Partitioning State Spaces for Distributed Model Checking

Mustapha Bourahla[1] and Mohamed Benmohamed[2]

[1]Computer Science Department
University of Biskra
BP 145 RP, Biskra 07000
ALGERIA
mbourahla@hotmail.com

[2]Computer Science Department
University of Constantine
Constantine, 25000
ALGERIA
ibnm@yahoo.fr

ABSTRACT. As the model-checking becomes increasingly used in the industry, 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 with hundreds of millions of states and transitions. The produced partitions will be used by distributed processes for parallel system analysis. This algorithm partitions the state space by performing a combination of abstraction-partition-refinement on its structure. The algorithm is designed by a way reducing 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.

RÉSUMÉ. Comme la vérification par modèle est devenue de plus en plus utilisée dans l'industrie, il y a un grand besoin des nouvelles méthodes pour faire face à des grands espaces d'états des systèmes concurrents. Nous proposons un nouveau algorithme pour le partitionnement des espaces d'états modélisant des conceptions industrielles contenant des centaines de millions d'états et de transitions. Les partitions produites seront utilisées par des processus distribués pour une analyse parallèle du système. Cet algorithme partage l'espace d'états en exécutant une combinaison d'abstraction, partitionnement, et raffinement sur sa structure. Les résultats expérimentaux montrent que cette méthode améliore la qualité des partitions, les coûts de la communication et puis la performance totale de l'analyse du système.

KEYWORDS: Distributed Model Checking, Abstraction, Partitioning, Refinement.

MOTS-CLÉS : Vérification Distribuée par Modèle, Abstraction, Partitionnement, Raffinement.

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 [5, 2]. Recently, a new promising method to tackle the state space explosion problem was introduced [8, 1, 3]. 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 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 of M machines. 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 of weighted Kripke structure (this is an extension of the Kripke structure where weights are associated with the states and with the transitions). 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.

2. Partioning State Spaces

The partitioning problem is defined as follows: Given a Kripke structure $K=(S,R)$ modelling a concurrent transition system, where S is a non-empty finite set of states, $R \subseteq S \times S$ is a total transition relation (i.e., $(\forall s \in S : (\exists s' \in S : (s,s') \in R))$), with $|S|=N$, partition S into M subsets, $S_0, \ldots, S_{M-1}$ such that $S_i \cap S_j = \emptyset$ for $i \neq j$, $|S_i| \cong N/M$, 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 $K=(S, R)$ 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 such 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 $P : S \rightarrow \{0, \ldots, M-1\}$, such 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 [7, 9, 10]. Formally, the partitioning algorithm works as follows: Consider a weighted Kripke structure $K_0 = (S_0, R_0)$, with weights both on states and transition edges. A partitioning algorithm consists of the following three phases.

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

The data structure used to store the state space consists of two tables. The first called *StateTable*, it stores information about states and the second 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`, `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 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 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 outgoing or 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`.

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_l to form a single state of the next level coarser structure K_{l+1}. Let S^s_l be the set of states of S_l combined to form state s of K_{l+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_l. 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_l. In the case where more than one state of S^s_l 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 $K_l = (S_l, R_l)$, 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 [6, 4, 7]. Thus, the next level coarser weighted Kripke structure K_{l+1} is constructed from K_l by finding a matching of K_l and collapsing the states being matched into multi-nodes. The unmatched states are simply copied over to K_{l+1}.

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*, such 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(`$K_l = (S_l, R_l)$`)`

```
{
 Matching:
 Create a random list RS of all the states in StateTable
 representing the set S_l
 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 ∪ {Σ_{(s',s'')|s''∈Adj(s)} w(s', s'') +
              Σ_{(s'',s')|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} ← ∅; R_{i+1} ← ∅; index ← 0
for each k ∈ {0, …, j-1} {
 S_{i+1} ← S_{i+1} ∪ {T[k]}
 SuccSet ← {Map[s]|s ∈ S_i ∧ Map[s] ≠ T[k] ∧
    ∀s'∈S_i : T[k] = Map[s'] ∧ (s',s)∈R_i}
 is(T[k]) ← index; ns(T[k]) ← Length(SuccSet)
 index ← index + ns(T[k])
 PredSet ← {Map[s]|s ∈ S_i ∧ Map[s] ≠ T[k] ∧
   ∀s' ∈ S_i : T[k] = Map[s'] ∧ (s, s') ∈ R_i}
 ip(T[k]) ← index; np(T[k]) ← Length(PredSet)
 index ← index + np(T[k])
 R_{i+1} ← R_{i+1} ∪ SuccSet ∪ PredSet
 sw(T[k]) ← sw(s_1) + sw(s_2) s.t. T[k] = Map[s_1] = Map[s_2]
 ctw(T[k]) ← ctw(s_1)+ctw(s_2)+w((s_1,s_2))+w((s_2,s_1)) s.t.
    T[k] = Map[s_1] = Map[s_2]
 aw(T[k]) ← aw(s_1) + aw(s_2) - w((s_1,s_2))-w((s_2,s_1)) s.t.
    T[k] = Map[s_1] = Map[s_2]
}
for each transition (q_1, q_2) ∈ R_{i+1} s.t.
 q_1 = Map[s_1] ∧ q_1 = Map[s_2] {
```

```
    w({q_i, q_j}) ← Σ_{s∈Map(s_i)∩Q'} w({s_i,s}) + Σ_{s∈Map(s_j)∩Q'} w({s_j,s})
  }
  return (S_{i+1}, R_{i+1})
}
```

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 $K_r=(S_r,R_r)$ 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.

We present our constructive partitioning algorithm which attempts to group strongly interconnected states 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 $\sum_{s_i,s_j\in S} C_{ij}$. 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($K_r = (S_r, R_r)$)

```
{
  m ← 0
  while S_r ≠ ∅ {
    Select a state s_i from S_r such that
    ∀s_j ∈ S_r ∧ s_j ≠ s_i : C_ij is the maximum
    P_r(s_i) ← m; S_r ← S_r\{s_i}
    repeat
      Select s_k ∈ S_r such that Σ_{s∈Post(s_k)∩S_r} C_kj is maximized
      In case of ties, select the state with the minimum
      total number of connections. This will tend to
      decrease inter-part connections.
      P_r(s_k) ← m; S_r ← S_r\{s_k}
    until Σ_{s∈P_r(m)} w(s) ≥ N/M ∨ S_r = ∅
    m++
  }
  return P_r
}
```

5. Refinement Phase

During the refinement phase, the partition P_z of the coarser weighted Kripke structure K_z is projected back to the original weighted Kripke structure, by going through the structures K_{z-1}, K_{z-2}, ..., 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 such that when swapped the resulting partition has a smaller *TransitionCut* number.

Consider a weighted Kripke structure $K_i=(S_i, R_i)$, and its partitioning function P_i. For each state $s \in S_i$ 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, $N(s)=\cup_{s' \in Adj(s)} P_i(s')$. Note that if s is an interior state of a partition, then $N(s)=\emptyset$. 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 $b \in N(s)$ we compute $ED^b_i(s)$ as the sum of the weights of the transitions $(s,s') \in R_i$ and the weights of the transitions $(s',s) \in R_i$ such that $P_i(s')=b$ (if the partition b is not specified, $ED_i(s)$ is computed for all neighbourhood partitions). Also we compute $ID_i(s)$ as the sum of the weights of the transitions $(s,s') \in R_i$ and the weights of the transitions $(s',s) \in R_i$ such that $P_i(s')=P_i(s)$. The quantity $ED^b_i(s)$ is called the external degree of s to partition b ($ED_i(s)$ 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 $b \in N(s)$ is $g^b(s)= ED^b_i(s) - ID_i(s)$.

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 $W_i : \{0, \ldots, M-1\} \rightarrow N$ (N is the set of natural numbers) be a total function, such that $W_i(a)$ is the weight of partition a of Kripke structure K_i, and let *BF* be the balancing factor ($0 \leq BF \leq 1$). A state s, whose weight is $w(s)$ can be moved from partition a to partition b only if

$W_i(b)+w(s) \leq (1+BF)*|S_i|/M$ and $W_i(a)-w(s) \geq (1-BF)*|S_i|/M$.

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 $(1+BF)*|S_i|/M$ or less than $(1-BF)*|S_i|/M$. Note that by adjusting the value of *BF*, we can vary the degree of imbalance among partitions.

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

```
{
 Projection:
 for each s ∈ S_i |P_i(s) ← P_{i+1}(Map_i[s])|
 for each s ∈ S_{i+1} {
  if (s_1, s_2 ∉ S_i ∧ Map_i[s_1] = s ∧ Map_i[s_2] = s) {
   if (ED_{i+1}(s) = 0) |ED_i(s_1) ← 0; ED_i(s_2) ← 0;
    ID_i(s_1) ← aw(s_1); ID_i(s_2) ← aw(s_2)|
   if (ID_{i+1}(s) = 0) |ID_i(s_1) ← ctw(s)-ctw(s_1)-ctw(s_2);
    ID_i(s_2) ← ctw(s)-ctw(s_1)-ctw(s_2)
    ED_i(s_1) ← aw(s_1)-ID_i(s_1); ED_i(s_2) ← aw(s_2)-ID_i(s_2)|
   if (ED_{i+1}(s)>0 ∧ ID_{i+1}(s)>0) |
    ID_i(s_1) ← Σ_{(s_1,s')∈E∧s'∈V_i} w((s_1,s')) +
                Σ_{(s',s)∈E∧s'∈V_i} w((s',s_1))
    ED_i(s_1) ← Σ_{(s_1,s')∈E∧s'∈V_i} w((s_1,s')) +
                Σ_{(s',s_1)∈E∧s'∈V_i} w((s',s_1))
    ED_i(s_2) ← ED_{i+1}(s) - ED_i(s_1)
    ID_i(s_2) ← 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 ∈ S_i {
  if (N(s) ≠ ∅) (N'(s) ← {p ∈ N(s)|BC_{(P_i(s),b)}(s) = true}
   ED*_i(s) ← max{ED^b_i(s)|b ∈ N'(s)|
   if ((ED*_i(s) > ID_i(s)) ∨
       (ED*_i(s) = ID_i(s) ∧ W_i(P_i(s)) - W_i(a) > w(s))) |
           P_i(s) ← a; Update(ED_i, ID_i)|
  }
 }
 return (P_i, ED_i, ID_i)
}
```

6. Conclusion

This paper presented a new scheme for partitioning the state space of concurrent transition systems to parts. Our concentration was the production of high quality partition and the reduction of the cross-border transitions during the refinement. The results of our experiments show that this new scheme produces good quality of partitioning that can be compared to other efficient approaches and it has small percentage of cross-border transitions.

References

1. 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.
2. 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.
3. Brim L., Crbova J., and Yorav K., "Using Assumptions to Distribute CTL Model Checking," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 4, 2002.
4. Bui T. and Jones C., "A Heuristic for Reducing Fill in Sparse Matrix Factorization," *In 6th SIAM Conf. Parallel Processing for Scientific Computing*, pp. 445-452, 1993.
5. 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.
6. Deo N., *Graph Theory with Applications to Engineering and Computer Science*, chapter 8. Automatic Computation. Prentice Hall, 1974.
7. Hendrickson B. and Leland R., "A Multilevel Algorithm for Partitioning Graphs," *Technical Report SAND93-1301*, Sandia National Laboratories, 1993.
8. Heyman T. *et al.*, "Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits," *Formal Methods in System Design*, vol. 21, no. 2, pp. 317-338, 2002.
9. Karypis G. and Kumar V., "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, 1998.
10. 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.
11. Papadimitriou C. H. and Steiglitz K., *Combinatorial Optimization*. Prentice Hall, 1982.