Technical Report: Multislicing

Vinit A. Ogale and Vijay K. Garg *
PDS Laboratory,
Dept. of Electrical and Computer Engineering,
University of Texas at Austin, TX 78712
{ogale,garg}@ece.utexas.edu

Abstract

In this paper, we examine the problem of detecting nested temporal predicates in the execution trace of a distributed program. We present a technique that allows efficient detection of a reasonably large class of predicates that we call the Multislicing Temporal Logic or MTL. Examples of valid MTL predicates are nested temporal predicates based on local variables with arbitrary negations, disjunctions, conjunctions and the possibly (EF or $\Diamond$) and invariant(AG or $\Box$) temporal operators. We do not know of any other technique that detects a similar class of predicates with a time complexity that is polynomial in the number of processes and events in the system. We introduce the concept of a *multislice* of an execution trace and use it for detecting predicates efficiently. A multislice is a set of traces (slices) that exactly represents all the global states which satisfy the predicate. We present an algorithm to compute the multislice of a computation and prove that its time complexity is polynomial with respect to the number of processes and events in the trace although it is not polynomial in the size of the formula. We compare the implementation of our technique with existing tools like POTA and Spin model checker by verifying traces generated by a Java implementation of the distributed dining philosophers algorithm.

1 Introduction

In a distributed system, it is of fundamental importance to determine whether a program executed correctly. The correctness properties or the predicates, which enable us to formally define a correct execution, often have temporal implications.

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, CNS-0509024, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

Hence, it is essential to have efficient techniques which can detect temporal predicates.

An execution trace, or a distributed computation can either be modeled as a total order, or, as a partial order on the set of events in the computation. Representing the computation as a total order could mask some of the bugs in other possible consistent interleavings. A partial order, in contrast, captures all the possible causally consistent interleavings. In this paper we use a partial order representation based on Lamport's *happened before* relation [14].

The drawback of using a partial order model is that the number of global states of the computation is exponential in the number of processes in the system. This makes predicate detection a hard problem in general [6, 25]. A number of strategies like symbolic representation of states and partial order reduction have been explored to tackle the state explosion problem [4, 10, 17, 21, 24, 26, 27]

In this paper, we present an algorithm for efficiently detecting all the temporal predicates that can be expressed in what we call the Multislicing Temporal Logic or MTL. An example of a valid MTL predicate would be a property based on local predicates and arbitrarily placed negations, disjunctions and conjunctions along with the possibly($\Diamond$) and invariant($\Box$) temporal operators (same as the EF and AG operators defined in [3]). Note that using a model checking based approach to detect predicates is inefficient and requires exponential time in the worst case.

Our approach extends a technique called *computation slicing* introduced in [7] which allows us to discard a large number of global states and still correctly detect the desired predicate. Informally, a *slice* with respect to a predicate is the smallest trace containing all the global states (cuts) in which the predicate holds. A slice is said to be *lean* when it contains *exactly* those global states that satisfy the predicate. Predicate detection using computation slicing traditionally involves computing the slice with respect to the predicate and examining the slice [22]. The drawback of this approach is that if the computed slice is not lean, then we are forced to examine each global state in the slice. This may require exponential time with respect to the number of processes in the system.

In this paper we suggest a new approach, *multislicing*, to address this issue. Instead of computing a single slice, which may not be lean, we compute a set of slices that are always lean. If the size of the set is small then is possible to efficiently detect the predicate. We show that it is possible to construct such a set in polynomial time with respect to the number of processes and events in the system if the predicate can be expressed in MTL. Note that the size of the computed multislice is not polynomial in the predicate size. We believe that many formulas used in practice are small in size and therefore our algorithms are feasible and applicable.

To summarize, this paper makes the following contributions:

- We introduce the idea of multislicing, i.e., maintaining a set of lean slices instead of a single, possibly non-lean, slice. We present an efficient algorithm to compute the multislice of a computation for MTL predicates, which lets us detect the predicates in $O(k.n^{2^k}.|E|)$ time where k is the number of operators in the predicate, E is the set of events in the computation and n is the number of processes. We do not know of any other algorithm that allows detection of nested temporal predicates containing disjunctions or negations in time that is polynomial in n and $|E|$.

- We compare our implementation of the multislicing algorithm with existing approaches by using Spin, POTA and multislicing to detect predicates in distributed programs. We have implemented a multislicing toolkit that can analyze traces generated by distributed Java programs [20].

  The remainder of the paper is organized as follows: Sections 2 and 3 discuss related work, a summary of slicing along with the model and notation used in this paper. Section 4 introduces the concept of multislicing and presents the main algorithms. In Section 5 we prove that the multislicing is efficient even when the predicate contains negations and disjunctions.

2 Related Work

A number of approaches for checking computations using temporal logic have been published. Temporal Rover [2], MaC [15] and JPaX [12] are some of the available tools. Many of the tools are based on total ordering of events and hence cannot be directly compared to our approach. These tools can miss potential bugs which would be detected by partial order representations. JMPaX [13] is based on a partial order model and supports temporal properties but its time complexity is exponential in the number of processes in the computation.

Another available option to verify computation traces is to use a model checking tool like Spin [8,9]. The computation trace needs to be converted to the Spin input computation and verification takes exponential time in the number of processes.

POTA [23] is another partial order based tool which uses computational slicing to detect predicates. POTA guarantees polynomial time complexity only if the predicate can be expressed in a subset of CTL [3] called *Regular CTL* or RCTL defined in [22]. Disjunctions and negations are not allowed in RCTL. If POTA is used with a logic that allows disjunctions or negations (like MTL), it uses a model checking algorithm to explore the reduced state space. Hence the asymptotic time complexity using POTA is exponential in the number of processes when the predicate contains disjunctions. Table 1 compares the time complexities of Spin,

Technique	Spin	POTA	Multislicing
RCTL	exponential in n	polynomial in n	polynomial in n
MTL	exponential in n	exponential in n	polynomial in n

Table 1: Time complexities (n = number of processes)

POTA and multislicing.

3 Model and Notation

3.1 System model and representation

We assume a loosely coupled, message-passing, asynchronous system model. A distributed program consists of n sequential programs $P_1, P_2, \ldots, P_n$. A computation is a single execution of such a program. A distributed computation ($\langle E, \rightarrow \rangle$) is modeled as a partial order on the set of events E, based on the happened before relation ($\rightarrow$) [14]. The *size of the computation* is the total number of events, $|E|$, in the computation.

Definition 1 *(Consistent Cut) A consistent cut C is a set of events in the computation which satisfies the following property: if an event e is contained in the set C, then all events in the computation that happened before e are contained in C.*
i.e., $\forall e_1, e_2 \in E : (e_2 \in C) \wedge (e_1 \rightarrow e_2) \Rightarrow e_1 \in C$

In Figure 1(i) the set $\{e_1, f_1\}$ is a consistent cut, while $\{e_1, e_2\}$ is not.

In the following discussion we mean 'consistent cut' whenever we simply say 'cut'. The set of all the consistent cuts in the computation is denoted by $\mathcal{C}$.

For notational convenience, we simply mention the maximal elements on each process that are elements of the cut to represent the cut. For example, the cut $\{e_1, e_2, f_1, f_2, f_3\}$ is written as $\{e_2, f_3\}$.

Definition 2 *Cut C_1 is less than or equal to cut C_2 if and only if, $C_1 \subseteq C_2$.*

A cut C satisfies a predicate P if the predicate is true in the global state represented by the cut. This is denoted by $C \models P$

We assume that each event in the computation has a vector clock [5, 16] associated with it. Each of these vector clocks correspond to a consistent cut in the computation.

The set of all consistent cuts in a computation forms a distributive lattice [7]. Birkhoff's representation theorem [1] states that a distributive lattice can be completely characterized by the set of its join irreducible elements (JIRRs). Join irreducible elements are elements of the lattice that cannot be expressed as the join of any two elements.[1] For example, in Figure 1 cuts $\{\}, \{f_1\}, \{f_2\}, \{f_3\}$, $\{e_1, f_1\}$, $\{e_2, f_1\}$, $\{e_3, f_1\}$ are join irreducible cuts. The cut, $\{e_1, f_2\}$ is not join irreducible because it can be expressed as the join (union) of cuts $\{f_2\}$ and $\{e_1, f_1\}$. The least consistent cut containing an event in the computation is a join irreducible element in the computational lattice [7].

Figure 1 shows a computation and the lattice of all the consistent cuts in the computation.

The *initial cut* is the least cut while the *final cut* is the greatest cut in the computational lattice. The initial is the empty set $\{\}$ and the final cut is the set of all events E. Detecting a predicate in a distributed computation is determining if the initial cut of the computation satisfies the predicate.

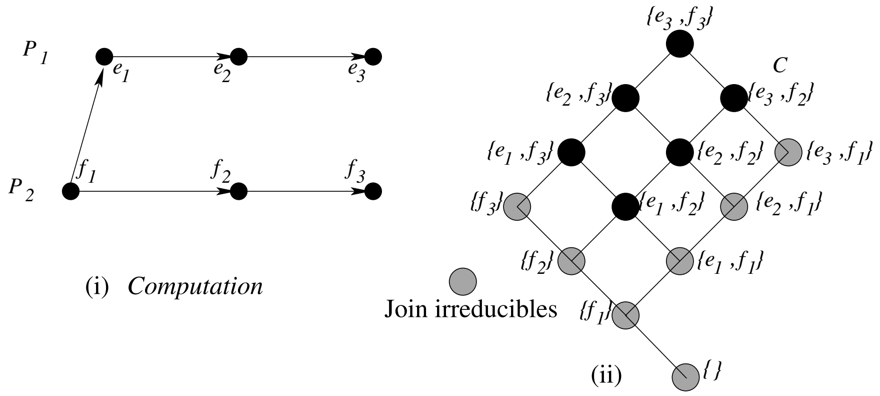


Figure 1: A computation and the lattice of its consistent cuts

3.2 Slicing [7, 19]

We now give a brief overview of *computational slicing*. Readers are referred to [19] for further details.

Definition 3 *(Slice) The slice $s[p]$ of a computation with respect to a predicate p is the poset of the join-irreducible consistent cuts of the smallest sublattice that contains all consistent cuts in the computation that satisfy p.*

Though the number of consistent cuts satisfying the predicate may be large, the slice of a predicate can be represented by the set of the join irreducible cuts in the

[1]Commonly, the bottom element is not considered to be a join irreducible element. However, in this paper, for notational convenience, we include the bottom element (the initial cut $\{\}$) in the set of join irreducible elements.

slice (including the bottom element or the initial cut $\{\}$ if applicable). The slice for the computation in Figure 2(i) for the predicate $(4 \leq x \leq 6) \ \wedge \ (y \neq 7)$ is shown in Figure 2(iii). *Slicing* is the operation of computing the slice for the given predicate.

Definition 4 *(Join-closed predicates) A predicate P is join-closed if all cuts that satisfy the predicate are closed under union.*
i.e., $(\forall C_1, C_2 \models P) : (C_1 \cup C_2) \models P$.

Definition 5 *(Meet-closed predicates) A predicate P is meet-closed if all cuts that satisfy the predicate are closed under intersection.*
i.e., $(\forall C_1, C_2 \models P) : (C_1 \cap C_2) \models P$.

Definition 6 *(Lean Slice) If every consistent cut in the slice $s[p]$ satisfies the predicate p, then the slice is said to be a lean slice.*

A lean slice is obtained whenever, the consistent cuts that satisfy the predicate have a lattice structure (closed under joins and meets). Such predicates are called regular predicates.

Definition 7 *(Regular Predicate) A predicate is regular if it is join-closed and meet-closed.*

If cuts C_1 and C_2 satisfy a regular predicate p, then by definition, $C_1 \cup C_2$ and $C_1 \cap C_2$ also satisfy p.

Consider the computation in Figure 2(i). Let $p \equiv (4 \leq x \leq 6) \wedge (y \neq 7))$.The cuts satisfying p are marked in Figure 2 (ii). The slice of p is the smallest computation which contains all the cuts that satisfy p. In this case the slice (Figure2 (iii)) contains exactly those cuts which satisfy p. Hence $s[p]$ is a lean slice.

Figure 2(iv) and 2(v) shows an example of the slice of a non-regular predicate $q \equiv (x = 6) \vee (y = 5)$. This slice is not lean since it contains the cuts $\{e_3, f_2\}$ and $\{e_3, f_3\}$ even though they do not satisfy the predicate q.

4 Multislicing

4.1 Logic model (MTL)

We now formally define Multislicing Temporal Logic(MTL).The atomic propositions in MTL are local predicates, i.e., properties that depend on a single process in the computation. Local predicates and their negations are regular predicates. Let AP be the set of all atomic propositions. Given the set of all consistent cuts,

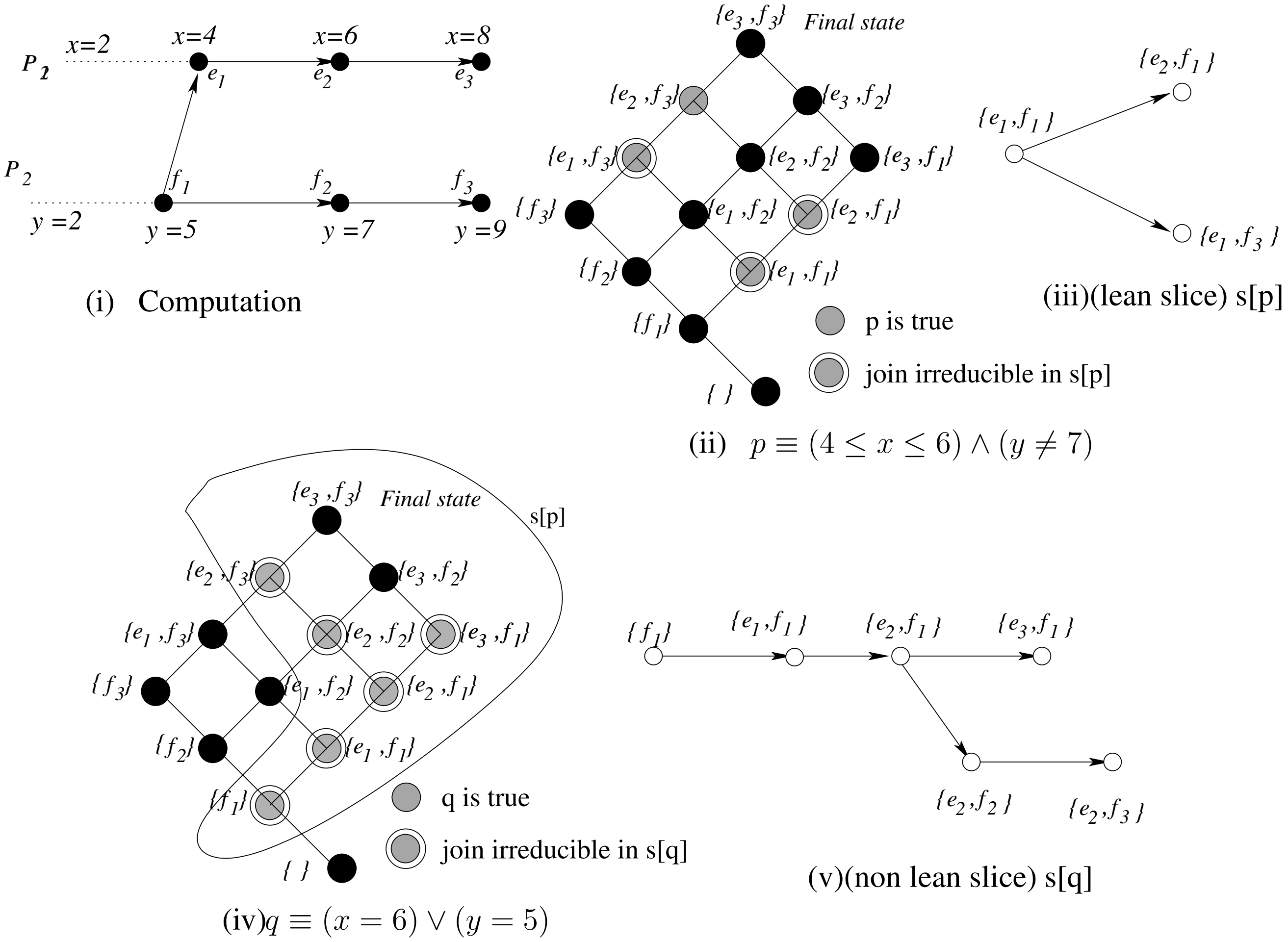


Figure 2: An example of a lean and non-lean slice

$\mathcal{C}$, of a computation, a labeling function $\lambda : \mathcal{C} \rightarrow 2^{AP}$ assigns to each consistent cut, the set of predicates from AP that hold in it. The operators $\wedge$ and $\vee$ represent the boolean conjunction and disjunction operators as usual , $\neg$ represent the negation of a predicate and we define the *possibly* ($\Diamond$) temporal operator (called EF in [17]).

Definition 8 *If $\mathcal{C}$ is the set of all consistent cuts of the computation then $\Diamond P$ holds at consistent cut C if and only if there exists $C' \in \mathcal{C}$ such that P is true at C' and $C \subseteq C'$.*

The formal MTL syntax is given below.

Definition 9 *A predicate in MTL is defined recursively as follows:*

1. $\forall l \in AP$ *,* l *is a local predicate*
2. *If* P *and* Q *are MTL predicates then* $P \vee Q$*,* $P \wedge Q$*,* $\Diamond P$ *and* $\neg P$ *are MTL predicates*

We now formally define the semantics of MTL. We use the notation cut $C \models P$ to mean that predicate P is true at consistent cut C and $\neg(C \models P)$ to mean that P is false at consistent cut C.

- $C \models l \Leftrightarrow l \in \lambda(C)$ for an atomic proposition l
- $C \models P \wedge Q \Leftrightarrow C \models P$ and $C \models Q$
- $C \models P \vee Q \Leftrightarrow C \models P$ or $C \models Q$
- $C \models \Diamond P \Leftrightarrow \exists C' \in \mathcal{C} : (C \subseteq C'$ and $C' \models P)$
- $C \models \neg P \Leftrightarrow \neg(C \models P)$

Note that the AG operator in CTL [17] can be written as $\neg\Diamond\neg$ in MTL.

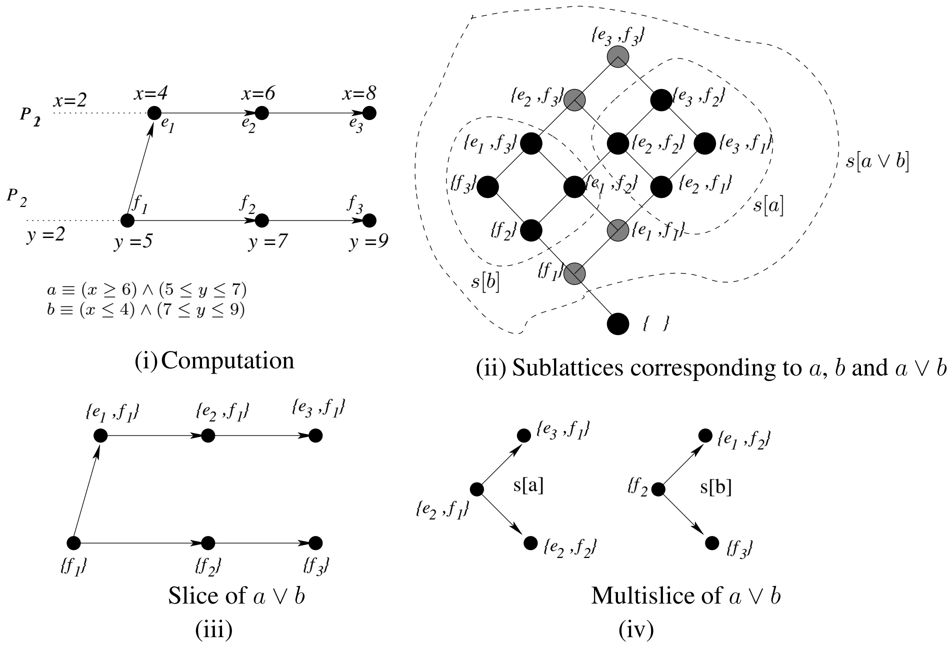
4.2 Multislice of a computation

The computation slice for a MTL predicate may not always be lean and current techniques to detect such predicates require exponential time in the worst case. To circumvent this problem we propose the multislicing approach. Instead of computing a single, possibly non-lean, slice we maintain a set of multiple lean slices corresponding to the predicate. This set is called a multislice of the computation. If the number of the slices in the multislice is bounded, then efficient predicate detection is possible since the slices are lean. In Section 5, we show that a multislice with polynomial (in the computation size) number of slices can be computed for MTL predicates.

Definition 10 *(Multislice) A multislice* S *of a computation* $\mathcal{C}$ *with respect to a predicate* P *is defined as a set of lean slices* $\{s_1, s_2, ..., s_k\}$ *corresponding to regular predicates* $p_1, p_2, ..., p_k$ *such that* $P = p_1 \vee p_2 \vee ... \vee p_k$*.*

Multislicing is the operation of computing a multislice of the computation for a given predicate. It follows from the definition that a multislice $S[P]$ satisfies the following properties:

1. If P is true at cut $C \in \mathcal{C}$ (i.e, $C \models P$) then $\exists s \in S[P]$ such that C belongs to s



(i) Computation

(ii) Sublattices corresponding to a, b and $a \vee b$

Slice of $a \vee b$
(iii)

Multislice of $a \vee b$
(iv)

Figure 3: An example of a multislice

2. $\forall s \in S[P] : C$ belongs to $s \Rightarrow C \models P$.

For example, consider the computation in Figure 3(i). If the predicate of interest is, say $a \vee b$, then the smallest sublattice that contains all cuts that satisfy $a \vee b$ is shown in Figure 3(ii). The corresponding slice is shown in Figure 3(iii). This slice contains many extraneous consistent cuts ($\{f_1\}, \{e_1, f_1\}, \{e_2, f_3\}, \{e_3, f_3\}$) that do not satisfy the predicate. The multislicing approach allows us to keep a set of slices and hence, instead of computing the non-lean slice for $a \vee b$, we add $s[a]$ and $s[b]$ to the multislice as shown in Figure 3(iv). Note that, the multislice

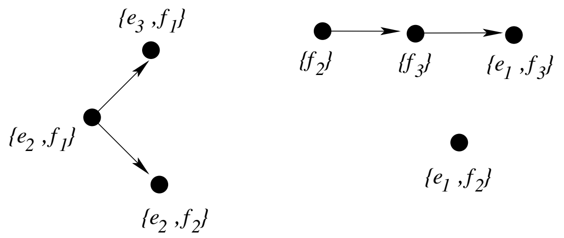


Figure 4: Multislices may not be unique - Another multislice for $a \vee b$ in Figure 3

for a predicate is not necessarily unique. Figure 4 shows another multislice for the computation in Figure 3 that is different from the multislice in Figure 3(iv).

4.3 Multislicing Algorithm

We now present an algorithm to compute a multislice for a predicate expressed in MTL. In the following discussion we mean a lean slice whenever we say 'slice'. Figure 5 outlines the main algorithm to compute a multislice of the computation. We assume that the input predicate has negations pushed in to the local predicates or the $\Diamond$ operators. In the following discussion, we shall often treat $\neg\Diamond$ as single operator.

We denote regular predicates and lean slices by small case letters (e.g. We use s or $s[p]$ to denote the slice of the computation for a regular predicate p). We use uppercase letters to represent multislices and MTL predicates. $S[P] = \cup\{s_i\}$ denotes a multislice corresponding to a MTL predicate $P = \bigvee p_i$. Given two slices s_1 and s_2 corresponding to regular predicates p_1 and p_2, $s_1 \wedge s_2$ represents the slice corresponding to $p_1 \wedge p_2$. Similarly $\Diamond s_1$ denotes the slice corresponding to $\Diamond p_1$ and $\neg\Diamond s$ represents a multislice corresponding to $\neg\Diamond p$.

```
/*The input predicate P_in has all negations pushed
inside to the ◇ operator or to the atomic propositions */
function computeMultiSlice(Predicate P_in)
S[P_in] : Set of slices
  Case 1 (Base case) : P_in == l or P_in == ¬l
      S[P_in] = {slice(P)}
  Case 2. P_in = P ∨ Q
      S[P] = computeMultiSlice(P); S[Q] = computeMultiSlice(Q)
      S[P_in] = {S[P] ∪ S[Q]}
  Case 3. P_in = P ∧ Q
      S[P] = computeMultiSlice(P); S[Q] = computeMultiSlice(Q)
      S[P_in] = {s_p ∧ s_q | ∀s_p ∈ S[P], s_q ∈ S[Q]}
  Case 4. P_in = ◇P
      S[P] = computeMultiSlice(P)
      S[P_in] = {◇s | s ∈ S[P]}
  Case 5. P_in = ¬◇P
      S[P] = computeMultiSlice(P)
      S[P_in] = {⋀(¬◇s) | s ∈ S[P]}

  return S[P_in]
end function
```

Figure 5: Multislicing a computation

A multislice represents all the cuts in the computation at which the predicate holds. To determine if the given MTL predicate holds at the initial state of the computation, we can simply check if the initial cut ($\perp$) belongs to any slice of the multislice. A multislice is nonempty if the predicate is true in any consistent cut of the computation.

We assume that for each atomic proposition l_i, the slice $s[l_i]$ can be computed in polynomial time. Efficient algorithms to compute $s[l_i]$ (or $s[\neg l_i]$) when the atomic propositions are local predicates, can be found in [19].

To compute $S[P \vee Q]$ we simply take $S[P] \cup S[Q]$(Case 2). The conjunction of two lean slices, $s_1 \wedge s_2$, can be computed by using the grafting algorithm described in [19]. The algorithm to compute EF of a regular predicate in [22] can be used to compute $\Diamond s_1$. Note that if s_1 and s_2 are lean slices, then $s_1 \wedge s_2$ and $\Diamond s_1$ are also lean [18, 22].

To compute $\neg\Diamond P$ where $P = p_1 \vee p_2 \vee \ldots \vee p_k$ is a MTL predicate, we compute the multislices for $\neg\Diamond p_1$, $\neg\Diamond p_2$ etc. and take their conjunction (Case 5). In Figure 6, we outline the algorithm to compute the multislice for $\neg\Diamond$ of a regular predicate. The main idea is, to find along each process, the least cut that does not satisfy $\Diamond P$. We get a set of at most n such cuts (called the *least cuts*) when the computation has n processes. Let $S' = \{C_1, C_2, \ldots, C_n\}$ be the set of least cuts. The output multislice S is given by the set of n slices $S = \{\texttt{intervalLatticeBetween}(C_i, E) | i = 1 \text{ to } n)$ where E is the final cut. Note that, $\texttt{intervalLatticeBetween}(C_1, C_2)$ returns the slice which with the contains all cuts between C_1 and C_2 with C_1, C_2 as the initial and final cuts respectively.

From the algorithm in Figure 6 it is clear that $\neg\Diamond p$ can have at most n slices. We prove in Section A.4 that the algorithm returns a multislice of $\neg\Diamond p$ when p is regular. This leads us to the result that the multislice of the MTL predicate $\neg\Diamond P$ where $P = p_1 \vee p_2 \vee \ldots \vee p_k$ can have n^k slices. Though it may appear that nested $\neg\Diamond$ operators could lead to exponential (in the size of n) blowup, we show later that the number of slices does not explode due to the property that the constituent slices of $\neg\Diamond P$ are *filters* [1] of the original computational lattice.

The algorithm in Figure 5 follows from the definition of a multislice and the semantics of MTL. A formal proof of correctness is included in the Appendix (Section A.4). In Section 5 we show that the size of the multislice computed by the algorithm in Figure 5 is polynomial with respect to the number of processes in the computation and the time complexity is $O(k.n^{2^k}.|E|)$. Note that the space required for the multislicing algorithm depends on the maximum number of slices in the multislice and is polynomial in n.

In Section A.1 we illustrate our multislicing algorithm with a step by step example of how a multislice is computed.

Input: Slice $s[p]$ corresponding to the regular predicate p
Original computation $s_{org} \equiv E$
Output: a multislice $S = S[\neg\Diamond p]$
Var: $C[1...n]$:cut (array of frontier events along each process)
Algorithm

```
For each process i:
    C = s[p].maxCutAlongProcess(i)
    C[i] = s[p].succesorEventOf(C[i])
    S = S ∪ intervalLatticeBetween(C, E)
        /* intervalLatticeBetween(C1, C2) contains all cuts between C1 and C2 */
    return S
```

Figure 6: $\neg\Diamond s$

5 Complexity Analysis

5.1 Complexity Analysis for Multislicing

We now show that a multislice can be computed in polynomial time with respect to the size of the computation for predicates belonging to MTL. The time complexity is not guaranteed to be polynomial in the size of the predicate.

We defined join-closed predicates in Section 3. In terms of the computational lattice, join-closed predicates are closed under the join operation [1] in the lattice. An example of a join-closed predicate is $\neg\Diamond$ of any predicate. Regular predicates are join-closed (and meet-closed).

Consider a multislice $S[P] = \bigvee_{1 \leq i \leq k'} s_i$, where $P = \bigvee_{1 \leq i \leq k'} p_i$. We can group together one or more slices in $S[P]$ such that each group corresponds to a join-closed predicate. For example, we could group together $S' = s_1, s_4$ if the predicate $P' = p_1 \vee p_4$ is join-closed. Let S'_1, $S'_2, \ldots, S'_m$ be such groups so that $S[P] = \{S'_1\} \cup \{S'_2\} \cup \ldots \cup \{S'_m\}$.

Definition 11 *(Join-closed size of a multislice)The join-closed size of a multislice is the minimum number of join-closed predicates that can represent the multislice. We denote the join-closed size of a multislice S by S^m.*

Note that $S^m \leq |S|$ where $|S|$ is the size of the multislice . Also S^m is an upper bound on the number of maximal elements in the multislice.

Lemma 1 *The join-closed size of a multislice of a MTL predicate P computed by the algorithm in Figure 5 is less than or equal to 2^k when the predicate P contains k operators.*

Proof: We prove this by induction on the number of operators k in the predicate (the size of the predicate).

Base case: When k is one, i.e., the predicate is an atomic proposition, the size of the multislice $|S| \leq 1$. Hence $S^m \leq 2^k$ when $k = 1$.

The algorithm allows the following operations

- $S[P] = s[l]$, This is the base case.
- $S = S_1 \vee S_2$
  $S^m = S_1^m + S_2^m \leq 2^{k_1} + 2^{k_2} \leq 2^k$ (since $k = k_1 + k_2 + 1$)
- $S = S_1 \wedge S_2$
  $S^m = S_1^m.S_2^m \leq 2^{k_1}.2^{k_2} \leq 2^k$ (since $k = k_1 + k_2 + 1$) ...we know $S^m = S_1^m.S_2^m$ from Lemma 5 in the Appendix.
- $S = \Diamond S_1$
  $S^m = S_1^m \leq 2^{k_1} \leq 2^k$ (since $k = k_1 + 1$)
- $S = \neg\Diamond S_1$
  $S_m \leq 1$ ...Lemmas 2 ,3 in the Appendix.

□

Note that the size of a multislice calculated by our algorithm in Figure 5 for predicates of the form $\Diamond P$ is $|S[\Diamond P]| = S^m[P]$.

Say $\Diamond P = p_1 \vee p_2 \vee \ldots \vee p_m$ where $m = S^m[\Diamond P] = S^m[P]$. Hence $S[\neg\Diamond P] = S[\neg\Diamond p_1] \wedge S[\neg\Diamond p_2] \wedge \ldots \wedge S[\neg\Diamond p_m]$. Since each of the multislices $S[\neg\Diamond p_i]$ could have at most n slices, the size of the multislice is $|S[\neg\Diamond P]| \leq n^m$.

Hence the size of a multislice of a predicate in the form $\neg\Diamond P$ is $n^{S^m[P]}$.

Theorem 1 *The multislice $S[P]$ computed by the algorithm in Figure 5 for a MTL predicate P with k operators has at most n^{2^k} slices*

Proof: Induction on k:

- $S = s[l]$, $|S|$ is always less than or equal to one in this case. This is the base case ($k = 1$).
- $S = S_1 \vee S_2$. $|S| = |S_1| + |S_2| \leq n^{2^{k_1}} + n^{2^{k_2}} \leq n^{2^k}$ (since $k = k_1 + k_2 + 1$)
- $S = S_1 \wedge S_2$. $|S| = |S_1|.|S_2| \leq n^{2^{k_1}}.n^{2^{k_2}} \leq n^{2^k}$ (since $k = k_1 + k_2 + 1$)
- $S = \Diamond S_1$. $|S| = |S_1^m| \leq 2^{k_1} \leq n^{2^k}$ (since $k = k_1 + 1$ and $S_1^m \leq 2^{k_1}$)
- $S = \neg\Diamond S_1$. $|S| = |n^{S_1^m}| \leq n^{2^{k_1}} \leq n^{2^k}$ (since $k = k_1 + 1$ and $S_1^m \leq 2^{k_1}$)

□

The time required to compute the conjunction two slices with respect to $\wedge$ is $O(|E|n^2))$ [19]. Case 3 in Figure 5 ($\vee$ of two MTL predicates) takes $O(|E_1|+|E_2|)$ time. It takes $O(|E|n)$ time to compute the slice with respect to the $\Diamond$ operator. The time required to compute the multislice corresponding to $\neg\Diamond p$ for a regular predicate p is $O(|E|n)$.

Theorem 2 *The time complexity of the algorithm in Figure 5 is polynomial in the number of events ($|E|$) and the number of processes (n) in the computation.*

Proof: The algorithm simplifies the predicate by computing the multislice for one operator at a time. Hence if there are k operators in all, it would require k steps to compute the multislice for the entire predicate. From Theorem 1, we know that after the k^{th} step at most n^{2^k} new slices are generated. The generation of each slice takes at most $|E|n^2$ time, where $|E|$ is the size of the computation and n is the number of processes in the computation. Hence, the total time taken to complete k steps (process each of the k operators) is $O(k.n^{2^k}.(|E|n^2)) = O(kn^{2^k}|E|)$. □

6 Experimental Evaluation

To examine the utility of multislicing in practical applications, we have implemented a multislicing toolkit to verify computation traces generated by distributed programs.

We used a Java implementation of the distributed dining philosophers algorithm from [11]. Faults were injected in the computation traces and the property under consideration was $(eat_0 \wedge eat_1) \vee (eat_1 \wedge eat_2) \wedge (\Diamond((eat_2 \wedge eat_3) \vee (eat_3 \wedge eat_4)))$. The trace was verified using, both, the multislicer and POTA [23]. Note that, for predicates containing disjunctions, POTA reduces the computation size and uses Spin [9] to check the predicate. The POTA/Spin combination performs well in some runs (when the slice generated is lean or empty) but it consistently runs out of memory in each configuration with 8 or more processes. The multislicer however scales well and we could use it to verify computations with 100 processes. The multislicing toolkit (including the Java source code) and other detailed experimental results are available for download at [20].

7 Conclusions

We conclude that it is possible to efficiently detect nested temporal predicates containing disjunctions and negations (along with conjunctions and $\Diamond$). We have introduced the notion of a multislice and have presented techniques to efficiently compute a multislice for a MTL predicate.

In the future, it will be useful to extend multislicing techniques for logics that include a wider range of temporal operators. Another aspect would be to identify sub-classes of MTL that can be detected even more efficiently(for example, in polynomial time with respect to the size of the predicate). It would also be interesting to examine how multislicing can be used verify traces representing non-terminating computations.

References

[1] DAVEY, B. A., AND PRIESTLEY, H. A. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[2] DRUSINSKY, D. The temporal rover and the atg rover. In *Spin Model Checking and Verification* (2000), vol. 1885 of LNCS, pp. 323–330.

[3] E. M. CLARKE, AND E. A. EMERSON. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proc. of the Workshop on Logics of Programs* (Yorktown Heights, New York, May 1981), vol. 131 of *Lecture Notes in Computer Science*.

[4] ESPARZA, J. Model checking using net unfoldings. In *Science Of Computer Programming* (1994), vol. 23(2), pp. 151–195.

[5] FIDGE, C. J. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices 24*, 1 (Jan. 1989), 183–194.

[6] GARG, V. K. *Elements of Distributed Computing*. Wiley & Sons, 2002.

[7] GARG, V. K., AND MITTAL, N. On slicing a distributed computation. In *21st International Conference on Distributed Computing Systems (ICDCS' 01)* (Washington - Brussels - Tokyo, Apr. 2001), IEEE, pp. 322–329.

[8] GERALD, H. The model checker spin. In *IEEE transactions on software engineering 23.5* (1997).

[9] GERALD, H. *The Spin Model Checker*. Addison-Wesley Professional, 2003.

[10] GODEFROID, P. *Partial-Order Methods for the Verification of Concurrent Systems*, vol. 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[11] HARTLEY, S. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

[12] HAVELUND, K., AND ROSU, G. Monitoring java programs with java pathexplorer. In *Runtime Verification 2001* (2001), vol. 55 of ENTCS.

[13] KOUSHIK SEN, G. R., AND AGHA, G. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)* (2005).

[14] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[15] M. KIM, S. KANNAN, I. L. O. S., AND VISWANATHAN, M. Java-mac: A run-time assurance tool for java programs. In *Runtime Verification 2001* (2001), vol. 55 of ENTCS.

[16] MATTERN, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms* (1989), Elsevier Science Publishers B.V. (North-Holland), pp. 215–226.

[17] MCMILLAN, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] MITTAL, N., AND GARG, V. K. On detecting global predicates in distributed computations. In *21st International Conference on Distributed Computing Systems (ICDCS' 01)* (Washington - Brussels - Tokyo, Apr. 2001), IEEE, pp. 3–10.

[19] MITTAL, N., AND GARG, V. K. Slicing a distributed computation: Techniques and theory. In *5th International Symposium on DIStributed Computing (DISC'01)* (Oct. 2001), pp. 78 – 92.

[20] OGALE, V. A., AND GARG, V. K. Multislicing toolkit. In *Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu/mslicer* (2006).

[21] PELED, D. All from one, one for all: On model checking using representatives. In *5th International Conference on Computer Aided Verification (CAV)* (1993), pp. 409–423.

[22] SEN, A., AND GARG, V. K. Detecting temporal logic predicates in distributed programs using computation slicing. In *7th International Conference on Principles of Distributed Systems* (Dec. 2003).

[23] SEN, A., AND GARG, V. K. Partial order trace analyzer (pota) for distributed programs. In *Proceedings of the Third International Workshop on Runtime Verification (RV)* (July 2003).

[24] STOLLER, S. D., AND LIU, Y. Efficient symbolic detection of global properties in distributed systems. In *10th International Conference on Computer Aided Verification (CAV)* (2000), vol. 1855 of LNCS, pp. 264–279.

[25] STOLLER, S. D., AND SCHNEIDER, F. B. Faster possibility detection by combining two approaches. In *Proc. of the 9th International Workshop on Distributed Algorithms* (Le Mont-Saint-Michel, France, Sept. 1995), Springer-Verlag, pp. 318–332.

[26] STOLLER, S. D., UNNIKRISHNAN, L., AND LIU, Y. A. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)* (July 2000), vol. 1855 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 264–279.

[27] VALMARI, A. A stubborn attack on state explosion. In *International Conference on Computer Aided Verification (CAV)* (1990), vol. 531 of LNCS, pp. 156–165.

A Appendix

A.1 Example of multislicing

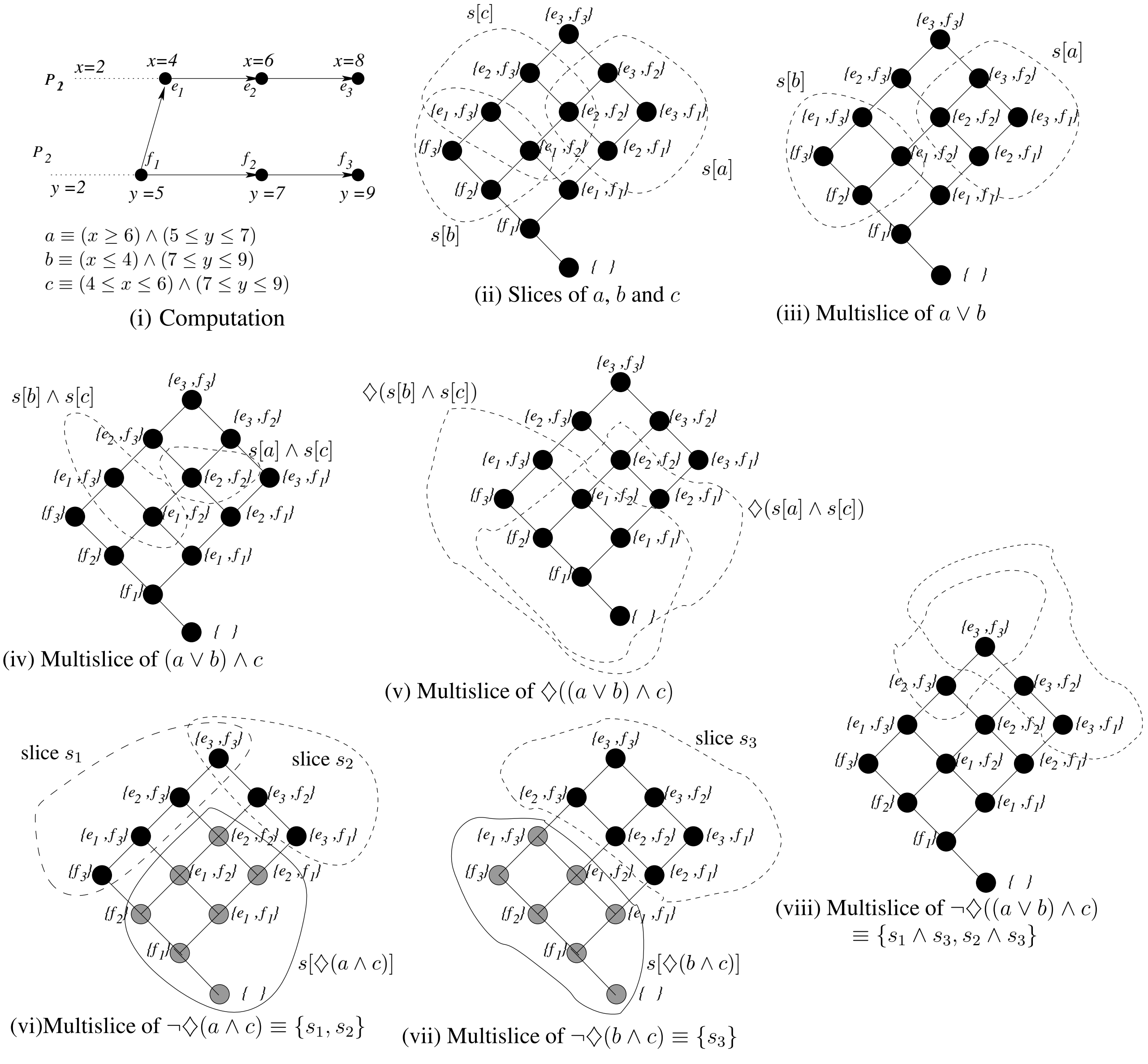


Figure 7: Computing the Multislice

A.2 An Example of Multislicing

Figure 7(i) depicts a computation and the predicate to be detected is $\neg\Diamond(a \vee b) \wedge c$. Figure 7(ii) depicts all the consistent cuts that satisfy local predicates a, b and c. We process the predicate recursively, inside out. We first compute the multislice for $P_1 = a \vee b$ which is simply the set $\{s[a], s[b]\}$ (Figure 7(iii)). In the next iteration to find the multislice for $P_2 = P_1 \wedge c$ we compute the slices $s_1 = s[a] \wedge s[c]$ and $s_2 = s[b] \wedge s[c]$ as shown in the Figure 7(iv). The multislice for $P = \Diamond F_2$ is the set $\{\Diamond(s[a] \wedge s[c]), \Diamond(s[b] \wedge s[c])\}$ as shown in Figure 7(v). The multislice for $\neg\Diamond((a \vee b) \wedge c))$ is given by the conjunction of the multislices $\neg\Diamond(a \wedge c)$ (Figure 7(vi)) and $\neg\Diamond(a \wedge c)$ (Figure 7(vii)). The final output multislice for the predicate $\neg\Diamond(a \vee b) \wedge c$ is shown in Figure 7(viii).

A.3 Proof of correctness of the multislicing algorithm

Theorem 3 *The algorithm in Figure 5 multislices the computation with respect to a MTL predicate F.*

Proof:

If F is not an atomic proposition or its negation, then it can be defined in terms of *smaller* predicates F' and F'' in one of the following four ways:

1. $F = F' \wedge F''$
2. $F = F' \vee F''$
3. $F = \Diamond F'$
4. $F = \neg\Diamond F'$

We show that the theorem holds for F, assuming that it holds for F' and F''.

Since the theorem holds for F' and F'', $S[F'] = \{s'_1, s'_2, \ldots, s'_{k'}\}$ where $s'_1, s'_2, \ldots, s'_{k'}$ are slices corresponding to predicates $p'_1, p'_2, \ldots, p'_{k'}$ such that $F' = p'_1 \vee p'_2 \vee \ldots \vee p'_{k'}$.

Similarly $S[F''] = \{s''_1, s''_2, \ldots, s''_{k''}\}$ where $s''_1, s''_2, \ldots, s''_{k''}$ are slices corresponding to predicates $p''_1, p''_2, \ldots, p''_{k''}$ such that $F'' = p''_1 \vee p''_2 \vee \ldots \vee p''_{k''}$.

- Case 1:

  $F = F' \wedge F''$
  $= (p'_1 \vee p'_2 \vee \ldots \vee p'_{k'}) \wedge (p''_1 \vee p''_2 \vee \ldots \vee p''_{k''})$
  $= ((p'_1 \wedge p''_1) \vee (p'_1 \wedge p''_2) \vee \ldots \vee (p'_1 \vee p''_{k''})) \vee ((p'_2 \wedge p''_1) \vee (p'_2 \wedge p''_2) \vee \ldots (p'_1 \vee p''_{k''})) \vee \ldots \vee ((p'_k \wedge p''_1) \vee (p'_k \wedge p''_2) \vee \ldots \vee (p'_{k'} \vee p''_{k''}))$

Since each of the disjuncts ($(p'_1 \wedge p''_1)$, etc.) is a regular predicate we get $F = (p_1 \vee p_2 \vee \ldots \vee p_k)$,where $k = k_1 * k_2$ and $p_1, p_2, \ldots, p_k$ are regular.

Rule 2 computes $S[F] = \{((s'_1 \wedge s''_1) \vee (s'_1 \wedge s''_2) \vee \ldots \vee (s'_1 \wedge s''_{k''})) \vee (((s'_2 \wedge s''_1) \vee (s'_2 \wedge s''_2) \vee \ldots \vee (s'_2 \wedge s''_{k''})) \vee \ldots \vee ((s'_{k'} \wedge s''_1) \vee (s'_{k'} \wedge s''_2) \vee \ldots \vee (s'_{k'} \wedge s''_{k''}\}$ $S[F] = \{p_1, p_2, \ldots, p_k\}$ where $F = p_1 \vee p_2 \vee \ldots \vee p_k$ and $p_1, p_2, \ldots, p_k$ are regular predicates.

Hence, by definition, $S[F]$ is a multislice for predicate F.

- Case 2:

  $F = F' \vee F''$
  $= (p'_1 \vee p'_2 \vee \ldots \vee p'_{k'}) \vee (p''_1 \vee p''_2 \vee \ldots \vee p''_{k''})$

  Rule 3 computes $S[F] = \{(s'_1, s'_2, \ldots, s_{k'})\} \cup (s''_1, s''_2, \ldots, s''_{k''})\}$ which is a multislice of F.

- Case 3:

  $F = \Diamond F_1$
  $= \Diamond(p'_1 \vee p'_2 \vee \ldots \vee p'_{k'})$
  $= \Diamond p'_1 \vee \Diamond p'_2 \vee \ldots \vee \Diamond p_{k'}$ since $\Diamond$ distributes over $\vee$

  According to Rule 4 $S[F] = \{\Diamond s'_1, \Diamond s'_2, \ldots, \Diamond s'_{k'}\}$ where $\Diamond s'_1, \Diamond s'_2, \ldots, \Diamond s'_{k'}$ are lean slices corresponding to regular predicates $\Diamond p'_1, \Diamond p'_2, \ldots, \Diamond p'_{k'}$

- Case 4:

  $F = \neg\Diamond F_1$
  $= \neg\Diamond(p'_1 \vee p'_2 \vee \ldots \vee p'_{k'})$
  $= \neg\Diamond p'_1 \wedge \neg\Diamond p'_2 \wedge \ldots \wedge \neg\Diamond p_{k'}$

  According to Rule 5 $S[F] = \{\neg\Diamond S[p_1] \wedge \neg\Diamond S[p_2], \ldots, \neg\Diamond S[p'_k]\}$ where $S[p_1], S[p_2], \ldots, S[P'_k]$ are multislices corresponding to regular predicates $p'_1, p'_2, \ldots, p'_{k'}$

  Therefore, by definition, $S[F]$ is a multislice for MTL predicate F.

Hence the algorithm in Figure 5 computes a multislice for the MTL predicate F. □

A.4 Proof of correctness of algorithm for $\neg\Diamond p$

We need to show that when p is regular $C \models \neg\Diamond p \iff C \in S$ where S is the output multislice of the algorithm in Figure 6. From the semantics of $\neg$ and $\Diamond$ we have $C \models \neg\Diamond p \iff \forall C_p : C_p \models p : \neg(C \leq C_p)$

Now the algorithm computes cuts $C_1, C_2, ..., C_n$ in the step $C[i] = s[p].\texttt{succesorEventOf}(C[i])$. We call these cuts *least cuts*. Least cuts have the following properties:

1. All cuts on process i that are less than or equal to the maximal cut satisfying p are less than C_i. i.e. all cuts less than C_i satisfy $\Diamond p$.

2. All cuts on process i that are greater than or equal to C_i are in the output multislice

3. The least cuts are the minimal join irreducible elements of the computational lattice that satisfy $\neg\Diamond P$.

A.4.1 To show: $C \in S \Rightarrow \forall C_p : C_p \models p : \neg(C \leq C_p)$

Assume that $C \leq C_p$. From the algorithm, $C \in S \Rightarrow (\exists i : C_i \leq C)$ where C_i is a least cut.
Hence $\exists C_i \leq C_p$ which violates the condition that C_i is the greater than the maximal cut satisfying p along process i.

A.4.2 To show: $(\forall C_p : C_p \models p : \neg(C \leq C_p) \Rightarrow C \in S$

Consider any cut C in the computation. C can be represented by the union of join irreducible cuts (Birkoff's representation theorem). Let $\mathcal{J}$ be the set of join irreducible cuts that represent C. Hence $C = \cup J_i$ where $J_i \in \mathcal{J}$. If any of the cuts in $\mathcal{J}$ is contained within the multislice of $\neg\Diamond p$ then that implies that there exists a least cut $C_i \subseteq J_i$ where C_i is a least cut and $J_i \in J$. Also $J_i \subseteq C$ hence $C \subseteq C_i$ and is included in the output slice by the algorithm.

If none of the cuts in $\mathcal{J}$ belong to the multislice for $\neg\Diamond p$ i.e., then we $J_i \in J \Rightarrow J_i \models \Diamond p$. Hence $\cup J_i$ where $J_i \in \mathcal{J}$ belongs to $\Diamond P$ (since the $\Diamond$ of a regular predicate is regular). Therefore C does not satisfy the predicate $\neg\Diamond p$.

A.5 Complexity analysis

Lemma 2 *Given any predicate P, the predicate corresponding to $\neg\Diamond P$ is join-closed.*

Proof: From the semantics of $\neg\Diamond$ we know that $C \models \neg\Diamond P \Rightarrow$ all cuts greater than C_1 satisfy $\neg\Diamond P$. Now $C_1 \cup C_2$ is greater than or equal to C_1 and C_2. Hence, $C_1, C_2 \models \neg\Diamond P \Rightarrow C_1 \cup C_2 \models \neg\Diamond P$ Therefore $\neg\Diamond P$ is join-closed. □

Lemma 3 *A multislice of a join-closed predicate is either empty or has a unique maximal element.*

Proof: Assume that C_1, C_2 are maximal elements of a join-closed predicate. Therefore $C_1 \cup C_2$ satisfies the predicate and is greater than or equal to C_1 and C_2. Since C_1, C_2 are maximal we have $C_1 = C_2 = C_1 \cup C_2$. □

Lemma 4 *The conjunction of any two join-closed predicates is also a join-closed predicate*

Proof: Let P_1, P_2 be join-closed and $P = P_1 \wedge P_2$. Hence $C \models P \Leftrightarrow (C \models P_1$ and $C \models P_2)$. We show that P is join-closed, i.e., $C_1 \models P$ and $C_2 \models P \Rightarrow (C_1 \cup C_2) \models P$.

$C_1, C_2 \models P$ implies that $C_1, C_2 \models P_1$ and $C_1, C_2 \models P_2$. Since P_1, P_2 are join closed therefore $(C_1 \cup C_2) \models P_1$ and $(C_1 \cup C_2) \models P_2)$. Hence $(C_1 \cup C_2) \models P)$. □

Lemma 5 *If S, S_1, S_2 are multislices such that $S = S_1 \wedge S_2$ then $S^m = S_1^m.S_2^m$*

Proof:

Let $S = S[P], S_1 = S[Q]$ and $S_2 = S[R]$ where $P = Q \wedge R$.

Now $Q = \bigvee Q'_i$ and $R = \bigvee R'_j$ where $1 \leq i \leq S_1^m, 1 \leq j \leq S_2^m$ Q'_i, R'_j etc are join closed predicates. Hence $P = \bigvee(Q'_i \wedge R'_j)$ where Q'_i and R'_j are join-closed, $Q'_i \in Q, R'_j \in R$

Now the conjunction of any two join-closed predicates is also a join-closed predicate (Lemma 4). Therefore P can be represented by $S_1^m.S_2^m$ join-closed predicates. Hence $S^m = S_1^m.S_2^m$ □