Verification of Cache Coherence Protocols wrt. Trace Filters

Parosh Aziz Abdulla*, Mohamed Faouzi Atig*, Zeinab Ganjei†, Ahmed Rezine† and Yunyun Zhu*
*Department of Information Technology
Uppsala University
Uppsala, Sweden
†Deptartment of Computer and Information Science
Linköping University
Linköping, Sweden

Abstract—**We address the problem of parameterized verification of cache coherence protocols for hardware accelerated transactional memories. In this setting, transactional memories leverage on the versioning capabilities of the underlying cache coherence protocol. The length of the transactions, their number, and the number of manipulated variables (i.e., cache lines) are parameters of the verification problem. Caches in such systems are finite-state automata communicating via broadcasts and shared variables. We augment our system with filters that restrict the set of possible executable traces according to existing conflict resolution policies. We show that the verification of coherence for parameterized cache protocols with filters can be reduced to systems with only a finite number of cache lines. For verification, we show how to account for the effect of the adopted filters in a symbolic backward reachability algorithm based on the framework of constrained monotonic abstraction. We have implemented our method and used it to verify transactional memory coherence protocols with respect to different conflict resolution policies.**

1. Introduction

The behavior of many types of systems can be described using one or more *parameters* such as the number of processes, or the number of variables that may be used in a given run of the system. Parameterized systems are ubiquitous and serve as natural models of mutual exclusion algorithms, bus protocols, distributed algorithms, telecommunication protocols, and cache coherence protocols. The goal of *parameterized verification* is to prove (or refute) the correctness of the system for all values of the parameters. For instance, in a cache coherence protocol, copies of a variable may exists in an arbitrary number of caches. It is then relevant to verify exclusive ownership of the cache line regardless of the number of caches in a particular session of the protocol. The state space of a such a system is infinite since we are dealing with an unbounded number of instances, namely one for each size.

Several techniques for the verification of parameterized systems have been developed during the last two decades [1], [2], [3], [4], [5]. One approach, related to this paper, is *monotonic abstraction* [6]. It defines an abstraction that allows to apply the framework of well quasi-ordered systems (wqo for short) [7] and based on backward reachability analysis in order to perform parameterized verification. Monotonic abstraction has been successfully applied to several non-trivial examples of mutual exclusion, leader election, and cache coherence protocols.

This paper addresses parameterized verification of transactional memory cache coherence protocols. Such protocols are not expected to guarantee coherence under arbitrary sequences of transitions. However, coherence should be guaranteed for all sequences that respect the transactional memory. Transactional memories usually make use of conflict tables in order to track read/write and write/write conflicts at a cache line granularity. Detected conflicts can be resolved according to different policies. For instance, in an eager policy, the conflict is resolved by aborting a transaction as soon as the conflict is detected. In a lazy policy, the resolution can wait until the commit before deciding on which transaction to abort.

Since the numbers of transactions, caches and cache lines are arbitrary, we need to consider systems that are parameterized in multiple dimensions. Furthermore, conflict policies can in general not be definable by finite-state automata since they quantify over the sets of threads and variables both of which are unbounded. Hence, parameterized verification of such systems is beyond the applicability of existing techniques. In this work, we present for the first time a method for automatic verification of cache coherence in the presence of transactional memories. We capture the conflict resolution mechanism, one for each policy, using so called *filters*, each of which is a set of forbidden "patterns". All traces of the system that do not match the patterns are allowed to occur. For instance, an eager conflict resolution will forbid traces where two different transactions continue running although a write/write conflict has been detected. Given a filter, we check reachability for the cache coherence protocol under the constraints imposed by the filter. For this we proceed in two steps. First, we give a small model theorem establishing that if coherence is violated then it is also violated using only a fixed small number of cache lines. Then we perform backward reachability analysis by modifying classical monotonic abstraction by accounting for information from the filters in order to exclude traces that are eliminated by the conflict resolution mechanism. We show that this is possible for the class of filters we use, and establish termination of the analysis.

We have implemented our approach and managed to show, for arbitrarily many caches on which arbitrary transactions are repeatedly run, that transactional memories such as FlexTM and DynTM with their proper cache coherence protocol extensions cannot violate coherence.

Related Work. To the best of our knowledge, this is the first work that considers parameterized verification of cache protocols in the presence of conflict policies.

Regular model checking [8], [9] performs parameterized verification by encoding the set of configurations using finite-state automata. The method has been augmented with techniques such as widening [10], [11], abstraction [12], and acceleration [13].

There are numerous techniques less general than regular model checking, but that are lighter and more dedicated to the problem of parameterized verification. The idea of *counter abstraction* is to keep track of the number of processes which satisfy a certain property [14], [15], [16], [17]. In general, counter abstraction is designed for systems with unstructured or clique architectures, but may be used for systems with other topologies too [18].

Several works reduce parameterized verification to the verification of finite-state models. Among these, the *invisible invariants* method [19], [20] and the work of [21] exploit cut-off properties to check invariants for mutual exclusion protocols.

Monotonic abstraction [6], [22], [23] combines regular model checking with abstraction in order to produce systems that have monotonic behaviors wrt. a wqo on the state-space.

Methods relying on dynamic detection of cutoff conditions are described in [1] and [24].

2. Motivating example

We use a hardware accelerated transactional memory in order to describe the different steps we use to establish coherence in the presence of execution filters.

An example of a hardware accelerated transactional memories. FlexTM [25] is a hardware accelerated transactional memory that orchestrates the execution of concurrent transactions by only allowing a subset of the possible traces (this subset includes the strictly serializable ones [26]). A finite but arbitrary number of caches participate in such executions. At most one transaction is run on each cache. Transactions can access arbitrarily many cache lines. The lines that do not fit in the caches are handled in software, and hence do not affect the cache coherence. We assume, to simplify the presentation, that the caches are large enough to hold all lines accessed by the transactions. Each transaction consists in some arbitrary sequence of read and write instructions on an arbitrary number of cache lines[1]. At any moment, transactions are either pending, committed or aborted. FlexTM tracks all transactions and might decide to abort a transaction based on some conflict resolution policy (e.g., lazy or eager). A transaction can therefore be aborted at any time, in which case a new arbitrary transaction might be started.

Like other hardware accelerated transactional memories [25], [27], FlexTM builds on the inherent versioning capabilities of the underlying cache coherence protocol. In the case of FlexTM, the MESI [28] protocol is extended. Schematically, FlexTM makes use of an extension of the MESI cache protocol, called TMESI [25] in order to maintain tentative versions of the accessed cache lines. In this protocol, a cache line can be in one of the states in $\{\mathtt{I}, \mathtt{S}, \mathtt{E}, \mathtt{M}, \mathtt{TI}, \mathtt{TMI}\}$. The four first states are the usual MESI states: Invalid, Shared, Exclusive and Modified. The last two ones are FlexTM additions. `TMI` and `TI` respectively correspond to a tentative written copy or to a read copy that is threatened by a tentative write by another transaction.

Table 1 depicts a run of two transactions reading and writing to cache lines l and l'. In the first transition (t_1), a read instruction from an invalid cache line (state `I`) results in an exclusive state `E`. We will say that the cache line “takes” the transition and changes its state from `I` to `E`. This transition is enabled if the state of the same line in all other caches is `I`. This appears in the transition because we forbid all other states using $f[\mathtt{S}, \mathtt{E}, \mathtt{M}, \mathtt{TI}, \mathtt{TMI}]$.

The second transition (t_2) is also a read. Here, a cache line that takes the transition moves from the invalid state to the shared one. This transition requires that at least another cache has the same line at a shared, exclusive, modified or threatened state (hence the $r[\mathtt{S}, \mathtt{E}, \mathtt{M}, \mathtt{TI}]$). In addition, the transition is not enabled if another cache associates the same line to `TMI` (hence the forbid $f[\mathtt{TMI}]$). If enabled, the transition t_2 performs a broadcast where it moves all exclusive or modified states to shared (hence the $b[(\mathtt{E}, \mathtt{S})(\mathtt{M}, \mathtt{S})]$). Except for the line firing the transition, a broadcast keeps all non mentioned lines unmodified. For instance, in this transition, the states of all `I` lines remain unchanged.

Transitions t_1, t_2, t_3, t_4 are said to be *horizontal transitions* because they focus on a particular cache line in all caches. More concretely, a cache line that “takes” such a transition changes state if there is at least another cache where the same line is at a state specified by the $r[\ ]$ part and if none of the other caches associates the same line to one of the states mentioned in $f[\ ]$. In this case, the line that takes the transition changes its state and moves the state of the same line in all other caches as described in the broadcast part $b[\ ]$.

Some transitions are said to be *vertical transitions* when they focus on all the lines of the same cache (as opposed to the same line in all caches). In FlexTM, commit and aborts correspond to vertical transitions. When a transaction is aborted (t_5), all lines in the cache running the transaction

1. Of course, transactions read and write variables, but as far as the cache protocol is concerned, these are tracked at a cache line granularity.

$$t_1 = \left(\texttt{I} \xrightarrow[read]{r[\ ]\ f[\texttt{S,E,M,TI,TMI}]\ b[\ ]} \texttt{E}\right)$$

$$t_2 = \left(\texttt{I} \xrightarrow[read]{r[\texttt{S,E,M,TI}]\ f[\texttt{TMI}]\ b[(\texttt{E,S})(\texttt{M,S})]} \texttt{S}\right)$$

$$t_3 = \left(\texttt{I} \xrightarrow[write]{r[\ ]\ f[\ ]\ b[(\texttt{S,I})(\texttt{E,I})(\texttt{M,I})]} \texttt{TMI}\right)$$

$$t_4 = \left(\texttt{I} \xrightarrow[read]{r[\texttt{TMI}]\ f[\ ]\ b[\ ]} \texttt{TI}\right)$$

$$t_5 = \left(\bullet \xrightarrow[abort]{b[(\texttt{TMI,I})(\texttt{TI,I})]} \bullet\right)$$

$$t_6 = \left(\bullet \xrightarrow[commit]{b[(\texttt{TMI,M})(\texttt{TI,I})]} \bullet\right)$$

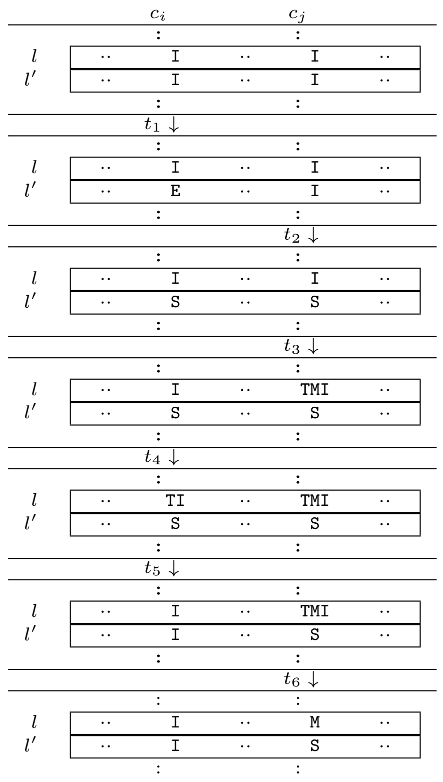

TABLE 1: A possible FlexTM [25] run is depicted to the right. At least two transactions are running on the caches c_i and c_j. In this execution, the c_i transaction tm_i reads line l', the c_j transaction tm_j reads line l' and writes line l, then tm_i reads l and is aborted by FlexTM before tm_j commits. This results in the TMESI transitions $t_1, \ldots t_6$ listed to the left.

that is to be aborted are invalidated. In a commit transition (t_6) all `TMI` lines are changed to `M`, and all `TI` lines are invalidated.

Coherence for transactional memory cache protocols. It turns out that cache coherence is violated if no restrictions are imposed on the sequences of horizontal (i.e., read and write) and vertical (i.e., abort and commit) transitions. For instance, assume that two transactions start running on caches c_1 and c_2 from a cache configuration where the line l is mapped to `I` in both caches (written $(\texttt{I}, \texttt{I})$). The sequence $(write, l, c_1)(write, l, c_2)(commit, c_1)(commit, c_2)$ where both transactions write the same l line and commit would result in executing transitions t_3, t_3, t_6 and t_6 by, respectively, caches c_1, c_2, c_1 and c_2. This sequence translates, for the l cache line, into the following states:

$$(\texttt{I}, \texttt{I}) \xrightarrow{write,l,c_1} (\texttt{TMI}, \texttt{I}) \xrightarrow{write,l,c_2} (\texttt{TMI}, \texttt{TMI})$$

$$(\texttt{TMI}, \texttt{TMI}) \xrightarrow{commit,c_1} (\texttt{M}, \texttt{TMI}) \xrightarrow{commit,c_2} (\texttt{M}, \texttt{M})$$

Coherence is violated in the last cache configuration. This is because the same cache line is mapped to the modified state `M` in two different caches. Intuitively, such configurations are bad because it is not clear which version to use if a transaction was to read a value as two possibly different versions coexist.

As it happens, FlexTM forbids such bad traces, based on some conflict resolution policy, by aborting transactions if certain conflicts arise.

In this work, we aim to show coherence in the presence of conflict resolution policies. Observe that the numbers of transactions, caches and cache lines are arbitrary. In other words, we are tackling coherence in the presence of conflict resolution policies for systems that are parameterized in the number of transactions, caches and cache lines.

Capturing transactional memory policies. FlexTM makes use of conflict tables in order to track read write and write write conflicts at a cache line granularity. Detected conflicts can be resolved according to different policies. For instance, in an eager policy, the conflict is resolved by aborting a transaction as soon as the conflict is detected. In a lazy policy, the resolution can wait until the commit before deciding on which transaction to abort.

We are interested in cache coherence in this work. We capture the conflict resolution mechanism using what we call *filters*. These consist in simple "patterns" that are going to be forbidden by the conflict resolution mechanisms. All traces that do not match the patterns are allowed. There will be simple patterns for each conflict resolution policy. Soundness requires that the patterns we use do not eliminate traces allowed by FlexTM. For instance, an eager conflict resolution will forbid traces where two different transactions continue running although a write write conflict has been detected.

Given such filters, we check reachability on the product of the cache coherence protocol and the filter and establish coherence for arbitrary transactions running on arbitrarily many caches and involving arbitrarily many cache lines.

3. Preliminaries

Let $\mathbb{N}$ denote the set of natural numbers. Given two natural numbers $i, j \in \mathbb{N}$, we use $[i, j]$ to denote the set $\{k \in \mathbb{N} \,|\, i \leq k \leq j\}$. For sets A and B, we use $f : A \mapsto B$ to denote that f is a function that maps any element from A to an element of B. Let $[A \mapsto B]$ denote the set of all functions from A to B. For $a \in A$ and $b \in B$, we use $f[a \leftarrow b]$ to denote the function f' where $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$. For a set $A' \subseteq A$, we use $f(A')$ to denote the set $\{f(a) \,|\, a \in A'\}$.

For a set Σ, we use Σ^* to denote the set of finite words over Σ. We use ϵ to denote the empty word. For a word $w \in \Sigma^*$, we use $|w|$ to denote its length (observe that $|\epsilon| = 0$). For $1 \leq i \leq |w|$, we use $w[i]$ to denote the letter at position i in w.

Let Θ be a subset of Σ. Given two words w and w', we define $w \sqsubseteq_\Theta w'$ to denote that there is a function $h : [1, |w|] \mapsto [1, |w'|]$ such that: (1) for every $i, j \in [1, |w|]$ such that $i < j$, $h(i) < h(j)$, (2) for every $i \in [1, |w|]$, $w'[h(i)] = w[i]$, and (3) $\{i \,|\, w'[i] \in \Theta\} \subseteq h([1, |w|])$.

4. Parameterized Cache Protocols with Filters

In this section, we introduce a formal model for parameterized cache protocols with filters, and define their coverability problem.

4.1. Parameterized Cache Protocols

A parameterized cache protocol consists of an arbitrary (but finite) number of caches. Each cache is a finite-state system manipulating an arbitrary (but finite) set of cache lines. Each cache can perform two kinds of operations: (1) *vertical* actions that only affect the states of the lines of one single cache, and (2) *horizontal* actions that affect the states of the same line but for different caches.

Formally, a parameterized cache protocol $\mathcal{P}$ is a tuple $(Q, A, \Delta, q_{\mathtt{init}})$ where Q is a finite set of states, A is a finite set of actions partitioned into two sets: the set of *vertical* actions $A_{\mathtt{ver}}$ and the set of *horizontal* actions $A_{\mathtt{hor}}$, $q_{\mathtt{init}} \in Q$ is the initial state, and Δ is a finite set of transitions. A transition can be of one of the following two forms: (1) $q \xrightarrow[a_{\mathtt{hor}}]{r[Q_1]\ f[Q_2]\ b\left[(q_1,q'_1),\ldots,(q_m,q'_m)\right]} q'$ or (2) $\bullet \xrightarrow[a_{\mathtt{ver}}]{b\left[(q_1,q'_1),\ldots,(q_m,q'_m)\right]} \bullet$ where: (i) q, q' in Q are cache line states, (ii) $a_{\mathtt{ver}}$ is a vertical action in $A_{\mathtt{ver}}$ and $a_{\mathtt{hor}}$ is a horizontal action in $A_{\mathtt{hor}}$, (iii) $Q_1 \subseteq Q$ is the set of *existentially required* states, (iv) $Q_2 \subseteq Q$ is the set of *universally forbidden* states, and (v) the sequence of pairs $(q_1, q'_1), \ldots, (q_m, q'_m) \in Q \times Q$, such that $q_i \neq q_j$ for all $i \neq j$, corresponds to a broadcast.

Let C be a finite set of caches and L be a finite set of cache lines. We write c to mean a cache in C and l to mean a cache line in L. A configuration ν over (C, L) is a mapping $\nu : C \mapsto [L \mapsto Q]$. We write $\nu_{(C,L)}$ to make the sets of caches and lines explicit. We use $\mathtt{cachesOf}(\nu_{(C,L)})$ and $\mathtt{linesOf}(\nu_{(C,L)})$ to respectively mean C and L. Let $\nu^{init}_{(C,L)}$ denote the configuration that associates $q_{\mathtt{init}}$ to all cache lines, i.e., $\nu^{\mathtt{init}}_{(C,L)}(c)(l) = q_{\mathtt{init}}$ for all $c \in C$ and $l \in L$.

Let $A^{\mathtt{ext}}_{\mathtt{ver}} = (A_{\mathtt{ver}} \times C)$ and $A^{\mathtt{ext}}_{\mathtt{hor}} = (A_{\mathtt{hor}} \times C \times L)$ respectively be the sets of extended vertical and horizontal actions over (C, L). Let $A^{\mathtt{ext}} = A^{\mathtt{ext}}_{\mathtt{ver}} \cup A^{\mathtt{ext}}_{\mathtt{hor}}$ be the set of extended actions. Given an extended action $\mathbf{a}$ of the form (a, c, l) or (a, c), we let $\mathtt{cacheOf}(\mathbf{a})$ mean the associated cache c.

Let ν and ν' be two configurations over (C, L). Let $\mathbf{a} \in A^{\mathtt{ext}}$ be an extended action. We use $\nu \xrightarrow{\mathbf{a}}_{(C,L)} \nu'$ to denote that one of the following cases holds:

Case 1: $\mathbf{a} = (a, c)$ for some vertical action $a \in A_{\mathtt{ver}}$ and cache $c \in C$, and there is a transition $t \in \Delta$ of the form $\bullet \xrightarrow[a_{\mathtt{ver}}]{b\left[(q_1,q'_1),\ldots,(q_m,q'_m)\right]} \bullet$ such that the following conditions are satisfied:

- For every cache line $l \in L$ such that $\nu(c)(l) = q_i$ for some $i \in [1, m]$, we have $\nu'(c)(l) = q'_i$. This corresponds to a transition resulting from a vertical action that changes the state of each cache line at q_i to q'_i.
- For every cache line $l \in L$ such that $\nu(c)(l) \notin \{q_i | i \in [1, m]\}$, we have $\nu'(c)(l) = \nu(c)(l)$, i.e., all the remaining cache lines keep their states.
- For every cache $c' \in C$ such that $c' \neq c$, we have $\nu'(c') = \nu(c')$, i.e., states of lines belonging to other caches remain unchanged.

Case 2: $\mathbf{a} = (a, c, l)$ for some $a \in A_{\mathtt{hor}}$, $c \in C$ and $l \in L$, and there are a transition $t \in \Delta$ of the form $q \xrightarrow[a]{r[Q_1]\ f[Q_2]\ b\left[(q_1,q'_1),\ldots,(q_m,q'_m)\right]} q'$, and a cache $c' \in C$, with $c \neq c'$, such that the following conditions are satisfied:

- $\nu(c)(l) = q$ and $\nu'(c) = \nu(c)[l \leftarrow q']$. The state of line l of the cache c changes from q to q'.
- $\nu(c')(l) \in Q_1$. This condition corresponds to the existential requirement. It states that the line l of at least another cache c' belongs to Q_1.
- $\nu(c'')(l) \notin Q_2$ for all $c'' \in C \backslash \{c, c'\}$. This condition corresponds to the universal requirement. It states that none of the lines l belonging to any cache other than c and c' is in Q_2.
- Any cache $c'' \in C \setminus \{c\}$ such that $\nu(c'')(l) = q_i$ for some $i \in [1, m]$, will change the state of l according to $\nu'(c'') = \nu(c'')[l \leftarrow q'_i]$. This corresponds to a horizontal broadcast where the state of the line l in any other cache is changed from q_i to q'_i.

- All other lines remain unchanged. In other words, for all caches $c'' \in C \setminus \{c\}$ with $\nu(c'')(l) \notin \{q_i | i \in [1, m]\}$ we have $\nu'(c'') = \nu(c'')$.

A trace $\sigma \in (A^{\text{ext}})^*$ over (C, L) is a sequence of *extended* actions. We use $\nu \xrightarrow{\sigma}_{(C,L)} \nu'$ to denote that one of the following two cases hold: (1) $\sigma = \epsilon$ and $\nu = \nu'$, or (2) there is a sequence of configurations $\nu_0, \ldots, \nu_n$ over (C, L) such that $\nu_0 = \nu$, $\nu_n = \nu'$, and for every $i \in [0, n-1]$, we have $\nu_i \xrightarrow{\mathbf{a}_i}_{(C,L)} \nu_{i+1}$ with $\sigma = \mathbf{a}_0 \mathbf{a}_1 \cdots \mathbf{a}_{n-1}$. In this case, we say that the configuration ν' is reachable from ν. Finally we say that the configuration ν' is reachable if it is reachable from $\nu^{\texttt{init}}_{(C,L)}$.

4.2. Filter Model

Let C be a finite set of caches and L be a finite set of cache lines. A pattern π over (C, L) is a finite sequence in $(A^{\text{ext}})^*$ of extended actions. We define a filter over (C, L) to be a finite set of *forbidden* patterns over (C, L).

Let C' be a set of caches and L' be a set of cache lines. Let σ be a trace over (C', L'). Let us assume that $\pi = \mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n$ and $\sigma = \mathbf{b}_1 \mathbf{b}_2 \cdots \mathbf{b}_m$. We say that the pattern π appears in σ (denoted by $\sigma \models \pi$) if and only if there are injective functions $\phi : C \mapsto C'$, $\psi : L \mapsto L'$ and $h : [1, n] \mapsto [1, m]$ such that:

- For every $i, j \in [1, n]$ such that $i < j$, $h(i) < h(j)$.
- For every $i \in [1, n]$, we have $\sigma[h(i)] = (a_i, \phi(c_i), \psi(l_i))$ if $\pi[i]$ is of the form (a_i, c_i, l_i) and $\sigma[h(i)] = (a_i, \phi(c_i))$ if $\pi[i]$ is of the form (a_i, c_i).
- For every $i \in [1, n]$ such that $\mathbf{a}_i$ is of the form (a_i, c_i, l_i) and there is an index j such that $i < j$ and $\texttt{cacheOf}(\mathbf{a}_j) = c_i$, we have $\mathbf{b}_k \notin (A_{\texttt{ver}} \times \phi(c_i))$ for all $h(i) < k < h(j')$ with j' is the minimal index such that $i < j'$ and $\texttt{cacheOf}(\mathbf{a}_{j'}) = c_i$.

A filter F over (C, L) is a finite set of *forbidden* patterns over (C, L). We say that a trace σ over (C', L') is *valid* with respect to a filter F if and only if $\sigma \not\models \pi$ for all $\pi \in F$.

4.3. Coverability Problem

Let ν and ν' be two configurations respectively over (C, L) and (C', L'). Let $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$ be two injective functions. We use $\nu \preceq_{(\phi,\psi)} \nu'$ to denote that for every cache $c \in C$ and every line $l \in L$, we have $\nu'(\phi(c))(\psi(l)) = \nu(c)(l)$. We use $\nu \preceq \nu'$ to denote that there are two injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$ such that $\nu \preceq_{(\phi,\psi)} \nu'$. Intuitively, this means that ν (modulo renaming of the caches and lines) is the restriction of ν' to the subsets of caches $\phi(C) \subseteq C'$ and lines $\psi(L) \subseteq L'$.

Let $\mathcal{P} = (Q, A, \Delta, q_{\texttt{init}})$ be a parameterized cache coherence protocol and F be a filter over a set of caches C and a set of lines L. The coverability problem for $\mathcal{P}$ with respect to the filter F and a configuration ν over (C, L), consists in checking whether there is a configuration ν' over (C', L'), with $\nu \preceq \nu'$, such that $\nu^{\texttt{init}}_{(C',L')} \xrightarrow{\sigma}_{(C',L')} \nu'$ for some trace σ over (C', L') with $\sigma \not\models \pi$ for any $\pi \in F$.

5. Small Model Theorem

In this section, we show that it is possible to restrict the analysis of the coverability problem for parameterized cache protocols to the subclass where only finite number of variables are used. Let $\mathcal{P} = (Q, A, \Delta, q_{\texttt{init}})$ be a parameterized cache protocol. We will first introduce some notations.

Notations. Let C and C' be two sets of caches and L and L' be two sets of cache lines. Given two injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$, we use $\sigma[\phi, \psi]$ to denote the trace σ' over (C', L') such that $|\sigma'| = |\sigma|$ and for every $i \in [1, |\sigma'|]$, $\sigma'[i] = (a, \phi(c), \psi(l))$ if $\sigma[i] = (a, c, l)$, and $\sigma'[i] = (a, \phi(c))$ if $\sigma[i] = (a, c)$. Given a trace τ over (C', L') We use $\tau[\phi^-, \psi^-]$ to denote the set of traces τ' over (C, L) such that $\tau'[\phi, \psi] \sqsubseteq_{(A_{\texttt{ver}} \times \phi(C))} \tau$. Intuitively, τ' corresponds to some trace obtained from τ by only deleting some horizontal actions and renaming caches and lines.

In the following, we will establish two closure properties of the considered cache protocols.

Closure property of the cache protocol. Our first property concerns the parameterized cache protocol. Intuitively, we show that if a configuration ν' is reachable and ν' is larger than a configuration ν (w.r.t. the ordering $\preceq$) then ν is also reachable.

Lemma 1. *Let C and C' be two sets of caches such that $|C| = |C'|$. Let L and L' be two sets of cache lines such that $|L| \le |L'|$. Let ν be a configuration over (C, L) and ν' be a configuration over (C', L'). If $\nu^{\texttt{init}}_{(C',L')} \xrightarrow{\sigma'}_{(C',L')} \nu'$ for some trace σ' over (C', L') and $\nu \preceq_{(\phi,\psi)} \nu'$ for some injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$, then $\nu^{\texttt{init}}_{(C,L)} \xrightarrow{\sigma}_{(C,L)} \nu$ with $\sigma \in \sigma'[\phi^-, \psi^-]$.*

Closure property of the filter. Our second property concerns the filter. We show that if a trace σ' is valid wrt. a filter then any trace which is an *extended-vertical-actions-preserving-subword* (modulo renaming of the caches) of σ' is also valid wrt. the filter.

Lemma 2. *Let C and C' be two sets of caches such that $|C| \le |C'|$. Let L and L' be two sets of cache lines such that $|L| \le |L'|$. Let $\phi : C \mapsto C'$ and $\psi : L \mapsto L'$ be two injective functions. Let σ' be a valid trace over (C', L') with respect to a given filter F. Then every trace $\sigma \in \sigma'[\phi^-, \psi^-]$ is valid with respect to the filter F.*

Bounding the number of cache lines. We are now ready to state our main theorem which is a consequence of Lemma 1 and Lemma 2. Intuitively, we will show that checking the coverability problem for parameterized cache protocols can be restricted to instances where the number of cache lines is bounded.

Theorem 3. *Let F be a filter over a set of caches C and a set of cache lines L. Let ν^{bad} be a configuration over (C, L). Let C' be a set of caches and L' be a set of cache lines. If $\nu^{\texttt{init}}_{(C',L')} \xrightarrow{\sigma'}_{(C',L')} \nu'$ for some valid trace σ' with respect to F and $\nu^{\mathsf{bad}} \preceq \nu'$, then there is a configuration ν over*

(C', L) such that $\nu^{\mathsf{bad}} \preceq \nu$ and $\nu^{\mathtt{init}}_{(C',L)} \xrightarrow{\sigma}_{(C',L)} \nu$ for some valid trace σ w.r.t. F.

As an immediate consequence of Theorem 3, we can restrict the coverability problem for parameterized cache protocols where the set of cache lines is restricted to L. More formally, we define the *restricted* coverability problem as follows: The *restricted* coverability problem for $\mathcal{P}$ wrt. a filter F and a configuration ν^{bad} over a set of caches C and a set of cache lines L, consists in checking whether there is a configuration ν over (C', L), such that: (1) $\nu^{\mathsf{bad}} \preceq_{(\phi,\psi)} \nu$ for some injective functions $\phi : C \mapsto C'$ and $\psi : L \mapsto L$ such that $\psi(l) = l$ for all $l \in L$, and (2) $\nu^{\mathtt{init}}_{(C',L)} \xrightarrow{\sigma}_{(C',L)} \nu$ for some trace σ over (C', L) with $\sigma \not\models \pi$ for any $\pi \in F$. As a corollary of Theorem 3, we have:

Corollary 4. *Let F be a filter over a set of caches C and a set of cache lines L. Let ν^{bad} be a configuration over (C, L). Then, the coverability problem for $\mathcal{P}$ wrt. F and ν^{bad} can be reduced to the* restricted *coverability problem for $\mathcal{P}$ wrt. F and ν^{bad}.*

As a consequence of Corollary 4, we will use from now on the term coverability problem to mean its restricted form.

6. Checking Trace Sensitive Coverability

Assume a cache protocol $\mathcal{P} = (Q, A, \Delta, q_{\mathtt{init}})$, a set of forbidden patterns F and a configuration ν^{bad} capturing some violation of cache coherence. Section 5 ensures that it is enough to check for the existence or absence of F-valid traces that cover ν^{bad} (i.e. violate coherence) on systems with the same number of cache lines as the number of lines in $\mathtt{linesOf}(\nu^{\mathsf{bad}})$. Observe that the length of the transactions and the number of caches (i.e. of concurrent transactions) is still arbitrary.

In fact, state reachability for any given two counters Minsky machine can be encoded using a parameterized cache protocol with a single cache line. The idea is to capture the value of each counter using the number of caches having their line at some cache state. Tests for zero are captured with the forbidding part of horizontal transitions. Coverability is therefore undecidable even in the case of a single cache line per cache. We use over-approximated systems where the analysis is exact and terminates and we refine the approximation in case of false positives.

The tail recursive procedure checkCov is used to check coherence. It takes three arguments. A cache protocol $\mathcal{P}$, a filter F, a configuration ν^{bad} and a preorder $\trianglelefteq$ on pairs of configurations and traces. All manipulated configurations have $L = \mathtt{linesOf}(\nu^{\mathsf{bad}})$ lines. The procedure is invoked with $checkCov(P, F, \nu^{\mathsf{bad}}, \trianglelefteq_0)$ where $(\nu, \sigma) \trianglelefteq_0 (\nu', \sigma')$ iff there are renamings $\phi : \mathtt{cachesOf}(\nu) \mapsto \mathtt{cachesOf}(\nu')$ and $\psi : L \mapsto L$ such that $\nu \preceq_{(\phi,\psi)} \nu'$ and $\mathtt{truncate}_F(\sigma[\phi, \psi]) \sqsubseteq_{(A_{\mathsf{ver}} \times \phi(\mathtt{cachesOf}(\nu)))} \mathtt{truncate}_F(\sigma')$ (see 3, 4.3). The result of $\mathtt{truncate}_F(\sigma)$ is defined to be the longest prefix of σ that does not contain more vertical instructions than the number of vertical instructions appearing in any of the patterns in F. Observe that such a prefix can be arbitrarily long. The idea is that the traces will be checked against the filter incrementally while being constructed, so we only need to check the "freshest" part of it. Intuitively, $(\nu, \sigma) \trianglelefteq_0 (\nu', \sigma')$ holds if, up to eliminating some caches, ν and ν' coincide and the $\mathtt{truncate}_F(\sigma)$ sequence can be obtained from the $\mathtt{truncate}_F(\sigma')$ sequence by deleting the same caches and some horizontal (but not vertical) instructions. The idea is that vertical instructions are not deleted from larger traces because this would not preserve F-validity. However, considering whole traces without applying $\mathtt{truncate}_F(\sigma)$ would result in a non wqo $\trianglelefteq_0$ for which there is no guarantee of termination even without refinement [7].

Lemma 5. *The preorder $\trianglelefteq_0$ is a wqo on $\{(\nu_{(C,L)}, \mathtt{truncate}_F(\sigma)) | \sigma \in ((A_{\mathsf{ver}} \times C) \cup (A_{\mathsf{hor}} \times C \times L))^*\}$.*

Procedure checkCov checks whether an F-valid trace σ can cover ν^{bad}. The procedure tracks pairs of the form (ν, σ), where ν is a configuration and σ is a trace. Intuitively, such a pair denotes all pairs (ν', σ') that are larger wrt. the current ordering, i.e. an upward closed set wrt. the current ordering $\trianglelefteq$. The procedure is a classical working list algorithm that maintains two sets of pairs, namely the working set W of pairs that have not been treated yet, and the visited set V of pairs that have been treated. The union of the two sets is minimal in the sense that one cannot find a pair of $\trianglelefteq$-related pairs. Given a pair in W (i.e., that has not been treated yet), the procedure computes the predecessor image wrt. each action that would not violate, given the trace in the pair, the filter F. For this reason, the trace σ that lead from ν^{bad} to the current configuration ν is maintained in each pair. Notice that the same configuration ν can participate in two $\trianglelefteq$-unrelated pairs (ν, σ) and (ν, σ'). The procedure

```
   Input: A protocol $\mathcal{P} = (Q, A, \Delta, q_{\mathtt{init}})$, a filter $F$, a bad configuration
          $\nu^{\mathsf{bad}}$ and a wqo $\trianglelefteq$ on pairs of configurations and $(A^{\mathtt{ext}})^*$
   Output: uncoverable or an $F$-valid trace covering $\nu^{\mathsf{bad}}$
1  W, V := $\{(\nu^{\mathsf{bad}}, \epsilon)\}, \{\}$;
2  while W is not empty do
3      remove a $(\nu, \sigma)$ from W and add it to V;
4      if $(\nu = \nu^{\mathtt{init}}_{\mathtt{cachesOf}(\nu),\mathtt{linesOf}(\nu)})$ then
5          if $\sigma$ is possible in $\mathcal{P}$ then return $\sigma$;
6          else return $checkCov(\mathcal{P}, F, \nu^{\mathsf{bad}}, \mathtt{strengthen}(\trianglelefteq, \sigma))$;
7      foreach $c \in \mathtt{cachesOf}(\nu) \cup \mathtt{newCache}(\mathtt{cachesOf}(\nu))$ do
8          $\Sigma := \{\}$;
9          foreach $a \in A_{hor}$ and $l \in \mathtt{linesOf}(\nu^{\mathsf{bad}})$ do add $(a, c, l)$ to $\Sigma$;
10         foreach $a \in A_{ver}$ do add $(a, c)$ to $\Sigma$ ;
11         foreach $\mathbf{a} \in \Sigma$ do
12             $\sigma' := \mathbf{a}\sigma$;
13             if $\sigma' \models \pi$ for some $\pi \in F$ then continue;
14             $\Gamma := \mathtt{minOf}_{\trianglelefteq}(\mathtt{preOf}(\mathbf{a}, \mathtt{upOf}_{\trianglelefteq}(\nu)))$;
15             foreach $\nu' \in \Gamma$ do
16                 if $(\nu'', \sigma'') \not\trianglelefteq (\nu', \sigma')$ for each $(\nu'', \sigma'') \in$ W $\cup$ V then
17                     remove from W $\cup$ V each $(\nu'', \sigma'')$ s.t. $(\nu', \sigma') \trianglelefteq (\nu'', \sigma'')$;
18                     add $(\nu', \sigma')$ to W;
19 return uncoverable
```

Procedure checkCov($\mathcal{P}, F, \nu^{\mathsf{bad}}, \trianglelefteq$)

makes use of the following operations:

1) at line 6, $\mathtt{strengthen}(\trianglelefteq, \sigma)$ is invoked in case the obtained trace σ is a false positive due to the application of the upward closure. It returns

a stronger ordering $\unlhd'$. The new ordering can be chosen to be a wqo in case $\unlhd$ is a wqo [29], [30].

2) at line 7, $\texttt{newCache}(C)$ returns a singleton c that is not in the set C of caches (i.e., $c \notin C$).
3) at line 14, $\texttt{upOf}_{\unlhd}(\nu)$ is the upward closure of ν wrt. the current ordering $\unlhd$.
4) at line 14, $\texttt{preOf}(\mathbf{a}, \Gamma)$ returns a representation of the (possibly infinite) set of configurations that can reach the upward set of configurations Γ in one step with the action $\mathbf{a}$.
5) at line 14, $\texttt{minOf}_{\unlhd}(\Gamma)$ returns a finite set of configurations that are pairwise $\unlhd$ unrelated and such that each element in Γ is larger than some of them.

Lemma 6. *The operations 1-5 are effectively computable.*

Assume each $\unlhd$ is a wqo and the operations are as stated above. Termination of each non recursive call to checkCov is obtained using a wqo argument. Intuitively each call to checkCov terminates and results in `uncoverable`, an F-valid trace, or in another call to checkCov with a stronger ordering. Indeed, an infinite execution that involves only a finite number of recursive calls would mean that there is a call where `W` never gets empty. This means that we keep on finding new pairs that cannot be eliminated by the elements in $\texttt{W} \cup \texttt{V}$ in lines 15-18. This infinite sequence of new elements contradicts that $\unlhd$ is a wqo.

Lemma 7. *Each infinite execution of checkCov contains an infinite number of recursive calls where each call is made with a preorder that is stronger than the orderings of the previous calls.*

Restriction to pairs corresponding to F-valid executions is obtained because lines 12-13, together with the fact that $\unlhd$ is stronger than $\unlhd_0$, ensure we discard actions and pairs that violate the filter. Soundness is guaranteed by the fact that line 14 computes an over-approximation of the predecessor configurations, that we consider all actions and that we eliminate pairs only if they denote less configurations and stronger traces. Returned traces are valid by construction.

Theorem 8. *If checkCov returns* `uncoverable`*, then none of the F-valid executions from $\nu^{\texttt{init}}$ cover $\nu^{\texttt{bad}}$. If it returns an F-valid trace σ, then $\nu^{\texttt{bad}}$ is coverable using σ.*

7. Experimental Results

We have implemented our techniques from Section 6 as an extension of the tool ZAAMA [30]. ZAAMA implements constrained monotonic abstraction [29]. The tool can address the parameterized verification problem for cache coherence protocols (without any restriction on the input sequence of traces). The input of our prototype includes the description of the parameterized cache protocol, the set bad configurations and the filter.

We have applied our prototype to a number of different cache coherence protocols and filters. In fact, we have considered two cache protocols: The TMESI protocol [25] and the UTCP protocol [27]. Both of them are adaptations of the well-known MESI protocol [28] to the case of transactional memories. TMESI is used in the hardware accelerated transactional memory FlexTM [25], while UTCP in the hybrid transactional memory DynTM [27].

These hardware accelerated transactional memories come with conflict resolution policies describing the set of forbidden traces. We model these policies using our filter models. FlexTM admits two conflict resolution policies which are the lazy and eager policies. In the lazy policy, the resolution can wait until the commit before deciding on which transaction to abort. While in the eager policy the conflict is resolved by aborting a transaction as soon as the conflict is detected. Therefore, FlexTM can be run with different modes. On the other hand, DynTM allows the eager and lazy modes to execute simultaneously. Furthermore, we have also defined a new filter for the lazy execution mode of FlexTM which allows the transactions whose read instructions precede all the conflicting writing instructions to survive when a conflicting transaction commits. For instance, the transaction running on c_1 would survive in $(read, l, c_1)(write, l, c_2)(commit, c_2)(commit, c_1)$. This behavior does not cause incoherent states and still satisfies the strict serializability definition [31]. We have also considered the filter allowing only strict serializable traces [26], [31].

The results of our analyses can be seen in Table 2. Our results show that TMESI (resp. UTCP) cannot violate coherence when run together with its proper filters, namely *lazy* FlexTM or *eager* FlexTM (resp. *eager* & *lazy* DynTM). To the best of our knowledge, this is the first time that coherence of such hardware accelerated transactional memories is proven automatically. Our results show that coherence is still preserved when TMESI is run together with the *new lazy* filter in spite of the fact that it allows for more traces than the ones allowed by the *lazy* FlexTM. Finally, our results show that both TMESI and UTCP become incoherent when considering only strict serializable traces.

All experiments were performed on a 2.9 Ghz Intel Core i7 with 8GB of RAM.

8. Conclusion

In this paper, we have addressed for the first time the parameterized verification of cache coherence protocols in the presence of transactional memories. We have first proposed a formal model for this class of systems in order to capture behaviours of parameterized cache coherence protocols as restricted by filters to capture transactional memories conflict resolution policies. Our first contribution was a small model theorem allowing us to restrict the analysis of such systems to only a fixed number of cache lines. Our second contribution was an non-trivial extension of the classical framework of monotonic abstraction in order to exclude the traces that are not allowed by our filter. Finally, we have implemented a prototype that is able to successfully establish or refute coherence for several challenging examples.

Cache protocol (filter)	#rules	# bad states	Reachable (Y/N)	Execution time
TMESI (eager FlexTM)	92	36	N	48.7s
TMESI (lazy FlexTM)	48	34	N	12.7s
UTCP (eager & lazy DynTM)	128	137	N	236.8s
UTCP (serial. filter)	70	47	Y, bad state (M, M)	117.3s
TMESI (new lazy filter)	47	34	N	13.5s
TMESI (serial. filter)	42	38	Y, bad state (M, M)	35.8s

TABLE 2: Experimental Results. The columns "#rules" and "# bad" states give the number of rules and the number of bad states used to model the cache coherence protocols, respectively. A "N" in the column "Reachable (Y/N)" means that the parameterized cache protocol with filter is coherent. A "Y" in the column "Reachable (Y/N)" means that the parameterized cache protocol with filter is not coherent and we provide the first reachable bad state. Finally, the column "Execution time" gives the running time in seconds.

A direction for future work is to address the problem of automatically synthesizing filters in order to ensure the coherence of a given cache protocol.

References

[1] A. Kaiser, D. Kroening, and T. Wahl, "Dynamic cutoff detection in parameterized concurrent programs," in *CAV'10*, ser. LNCS, vol. 6174. Springer, 2010, pp. 645–659.

[2] P. Liu and T. Wahl, "Infinite-state backward exploration of boolean broadcast programs," in *FMCAD*, 2014.

[3] E. M. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parameterized verification," in *VMCAI'06*, ser. LNCS, vol. 3855. Springer, 2006, pp. 126–141.

[4] D. Sethi, M. Talupur, and S. Malik, "Using flow specifications of parameterized cache coherence protocols for verifying deadlock freedom," in *ATVA*, ser. LNCS, vol. 8837, 2014.

[5] K. L. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *CHARME*, ser. Lecture Notes in Computer Science, vol. 2144, 2001.

[6] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine, "Regular model checking without transducers (on efficient verification of parameterized systems)," in *TACAS'07*, ser. LNCS, vol. 4424. Springer, 2007, pp. 721–736.

[7] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *LICS'96*, 1996, pp. 313–321.

[8] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar, "Symbolic model checking with rich assertional languages," *Theor. Comput. Sci.*, vol. 256, no. 1-2, pp. 93–112, 2001.

[9] D. Dams, Y. Lakhnech, and M. Steffen, "Iterating transducers," in *CAV'01*, ser. LNCS, vol. 2102. Springer, 2001.

[10] B. Boigelot, A. Legay, and P. Wolper, "Iterating transducers in the large," in *CAV'03*, ser. LNCS, vol. 2725. Springer, 2003, pp. 223–235.

[11] T. Touili, "Regular Model Checking using Widening Techniques," *ENTCS*, vol. 50, no. 4, 2001, proc. of VEPAS'01.

[12] A. Bouajjani, P. Habermehl, and T. Vojnar, "Abstract regular model checking," in *CAV'04*, ser. LNCS, vol. 3114. Springer, 2004, pp. 372–386.

[13] P. A. Abdulla, A. Legay, J. d'Orso, and A. Rezine, "Simulation-based iteration of tree transducers," in *TACAS'05*, ser. Lecture Notes in Computer Science, vol. 3440. Springer, 2005, pp. 30–44.

[14] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, no. 3, pp. 675–735, 1992.

[15] G. Delzanno, "Automatic verification of cache coherence protocols," in *CAV'00*, ser. LNCS, Emerson and Sistla, Eds., vol. 1855. Springer, 2000, pp. 53–68.

[16] ——, "Verification of consistency protocols via infinite-state symbolic model checking," in *FORTE'00*, ser. IFIP Conference Proceedings, vol. 183. Kluwer, 2000, pp. 171–186.

[17] A. Pnueli, J. Xu, and L. Zuck, "Liveness with (0,1,infinity)-counter abstraction," in *CAV'02*, ser. LNCS, vol. 2404. Springer, 2002.

[18] P. Ganty and A. Rezine, "Ordered counter-abstraction," in *Language and Automata Theory and Applications*, ser. LNCS. Springer International Publishing, 2014, vol. 8370, pp. 396–408.

[19] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions," in *CAV'01*, ser. LNCS, vol. 2102. Springer, 2001, pp. 221–234.

[20] A. Pnueli, S. Ruah, and L. D. Zuck, "Automatic deductive verification with invisible invariants," in *TACAS'01*, ser. LNCS, vol. 2031. Springer, 2001, pp. 82–97.

[21] K. S. Namjoshi, "Symmetry and completeness in the analysis of parameterized systems," in *VMCAI'07*, ser. LNCS, vol. 4349. Springer, 2007, pp. 299–313.

[22] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine, "Handling parameterized systems with non-atomic global conditions," in *VMCAI'08*, ser. LNCS, vol. 4905. Springer, 2008, pp. 22–36.

[23] N. Yonesaki and T. Katayama, "Functional specification of synchronized processes based on modal logic," in *IEEE 6th International Conference on Software Engineering*, 1982, pp. 208–217.

[24] P. A. Abdulla, F. Haziza, and L. Holík, "All for the price of few (parameterized verification through view abstraction)," in *VMCAI*, ser. LNCS, vol. 7737, 2013, pp. 476–495.

[25] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *ISCA'08*. IEEE Computer Society, 2008, pp. 139–150.

[26] P. A. Abdulla, S. Dwarkadas, A. Rezine, A. Shriraman, and Y. Zhu, "Verifying safety and liveness for the flextm hybrid transactional memory," in *DATE 13*. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 785–790.

[27] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *MICRO 10*. IEEE Computer Society, 2010, pp. 27–38.

[28] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *ISCA 84*. ACM, 1984, pp. 348–354.

[29] P. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine, "Constrained monotonic abstraction: A cegar for parameterized verification," in *CONCUR 2010*, ser. LNCS. Springer Berlin Heidelberg, 2010, vol. 6269, pp. 86–101.

[30] Z. Ganjei, A. Rezine, P. Eles, and Z. Peng, "Abstracting and counting synchronizing processes," in *VMCAI 05*, ser. LNCS, vol. 8931. Springer, 2015, pp. 227–244.

[31] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.