SigrefMC: multi-core signature refinement

Tom van Dijk and Jaco van de Pol

Formal Methods & Tools, University of Twente

October 19, 2015

Contents

1 Introduction

This report is the documentation for the SigrefMC tool, developed by Tom van Dijk at the Formal Methods & Tools research group of the University of Twente. SigrefMC implements multi-core signature-based partition refinement using the multi-core (MT)BDD package Sylvan [5].

SigrefMC performs bisimulation minimisation for labeled transition systems (LTSs), continuous-time Markov chains (CTMCs) and interactive Markov chains (IMCs), which combines the features of LTSs and CTMCs. These allow the analysis of quantitative properties, e.g. performance and dependability. SigrefMC implements strong bisimulation and branching bisimulation. Strong bisimulation preserves both internal behavior (τ-transitions) and observable behavior, while branching bisimulation abstracts from internal behavior. The advantage of branching bisimulation compared to other variations of weak bisimulation is that it preserves the branching structure of the LTS, thus preserving certain interesting properties such as CTL* without next-state operator.

The tool supports two input formats, the XML format used by the original Sigref tool [9], and the BDD format that the LTSmin toolset [3] generates for various model checking languages. SigrefMC supports both the floating-point and the rational representation of rates in continuous-time transitions.

One of the design goals of this tool is to encourage researchers to extend it for their own file formats and notions of bisimulation, and to integrate it in other toolsets. Therefore, SigrefMC is freely available online[1] and licensed with the Apache 2.0 license.

[1] https://github.com/utwente-fmt/sigrefmc

2 Background

This chapter contains a condensed version of several sections from [6].

2.1 Bisimulation minimisation

One core challenge in model checking is the state space explosion problem. The space and time requirements of model checking increase exponentially with the size of the models. Bisimulation minimisation computes the smallest equivalent model (maximal bisimulation) under some notion of equivalence, which can significantly reduce the number of states. This technique is also used to abstract models from internal behavior, when only observable behavior is relevant.

The maximal bisimulation of a model is typically computed using partition refinement. Starting with an initially coarse partition (e.g. all states are equivalent), the partition is refined until states in each equivalence class can no longer be distinguished. The result is the maximal bisimulation with respect to the initial partition. Another well-known method to deal with very large state spaces is symbolic model checking, where sets of states are represented by their characteristic function, which is efficiently stored using binary decision diagrams (BDDs).

One particular application of symbolic bisimulation minimisation is as a bridge between symbolical models and explicit-state analysis algorithms. Such models can have very large state spaces that are efficiently encoded using BDDs. If the minimised model is sufficiently small, then it can be analyzed efficiently using explicit-state algorithms. The symbolic representation of the maximal bisimulation, when effective, often tends to be much larger than the original model.

Blom and Orzan [1] introduced a signature-based method for partition refinement, which assigns states to equivalence classes according to a characterizing signature. This method easily extends to various types of bisimulation. Wimmer et al. [8, 9] implemented symbolic bisimulation minimisation based on the signatures introduced by Blom. Their tool is called Sigref and their work is the basis for SigrefMC.

To take advantage of computer systems with multiple processors, developing scalable parallel algorithms is the way forward. In [5], we implemented the multi-core BDD package Sylvan, applying parallelism to symbolic model checking. SigrefMC is based on Sylvan to implement multi-core signature-based symbolic partition refinement.

2.2 Definitions

We recall the basic definitions of partitions, of LTSs, of CTMCs, of IMCs, and of various bisimulations as in [1, 2, 8, 9, 10].

Definition 1. *Given a set S, a partition π of S is a subset $\pi \subseteq 2^S$ such that*

$$\bigcup_{C \in \pi} C = S \qquad \text{and} \qquad \forall C, C' \in \pi \colon (C = C' \vee C \cap C' = \emptyset).$$

If π' and π are two partitions, then π' is a refinement of π, written $\pi' \sqsubseteq \pi$, if each block of π' is contained in a block of π. The elements of π are called equivalence classes or blocks. Each equivalence relation $\equiv$ is associated with a partition $\pi = S/\!\equiv$. In this paper, we use π and $\equiv$ interchangeably.

Definition 2. *A labeled transition system (LTS) is a tuple $(S, \mathsf{Act}, \rightarrow)$, consisting of a set of states S, a set of labels Act that may contain the non-observable action τ, and transitions $\rightarrow \subseteq S \times \mathsf{Act} \times S$.*

We write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. and $s \xrightarrow{\tau} \!\!\!\!\!\rightarrow$ when s has no outgoing τ-transitions. We use $\xrightarrow{a*}$ to denote the transitive reflexive closure of $\xrightarrow{a}$. Given an equivalence relation $\equiv$, we write $\underset{\equiv}{\xrightarrow{a}}$ for $\xrightarrow{a} \cap \equiv$, i.e., transitions between equivalent states, called *inert* transitions. We use $\underset{\equiv}{\xrightarrow{a*}}$ for the transitive reflexive closure of $\underset{\equiv}{\xrightarrow{a}}$.

Definition 3. *A continuous-time Markov chain (CTMC) is a tuple $(S, \Rightarrow)$, consisting of a set of states S and Markovian transitions $\Rightarrow \subseteq S \times \mathbb{R}^{>0} \times S$.*

We write $s \overset{\lambda}{\Rightarrow} t$ for $(s, \lambda, t) \in \Rightarrow$. The interpretation of $s \overset{\lambda}{\Rightarrow} t$ is that the CTMC can switch from s to t within d time units with probability $1 - e^{-\lambda \cdot d}$. For a state s, let $\mathbf{R}(s)(s') = \sum\{\lambda \mid s \overset{\lambda}{\Rightarrow} s'\}$ be the rate to move from state s to state s', and let $\mathbf{R}(s)(C) = \sum_{s' \in C} \mathbf{R}(s)(s')$ be the cumulative rate to reach a set of states $C \subseteq S$ from state s.

Definition 4. *An interactive Markov chain (IMC) is a tuple $(S, \mathsf{Act}, \rightarrow, \Rightarrow)$, consisting of a set of states S, a set of labels Act that may contain the non-observable action τ, transitions $\rightarrow \subseteq S \times \mathsf{Act} \times S$, and Markovian transitions $\Rightarrow \subseteq S \times \mathbb{R}^{>0} \times S$.*

An IMC basically combines the features of an LTS and a CTMC. One feature of IMCs is the *maximal progress assumption*. Internal interactive transitions, i.e. τ-transitions, can be assumed to take place immediately, while the probability that a Markovian transition executes immediately is zero. Therefore, we may remove all Markovian transitions from states that have outgoing τ-transitions: $s \xrightarrow{\tau}$ implies $\mathbf{R}(s)(S) = 0$. We call IMCs to which this operation has been applied *maximal-progress-cut* (mp-cut) IMCs.

For LTSs, strong and branching bisimulation are typically defined as follows:

Definition 5. *An equivalence relation $\equiv_S$ is a strong bisimulation on an LTS if for all states s, t, s' with $s \equiv_S t$ and for all $s \xrightarrow{a} s'$, there exists a state t' with $t \xrightarrow{a} t'$ and $s' \equiv_S t'$.*

Definition 6. *An equivalence relation $\equiv_B$ is a branching bisimulation on an LTS if for all states s, t, s' with $s \equiv_B t$ and for all $s \xrightarrow{a} s'$, either*

- *$a = \tau$ and $s' \equiv_B t$, or*
- *there exist states t', t'' with $t \xrightarrow{\tau^*} t' \xrightarrow{a} t''$ and $t \equiv_B t'$ and $s' \equiv_B t''$.*

For CTMCs, strong bisimulation is defined as follows:

Definition 7. *An equivalence relation $\equiv_S$ is a strong bisimulation on a CTMC if for all $(s,t) \in \equiv_S$ and for all classes $C \in S/\equiv_S$, $\boldsymbol{R}(s)(C) = \boldsymbol{R}(t)(C)$.*

For mp-cut IMCs, strong and branching bisimulation are defined as follows:

Definition 8. *An equivalence relation $\equiv_S$ is a strong bisimulation on an mp-cut IMC if for all $(s,t) \in \equiv_S$ and for all classes $C \in S/\equiv_S$*

- *$s \xrightarrow{a} s'$ for some $s' \in C$ implies $t \xrightarrow{a} t'$ for some $t' \in C$*
- *$\boldsymbol{R}(s)(C) = \boldsymbol{R}(t)(C)$*

Definition 9. *An equivalence relation $\equiv_B$ is a branching bisimulation on an mp-cut IMC if for all $(s,t) \in \equiv_B$ and for all classes $C \in S/\equiv_B$*

- *$s \xrightarrow{a} s'$ for some $s' \in C$ implies*
  - *$a = \tau$ and $(s,s') \in \equiv_B$, or*
  - *there exist states $t', t'' \in S$ with $t \xrightarrow{\tau^*} t' \xrightarrow{a} t''$ and $(t,t') \in \equiv_B$ and $t'' \in C$.*
- *$\boldsymbol{R}(s)(C) > 0$ implies*
  - *$\boldsymbol{R}(s)(C) = \boldsymbol{R}(t')(C)$ for some $t' \in S$ such that $t \xrightarrow{\tau^*} t' \not\xrightarrow{\tau}$ and $(t,t') \in \equiv_B$.*
- *$s \not\xrightarrow{\tau}$ implies $t \xrightarrow{\tau^*} t' \not\xrightarrow{\tau}$ for some t'*

2.3 Signature-based Bisimulation

Blom and Orzan [1] introduced a signature-based approach to compute the maximal bisimulation of an LTS, which was further developed into a symbolic method by Wimmer et al. [9]. Each state is characterized by a *signature*, which is the same for all equivalent states in a bisimulation. These signatures are used to refine a partition of the state space until a fixed point is reached, which is the maximal bisimulation.

In the literature, multiple signatures are sometimes used that together fully characterize states, for example based on the state labels, based on the rates of continous-time transitions, and based on the enabled interactive transitions. In the current paper, these multiple signatures are considered elements of a single signature that fully characterizes each state.

Definition 10. *A signature $\sigma(\pi)(s)$ is a tuple of functions $f_i(\pi)(s)$, that together characterize each state s with respect to a partition π.*

Two signatures $\sigma(\pi)(s)$ and $\sigma(\pi)(t)$ are equivalent, if and only if for all f_i, $f_i(\pi)(s) = f_i(\pi)(t)$.

The signatures of five bisimulations from Section 2.2 are known from the literature. For all actions $a \in \mathsf{Act}$ and equivalence classes $C \in \pi$, we define

- $\mathbf{T}(\pi)(s) = \{(a, C) \mid \exists s' \in C\colon s \xrightarrow{a} s'\}$
- $\mathbf{B}(\pi)(s) = \{(a, C) \mid \exists s' \in C\colon s \xrightarrow[\pi]{\tau^*} \xrightarrow{a} s' \wedge \neg(a = \tau \wedge s \in C)\}$
- $\mathbf{R}^s(\pi)(s) = C \mapsto \mathbf{R}(s)(C)$
- $\mathbf{R}^b(\pi)(s) = C \mapsto \max(\{\mathbf{R}(s')(C) \mid \exists s'\colon s \xrightarrow[\pi]{\tau^*} s' \not\xrightarrow{\tau}\})$

The five bisimulations are associated with the following signatures:

Strong bisimulation for an LTS	$(\mathbf{T})$	[9]
Branching bisimulation for an LTS	$(\mathbf{B})$	[9]
Strong bisimulation for a CTMC	$(\mathbf{R}^s)$	[7]
Strong bisimulation for an mp-cut IMC	$(\mathbf{T}, \mathbf{R}^s)$	[10]
Branching bisimulation for an mp-cut IMC	$(\mathbf{B}, \mathbf{R}^b, s \xrightarrow{\tau^*} \not\xrightarrow{\tau})$	[10]

Functions $\mathbf{T}$ and $\mathbf{B}$ assign to each state s all actions a and equivalence classes $C \in \pi$, such that state s can reach C by an action a either directly ($\mathbf{T}$) or via any number of inert τ-steps ($\mathbf{B}$). $\mathbf{R}^s$ equals $\mathbf{R}$ but with the domain restricted to the equivalence classes $C \in \pi$, and represents the cumulative rate with which state s can go to states in C. $\mathbf{R}^b$ equals $\mathbf{R}^s$ for states $s \not\xrightarrow{\tau}$, and takes the highest "reachable rate" for states with inert τ-transitions. In branching bisimulation for mp-cut IMCs, the "highest reachable rate" is by definition the rate that all states $s \not\xrightarrow{\tau}$ in C have. The element $s \xrightarrow{\tau^*} \not\xrightarrow{\tau}$ distinguishes time-convergent states from time-divergent states [10], and is independent of the partition.

For the bisimulations of Definitions 5–9, we state:

Lemma 2.3.1. *A partition π is a bisimulation, if and only if for all s and t that are equivalent in π, $\sigma(\pi)(s) = \sigma(\pi)(t)$.*

For the above definitions it is fairly straightforward to prove that they are equivalent to the classical definitions of bisimulation. See e.g. [1, 9] for the bisimulations on LTSs and [10] for the bisimulations on IMCs.

2.4 Partition refinement

The definition of signature-based partition refinement is as follows.

Definition 11 (Partition refinement with full signatures)**.**

$$\begin{aligned} \mathrm{sigref}(\pi, \sigma) &:= \{\{t \in S \mid \sigma(\pi)(s) = \sigma(\pi)(t)\} \mid s \in S\} \\ \pi^0 &:= \{S\} \\ \pi^{n+1} &:= \mathrm{sigref}(\pi^n, \sigma) \end{aligned}$$

The algorithm iteratively refines the initial coarsest partition $\{S\}$ according to the signatures of the states, until some fixed point $\pi^{n+1} = \pi^n$ is obtained. This fixed point is the maximal bisimulation for "monotone signatures":

Definition 12. *A signature is monotone if for all π, π' with $\pi' \sqsubseteq \pi$, whenever $\sigma(\pi')(s) = \sigma(\pi')(t)$, also $\sigma(\pi)(s) = \sigma(\pi)(t)$.*

For all monotone signatures, the sigref operator is monotone: $\pi \sqsubseteq \pi'$ implies $\text{sigref}(\pi, \sigma) \sqsubseteq \text{sigref}(\pi', \sigma)$. Hence, following Kleene's fixed point theorem, the procedure above reaches the greatest fixed point.

In Definition 11, the full signature is computed in every iteration. We propose to apply partition refinement using parts of the signature. By definition, $\sigma(\pi)(s) = \sigma(\pi)(t)$ if and only if for all parts $f_i(\pi)(s) = f_i(\pi)(t)$.

Definition 13 (Partition refinement with partial signatures)**.**

$$\begin{aligned} \text{sigref}(\pi, f_i) &:= \{\{t \in S \mid f_i(\pi)(s) = f_i(\pi)(t) \wedge s \equiv_\pi t\} \mid s \in S\} \\ \pi^0 &:= \{S\} \\ \pi^{n+1} &:= \text{sigref}(\pi^n, f_i) \qquad (\text{select } f_i \in \sigma) \end{aligned}$$

We always select some f_i that refines the partition π. A fixed point is reached only when no f_i refines the partition further: $\forall f_i \in \sigma\colon \text{sigref}(\pi^n, f_i) = \pi^n$. The extra clause $s \equiv_\pi t$ ensures that every application of sigref refines the partition.

Theorem 2.4.1. *If all parts f_i are monotone, Def. 13 yields the greatest fixed point.*

Proof. The procedure terminates since the chain is decreasing ($\pi^{n+1} \sqsubseteq \pi^n$), due to the added clause $s \equiv_\pi t$. We reach some fixed point π^n, since $\forall f_i \in \sigma\colon \text{sigref}(\pi^n, f_i) = \pi^n$ implies $\text{sigref}(\pi^n, \sigma) = \pi^n$. Finally, to prove that we get the *greatest* fixed point, assume there exists another fixed point $\xi = \text{sigref}(\xi, \sigma)$. Then also $\xi = \text{sigref}(\xi, f_i)$ for all i. We prove that $\xi \sqsubseteq \pi^n$ by induction on n. Initially, $\xi \sqsubseteq S = \pi^0$. Assume $\xi \sqsubseteq \pi^n$, then for the selected i, $\xi = \text{sigref}(\xi, f_i) \sqsubseteq \text{sigref}(\pi^n, f_i) = \pi^{n+1}$, using monotonicity of f_i. □

There are several advantages to this approach due to its flexibility. First, for any f_i that is independent of the partition, refinement with respect to that f_i only needs to be applied once. Furthermore, refinements can be applied according to different strategies. For instance, for the strong bisimulation of an mp-cut IMC, one could refine w.r.t. $\mathbf{T}$ until there is no more refinement, then w.r.t. $\mathbf{R}^s$ until there is no more refinement, then repeat until neither $\mathbf{T}$ nor $\mathbf{R}^s$ refines the partition. Finally, computing the full signature is the most memory-intensive operation in symbolic signature-based partition refinement. If the partial signatures are smaller than the full signature, then larger models can be minimised.

3 Implementation

This chapter describes the technical details of the implementation of signature-based partition refinement. It contains a condensed version of several sections from [6].

3.1 Decision diagram algorithms in Sylvan

In symbolic model checking, sets of states and transitions are represented by their characteristic function, rather than stored individually. With states described by N Boolean variables, a set $S \subseteq \mathbb{B}^N$ can be represented by its characteristic function $f : \mathbb{B}^N \to \mathbb{B}$, where $S = \{s \mid f(s)\}$. Binary decision diagrams (BDDs) are a concise and canonical representation of Boolean functions.

An (ordered) BDD is a directed acyclic graph with leaves 0 and 1. Each internal node has a variable label x_i and two outgoing edges labeled 0 and 1. Variables are encountered along each path according to a fixed variable ordering. Duplicate nodes and nodes with two identical outgoing edges are forbidden. It is well known that for a fixed variable ordering, every Boolean function is represented by a unique BDD.

In addition to BDDs with leaves 0 and 1, multi-terminal binary decision diagrams have been proposed with leaves other than 0 and 1, representing functions from the Boolean space $\mathbb{B}^N$ onto any finite set. For example, MTBDDs can have leaves representing integers (encoding $\mathbb{B}^N \to \mathbb{N}$), floating-point numbers (encoding $\mathbb{B}^N \to \mathbb{R}$) and rational numbers (encoding $\mathbb{B}^N \to \mathbb{Q}$). Partial functions are supported using a terminal leaf $\perp$.

Sylvan [5] implements parallelized operations on decision diagrams using parallel data structures and work-stealing. Work-stealing is a load balancing method for task-based parallelism. Recursive operations, such as most BDD operations, implicitly form a tree of tasks. Independent subtasks are stored in queues and idle processors steal tasks from the queues of busy processors.

Algorithm 1 describes the implementation of a generic binary operation F. BDD operations mainly consist of consulting an operation cache, performing some recursive step, and creating new BDD nodes using a unique table. The operation cache is required to reduce the time complexity of BDD operations from exponential to polynomial in the size of the BDDs. Sylvan uses a single shared unique table for all BDD nodes and a single shared operation cache for all operations. To obtain high performance in a multi-core environment, the datastructures for the BDD nodes and the operation cache must be highly scalable. Sylvan implements several non-blocking datastructures to enable good speedups.

To compute symbolic signature-based partition refinement, several basic operations must be supported by the BDD package (see also [9]). Sylvan implements basic operations

```
1 def apply(x, y, F):
2     if (x, y, F) ∈ cache :  return cache[(x, y, F)]          /* get from cache */
3     if x and y are terminals :  return F(x, y)              /* apply operator F */
4     v = topVar(x,y)
5     low ← apply(x_{v=0}, y_{v=0}, F)                        /* execute in parallel */
6     high ← apply(x_{v=1}, y_{v=1}, F)
7     result ← BDDnode(v, low, high)                          /* compute result */
8     cache[(x, y, F)] ← result                               /* put in cache */
9     return result
```

Algorithm 1: Generic algorithm that applies a binary operator F to BDDs x and y.

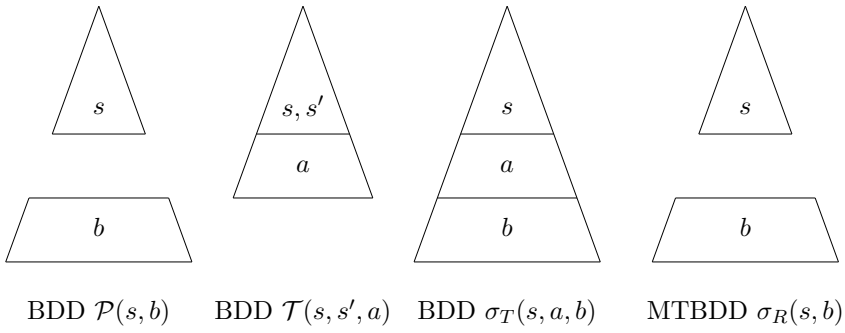

Figure 3.1: Schematic overview of the BDDs in signature refinement

such as $\wedge$ and `if-then-else`, and existential quantification $\exists$. Negation $\neg$ is performed in constant time using complement edges. To compute relational products of transition systems, there are operations `relnext` (to compute successors) and `relprev` (to compute predecessors and to concatenate relations), which combine the relational product with variable renaming. The operation `and_exists` computes the traditional relational product without variable renaming. Similar operations are also implemented for MTBDDs. Sylvan is designed to support custom BDD algorithms. We present two custom algorithms below.

3.2 Encoding of signature refinement

We implement symbolic signature refinement similar to [9]. Unlike [9], we do not refine the partition with respect to a single block, but with respect to all blocks simultaneously. We use a binary encoding with variables s for the current state, s' for the next state, a for the action labels and b for the blocks. We order BDD variables a and b after s and s', since this is required to efficiently replace signatures (a, b) by new block numbers b (see below). Variables s and s' are interleaved, which is common in the context of transition systems.

To perform symbolic bisimulation we represent a number of sets by their characteristic functions. See also Figure 3.1.

- A set of states is represented by a BDD $\mathcal{S}(s)$;
- Transitions are represented by a BDD $\mathcal{T}(s, s', a)$;
- Markovian transitions are represented by an MTBDD $\mathcal{R}(s, s')$, with leaves containing rational numbers ($\mathbb{Q}$);
- Signatures $\mathbf{T}$ and $\mathbf{B}$ are represented by a BDD $\sigma_T(s, a, b)$;
- Signatures $\mathbf{R}^s$ and $\mathbf{R}^b$ are represented by an MTBDD $\sigma_R(s, b)$.

In the literature, three methods have been proposed to represent π.

1. As an equivalence relation, using a BDD $\mathcal{E}(s, s') = 1$ iff $s \equiv_\pi s'$.
2. As a partition, by assigning each block a unique number, encoded with variables b, using a BDD $\mathcal{P}(s, b) = 1$ iff $s \in C_b$.
3. Using $k = \lceil \log_2 n \rceil$ BDDs $\mathcal{P}_0, \ldots, \mathcal{P}_{k-1}$ such that $\mathcal{P}_i(s) = 1$ iff $s \in C_b$ and the i^{th} bit of b is 1. This requires significant time to restore blocks for the refinement procedure, but can require less memory.

We choose to use method 2, since in practice the BDD of $\mathcal{P}(s, b)$ is smaller than the BDD of $\mathcal{E}(s, s')$. Using $\mathcal{P}(s, b)$ also has the advantage of straight-forward signature computation. The logarithmic representation is incompatible with our approach, since we refine all blocks simultaneously. Their approach involves restoring individual blocks to the $\mathcal{P}(s, b)$ representation, performing a refinement step, and compacting the result to the logarithmic representation. Restoring all blocks simply computes the full $\mathcal{P}(s, b)$.

We represent Markovian transitions using rational numbers, since they offer better precision than floating-point numbers. The manipulation of floating-point numbers typically introduces tiny rounding errors, resulting in different results of similar computations. This significantly affects bisimulation reduction, often resulting in finer partitions than the maximal bisimulation [7].

3.3 The `refine` algorithm

Partition refinement consists of two steps: computing the signatures and computing the next partition. Given the signatures σ_T and/or σ_R for the current partition π, the new partition can be computed as follows.

Since the chosen variable ordering has variables s, s' before a, b, each path in σ ends in a (MT)BDD representing the signature for the states encoded by that path. For σ_T, every path that assigns values to s ends in a BDD on a, b. For σ_R, every path that assigns values to s ends in a MTBDD on b with rational leaves.

```
1  def refine(σ, P):
2      if (σ, P, iter) ∈ cache :  return cache[(σ, P, iter)]
3      v = topVar(σ, P)
4      if v equals s_i for some i :
           # match paths on s in σ and P
5          low ← refine(σ_{s_i=0}, P_{s_i=0})
6          high ← refine(σ_{s_i=1}, P_{s_i=1})
7          result ← BDDnode(s_i, low, high)
8      else:
           # σ now encodes the state signature
           # P now encodes the previous block
9          B ← decodeBlock(P)
           # try to claim block B if still free
10         if blocks[B].sig = ⊥ : cas(blocks[B].sig, ⊥, σ)
11         if blocks[B].sig = σ : result ← P
12         else:
13             B ← search_or_insert(σ, B)
14             result ← encodeBlock(B)
15     cache[(σ, P, iter)] ← result
16     return result
```

Algorithm 2: `refine`, the (MT)BDD operation that assigns block numbers to signatures, given a signature σ and the previous partition $\mathcal{P}$.

Wimmer et al. [9] present a BDD operation `refine` that "replaces" these sub-(MT)BDDs by the BDD representing a unique block number for each distinct signature. The result is the BDD of the next partition. They use a global counter and a hash table to associate each signature with a unique block number. This algorithm has the disadvantage that block number assignments are unstable. There is no guarantee that a stable block has the same block number in the next iteration. This has implications for the computation of the new signatures. When the block number of a stable block changes, cached results of signature computation in earlier iterations cannot be reused.

We modify the `refine` algorithm to use the current partition to reuse the previous block number of each state. This also allows refining a partition with respect to only a part of the signature, as described in Section 2.3. The modification is applied such that it can be parallelized in Sylvan. See Algorithm 2.

The algorithm has two input parameters: the (MT)BDD σ which encodes the (partial) signature for the current partition, and the BDD $\mathcal{P}$ which encodes the current partition. The algorithm uses a global counter iter, which is the current iteration of partition refinement. This is necessary since the cached results of the previous iteration cannot be reused. It also uses and updates an array `blocks`, which contains the signature of each block in the new partition. This array is cleared between iterations of partition refinement.

The implementation is similar to other BDD operations, featuring the use of the operation cache (lines 2 and 15) and a recursion step for variables in s (lines 3–7), with the two recursive operations executed in parallel. `refine` simultaneously descends in σ and $\mathcal{P}$ (lines 5–6), matching the valuation of s_i in σ and $\mathcal{P}$. Block assignment happens at lines 9–14. We rely on the well-known atomic operation `compare_and_swap` (`cas`), which atomically compares and modifies a value in memory. This is necessary so the algorithm is still correct when parallelized. We use `cas` to claim a block number for the signature (line 10). If the block number is already used for a different signature, then this block is being refined and we call a method `search_or_insert` to assign a new block number.

Different implementations of `search_and_insert` are possible. We implemented a parallel hash table that uses a global counter for the next block number when inserting a new pair (σ, B), similar to [9]. An alternative implementation that performed better in our experiments integrates the `blocks` array with a skip list. A skip list is a probabilistic multi-level ordered linked list. See [4].

Our implementation of the skip list is restricted to at most 5 levels and supports only the `insert` operation. We use a short-lived local lock at the lowest level to insert buckets, and lock-free insertions using `cas` at higher levels. Furthermore, by restricting the number of blocks to at most 2^{31}, we only need 32 bytes for each bucket in the skip list:

```
struct { uint64_t sig; uint32_t prev_block; uint32_t next[5]; }
```

Each bucket in the skip list contains the pair (σ, B) (`sig` and `prev_block`) and the 31-bit indices of the next bucket at each level. The highest bit of `next[0]` is used as a lock, which is released when setting `next[0]` to a new value.

We implement `search_or_insert` as follows. We traverse the skip list until either a bucket with (σ, B) is found (and returned), or the bucket B' that would immediately precede a bucket with (σ, B). We use `cas` to set the highest bit of `next[0]` and lock bucket B'. This ensures that no other thread can insert (σ, B) (or any other pair directly preceded by bucket B') simultaneously. A new block number B'' is generated using a global counter `next_block`, which is increased atomically with `cas`. Bucket B'' is initialized and inserted into the skip list by updating `next[0]` of B' with B'', which also releases the lock on B'. Finally, the new bucket is inserted at a random number of higher levels using `cas`.

3.4 Computing inert transitions

To compute the set of inert τ-transitions for branching bisimulation, i.e., $s \xrightarrow[\pi]{\tau} s'$, or more generally, to compute any inert transition relation $\rightarrow \cap \equiv$ where $\equiv$ is the equivalence relation corresponding to π computed by $\mathcal{E}(s, s') = \exists b\colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$, the expression $\mathcal{T}(s, s') \wedge \exists b\colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$ must be computed. We compute $\rightarrow \cap \equiv$ directly using a custom BDD algorithm. The `inert` algorithm takes parameters $\mathcal{T}(s, s')$ ($\mathcal{T}$ may contain other variables ordered after s, s') and two copies of $\mathcal{P}(s, b)$: $\mathcal{P}^s$ and $\mathcal{P}^{s'}$. The algorithm matches $\mathcal{T}$ and $\mathcal{P}^s$ on valuations of variables s, and $\mathcal{T}$ and $\mathcal{P}^{s'}$ on valuations of variables

```
1  def inert(T, P^s, P^s'):
2      if (T, P^s, P^s') ∈ cache :  return cache[(T, P^s, P^s')]
       # find highest variable, interpreting s_i in P^s' as s'_i
3      v = topVar(T, P^s, P^s')
4      if v equals s_i for some i :
           # match s_i in T with P^s
5          low ← inert(T_{s_i=0}, P^s_{s_i=0}, P^s')
6          high ← inert(T_{s_i=1}, P^s_{s_i=1}, P^s')
7          result ← BDDnode(s_i, low, high)
8      elif v equals s'_i for some i :
           # match s'_i in T with s_i in P^s'
9          low ← inert(T_{s'_i=0}, P^s, P^s'_{s_i=0})
10         high ← inert(T_{s'_i=1}, P^s, P^s'_{s_i=1})
11         result ← BDDnode(s'_i, low, high)
12     else:
           # match the blocks P^s and P^s'
13         if P^s ≠ P^s' :  result ← False
14         else:  result ← T
15     cache[(T, P^s, P^s'] ← result
16     return result
```

Algorithm 3: Computes the inert transitions of a transition relation $\mathcal{T}$ according to the block assignments to current states ($\mathcal{P}^s$) and next states ($\mathcal{P}^{s'}$).

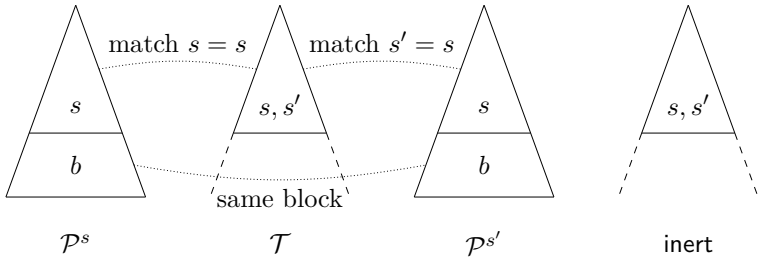

Figure 3.2: Schematic overview of the BDDs in the `inert` algorithm

s'. See Algorithm 3, and also Figure 3.2 for a schematic overview. When in the recursive call all valuations to s and s' have been matched, with $S_s, S_{s'} \subseteq S$ the sets of states represented by these valuations, then $\mathcal{T}$ is the set of actions that label the transitions between states in S_s and $S_{s'}$, $\mathcal{P}^s$ is the block that contains all S_s and $\mathcal{P}^{s'}$ is the block that contains all $S_{s'}$. Then if $\mathcal{P}^s \neq \mathcal{P}^{s'}$, the transitions are not inert and `inert` returns `False`, removing the transition from $\mathcal{T}$. Otherwise, $\mathcal{T}$ (which may still contain other variables ordered after s, s', such as action labels), is returned.

4 Usage

4.1 Compiling

To compile SigrefMC, the framework `CMake` is required.

```
> cmake .
> make
```

A successful compilation produces two files called `sigrefmc` and `sigrefmc_ht` which reside in the subdirectory `src`. The `sigrefmc_ht` program is identical to `sigrefmc`, except that it uses a hash table instead of a skip list in the `refine` algorithm. Type

```
> src/sigrefmc --help
```

or

```
> src/sigrefmc --usage
```

for a list of command-line options that the program expects.

4.2 Command-Line Parameters

The currently understood command-line parameters are:

⟨*filename*⟩
: Tells `sigrefmc` to look for the specification of the transition system to perform bisimulation minimisation on in the file ⟨*filename*⟩.

`-b` ⟨*bisimulation*⟩
: Sets the bisimulation type. With bisimulation type 1, branching bisimulation will be applied to LTS and IMC models. With bisimulation type 2, strong bisimulation will be applied to LTS and IMC models. This option is ignored for CTMC models, for which always strong bisimulation will be applied. For LTS and IMC models, the default bisimulation is branching bisimulation.

`-v` ⟨*verbosity*⟩
: Sets the verbosity level, valid arguments are 0–2, the default is 0.

`-w` ⟨*workers*⟩
: Sets the number of workers, with 0 for auto-detect, and 1 for sequential execution. The default is 0.

`-p` ⟨*filename*⟩
: If the `gperftools` library is installed, then this option allows CPU profiling to the file ⟨*filename*⟩.

`-l` ⟨*leaftype*⟩
: Sets the leaf type for Markovian transitions in CTMC and IMC models. Use `floating` for floating points, and `fraction` for the 32-bit-32-bit representation that Sylvan offers. The `gmp` option is not yet supported, due to a technicality in Sylvan's support for custom MTBDD leaves. The default leaf type is `floating`.

`-c` ⟨*closure*⟩
: Sets the algorithm for closure of a transition relation in branching bisimulation. Valid options are `fixpoint`, `squaring` and `recursive`. With the options `squaring` and `recursive`, the closure of the τ-transitions is precomputed before the refinement loop. The `squaring` method uses iterative squaring. The `recursive` method uses an obscure recursive-descent algorithm. The `fixpoint` method simply performs reachability to append all states that are backwards reachable via τ-transitions.

`-m`
: For LTSmin LTS models. If chosen, all transition relations are merged before the bisimulation minimisation. Currently, LTS bisimulation minimisation has rudimentary support for multiple transition relations.

`-q` ⟨*quotient*⟩
: Currently not implemented. Quotient extraction is implemented, but it is not yet part of the final distribution as it requires some code cleanup. Quotient extraction is a fairly straight-forward algorithm that given a LTS/CTMC/IMC and a partition, computes the new LTS/CTMC/IMC, either in a symbolic format (not recommended due to blowup) or in explicit format.

5 Extending the tool

This chapter briefly outlines how to extend SigrefMC.

5.1 Files

Filename	Description
`bisimulation.h`	Header file containing bisimulation function definitions.
`bisim_ctmc.cpp`	Implementation of (strong) bisimulation for CTMCs.
`bisim_imc.cpp`	Implementation of strong and branching bisimulation for IMCs.
`bisim_lts.cpp`	Implementation of strong and branching bisimulation for LTSs.
`blocks.h`	Header file for encode_blocks and decode_blocks.
`blocks.c`	Implementation of encode_blocks and decode_blocks.
`inert.h`	Header file of the `inert` algorithm.
`inert.c`	Implementation of the `inert` algorithm.
`refine.h`	Header file of the `refine` algorithm.
`refine_ht.c`	Implementation of `refine` using a hash table.
`refine_sl.c`	Implementation of `refine` using a skip list.
`sigref_util.h`	Header file for several utility functions.
`sigref_util.cpp`	Implementation of several utility functions.
`getrss.h`	Header file for computing memory usage of programs.
`getrss.c`	Implementation of computing memory usage of programs.
`parse_bdd.hpp`	Header file for models in the LTSmin file format.
`parse_bdd.cpp`	Parser for models in the LTSmin file format.
`parse_xml.hpp`	Header file for models in the Sigref XML file format.
`parse_xml.cpp`	Parser for models in the Sigref XML file format.
`systems.hpp`	Definitions of C++ interfaces for parsers.
`sigref.h`	Header file for sigrefmc main program.
`sigref.cpp`	Main sigrefmc file.

5.2 Extending the tool

To support other file formats, create a new parser (in `parse_x.cpp` and `parse_x.hpp`), and modify `sigref.cpp` to include and call the new parser.

For other notions of bisimulation, modify `bisimulation.h` by adding a new bisimulation task, modify `sigref.cpp` to actually call the task, and implement the bisimulation minimisation in a new `bisim_xxx.cpp` file. You can use `bisim_ctmc.cpp` as an example, as it is the simplest implementation of bisimulation minimisation.

To support other systems, modify `systems.hpp`, create a parser for the system, and create a bisimulation implementation for the system.

6 Known limitations

This version of SigrefMC still has a few limitations.

- Sylvan currently supports rational numbers using 32-bit integers. The effect of this limitation is that fractions that require more than 32 bits in their numerator or denominator cannot be represented yet. Rudimentary support for GMP objects exists, but is not ideal and requires an extra hash table between the GMP library and Sylvan. True support requires a modification of the hash table inside Sylvan.
- Quotient extraction has been implemented, but it is not yet part of the final distribution as it requires some code cleanup. Quotient extraction is a fairly straightforward algorithm that given a LTS/CTMC/IMC and a partition, computes the new LTS/CTMC/IMC, either in a symbolic format (not recommended due to blowup) or in explicit format.
- The maximum number of blocks is 2^{31}. This is partially due to memory limitations (for $2,147,483,648$ blocks, we already require 12 gigabytes to store all signature-partition pairs, ignoring additional overhead by the skip list/hash table) and due to the design of the skip list. The limitation can be lifted easily, but then SigrefMC requires twice as much memory.

Bibliography

[1] Stefan Blom and Simona Orzan. Distributed Branching Bisimulation Reduction of State Spaces. *ENTCS*, 89(1):99–113, 2003.

[2] Holger Hermanns and Joost-Pieter Katoen. The How and Why of Interactive Markov Chains. In *8th Intl. Symp. Formal Methods for Components and Objects*, volume 6286 of *LNCS*, pages 311–337. Springer, 2009.

[3] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco C. van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *TACAS 2015*, pages 692–707, 2015.

[4] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[5] Tom van Dijk and Jaco C. van de Pol. Sylvan: Multi-Core Decision Diagrams. In *TACAS 2015*, pages 677–691, 2015.

[6] Tom van Dijk and Jaco C. van de Pol. Multi-Core Symbolic Bisimulation Minimisation. In *TACAS 2016*, 2016. In submission.

[7] Ralf Wimmer and Bernd Becker. Correctness Issues of Symbolic Bisimulation Computation for Markov Chains. In *MMB&DFT*, volume 5987 of *LNCS*, pages 287–301. Springer, 2010.

[8] Ralf Wimmer, Marc Herbstritt, and Bernd Becker. Optimization techniques for BDD-based bisimulation computation. In *17th GLSVLSI*, pages 405–410. ACM, 2007.

[9] Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. Sigref- A Symbolic Bisimulation Tool Box. In *ATVA*, volume 4218 of *LNCS*, pages 477–492. Springer, 2006.

[10] Ralf Wimmer, Holger Hermanns, Marc Herbstritt, and Bernd Becker. Towards Symbolic Stochastic Aggregation. Technical report, SFB/TR 14 AVACS, 2007.