Partitioning State Spaces of Concurrent Transition Systems

Mustapha Bourahla
Computer Science Department, University of Biskra, Algeria

Abstract: *As the model-checking becomes increasingly used in the industry as an analysis support, there is a big need for efficient new methods to deal with the large real-size of concurrent transition systems. We propose a new algorithm for partitioning the large state space modelling industrial designs as concurrent transition systems with hundreds of millions of states and transitions. The produced partitions will be used by distributed processes for parallel system analysis. The state space is supposed to be represented by a weighted Kripke structure (this is an extension of the Kripke structure where weights are associated with the states and with the transitions). This algorithm partitions the weighted Kripke structure by performing a combination of abstraction-partition-refinement on this structure. The algorithm is designed in a way that reduces the communication overhead between the processes. The experimental results on large real designs show that this method improves the quality of partitions, the communication overhead and then the overall performance of the system analysis.*

Keywords: *Concurrent transition systems, distributed and parallel analysis*, *abstraction, partitioning, refinement.*

Received February 6, 2004; accepted July 4, 2004

1. Introduction

As formal verification becomes increasingly used in the industry as a part of the design process, there is a constant need for efficient tool support to deal with real-size applications. There are many methods proposed to overcome this problem, including abstraction, partial order reduction, equivalence-based reduction, modular methods, and symmetry [3, 6]. Recently, a new promising method to tackle the state space explosion problem was introduced [2, 4, 10]. This method is based on the use of multiprocessor systems or workstation clusters. These systems often boast a very large (distributed) main memory. Furthermore, the large computational power of such systems also helps in effectively reducing model checking time.

In this paper, we develop an efficient algorithm for partitioning the state space in terms of computation and communication. The following is a detailed description of our approach. The state space on which the analysis will be performed, is partitioned into M parts, where each part is owned by one process in the network. In order to increase the performance of the parallel analysis, it is essential to achieve a good load balancing between the M machines, meaning that the M parts of the distributed state space should contain nearly the same number of states. The quality of a partitioning algorithm could also be estimated according to the number of cross-border transitions of the partitioned state space (i. e., transitions having the source state in a component and the target state in another component). This number should be as small as possible, since it has effect on the number of messages sent over the network during the system analysis. The state space is represented by a simple structure (weighted Kripke structure) which is represented by a data structure doesn't consume large memory. We adopted a static partition scheme, which avoids the potential communication overhead occurring in dynamic load balancing schemes. This partitioning scheme has an adaptive cost which yields nearly equal partitions with small number of cross-border transitions. Then, the problem is to choose an appropriate partition algorithm associating to each state a machine index. The result of this algorithm is a partitioning function P.

Our algorithm for partitioning is performed on three steps. The first step is the abstraction of the state space represented by a weighted Kripke structure using the matching notion [5, 7, 9] of pairs of states making a transition in the model (one is the source and the other is the target). This abstraction will continue until reduction of the state space to a certain number of states small enough to do the partitioning very easily. After partitioning this much smaller weighted Kripke structure to M partitions, this partitioning is projected back towards the original weighted Kripke structure (finer structure), by periodically performing refinements on the projected structures. When the partitioning function P: $S \rightarrow \{0, \ldots, M-1\}$ is produced, we proceed by partitioning the weighted Kripke structure to M components where the border states are duplicated by a fashion satisfying the balancing condition.

The results of our experiments on real designs show that this new scheme produces good quality of partitioning that can be compared to other efficient approaches. This partitioning has small percentage of cross-border transitions compared to other reported results. It requires substantially small time.

2. Partitioning State Spaces

The partitioning problem is defined as follows: Given a Kripke structure K = (S, I, R, L) modelling a concurrent transition system over a set of Atomic Propositions (AP), where:

- S is a non-empty finite set of states.
- $I \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a total transition relation (i. e., $(\forall s \in S: (\exists s' \in S : (s, s') \in R))$).
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

With $|S| = N$, partition S into M subsets, $S_0, S_1, \ldots, S_{M-1}$ such that $S_i \cap S_j = \varnothing$ for $i \neq j$, $|S_i|$ (N/M, and $\cup_i S_i = S$, and the number of transitions crossing the border is minimized.

We define a weighted Kripke structure as Kripke structure K = (S, I, R, L) where weights are associated with each state $s \in S$ and each transition $(s, s') \in R$. A weighted state $s \in S$ is a state collapsing (abstracting) states of the original model and its weight represents their number. The weighted state collapsing the original initial state is the initial weighted state. The weight of a transition between two weighted states s, $s' \in S$, represents the number of transitions between the states composing the weighted state s and the weighted state s'. Now, a Kripke structure K can be viewed as a weighted Kripke structure, where all the state weights and transition weights are equal to one.

The partitioning problem can be naturally extended to weighted Kripke structures. In this case, the goal is to partition the states into M disjoint subsets so that the sum of the state weights in each subset is the same, and the sum of the transitions weights which crossing the border is minimized. A partition of S is commonly represented by a total partition function $P: S \rightarrow \{0, \ldots, M - 1\}$, so that for every state $s \in S$, P (s) is an integer between 0 and M - 1, indicating the partition at which state s belongs. Given a partition function P, the number of transitions crossing the border is called the *TransitionCut* of the partition.

The idea of our algorithm of partitioning is inspired by good works done for partitioning graphs [9, 11, 12]. The basic structure which we propose for the partitioning algorithm is very simple. The original weighted Kripke structure K is first abstracted down to a sufficient number of states, a partition of this much smaller structure is computed, and then this partition is projected back towards the original structure (finer Kripke structure). At each step of the structure abstracting, the partition is further refined. Since the finer structure has more degrees of freedom, such refinements usually decrease the *TransitionCut* number. Formally, the partitioning algorithm works as follows: Consider a weighted Kripke structure $K_0 = (S_0, I_0, R_0, L_0)$, with weights both on states and transition edges. A partitioning algorithm consists of the following three phases (each one is described in detail in a separate section):

1. Abstracting phase in which the Kripke structure K_0 is transformed into a sequence of smaller structures $K_1, K_2, \ldots, K_r$ such that $|S_0| > |S_1| > |S_2| > \ldots > |S_r|$.
2. Partitioning phase where, a partition function P_r of the structure $K_r = (S_r, I_r, R_r, L_r)$ is computed that partitions S_r into M parts.
3. Refinement phase where, the partition function P_r of K_r is projected back to K_0 by going through intermediate partition functions $P_{r-1}, P_{r-2}, \ldots, P_1, P_0$.

The data structure used to store the state space consists of two tables. The first is called *StateTable*, it stores information about states and the second is called *TransitionTable*, it stores the transitions. For each state $s \in S$ (which is an index in $\{0, \ldots, N-1\}$, N is the number of states), StateTable[s] contains the following informations. sw the weight of s, ns (and np) the number of transitions outgoing (ingoing) from s (the number of successor (predecessor) states of s), is (and ip) the index into *TransitionTable* that is the beginning of the transitions table of successor (predecessor) states of s, ctw the weight of the transitions that have been contracted to create s (if s is collapsing state), and aw the sum of the weight of the transitions adjacent (outgoing and ingoing) to s. The table *TransitionTable* is fragmented to many portions. Each portion represents the transitions of a state $s \in S$ to/from its adjacent (successor and predecessor) states. Thus, there are two information: The first is the state with which the transition is made. The second information indicates if the transition edge is an outgoing or an ingoing edge. We define the function Adj: $S \rightarrow 2^S$ associated to the table *TransitionTable*. Adj (s) gives the set of states that are connected (adjacent) to s. These information are used during different phases of the algorithm, and greatly improve the performance. Also, they are computed incrementally during the abstraction, hence they do not increase the overall run time of the algorithm.

3. Abstracting Phase

During the abstracting phase, a sequence of smaller weighted Kripke structures, each with fewer states, is constructed. Structure abstracting can be achieved by combining a set of states of a weighted Kripke structure K_i to form a single state of the next level coarser structure K_{i+1}. Let S^s_i be the set of states of S_i

combined to form state s of K_{i+1}. We will refer to state s as a multi-node. In order for a partition of a coarser weighted Kripke structure to be good with respect to the original structure, the weight of state s is set equal to the sum of the weights of the states in S^s_i. Also, in order to preserve the connectivity information in the coarser structure, the transitions of s are the union of the transitions of the states in S^s_i. In the case where more than one state of S^s_i contain transitions to the same state s', the weight of the transition of s is equal to the sum of the weights of these transitions. This is useful when we evaluate the quality of a partition at a coarser weighted Kripke structure. The *TransitionCut* number of the partition in a coarser structure will be equal to the *TransitionCut* number of the same partition in the finer structure.

Given a weighted Kripke structure $K_i = (S_i, I_i, R_i, L_i)$, a coarser Kripke structure can be obtained by collapsing adjacent states. Two states are adjacent if and only if there is a transition between these two states. Thus, the transition between two states is collapsed and a multi-node consisting of these two states is created. This transition collapsing idea can be formally defined in terms of matchings [5, 7, 9].

A matching of a weighted Kripke structure, is a set of transitions, no two of which are incident on the same state (a transition is incident on a state if the state is the source or the target of this transition). A matching is maximal if any transition in the structure that is not in the matching has at least one of its endpoints matched. The maximal matching that has the maximum number of transitions is called maximum matching.

Thus, the next level coarser weighted Kripke structure K_{i+1} is constructed from K_i by finding a matching of K_i and collapsing the states being matched into multi-nodes. The unmatched states are simply copied over to K_{i+1}. Since the goal of collapsing states using matchings is to decrease the size of the structure K_i, the matching should contain a large number of transitions. For this reason, maximal matchings are used to obtain the successively coarse structures. Note that depending on how matchings are computed, the number of transitions belonging to the maximal matching may be different. However, because the complexity of computing a maximum matching [13] is in general higher than that of computing a maximal matching, the latter is preferred. Since maximal matchings are used to abstract the structure, the number of states in K_{i+1} cannot be less than half the number of states in K_i; thus, it will require at least O (log (N/N')) steps to abstract K_0 down to a structure with N' states. However, depending on the connectivity of K_i, the size of the maximal matching may be much smaller than $|S_i|/2$. In this case, the ratio of the number of states from K_i to K_{i+1} may be much smaller than 2. If the ratio becomes lower than a threshold, then it is better to stop the abstracting phase. However, this type of pathological condition usually arises after many abstracting levels, in which case K_i is already fairly small; thus, aborting the abstracting does not affect the overall performance of the algorithm.

The abstraction algorithm consists of two stages: The matching stage and the contraction stage where, a coarser structure is created by contracting the states as dictated by the matching. The output of the matching stage, is two vectors *Match* and *Map*, so that for each state s, Match [s] stores the state with which s has been matched (or s itself if it is unmatched), and Map [s] stores the given label of s in the coarser structure which is assigned a sequential number (if Match [s] = s' then Map [s] = Map [s']). During the contraction stage, the *Match* and *Map* vectors are used to abstract the structure.

Algorithm 1: *Abstract* ($K_i = (S_i, I_i, R_i, L_i)$)

```
{
 Matching:
   Create a random list RS of all the states in
   StateTable representing the set S_i
 for each state s ∈ RS
 {
   Match [s] ← s
   Map [s] ← -1
 }
 for each state s ∈ RS
 {
   if (Match [s] = s)
    {
      H ← {s' | Match [s'] = s' ∧ s' ∈ Adj (s)}
      Let MW be the maximum weight of the
      contracted transitions in H
      H ← H \ {s' ∈ H | ctw (s') < MW}
      SW ← ∅
      for each state s' ∈ H
       {
         SW ← SW ∪ {
              ∑(s', s'') ( Ri | s'' ∈ Adj (s) w (s', s'') +
              ∑(s'', s') ( Ri | s'' ∈ Adj (s) w (s'', s')}
       }
      Let s' ∈ H be the state corresponding to the
         maximum in SW
      Match [s] ← s'
    }
 }
 j ← 0
 for each state s ∈ RS
 {
   if (Map [s] = -1)
    {
      T [j] ← Map [s] ← Map [Match [s]] ← q_j
      j++
    }
 }
```

Contraction:

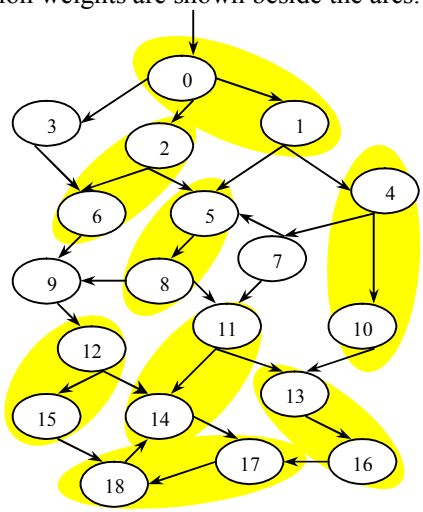
$S_{i+1} \leftarrow \varnothing$; $R_{i+1} \leftarrow \varnothing$; $index \leftarrow 0$
for each $k \in \{0, \ldots, j-1\}$
{
$S_{i+1} \leftarrow S_{i+1} \cup \{T[k]\}$
$SuccSet \leftarrow \{Map[s] \mid s \in S_i \wedge Map[s] \neq T[k] \wedge \forall s' \in S_i: T[k] = Map[s'] \wedge (s', s) \in R_i\}$
$is(T[k]) \leftarrow index$
$ns(T[k]) \leftarrow Length(SuccSet)$
$index \leftarrow index + ns(T[k])$
$PredSet \leftarrow \{Map[s] \mid s \in S_i \wedge Map[s] \neq T[k] \wedge \forall s' \in Si: T[k] = Map[s'] \wedge (s, s') \in R_i\}$
$ip(T[k]) \leftarrow index$
$np(T[k]) \leftarrow Length(PredSet)$
$index \leftarrow index + np(T[k])$
$R_{i+1} \leftarrow R_{i+1} \cup SuccSet \cup PredSet$
$sw(T[k]) \leftarrow sw(s_1) + sw(s_2)$ *s. t.* $T[k] = Map[s_1] = Map[s_2]$
$ctw(T[k]) \leftarrow ctw(s_1) + ctw(s_2) + w((s_1, s_2)) + w((s_2, s_1))$ *s. t.* $T[k] = Map[s1] = Map[s2]$
$aw(T[k]) (aw(s1) + aw(s2) - w((s1, s2)) - w((s2, s1))$ *s. t.* $T[k] = Map[s1] = Map[s2]$
}
for each transition $(q_1, q_2) \in R_{i+1}$ *s. t.* $q_1 = Map[s_1] \wedge q1 = Map[s2]$
{
$w((q1, q2)) \leftarrow \sum_{s \mid Map[s] = q2} w((s_1, s)) + \sum_{s \mid Map[s] = q2} w((s_2, s))$
}
$I_{i+1} = \{q \mid q = Map[s] \wedge s \in I_i\}$
$L_{i+1}: Q \rightarrow 2^{AP}$ *s. t.*
$L_{i+1}(q) = L_i(s_1) \cup L_i(s_2)$, *where* $Map[s_1] = Map[s_2] = q$
return $(S_{i+1}, I_{i+1}, R_{i+1}, L_{i+1})$
}

The idea of matching stage is to minimize the *TransitionCut* number by selecting a maximal matching whose transition edges have a large weight, thus we can decrease the transition weight of the coarser structure. Finding a maximal matching that contains transitions with large weight, is computed using the randomized algorithm described above. The states are visited in random order to find their set of unmatched adjacent states with maximum weight of connection. We select the state that has the maximum sum of weights of the transitions connecting this state to the adjacent states of the state considered for matching. The example below clarifies this idea.

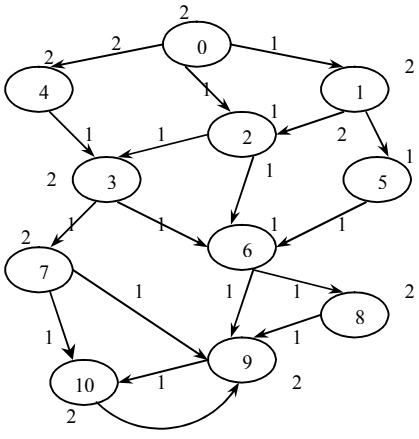
To efficiently implement the above operation for all matched states, we have used a table to keep track of the states seen so far. These data structures allow us to implement structure contraction by visiting each transition only once; thus, structure contraction takes time proportional to the number of transitions. Then, the complexity of this algorithm is O (|R|).

Example 1: The weighted Kripke structure shown in Figure 1-a (the weights are all equal to one) is abstracted using the described algorithm with the following visit order: 14, 8, 5, 11, 1, 0, 2, 10, 6, 13, 15, 16, 3, 9, 18, 17, 4, 7, 12. The result of matching is illustrated by colouring the matched states.

The weighted Kripke structure shown on bottom of Figure 1 is the result of the contraction stage described in the algorithm of abstraction. The state weights are shown beside the circles representing the states and the transition weights are shown beside the arcs.



(a) Matching



(b) Contraction

Figure 1. Example of the abstraction: (a) Matching and (b) Contraction.

4. Partitioning Phase

The second phase of the partitioning algorithm computes a high-quality partition (i. e., small *TransitionCut* number) P_r of the coarse weighted Kripke structure $K_r = (S_r, I_r, R_r, L_r)$ such that each part contains roughly N/M of the state weight of the original structure. Since during abstracting, the weights

of the states and transitions of the coarser structure were set to reflect the weights of the states and transitions of the finer structure, K_r contains sufficient information to intelligently enforce the balanced partition and the small *TransitionCut* number requirements.

The Karnighan-Lin partitioning algorithm [12] is a known iterative procedure which starts with an initial partition and attempts to improve the solution at each step by swapping a pair of states from the two parts. Here, we present a different constructive partitioning algorithm which attempts to group strongly interconnected states (i. e., states among which there are many interconnections) into parts. We can define C_{ij} the number of connections (transitions) between states s_i and s_j. $C_{ij} = ctw(s_i) + ctw(s_j) + w((s_i, s_j)) + w((s_j, s_i))$. A set S of states has the weight $\sum_{si, sj \in S} C_{ij}$. A partitioning algorithm usually attempts to find an admissible part of large weight. The effect of finding parts with large weights is to decrease the number of interconnections between parts.

Algorithm 2: Partitioning ($K_r = (S_r, I_r, R_r, L_r)$)

```
{
  m ← 0
  while Sr ≠ ∅
  {
    Select a state si from Sr such that
     ∀sj ∈ Sr ∧ si ≠ sj: Cij is the maximum
    Pr (si) ← m
    Sr ← Sr\ {si}
    repeat
      Select sk ∈ Sr such that
      ∑sj|P(sj) = m Ckj is maximized
      In case of ties, select the state with the minimum
         total number of connections. This will tend to
         decrease inter-part connections.
      Pr (sk) ← m
      Sr ← Sr\{sk}
    until ∑s| P (s) = m w (s) ≥ N / M ∨ Sr = ∅
    m++
  }
  return Pr
}
```

Example 2: Figure 2 shows the result of partitioning the abstracted weighted Kripke structure of Figure 1-b to three partitions. This corresponds to the partitioning function P_r returned by the partition algorithm and which is defined as follows. $P_r(s) = a \mid s \in \{0, 1, 3\}$, $P_r(s) = b \mid s \in \{4, 7, 9\}$, $P_r(s) = c \mid s \in \{2, 5, 6, 8, 10\}$.

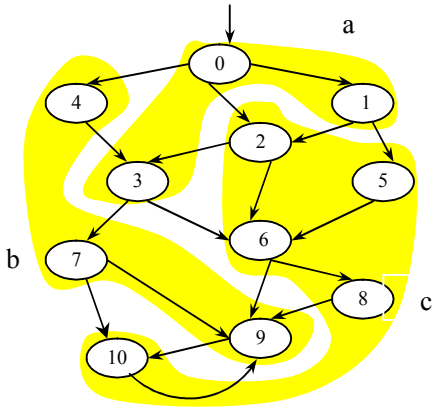


Figure 2. Partitioning to three partitions.

5. Refinement Phase

During the refinement phase, the partition P_r of the coarser weighted Kripke structure K_r is projected back to the original weighted Kripke structure, by going through the structures $K_{r-1}, K_{r-2}, \ldots, K_0$. Since each state of K_{i+1} contains a distinct subset of states of K_i, obtaining P_i from P_{i+1} is done by simply assigning the set of states S^s_i collapsed to $s \in K_{i+1}$ to the partition $P_{i+1}(s)$ (i. e., $P_i(s') = P_{i+1}(s)$, $\forall s' \in S^s_i$). Even though P_{i+1} is a local minimum partition of K_{i+1}, the projected partition P_i may not be at a local minimum with respect to K_i. Since K_i is finer, it has more degrees of freedom (more detailed information was abstracted) that can be used to improve P_i, and decrease the *TransitionCut* number. Hence, it may still be possible to improve the projected partition of K_{i+1} by local refinement heuristics. For this reason, after projecting a partition, a partition refinement algorithm is used. The basic purpose of a partition refinement algorithm is to select two subsets of states, one from each part so that when swapped the resulting partition has a smaller *TransitionCut* number. Specifically, if A and B are the two parts of a partition, a refinement algorithm selects $A' \subset A$ and $B' \subset B$ such that $A \setminus A' \cup B'$ and $B \setminus B' \cup A'$ yields a partition with a smaller *TransitionCut* number.

Consider a weighted Kripke structure $K_i = (S_i, I_i, R_i, L_i)$, and its partitioning function P_i. For each state s (Si we define the neighbourhood N(s) of s to be the union of the partitions that the states adjacent to s (i. e., Adj (s)) belong to. That is, N (s) = (s'(Adj (s) Pi (s'). Note that if s is an interior state of a partition, then N (s) = (. On the other hand, the cardinality of N (s) can be as high as Adj (s), for the case in which each state adjacent to s belongs to a different partition. During refinement, s can move to any of the partitions in N (s). For each state s we compute the gains of moving s to one of its neighbour partitions. In particular, for

every b (N(s) we compute EDbi (s) as the sum of the weights of the transitions (s, s') (Ri and the weights of the transitions (s', s) (Ri such that Pi (s') = b (if the partition b is not specified, EDi(s) is computed for all neighbourhood partitions). Also we compute IDi(s) as the sum of the weights of the transitions (s, s') (Ri and the weights of the transitions (s', s) (Ri, such that Pi(s') = Pi(s). The quantity EDbi(s) is called the external degree of s to partition b (EDi(s) is the external degree of s), while the quantity ID(s) is called the internal degree of s. Given these definitions, the gain of moving state s to partition b (N(s) is gb (s) = EDbi(s) - IDi(s).

However, in addition to decreasing the *TransitionCut* number, moving a state from one partition to another must not create partitions whose size is unbalanced. In particular, our partitioning refinement algorithm moves a state only if it satisfies the following balancing condition. Let W_i: {0, …, M-1} $\rightarrow$ N (N is the set of natural numbers) be a total function, so that W_i (a) is the weight of partition a of Kripke structure K_i, and let *BF* be the balancing factor ($0 \le BF \le 1$). A state s, whose weight is w (s) can be moved from partition a to partition b only if

$$W_i(b) + w(s) \le (1 + BF) * |S_0| / M$$
$$And$$
$$W_i(a) - w(s) \ge (1 - BF) * |S_0| / M$$

We note this condition by $BC_{[a/b]}$ (s). This condition ensures that movement of a node into a partition does not make its weight higher than $(1 + BF) * |S_0| / M$ or less than $(1 - BF) * |S_0| / M$. Note that by adjusting the value of *BF*, we can vary the degree of imbalance among partitions. If BF = 0, then the refinement algorithm tries to make each partition of equal weight. In our experiments we found that letting *BF* to be about 5%, tends to improve the quality of the partitioning and it minimizes the load imbalance.

Algorithm 3: Refinement ($K_{i+1} = (S_{i+1}, I_{i+1}, R_{i+1}, L_{i+1})$, P_{i+1}, ED_{i+1}, ID_{i+1})

{
  Projection:
  for each $s \in S_i$
  $\{P_i(s) \leftarrow P_{i+1}(Map_i[s])\}$
  for each $s \in S_{i+1}$
  {
    if $(s_1, s_2 \in S_i \wedge Map_i[s_1] = s \wedge Map_i[s_2] = s)$
    {
      if $(ED_{i+1}(s) = 0)$
      {
        $ED_i(s_1) \leftarrow 0$
        $ED_i(s_2) \leftarrow 0$
        $ID_i(s_1) \leftarrow aw(s_1)$
        $ID_i(s_2) \leftarrow aw(s_2)$
      }
      if $(ID_{i+1}(s) = 0)$
      {
        $ID_i(s_1) \leftarrow ctw(s) - ctw(s_1) - ctw(s_2)$
        $ID_i(s_2) \leftarrow ctw(s) - ctw(s_1) - ctw(s_2)$
        $ED_i(s_1) \leftarrow aw(s_1) - Idi(s_1)$
        $ED_i(s_2) \leftarrow aw(s_2) - Idi(s_2)$
      }
      if $(ED_{i+1}(s) > 0$ (IDi + 1(s) > 0)
      {
        IDi (s1) (
        $((s1, s') \in Ri \wedge Pi(s_1) = Pi(s')\ w((s_1, s')) +$
        $\sum_{(s', s1) \in Ri \wedge Pi(s1) = Pi(s')} w((s', s_1))$
        $ED_i(s_1) \leftarrow$
        $\sum_{(s1, s') \in Ri \wedge Pi(s1) \neq Pi(s')} w((s_1, s')) +$
        $\sum_{(s', s1) \in Ri \wedge Pi(s1) \neq Pi(s')} w((s', s_1))$
        $ED_i(s_2) \leftarrow ED_{i+1}(s) - ED_i(s_1)$
        $ID_i(s_2) \leftarrow ID_{i+1}(s) - ID_i(s_1) -$
        $w((s_1, s_2)) - w((s_2, s_1))$
      }
    }
  }
  Refinement:
  The states in S_i *are checked in random order*
  for each state $s \in S_i$
  {
    if $(N(s) \neq \varnothing)$
    {
      $N'(s) \leftarrow \{p \in N(s) \mid BC_{[Pi(s)/p]}(s) = true\}$
      $ED^a{}_i(s) \leftarrow max\{ED^b{}_i(s) \mid b \in N'(s)\}$
      if $((ED^a{}_i(s) > ID_i(s)) \vee (ED^a{}_i(s) = ID_i(s) \wedge$
      $W_i(P_i(s)) - W_i(a) > w(s)))$
      {
        $P_i(s) \leftarrow a$
        Update (ED_i, ID_i)
      }
    }
  }
  return P
}

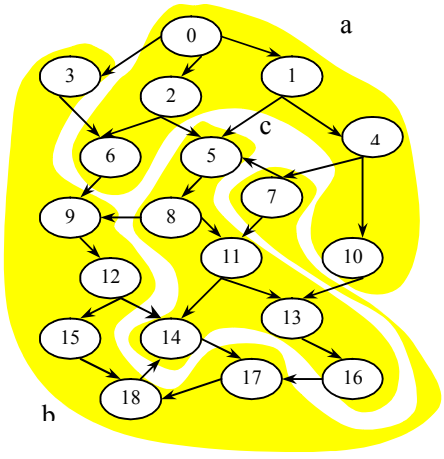
The refinement consists of two separate stages: The partition P_{i+1} of the weighted Kripke structure K_{i+1} is projected back to K_i, then P_i is refined. Refinement is swapping states between partitions which is based on the reduction in the transitions crossing the border. The selections of states to be swapped are driven by the gain value of the states. The gain values are computed using two arrays ID and ED where for each state s,

$$ID[s] = \sum_{(s, s') \in R} (P(s) = P(s')\ w((s, s')) + \sum_{(s', s) \in R \wedge P(s) = P(s')} w((s', s)).$$

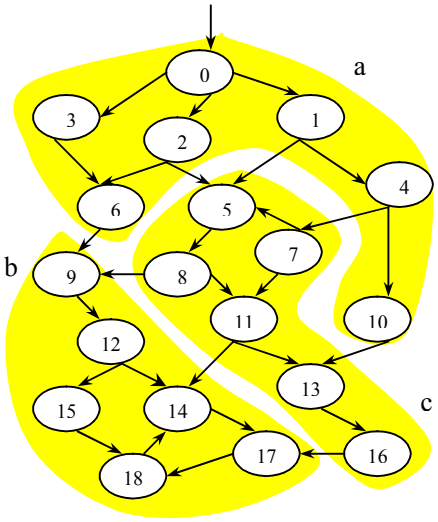
$$ED[s] = \sum_{(s, s') \in R \wedge P(s) \neq P(s')} w((s, s')) + \sum_{(s', s) \in R \wedge P(s) \neq P(s')} w((s', s)).$$

Given these arrays, the gain of a state s is given by g_s = ED [s] – ID [s]. The *TransitionCut* number is given by ($\sum_s$ED [s]) / 2. In our implementation, after partitioning the coarse weighted Kripke structure K_r, the internal (ID) and external (ED) degrees of the states of K_r are computed. The internal and external degrees of all other structures K_i with i < r, are computed incrementally during the projection stage. That is, the algorithm moves s to a partition that leads to the largest reduction in the cross border transition without violating the balance condition. If no reduction in the transition crossing the border is possible, by moving s, then s is moved to the partition (if any) that leads to no increase in the transitions crossing the border but improves the balance. After moving state s, the algorithm updates the internal and external degrees of the states adjacent to s to reflect the change in the partition.

Example 3: Figure 3 shows the projection of the partition illustrated in Figure 2. For refinement, we assume the balancing factor BF = 5%. N (3) = {a}, N' (3) = {a}, ID (3) = 0, and ED^a (3) = 2 then the state 3 can be moved from the partition b to the partition a. The other states that can be moved in this refinement is the states 7 and 14. Figure 3-b shows the result of this refinement. The gain is a very balanced partition with reduction of the *TransitionCut* number by 6.



(a) Projection.



(b) Refinement.

Figure 3. Projection and refinement.

6. Experimental Results

We experimented the implementation of the partitioning algorithm on three state spaces of three industrial-sized protocols:

1. The HAVI protocol [14], standardized by several companies, among which Philips, in order to solve interoperability problems for home audio-video networks. HAVI provides a distributed platform for developing applications on top of home networks containing heterogeneous electronic devices and allowing dynamic plug-and-play changes in the network configuration. We considered a configuration of the HAVI protocol with 2 device control managers which has 1,039,017 states and 3,371,039 transitions.
2. The TOKENRING leader election protocol [8] for unidirectional ring networks. We considered a configuration of the TOKENRING protocol with 3 stations. This configuration has 12,362,489 states and 45,291,166 transitions.
3. The arbitration protocol for the SCSI-2 bus [1], which is designed to provide an efficient peer-to-peer I/O bus for interconnecting computers and peripheral devices (magnetic and optical disks, tapes, printers, etc.). We have used a specification with 6 disks (1,202,208 states and 13,817,802 transitions).

Figure 4 shows the time taken by each phase (abstraction, partition and refinement) for partitioning the state space of the HAVI protocol. We have abstracted the whole space to 5000 states. We have partitioned the state space to 2, 4, 6 and then 8 partitions. The abstraction phase takes approximately 8 minutes. The partition and refinement phases are faster.

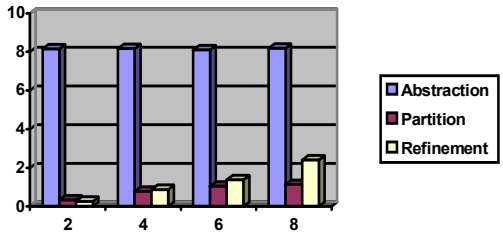


Figure 4. Partitioning time of the HAVI protocol to 2, 4, 6, and 8 partitions.

Figure 5 shows the percentage of cross-border transitions for the three systems. The average percentage is about 12%. This partitioning has small percentage of cross-border transitions compared to other reported results [2].

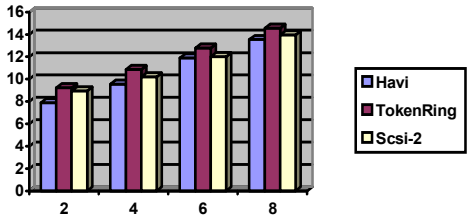


Figure 5. Transition cut of the three state spaces.

7. Conclusion

This paper presented a solution to the state explosion problem during the analysis of real-size concurrent transition systems. This solution is based on new scheme for partitioning the state space to parts. These parts will be used by a network of processes running in parallel. This algorithm is designed by a way reducing the communication overhead between the different processes.

Our concentration for the partitioning algorithm was the production of high quality partition. Its adaptability makes it suitable for exploiting the resources of very large environments. Other objective of our partitioning algorithm was the reduction of the cross-border transitions during the refinement.

References

[1] ANSI, "Small Computer System Interface-2," *Standard X3.131-1994*, January 1994.

[2] Bell A. and Haverkort B. R., "Sequential and Distributed Model Checking of Petri Net Specifications," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 4, 2002.

[3] Bourahla M. and Benmohamed M., "Predicate Abstraction and Refinement for Model Checking VHDL State Machines," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, 2002.

[4] Brim L., Crhova J., and Yorav K., "Using Assumptions to Distribute CTL Model Checking," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 4, 2002.

[5] Bui T. and Jones C., "A Heuristic for Reducing Fill in Sparse Matrix Factorization," *in Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pp. 445-452, 1993.

[6] Clarke E. M., Grumberg O., and Long D. E., "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512-1542, 1994.

[7] Deo N., *Graph Theory with Applications to Engineering and Computer Science*, Automatic Computation, Prentice Hall, 1974.

[8] Garavel H. and Mounier L., "Specification and Verification of Various Distributed Leader Election Algorithms for Unidirectional Ring Networks," *Science of Computer Programming*, vol. 29, no. 1, pp. 171-197, 1997.

[9] Hendrickson B. and Leland R., "A Multilevel Algorithm for Partitioning Graphs," *Technical Report SAND93-1301*, Sandia National Laboratories, 1993.

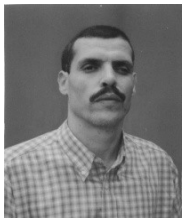
[10] Heyman T., Geist D., Grumberg O., and Schuster A., "Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits," *Formal Methods in System Design*, vol. 21, no. 2, pp. 317-338, 2002.

[11] Karypis G. and Kumar V., "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, 1998.

[12] Kernighan B. W. and Lin S., "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291-307, 1970.

[13] Papadimitriou C. H. and Steiglitz K., *Combinatorial Optimization*, Prentice Hall, 1982.

[14] Romijn J., "Model Checking the HAVi Leader Election Protocol," *Technical Report SEN-R9915*, CWI, Amesterdam, The Netherlands, June 1999.



Mustapha Bourahla has been a teacher-researcher since 1994 at the Computer Science Department, University of Biskra, Algeria. He was the Computer Science Department head, University of Biskra. He has been a member of the scientific committee since 1999. He holds MSc degree in computer science from the University of Montreal, Canada, 1989. He was a member of VHDL research group at Bell-Northern Research, Ottawa, Canada from

1989 until 1993. He is expected to finish his PhD in 2005.