal{F}', \mathbb{K}', \phi')$, where $\mathcal{F}'$, $\mathbb{K}'$, and ϕ' are from Examples 1 and 2. We first check $\boldsymbol{\alpha}^{\text{join}}_{\mathbb{K}'}(\mathcal{F}') \models \phi'$. The following counterexample is found: $c = s_0 \to s_1 \to s_0 \to \ldots$. The associated feature expression in $\mathcal{F}'$ is: c. So, this is a genuine counterexample for variants $[\![c]\!] \cap \mathbb{K}' = \{c \wedge (d = 0), c \wedge (d = 1), c \wedge (d = 2), c \wedge (d = 3)\}$. In the next iteration, we check $\pi_{[\![\neg c]\!] \cap \mathbb{K}'}(\mathcal{F}') \models \phi'$? We obtain the counterexample: $c = s_0 \to s_1 \to s_3 \to s_5 \to s_0 \to \ldots$, with the associated feature expression $(d=2) \wedge (d=0)$. This is a spurious counterexample with the interpolant $(d=0)$. In the next iteration, we consider calls to check $\pi_{[\![\neg c \wedge (d=0)]\!] \cap \mathbb{K}'}(\mathcal{F}') \models \phi'$ and $\pi_{[\![\neg c \wedge (d \neq 0)]\!] \cap \mathbb{K}'}(\mathcal{F}') \models \phi'$? We obtain that $\boldsymbol{\alpha}^{\text{join}}_{[\![\neg c \wedge (d \neq 0)]\!]}(\pi_{[\![\neg c \wedge (d \neq 0)]\!]}(\mathcal{F}')) \models \phi'$ holds (see $\boldsymbol{\alpha}^{\text{join}}_{[\![\neg c \wedge (d \neq 0)]\!]}(\pi_{[\![\neg c \wedge (d \neq 0)]\!]}(\mathcal{F}'))$ in Fig. 4). Thus, `ARP` ends by reporting correct variants $[\![\neg c \wedge (d \neq 0)]\!] = \{\neg c \wedge (d=1), \neg c \wedge (d=2), \neg c \wedge (d=3)\}$.

5 Syntactic Transformations

We introduce the modelling language Promela for writing sketches and model families, describe how to translate sketches into model families, and how to construct abstract models and projections out of model families.

5.1 Promela language

We now present the syntax of the Promela language.

Syntax.

Promela is a non-deterministic modelling language of the SPIN model checker [20], which is designed for describing systems composed of concurrent processes that communicate asynchronously. A Promela model, P, consists of a finite set of processes to be executed concurrently. The basic statements of processes are: `skip`, `break`, `x` := *expr*, $c?x$, $c!expr$, $s_1 ; s_2$, $\texttt{if} :: g_1 \to s_1 \cdots :: g_n \to s_n\, \texttt{fi}$, $\texttt{do} :: g_1 \to s_1 \cdots :: g_n \to s_n\, \texttt{od}$, where `x` is a variable, *expr* is an expression, c is a channel, and g_i are conditions over variables.

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 $[n_i, n_i']$.

Model Families.

To encode multiple variants, a new compile-time conditional statement is included. The new

guarded-by-features statement is of the form:

$$\texttt{\#if} :: \psi_1 \to s_1 \ldots :: \psi_n \to s_n \ \texttt{\#endif}$$

where $\psi_1, \ldots, \psi_n$ are feature expressions defined over $\mathbb{F}$. Actually, this is the only place where features may be used. If presence condition ψ_i is satisfied by a configuration $k \in \mathbb{K}$ the statement s_i will be included in the variant corresponding to k. Hence, "**#if**" plays the same role as "**#if**" directives in C preprocessor CPP [10].

5.2 Syntactic Transformations

We now present the syntactic transformations of the PROMELA models used in this work.

From Sketches to Model Families.

Our aim is to transform a sketch $\hat{P}$ with a set of holes $\texttt{??}_1^{[n_1,n_1']}, \ldots, \texttt{??}_m^{[n_m,n_m']}$, into a model family $\overline{P}$ with a set of numerical features $A_1, \ldots, A_m$ with domains $[n_1, n_1'], \ldots, [n_m, n_m']$, respectively. The set of configurations $\mathbb{K}$ includes all possible combinations of feature's values. We now define a rewrite rule for eliminating holes ?? from a model sketch. The rewrite rule is:

$$s[\texttt{??}^{[n,n']}] \leadsto \texttt{\#if} :: (\texttt{A}{=}n) \to s[n] \ldots :: (\texttt{A}{=}n') \to s[n'] \ \texttt{\#endif} \quad (1)$$

where $s[-]$ is a (non-compound) basic statement with a single expression – in it, $\texttt{??}^{[n,n']}$ 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 (1) is that if the current sketch being transformed matches the abstract syntax tree node of the shape $s[\texttt{??}^{[n,n']}]$ then replace $s[\texttt{??}^{[n,n']}]$ according to the rule (1). We write $\texttt{Rewrite}(\hat{P})$ to be the final model family obtained by repeatedly applying the rule (1) on sketch $\hat{P}$ and on its transformed versions until we reach a point where this rule can not be applied.

Let H be a set of holes in the sketch $\hat{P}$. We define a *control function* $\varphi : \Phi = H \to \mathbb{Z}$ to describe the value of each hole in the sketch. We write $[\![\hat{P}]\!]_{TS}^{\varphi}$ for TS obtained by replacing holes in $\hat{P}$ according to φ. By induction on the structure of $\hat{P}$, we can show:

Theorem 4 *Let $\hat{P}$ be a sketch and φ be a control function, s.t. features $A_1, \ldots, A_n$ correspond to holes $\texttt{??}_1, \ldots, \texttt{??}_n$. We define a configuration $k \in \mathbb{K}$, s.t. $k(A_i) = \varphi(\texttt{??}_i)$ for $1 \leq i \leq n$. Let $\overline{P} = \texttt{Rewrite}(\hat{P})$. We have: $[\![\hat{P}]\!]_{TS}^{\varphi} \equiv [\![\pi_k([\![\overline{P}]\!]_{FTS})]\!]_{TS}$.*

Example 5 The sketch SIMPLE in Section 2 is encoded as the model family SIMPLE in Fig. 1a.

From Model Families to Projections and Abstract Models.

We present two syntactic transformations of model families $\overline{P} = \texttt{Rewrite}(\hat{P})$ obtained from PROMELA sketches $\hat{P}$: projection $\pi_{[\![\psi]\!]}(\overline{P})$ and variability abstraction $\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}(\overline{P})$.

The projection $\pi_{[\![\psi]\!]}(\overline{P})$ is obtained by defining a translation recursively over the structure of ψ. Let ψ be of the form $(\texttt{A} < m)$. The rewrite rule is:

$$\begin{array}{l} \texttt{\#if} :: (\texttt{A}{=}n) \to s[n] \ldots :: (\texttt{A}{=}m) \to s[m] \ldots :: (\texttt{A}{=}n') \to s[n'] \texttt{\#end} \leadsto \\ \texttt{\#if} :: (\texttt{A}{=}n) \to s[n] \ldots :: \mathsf{false} \to s[m] \ldots :: \mathsf{false} \to s[n'] \texttt{\#end} \end{array} \quad (2)$$

That is, all guards that do not satisfy $(\texttt{A} < m)$ are replaced with false. The rewrite rule for $(A = m)$ is defined similarly. Let ψ be a feature expression of the form $\neg\psi'$. We first transform $\overline{P}$ by applying the projection ψ', then in all **#if**-s obtained from the projection ψ' we change the guards: 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.

The abstract model $\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}(\overline{P})$ is obtained by resolving all "**#if**"-s. The rewrite rule is:

$$\begin{array}{l} \texttt{\#if} :: \psi_1 \to s_1 :: \ldots :: \psi_n \to s_n \ \texttt{\#endif} \leadsto \\ \texttt{if} :: \boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}(\psi_1) \to s_1 :: \ldots :: \boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}(\psi_n) \to s_n \ \texttt{fi} \end{array} \quad (3)$$

where all guards in the new **if** are set to *true* or *false* depending whether there is some valid configurations that satisfies that guard or not.

Let $\overline{P}$ be a PROMELA family and let $[\![\overline{P}]\!]_{FTS}$ be the FTS obtained by its compilation. The following correctness result states that the abstract model obtained by applying $\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}$ on $[\![\overline{P}]\!]_{FTS}$ coincides with the TS obtained by compiling $\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}(\overline{P})$. The same applies for projections $\pi_{[\![\psi]\!]}$. The following result follows by observation that feature expressions are introduced in FTSs and PROMELA model families only through "**#if**"-s.

Theorem 5 *$\pi_{[\![\psi]\!]}([\![\overline{P}]\!]_{FTS}) \equiv [\![\pi_{[\![\psi]\!]}(\overline{P})]\!]_{FTS}$ and $\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}([\![\overline{P}]\!]_{FTS}) \equiv [\![\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}(\overline{P})]\!]_{TS}$.*

Example 6 For the model family SIMPLE in Fig. 1a, the abstract model $\boldsymbol{\alpha}_{\mathbb{K}}^{\text{join}}$(SIMPLE) is given in Fig. 1b, while $\boldsymbol{\alpha}^{\text{join}}_{\text{Min}\leq\mathtt{A}\leq 2}(\pi_{\text{Min}\leq\mathtt{A}\leq 2}(\text{SIMPLE}))$ is in Fig. 1c.

6 Synthesis Algorithm

The synthesis algorithm SYNTHESIZE($\hat{P}$) for solving a sketch $\hat{P}$ is described by Algorithm 2. The sketch $\hat{P}$ is first encoded as a model family $\overline{P} = \mathtt{Rewrite}(\hat{P})$ (Line 1). Then, we call function $\mathtt{ARP}(\overline{P}, \mathbb{K}, \phi)$, which takes as input the model family $\overline{P}$, its configuration set $\mathbb{K}$, and the property to verify ϕ, and returns as solution a set of variants $\mathbb{K}' \subseteq \mathbb{K}$ that satisfy ϕ (Line 2). Finally, the inferred set of correct variants $\mathbb{K}' \subseteq \mathbb{K}$ is returned as solution of the given model sketching problem (Line 3).

Algorithm 2 SYNTHESIZE($\hat{P}$)

Input: A sketch $\hat{P}$
Output: Correct realizations of $\hat{P}$
1: $\overline{P} = \mathtt{Rewrite}(\hat{P})$
2: $\mathbb{K}' = \mathtt{ARP}(\overline{P}, \mathbb{K}, \phi)$
3: **return** $\mathbb{K}'$

The correctness of SYNTHESIZE($\hat{P}$) follows from the correctness of Rewrite (see Theorem 4), ARP (see Theorem 3) and syntactic transformations (see Theorem 5).

7 Evaluation

We now evaluate our approach for model sketching by abstraction refinement for lifted model checking.

Implementation

We have developed a prototype model synthesizer, called PromelaSketcher, for resolving Promela sketches. It uses the ANTLR parser generator (https://www.antlr.org/) for processing Promela code, while projections and abstractions of #if-enriched Promela code are implemented using source-to-source transformations as described in Section 5. It calls the SPIN model checker [20] to verify the generated Promela abstract models. Our tool is written in Java and consists of around 2K LOC.

Experiment setup and Benchmarks

All experiments are run on 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM LUbuntu 20.10, with 8 GB memory. All times are reported as average over five independent executions. The tool is available: https://github.com/aleksdimovski/Promela_sketcher. We compare our approach with the Brute force enumeration and the lifted model checker ProVeLine. The Brute force enumeration approach generates all possible sketch realizations and verifies them using SPIN one by one. The lifted model checker ProVeLine takes as input Promela model families obtained by transforming model sketches $\overline{P} = \mathtt{Rewrite}(\hat{P})$, compiles them into FTSs, and applies specialized lifted model checking algorithms directly over the obtained FTSs, thus returning the set of correct variants. Hence, ProVeLine is called only once for each benchmark. 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-bits, 4-bits, and 8-bits. Note that a sketch with n-bits holes is translated into a family with features that have 2^n possible values. We only measure the model checking SPIN times to generate a process analyser (pan) and to execute it.

Performance Results

Table 1 shows the performance results of synthesizing our benchmarks.

The Loop sketch [24] in Fig 5 contains one hole ?? represented by feature A. The coarsest abstract model has an if statement with one optional sequence 'do :: (x>n) $\rightarrow$ ...od' for each possible value n of feature A. PromelaSketcher reports counterexamples for the cases $(A = Min), \ldots, (A = 4)$, and then we obtain the correct solution for the abstraction $(5 \leq A \leq Max)$.

The Welfare sketch [20] is a problem due to Feijen. There are three ordered lists of integers a, b, and c. At least one element appears in all three lists. Find the smallest indices i, j, and k, such that a[i]=b[j]=c[k]. That is, we want to find the first element that appears in all three lists. The list c is initialized in such a way that concrete values assigned to the first $n-1$ elements do not appear in lists a and b, and the last n-th element of c is assigned to the hole ??. Hence, the hole ?? should

```
init {
  byte x:=10;
  int y:=0;
  do :: (x>??) → x--;
                y++
     :: else → break
  od;
  assert (y < 6) }
```

Fig. 5: Loop.

```
int a[5], b[5], c[5];
init {
  byte i, j, k;
  a[0]:=1; ... a[4]:=18;
  b[0]:=4; ... a[4]:=25;
  c[0]:=5; ... c[4]:=??;
  do :: a[i]<b[j] ∧ i<4 → i++;
     :: b[j]<c[k] ∧ j<4 → j++;
     :: c[k]<a[i] ∧ k<4 → k++;
     :: else → break od;
  assert(a[i]=b[j] ∧b[j]=c[k]
             ∧c[k]=a[i]) }
```

Fig. 6: Welfare.

```
byte N:=4, MAX, distance[16];
byte city, dest, tour, seen;
bool visited[4];
#define Dist(a,b) distance[4*a+b]
inline travel2(dest) {
  (city != dest ∧ tour ≤ MAX) →
  tour := tour + Dist(city,dest)
  city := dest
  if :: (¬visited[city]) →
        visited[city]:=true; seen++
     :: else → break fi}
init {
  MAX:=??;
  Dist(0,1)=20; ... Dist(3,2)=12;
  do :: select (dest: 0 .. (N-1)) →
                   travel2(dest) od;
  ltl p {[](seen<N ∧ tour>MAX) } }
```

Fig. 7: Salesman.

be replaced with the smallest value that appear also in lists a and b. PromelaSketcher successfully partitions the configuration space and finds the correct solutions for various values assigned to lists a, b, and c.

The Salesman sketch [20] is a well-known optimisation problem, whose Promela solution is given in Fig. 7. Given a list of N cities and distances between each pair of cities, it asks to find the shortest possible tour that visits each city and returns to the origin city. We now use our approach to find the shortest tour through the cities. We initialize variable MAX to an integer hole ??. Whenever there exists a shorter tour than the one assigned to MAX, the given LTL property p fails and a counterexample is reported. Therefore, the LTL property p will be correct only when MAX is initialized to the value less or equal to the shortest possible tour. PromelaSketcher successfully finds this value for ?? in only two iterations. In the first iteration, it reports a counterexample with a tour of length $(n + 1)$ that is greater than the shortest possible tour that is of length n. Then, in the second iteration, the abstraction $(A \leq n)$ satisfies the property p.

In summary, 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. We can also see that PromelaSketcher outperforms ProVeLine for 3-bits benchmarks. For bigger benchmarks, ProVeLine crashes with an out-of-memory error.

8 Related Work

One of the earliest sketching approaches is the Sketch tool [24] that uses SAT-based counterexample-guided inductive synthesis to resolve program sketches in C. Recently, there have been proposed several works that resolve program sketches using lifted (SPL) static analysis. The work [17] uses a forward numerical lifted analysis [10] based on abstract interpretation to resolve numerical sketches with respect to Boolean specifications, while the works [8, 9] use a combination of forward and backward [6, 21] analyses to resolve sketches with respect to Boolean and quantitative specification. In the context of model sketching, Ceska et. al [2] have proposed two approaches to the automated synthesis of partial probabilistic models: the first one is based on abstraction refinement, and the second approach is based on inductive synthesis.

Classen et al. [3, 5] have shown how specifically designed lifted model checking algorithms can be used for verifying LTL and CTL properties of FTSs. The variability abstractions and the abstraction-refinement procedures are proposed

Table 1: Performance results. All times in sec.

Bench.	3 bits						4 bits						8 bits					
	Promela Sketcher		Brute force		ProVeLine		Promela Sketcher		Brute force		ProVeLine		Promela Sketcher		Brute force		ProVeLine	
	Call	Time	Call	Time	Call	Time	Call	Time	Call	Time	Call	Time	Call	Time	Call	Time	Call	Time
Simple	2	0.122	8	0.303	1	0.144	2	0.123	16	0.610	1	crash	2	0.127	256	10.04	1	crash
Loop	4	0.262	8	0.303	1	0.167	4	0.263	16	0.610	1	crash	4	0.653	256	9.739	1	crash
Welfare	4	0.277	8	0.291	1	0.351	5	0.278	16	0.617	1	crash	10	0.539	256	9.849	1	crash
Salesman	2	0.148	8	0.319	1	0.221	2	0.142	16	0.636	1	crash	2	0.142	256	10.29	1	crash

for efficient lifted model checking of LTL [12, 14] and CTL [11, 15, 16].

9 Conclusion

In this paper, we employ techniques from product-line model checking for resolving model sketches.

References

[1] Baier C, Katoen J (2008) Principles of model checking. MIT Press

[2] Ceska M, Dehnert C, Jansen N, et al (2019) Model repair revamped - - on the automated synthesis of markov chains -. In: Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday, LNCS, vol 11500. Springer, pp 107–125, https://doi.org/10.1007/978-3-030-31514-6_7, URL https://doi.org/10.1007/978-3-030-31514-6_7

[3] Classen A, Cordy M, Heymans P, et al (2012) Model checking software product lines with SNIP. STTT 14(5):589–612. https://doi.org/10.1007/s10009-012-0234-1, URL http://dx.doi.org/10.1007/s10009-012-0234-1

[4] Clements P, Northrop L (2001) Software Product Lines: Practices and Patterns. Addison-Wesley

[5] Cordy M, Schobbens P, Heymans P, et al (2013) Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In: 35th International Conference on Software Engineering, ICSE '13. IEEE Computer Society, pp 472–481, https://doi.org/10.1109/ICSE.2013.6606593, URL https://doi.org/10.1109/ICSE.2013.6606593

[6] Dimovski AS (2021) Lifted termination analysis by abstract interpretation and its applications. In: GPCE '21: Concepts and Experiences. ACM, pp 96–109, https://doi.org/10.1145/3486609.3487202, URL https://doi.org/10.1145/3486609.3487202

[7] Dimovski AS (2022) Model sketching by abstraction refinement for lifted model checking. In: SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, 2022. ACM, pp 1845–1848, https://doi.org/10.1145/3477314.3507170, URL https://doi.org/10.1145/3477314.3507170

[8] Dimovski AS (2022) Quantitative program sketching using lifted static analysis. In: 25th International Conference, FASE 2022, Proceedings, LNCS, vol 13241. Springer, pp 102–122, https://doi.org/10.1007/978-3-030-99429-7_6, URL https://doi.org/10.1007/978-3-030-99429-7_6

[9] Dimovski AS (2023) Quantitative program sketching using decision tree-based lifted analysis. J Comput Lang 75:101,206. https://doi.org/10.1016/j.cola.2023.101206, URL https://doi.org/10.1016/j.cola.2023.101206

[10] Dimovski AS, Apel S (2021) Lifted static analysis of dynamic program families by abstract interpretation. In: 35th European Conf. on Object-Oriented Programming, ECOOP 2021, LIPIcs, vol 194. Schloss Dagstuhl, pp 14:1–14:28, https://doi.org/10.

4230/LIPIcs.ECOOP.2021.14, URL https://doi.org/10.4230/LIPIcs.ECOOP.2021.14

[11] Dimovski AS, Wasowski A (2017) From transition systems to variability models and from lifted model checking back to UPPAAL. In: Models, Algorithms, Logics and Tools, LNCS, vol 10460. Springer, pp 249–268, https://doi.org/10.1007/978-3-319-63121-9_13, URL https://doi.org/10.1007/978-3-319-63121-9_13

[12] Dimovski AS, Wasowski A (2017) Variability-specific abstraction refinement for family-based model checking. In: 20th International Conference, FASE 2017, Proceedings, LNCS, vol 10202. Springer, pp 406–423, https://doi.org/10.1007/978-3-662-54494-5_24, URL http://dx.doi.org/10.1007/978-3-662-54494-5_24

[13] Dimovski AS, Al-Sibahi AS, Brabrand C, et al (2015) Family-based model checking without a family-based model checker. In: 22nd Int. Symposium, SPIN 2015, Proceedings, LNCS, vol 9232. Springer, pp 282–299, https://doi.org/10.1007/978-3-319-23404-5_18, URL http://dx.doi.org/10.1007/978-3-319-23404-5_18

[14] Dimovski AS, Al-Sibahi AS, Brabrand C, et al (2017) Efficient family-based model checking via variability abstractions. STTT 19(5):585–603. https://doi.org/10.1007/s10009-016-0425-2, URL https://doi.org/10.1007/s10009-016-0425-2

[15] Dimovski AS, Legay A, Wasowski A (2019) Variability abstraction and refinement for game-based lifted model checking of full CTL. In: 22nd International Conference, FASE 2019, Proceedings, LNCS, vol 11424. Springer, pp 192–209, https://doi.org/10.1007/978-3-030-16722-6_11, URL https://doi.org/10.1007/978-3-030-16722-6_11

[16] Dimovski AS, Legay A, Wasowski A (2020) Generalized abstraction-refinement for game-based CTL lifted model checking. Theor Comput Sci 837:181–206. https://doi.org/10.1016/j.tcs.2020.06.011, URL https://doi.org/10.1016/j.tcs.2020.06.011

[17] Dimovski AS, Apel S, Legay A (2021) Program sketching using lifted analysis for numerical program families. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings, LNCS, vol 12673. Springer, pp 95–112, https://doi.org/10.1007/978-3-030-76384-8_7, URL https://doi.org/10.1007/978-3-030-76384-8_7

[18] Ghosh D, Singh J (2020) Effective spectrum-based technique for software fault finding. Int j inf tecnol 12:677–682. https://doi.org/10.1007/s41870-019-00347-1, URL https://doi.org/10.1007/s41870-019-00347-1

[19] Goyal S, Bhatia PK (2021) Software fault prediction using lion optimization algorithm. Int j inf tecnol 13:2185–2190. https://doi.org/10.1007/s41870-021-00804-w, URL https://doi.org/10.1007/s41870-021-00804-w

[20] Holzmann GJ (2004) The SPIN Model Checker - primer and reference manual. Addison-Wesley

[21] MD. Obaidullah Al-Faruk MASK. M. Akib Hussain, Tonni SM (2020) Bfm: a forward backward string matching algorithm with improved shifting for information retrieval. Int j inf tecnol 12:479–483. https://doi.org/10.1007/s41870-019-00371-1, URL https://doi.org/10.1007/s41870-019-00371-1

[22] Mohan GB, Kumar RP (2023) Lattice abstraction-based content summarization using baseline abstractive lexical chaining progress. Int j inf tecnol 15:369–378. https://doi.org/10.1007/s41870-022-01080-y, URL https://doi.org/10.1007/s41870-022-01080-y

[23] Shozab Khurshid AKS, Iqbal J (2021) Effort based software reliability model with fault reduction factor, change point and imperfect debugging. Int j inf tecnol 13:331–340. https://doi.org/10.1007/s41870-019-00286-x, URL https://doi.org/10.1007/s41870-019-00286-x

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