Efficient CSP Model Checking for Symbolic State-space Exploration with Application in Bacterial Chemo-Taxis Modelings

Jing GUO (guo_jing163@163.com)
Jinling Institute of Technology

Software article

Keywords: Saturation checking, Event locality, Biological model, CSP theory, Concurrent system

Posted Date: November 17th, 2020

DOI: https://doi.org/10.21203/rs.3.rs-105136/v1

Efficient CSP Model Checking for Symbolic State-space Exploration with application in bacterial chemo-taxis modelings

JING GUO*
School of Software Engineering, Jinling Institute of Technology, Nanjing 211169, P. R. China
Email: guo_jing@jit. edu. cn

November 7, 2020

Abstract

Event locality is a class of theory that can clearly minimizes the memory and time required to store the state space. We present the multi-way decision diagrams to encode the next-state functions based on event locality and CSP's stable failure. Instead of studying symbolic overall model, we apply the above thinking to the set of sub-models. Furthermore, we focus on saturation algorithms for CSP theory. And the algorithms consist of two phases, the universal process's saturation checking and the single CSP symbol's saturation checking. In the former case, the algorithms discuss better iteration strategy and recursive local fix-point computations. In the latter case, the the single CSP symbol's saturation checking calls the former algorithms, being combined preformed events and refusal events. Finally, we discuss bacterial chemotaxis modeled in CSP's framework and saturation checking used to check the predefined properties. The all algorithms are implemented in CSP tool FDR, the running results show that our algorithms often perform significantly faster than other conventional algorithms.

Keywords: Saturation checking, Event locality, Biological model, CSP theory, Concurrent system

1 Introduction

For safety-critical systems, such as railway system and flight system, small failure may lead to huge economic losses and even threaten the human life. Such systems require a relatively high reliability. However, testing and debugging are traditional method to verify the key properties in these systems, and only a little behaviors of systems can be examined. This requires researchers to employ new available means to overcome difficulties.

Formal methods are differentiated from other methods in that their specification and verification are based on a rigorous mathematical foundation, which guarantees a high degree of correctness. The internal algorithms in verification rely on the knowledge of reachable state space of systems. Unfortunately, many of them do not support very large state space. So researchers fall into two categories:explicit techniques and symbolic techniques, according to whether state-space are generated and represented. Explicit techniques store states by graphs, trees, where each state is explored linearly. Symbolic techniques store sets of states by BDDs (Binary Decision Diagrams), where the reachable set of states are explored sub-linearly. But the combinatorially expanding state space is solved difficultly for both techniques. There are some progresses in symbolic techniques, but the limitations of memory and time are still remain.

Researchers have addressed the limitations in a number of aspects. One direction has focused on disjunction-partitioned transitions, which shift from global transitions to local transitions. The local transitions mean only finding part of transitions, rather than all of transitions. They have touched on the concept of event locality inherent in asynchronous systems. Event locality shows that researchers can explore the given subsystems which can be fired by some events [1]. Furthermore, it is applied in the area of generating state-space recently, also studied by MDDs [2–4]. The application breaks through the restriction of the traditional symbolic techniques that explore reachable states and often is limited to a relatively specific range.

The above studying target the reduction of practicable state-space, whereas the saturation based algorithms are committed to finding the set of states satisfy the properties. In contrast with breath-first search, they can employ the idea of "event locality", named "saturation-based iteration strategy" [5–8]. The advantages of strategy are clear, that are they can speed up the operation by only exploit the relevant sub-models [9–12].

*This paper was supported by High-level Talents Research Project of Jinling Institute of Technology (No. : jit-b-201817) and Universities Philosophy and Social Science Researches in Jiangsu Province (2020SJA0534).

CSP is a high-level formal language whose fundamental thought is the refinement of processes due to traces and failures. The CSP theory has advantages on describing the various interactions and verifying the concurrent properties of processes. In this paper, we attempt to apply the saturation-based iteration strategy to CSP model checking. It may been put into full application through the two technologies of Kronecker encoding of the next-state function and saturation-based iteration strategy [13]. The former technology need professional mathematics knowledge [14,15], and the steps are more complicated. Thus, we try to reduce unnecessary steps. In view of the fact that an modified constrained saturation based exploration has state-insertion operations, we incorporate them into the framework of the CSP's traces and stable failure.

The remainder of this paper is organized as follows. Sect. 2 gives the background on CSP, MDDs, discrete-state model. In Sect. 3, we use next-event functions instead of next-state functions in MDDs, where the locality theory is based on stable failure. Sect. 4 describes the conventional checking for CSP. Sect. 5 proposes our framework of saturation checking of CSP and provides the detailed algorithms on it. Sect. 6 proposes the application on the biological system-arterial chemotaxis. In Sect. 7, the new approaches are apply to typical models. According to the results reported in the table 1, our approaches are more memory-efficient and requires less time. Finally, Sect. 8 states our conclusions and future research thinking.

2 Background and related work

2.1 Communicating sequential process (CSP)

In CSP, the event is a smallest unit that allows to describe the action of system, and the process consists of many events. Event occurs instantaneously and lasts a very short time, so we do not care about the duration, but rather, temporal logic is what we are concerned with. In this paper, we write the event in lower-case letters and the process in capitals. In the aspect of syntax, the CSP defines the structure of each symbol in detail. For example,

- $a \to P$: The process first executes the event a, then behaves like process P. If the process does nothing, the P is never performed.
- $x : A \to P\,(x)$: The process first executes a event of event set A, then behaves like $P(x)$ according to what event of A is firstly choosed.
- $c!v \to P/c?x : T \to P\,(x)$: The input event $c?x : T$ or the output event $c!v$ are executed before the P or $P(x)$, respectively.
- $P\backslash A$: In addition to hiding the event set A, the process performs P.

  So far, the symbols we have described above are related to one process whose events are given in a linear fashion. In the following, we introduce the processes which give alternative behaviors.
- $P\Box Q$: $P\Box Q$ executes the events of P(or Q)sequentially if the first external event is from P(or Q).
- $P \sqcap Q$: There is non-deterministic internal choice involved in the $\sqcap$ operator. $P \sqcap Q$ behaves like P or Q non-deterministically. The first event is internal event, which is not exposed to the interface.
- $P\|Q$: Suppose process P contains event set E_P, and process Q contains event set E_Q. $\|$ means P and Q execute in parallel, but both processes agree on the events in $E_P \cap E_Q$.
- $P_A\|_B Q$: P and Q perform the event set B and the event set A, respectively, but processes agree on the events in $A \cap B$.
- $P|||Q$: The process carries out P and Q independently. But P and Q do not communicate any common events.
- $P_1; P_2$: Process P_1 and P_2 are executed sequentially. For environment, P_1 is offered firstly and then P_2 is given. $P_1; P_2$ perform P_2 until P_1's execution is completed.
- $P_1 \Delta P_2$: In $P_1 \Delta P_2$, P_2 can interrupt P_1's execution at any time.

In the above theoretical framework, we introduce the different operators from the view of process. But this theory alone is not broad enough to describe the behaviour of system. So refusals set is introduced. Once the events are chosen, there are some events that not required to perform. These events are refusal events.

The refusal theory is more interested in the refusal event set after performing the trace. So to describe the idea, we introduce the pair (Tr, X) which indicates that the process first communicates the events of trace Tr then to refuse the event set X. It is defined as follows:

$$(\mathrm{Tr}, X) \in SF \Rightarrow Tr \in T,$$

$$(\mathrm{Tr}, X) \in SF \wedge X' \subseteq X \Rightarrow (\mathrm{Tr}, X') \in SF,$$

where SF is stable failure. It is easier to understood by the following expressions:

$$\exists P'' \cdot P \xrightarrow{\mathrm{Tr}} P'' \wedge P'' \downarrow \wedge P'' \ \text{ref}\ X,$$

$$P'' \ \text{ref}\ X \xrightarrow{\mathrm{Tr}} \exists P' \wedge P'' \stackrel{\langle\rangle}{\Longrightarrow} P' \downarrow \wedge \forall a \in X \cdot \neg \left(P' \xrightarrow{a}\right).$$

Essentially, this implies that P'' refuses the event set X after P performs Tr and behaves like P''. However, P'' means that P'' is a stable process which contains no internal events.

2.2 Discrete-state model

Based on state model $\left\langle \hat{S}, s_{\mathrm{init}}, \aleph_\alpha, \varepsilon \right\rangle$, we define discrete-state model

$$\left\langle \hat{S}, s_{\mathrm{init}}, \aleph_{\alpha,X}, \varepsilon, X \right\rangle,$$

where $\hat{S}$ is the state space in which each $s \in \hat{S} = \langle s_k, s_{k-1}, ..., s_1 \rangle$, which implies the global state $s \in \hat{S}$ is partitioned into k local states for k submodels; the initial states $s_{init} \in \hat{S}$;ε is a finite set of asynchronous events;X is the refusal event set;the next-state function $\aleph_{\alpha,X}$ is based on $\aleph_\alpha$, which means that α is disabled before the event set X is refused in next-state function $\aleph$.

Accordingly, $\aleph_\alpha^{-1}$ and $\aleph_{\alpha,X}^{-1}$ express inverse next-state functions. In many cases, the next-state function $\aleph$ is expressed as a union $\aleph = \cup_{e \in \xi} \aleph(e, s)$, where $\aleph(e, s)$ is the set of successor states reachable from state s via event e. If $\aleph(e, s)$ holds, event e is enabled in s, otherwise, event e is disabled. The next-state $\aleph(e, s)$ can also expressed as the cross-product of k local functions, i. e. ,

$$\aleph(e, s) = \aleph_k(e, s_k) \times \aleph_{k-1}(e, s_{k-1}) \times ... \times \aleph_1(e, s_1).$$

The reachable state space is often referred to as the smallest set

$$S = \{s\} \cup \aleph(e, s) \cup \aleph(\aleph(e, s)) \cup ... = \aleph^*(e, s),$$

where s is usually the initial state s_{init} and S is a forward and transitive closure. Moreover, the relationship between local states and global states can be represented by encoding the characteristic function

$$f_s : \{0, 1, ..., \aleph_k - 1\} \times ... \times \{0, 1, ..., \aleph_1 - 1\} \to \{0, 1\},$$

$$f_s(s_k, s_{k-1}, ..., s_1) = \begin{cases} 1, & \text{if } (s_k, s_{k-1}, ..., s_1) \in S, \\ 0, & \text{otherwise.} \end{cases}$$

Instead of studying symbolic overall model, we apply the partition is a thought that enables us to use symbolic techniques based on decision diagrams.

2.3 Multi-way decision diagrams (MDDs)

In above introductions, we specify the finite-state asynchronous system. We are not restricted to the whole system, however, we can extend our notation to systems and subsystems, global states and local states, as well as next-state functions and inverse next-state functions.

BDDs are the most widely used directed, acyclic graphes that are widely used to store and manipulate sets of states. Instead of studying more details on BDDs, we employ the version of MDDs, which extend BDDs with discrete integer-valued variables. More specifically, they are in form of quasi-reduced canonical formats.

Formally, a quasi-reduced MDD is a directed acyclic edge-labeled graph, where

- Nodes are distributed in the several levels, which expressed as $\{k, ..., 1, 0\}$.
- Only one root node $\langle k|r \rangle$ is at level k, whereas one or more nodes belong to levels $k-1$ through 1. Two terminal nodes $\langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$ are at level 0.
- Several non-terminal nodes have n_k outgoing arcs at levels $k-1$ through 1. The n_k outgoing arcs are identified as $\{0, ..., 1, n_{k-1}\}$.

The node at level k is $\langle k|p\rangle$, where p is defined as index of node $\langle k|p\rangle$ to make it different from other nodes. Node $\langle k|p\rangle$ can be reached from a sequence $(i_k, ..., i_1) \in S_k \times ... \times S_1$, where we argue that $S_k \times ... \times S_1$ to represent cross-product, that is of k local state spaces:

$$\langle k|p\rangle [i_k, i_{k-1}, ..., i_1] = \langle k-1| (\langle k|p\rangle [i_k])\rangle [i_{k-1}, ..., i_1] .$$

The node at level k is recursively defined as

$$\langle k|p\rangle [i_k, i_{k-1}, ..., i_1] = \bigcup_{i_k \in S_k} \{i_k\} \times \langle k-1| \langle k|p\rangle [i_k]\rangle .$$

Due to the complexity, the model we often considered is partitioned into k smaller interacting sub-models. Likewise, the global state $s \in \hat{S}$ is also written as $s = \langle i_k, i_{k-1}, ..., i_1\rangle$, which stands for local states.

3 Next-event functions of MDDs

The next-state functions of MDDs are defined via concept of local transitions between states. We have Level L through 1 and Level L' through $1'$ defined that correspond to "from" states and"to" states. In asynchronous system, locality is a key property built on the fact that events are independent of most levels. Even a event fired, the value of i_k ($\forall i_k \in S_k$) may be not changed.

The local state analysis with parameter is that:

$$\aleph_{k,\alpha} : S_{k,\alpha} \to 2^{S_k},$$

which means the local influence for event α on sub-model k. $\aleph_{k,\alpha}$ is the next-state function corresponding to the event α of the event set ξ at level k.

The impact of event α on the global state is:

$$\aleph_\alpha (i_k, ..., i_1) = \aleph_{k,\alpha} (i_k) \times ... \times \aleph_{1,\alpha} (i_1) .$$

Conceptually, for the global state, we can compute each local state $(i_k, ..., i_1)$ at level $k, ..., 1$ encoding the effect of firing α in them.

In this paper, we use the next-event function to describe the next event's forward and backward transition instead of states. For convenience, the next-event function is also denoted by $\aleph$. The failure theory defines refusal event sets in processes and uses trace Tr to record the performed events. Process executes trace Tr before refusal event set. On infiltrating the idea of failure theory into next-state function to redefine the impact of trace Tr on the global state:

$$\aleph_{(tr,X)} (i_k, ..., i_1) = \aleph_{k,(tr,X)} (i_k) \times ... \times \aleph_{1,(tr,X)} (i_1) .$$

We denote transition $(\alpha_1 \to \alpha_2 \to ... \to \alpha_n)$, where n is the number of events contained in the trace, and $\alpha \in \varepsilon$. Additionally, we discuss global events $\alpha_1 = (\alpha_1^1, ..., \alpha_1^k)$,..., $\alpha_n = (\alpha_n^1 ..., \alpha_n^k)$ on the subsystems.

For instance, local event α_1^1 indicates the effect of the global event α_1 on the subsystem 1. Every global event is composed of k local events. Similar to this, global refusal event set X is expressed as $X = (X_1, ..., X_k)$, where $X_1, ..., X_k$ is local refusal event at level $1, ..., k$.

In the saturation strategy, we fire events node-wise, bottom-up per iteration. It has several key properties. Firstly, it allows a "from the button level to the top level" order of nodes which satisfies the assigned condition. Moreover, instead of firing events in level-wise heavyweight image computation of global strategies, we fire events in chained lightweight of local next-state function. The node α is saturated if a fix-point is reached with respect to firing any irrelevant events of all levels above k: $\forall k_1, k \geq k_1 \geq 1, \forall \alpha \in \varepsilon_{k_1}, S_k \times ... \times S_{k+1} \times B(\alpha) \geq \aleph_\alpha (S_k \times ... \times S_{k+1} \times B(\alpha))$ We encode sub-states below node α by

$$B(\alpha) = \bigcup_{i_k \in S_k} \{i_k\} \times B(\alpha [i_k]) ,$$

and k refers to the level of node α. The image computation $\aleph$ is the set of states:

$$\aleph (S_y) = \{j : \exists i \in S_y, (i, j) \in \aleph\} , S_y \in S_1, ..., S_k, 1 \leq y \leq k.$$

The locality and saturation apply to complicated asynchronous systems. We redefine the MDD encoding local next-event function and stable failure theory.

Denote by $Top(Tr, X)/Bot(Tr, X)$ the highest/maximum levels on which α affects, with

-The root node of MDD is selected from $Top(\alpha)$, $Top(A)$ and $Top(tr, X)$.
-$Top(\alpha)$ are the highest/maximum levels on which α affects.
-$Top(A)$ are the highest/maximum levels on which the event set A affects, with

$$Bot(A) = \min_{\alpha \in A} Bot(\alpha).$$

Affection means that its firing will change the value of i_k at level $k(1 \leq k \leq K)$. Let join the record of Tr and refusal event set in the transition. Denote by $Top(tr, X)/Bot(tr, X)$ the highest/lowest levels on which the stable failure (Tr, X) affects. The root node of MDD is selected from $Top(\alpha)$, $Top(A)$, $Top(Tr, X)$. Let

$$Inter(\alpha) = [Bot(\alpha), Top(\alpha)],$$

$$Inter(A) = [Bot(A), Top(A)],$$

$$Inter(tr, X) = [Bot(tr, X), Top(tr, X)].$$

The value of $Inter(\alpha)$, $Inter(A)$ and $Inter(tr, X)$ present the characteristics of discretization. Now, we can build a MDD for each event's transition. For the event MDDs can be merged by levels:
If the "from" state level $k \in Inter(\alpha)$, k appears on every path that α affects.
If the "to" state level $k' \in Inter(\alpha)$, k' could appear on every path that α affects, or not.

4 Conventional saturation checking for CSP

In Conventional checking of CSP, we formulate the process in 2 steps: trace Tr checking and refusal event set checking. In order to perform local events at each level on each sub-model, we encode local next-event functions each $\aleph_{k,\alpha}$ as a incidence matrices:

$$\aleph_{k,\alpha}[i_k, j_k] = 1 \Leftrightarrow j_k \in \aleph_{k,\alpha}(i_k).$$

Similarly, the inclusion holds at each level:

$$\begin{array}{cccc} \alpha_2^1 \in \aleph_{1,\alpha_1^1}(i_1), & \alpha_3^1 \in \aleph_{1,\alpha_2^1}(i_1), & ..., & \alpha_n^1 \in \aleph_{1,\alpha_{n-1}^1}(i_1) \\ \vdots & & & \vdots \\ \alpha_2^k \in \aleph_{k,\alpha_1^k}(i_k), & \alpha_3^k \in \aleph_{k,\alpha_2^k}(i_k), & ..., & \alpha_n^k \in \aleph_{k,\alpha_{n-1}^k}(i_k) \end{array}.$$

We explore the state-space by inverse next-event functions. For CSP, the elementary unit is process. So we start our checking from process checking. The stable failure of P is expressed as $(\mathrm{Tr}(P), X(P))$. The event array of P is defined as $\{\alpha_1 ... \alpha_n\}$ in chronological order, where n is the number of events. In fact, every event corresponds to a global state. After completing the process event sequences of P, the process refuses to perform any event contained in $X(P)$. $X(P)$ lists as $\left\{X_1^P, X_2^P, ..., X_m^P\right\}$, where m is the count of $X(P)$.

Algorithm 1 A traditional saturation checking procedure of P

$Sat_1(P, \mathrm{Tr}(P), X(P))$: set of states
Declare χ: set of states
$\chi \leftarrow \chi(\alpha_n)$
iff $\aleph(\chi) \cap X(P) = \phi$
then
  $i = n - 1$
  repeat
    get event from the event array of P in reverse order
    set $\mathrm{Tr}(P) = \{\alpha_1, ..., \alpha_n\}$
    $\chi \leftarrow \chi \cup \left(\aleph^{-1}(\chi) \cap \chi(\alpha_i)\right)$
    $i = i - 1$
  until χ dose not change
return χ

In Algorithm 1, we start from $\chi(\alpha_n)$, which is the set of states satisfying the last event of event array, then compute whether intersection of the next state set of α_n and refusal event set $X(P)$ is empty. After that, the intersection of the afterimage of the explored states and the set of states satisfying the i^{th} event of event array are added to the event set χ repeatedly until χ does not change. The new event is alive on the promise of the occurrence of previous events. The chain of events is arranged in the sequence of the time when event occurred.

In our approach, we assume that the process can be partitioned into n events. Each $enabled(P)$ is expressed as a conjunction of single enabled events. Hence, we express the enabled process, the condition under which the process is triggered successfully, as

$$enabled(P) \Leftrightarrow \Big(\aleph(\alpha_n) \cap X(P) = \emptyset \quad \rightarrow \\ enabled(\alpha_1, \emptyset) \wedge enabled(\alpha_2, \aleph(\alpha_1)) \wedge ... \wedge enabled(\alpha_n, \aleph(\alpha_{n-1}))\Big),$$

and

$$produce(P) \Leftrightarrow \alpha_2 \wedge \alpha_3 \wedge ... \wedge \alpha_n,$$

where $\alpha_i = produce(\alpha_{i-1}, \aleph(\alpha_{i-1})), i = 2, 3, ..., n.$

$$produce(P)^{-1} \Leftrightarrow \alpha_2 \wedge \alpha_3 \wedge ... \wedge \alpha_{n-1},$$

where $\alpha_j = produce^{-1}(\alpha_{j+1}, \aleph(\alpha_{j+1})), i = 1, 2, ..., n-1.$

To describe the connection between the new events and old events, we also study the chain of events by utilizing next-event functions. Note that this requires us to identify events(the member of process) that can occur, because the events actually executed are changed. A process for both the enabling events and the events that have bubbled to the surface, which only affects specific sets, is expressed by making use of the definition just introduced. One important point corresponds to the fact that environment allows to "enable" the events of P. In other words, the refusal set X and the events of P have no elements in common. The function "produce" means that the events satisfied equation really happened. Even if the events have been enabled, they may not happened.

5 Saturation checking for CSP

The next event α_j should not only be a "member" of process P, but also satisfies the next-event function whose independent variable is α_j. That is,

$$\alpha_j \in \aleph_P^{-1}(\alpha_i) \Leftrightarrow (\alpha_j \in P) \wedge (\alpha_j \in \aleph^{-1}(\alpha_i). \tag{1}$$

Thus, the MDD structure for (1) is the common part of two corresponding structures. The inverse next-event function is considered as a form of dynamic MDD structure. And the events of process resulted from different dynamic actions of submodels. So the MDD encoding next-event function is a composite of multiple level structure. The above approaches have to rely on intersection after each event, which has been fired in inverse order of Tr(P). However, the challenge arise from the expensive operation of intersection and too large a space range. To overcome these difficulties, we modify the algorithms heavily. In our implementation, the levels of MDD rely on hierarchical decomposition of the process P. We add constraints to inverse next-event function, where the process P holds in all nodes along the paths in MDD. Its saturation checking is unlike the traditional property checking, as the executing events of process are changing. The liveness property and reachable property are the two significant proven natures in formal methods. And the process P is consistently satisfied, that is a kind of liveness at some degrees. We attempt to integrate liveness into inverse next-event function:

Algorithm 2 A new saturation checking procedure of P

```
mdd satP_1(mdd p)
for each α = α_n do
  ℵ_{α,p} ← nextState(P, ℵ_d);
if (ℵ_{α,p} ∩ MDD_x) = ∅ then i = n − 1;
repeat
  ℵ^{-1}_{α,p} ← nextState(P, ℵ_{α_i})^{-1});
  mdd s = intersection(ℵ^{-1}_{α_i,p});
until i = 1;
return s;
```

```
mdd nextState(mdd p_1, relationship ℵ)
represent relationship ℵ as mdd v
resolve the event α into its component events
if p_1 == 1 and v == 1 then return 1;
```

check the cache
if solution for p and $\aleph$ exist **then** return solution;
mdd $M \leftarrow 0$;
int $\hat{p} \leftarrow$ the number of p_1's level;
int $N \leftarrow$ the number of v's level;
if $\hat{p} == N$ **then** continue the process of the nextState;
elseif return 0;
do while $l_p \leftarrow p_1$'s level $l_v \leftarrow v$'s level from high level to low level
if $l_p > l_v$ **then** for each $\alpha_p^{'} \in \varepsilon_{l_p}$ **do**
$M \leftarrow nextState(\alpha_p^{'} \leftarrow dw, v)$;
elseif $l_p < l_v$ **then** $\alpha_N^{'} \in \varepsilon_{l_n}$ **do**
$M \leftarrow nextState(p_1, \alpha_n^{'} \leftarrow dw)$;
elseif $l_p = l_v$ **then** for each $\alpha_p^{'} \in \varepsilon_{l_p}$, $\alpha_n^{'} \in \varepsilon_{l_n}$ **do**
$M \leftarrow nextState(\alpha_p^{'} \leftarrow dw, \alpha_n^{'} \leftarrow dw)$;
add M to cache as solutions for p_1 and $\aleph$
return M;

We assume that the process P is referred as a MDD pattern p. The term α_n means in general the last event of process P. We translate a compound event into equivalent small events. There are overlaps between MDD encoding process P and MDD structure encoding the verified system. For convenience, the result is answered by combing simple results. In other words, we determine the satisfactory repeating elements at each level, lastly, aggregate these results. To conserve strong range in traditional methods, the contained events are used to bound the implementation. The process P confirms the extent of checking. Every subsystem restricts the iteration to every transition. In this paper, we can sure about the location of last event in the MDD or candidated events which may be satisfied. We must confirm all events are included in the system to address. The steps can be summarized into three steps:

(1)Check if the refusal events are contained in the forward transitive MDD;
(2)Evaluate the sub-MDD in each transition;
(3)Assign the sub-MDD to every level on the basis of the second step;

Algorithm 3 Another new saturation checking procedure of P

---

mdd TransProd (mdd P_2, mdd v_2, relationship N, flag f)
represent relationship $\aleph$ as MDD v_1
if $v_2 == 1$ and $v_1 == 1$ **then** return 1;
check the cache
if solution for p_2, v_2 and $\aleph$ exist **then**
return solution;
mdd $M \leftarrow 0$;
int $V = v'_{2,s}$ level;
do while $l_{v_2} \leftarrow v'_2$ level from high level to low level
for each $\alpha_{v_2} \in \varepsilon_{v_2}$ **do**
if $f == 1$ **then**
check **if** produce $(\alpha_{v_2}, \aleph(\alpha_{v_2})) \in l_{p_2}$ (l_{p_2} is the l_{v_2}th level of MDD p_2)
then mdd $z \leftarrow$ TransProd(produce$(\alpha_{v_2}, \aleph(\alpha_{v_2})), \alpha_{v_2}, \aleph(\alpha_{v_2}, 1))$;
else
check **if** produce$^{-1}(\alpha_{v_2}, \aleph(\alpha_{v_2})) \in l_{p_2}$ (l_{p_2} is the l_{v_2}th level of MDD p_2)
then mdd $z \leftarrow$ TransProd(produce$^{-1}(\alpha_{v_2}, \aleph^{-1}(\alpha_{v_2})), \alpha_{v_2}, \aleph^{-1}(\alpha_{v_2}, 0))$;
if $z! = 0$ and $M! = 0$ **then** union(z, M)
if $M == 0$ **then**
$M \leftarrow NewNode(l_{v_2})$;
$M \leftarrow$ nextState saturate$(p_2, \aleph, f)$;
add the solution to the cache;
return M;

---

mdd nextStateSaturate(mdd p_2, mdd v_2, flag f)
check the Cache
if exist the solution for p_2 and v_2 exist **then** return the solution;
mdd $M \leftarrow NewNode(l_{v_2})$;
for all α_{v_2} in l_{v_2}

if $\alpha_{v_2} \in l_{v_2}$
then continue the process of nextStateSaturate
else return 0;
repeat
if $f == 1$ and produce$(\alpha_{v_2}, \aleph(\alpha_{v_2})) \neq 0$ **then**
check **if** produce$(\alpha_{v_2}, \aleph(\alpha_{v_2})) \in l_{v_2}$
then mdd $z \leftarrow$ TransProd(produce$(\alpha_{v_2}, \aleph(\alpha_{v_2})), \alpha_{v_2}, \aleph(\alpha_{v_2}, 1))$;
elseif $f == 0$ and $produce^{-1}(\alpha_{v_2}, \aleph(\alpha_{v_2})) \neq 0$
check **if** $produce^{-1}(\alpha_{v_2}, \aleph(\alpha_{v_2})) \in l_{v_2}$
then mdd $z \leftarrow$ TransProd$(produce^{-1}(\alpha_{v_2}, \aleph^{-1}(\alpha_{v_2})), \alpha_{v_2}, \aleph^{-1}(\alpha_{v_2}, 1))$;
$M \leftarrow union(z, M)$;
until M do not change;
add M to Cache as solution for p_2 and v_2.
return M;

---

mdd SatP_2(mdd P)
for each $\alpha == \alpha_n$
do $\aleph_{\alpha,p} \leftarrow$ nextStateSaturate$(P, \alpha_n, 1)$
if $(\aleph_{\alpha,p} \cap mdd_x) = \emptyset$
then $i = n - 1$
repeat
mdd $\aleph^{-1}_{\alpha,p} \leftarrow$ nextStateSaturate$(P, \alpha_i, 0)$
mdd $s =$ intersection$(\aleph^{-1}_{\alpha,p})$
until $i = 1$;
return s;

The second method has a choice of forward transition or backward transition. We test whether the newly created events belong to the collection of events at their own level step by step. We first set up the set of events $Tr(P)$ and refusal set $X(P)$, then saturate the MDD structure that convert the event of P into the set of explored events, finally to combine the results and the saturated pattern representing events. During the procedure, we narrow the search space, which presents transition relationship. The method retains the advantages of event locality and local fixpoint computation by testing the newly generated local event per iteration.

On the basis of the above theory, we overwrite saturation-based CSP model checking in the framework of stable failure. The detailed algorithms are as follows:

Algorithm 4 The detailed algorithms for the saturation-based CSP model checking

---

Sat$(a \rightarrow P)$
if Tr$(a \rightarrow P) = \langle\rangle$ **then**
check **if** $a \notin X(P)$ **then** return true;
else set
$P' = a \cup P$;
Tr$(P') = a \cup$ Tr$(P) = \langle a, \alpha_1, ..., \alpha_n \rangle$;
return $\begin{pmatrix} \text{Sat}_1\left(P', \text{Tr}\left(P'\right), X\left(P'\right)\right) \text{ or} \\ \text{Sat}P_1(\text{mdd } P') \text{ or Sat}P_2(\text{mdd } P') \end{pmatrix}$;

Sat$(x : A \rightarrow P\,(x))$
if Tr$(x : A \rightarrow P\,(x)) = \langle\rangle$ **then**
check **if** $A \cap X(P) = \{\}$ **then** return true;
else set
$a =$ head(Tr $(x : A \rightarrow P\,(x))$);
Tr$(P_1) =$ tail(Tr $(x : A \rightarrow P\,(x))$);
$P_1 = P\,(a)$;
$P' = a \cup P_1$;
Tr$(P') = a \cup$ Tr (P_1);
$X(P') = X\,(P_1)$;
return $\begin{pmatrix} \text{Sat}_1\left(P', \text{Tr}\left(P'\right), X\left(P'\right)\right) \text{ or} \\ \text{Sat}P_1(\text{mdd } P') \text{ or Sat}P_2(\text{mdd } P') \end{pmatrix}$;

Sat$(c!v \rightarrow P)$
ifTr$(c!v \rightarrow P) = \langle\rangle$ then

check **if** $c.v \notin X(P)$ **then** return true;
else set
$P' = a \cup P_1$;
$\mathrm{Tr}(P') = a \cup \mathrm{Tr}(P_1) = \langle c.v, \alpha_1, ..., \alpha_n \rangle$;
$X(P') = X(P_1)$;
return $\begin{pmatrix} \mathrm{Sat}_1(P', \mathrm{Tr}(P'), X(P')) \text{ or} \\ \mathrm{Sat}P_1(\mathrm{mdd}\ P') \text{ or } \mathrm{Sat}P_2(\mathrm{mdd}\ P') \end{pmatrix}$;

$\mathrm{Sat}(c?x : T \to P(x))$
if $\mathrm{Tr}(c?x : T \to P(x)) = \langle\rangle$ **then**
check **if** $in.T \cap X(P) = \{\}$ **then** return true;
else set
$c.v = \mathrm{head}(c?x : T \to P(x))$;
$\mathrm{Tr}(P_1) = \mathrm{tail}(\mathrm{Tr}(c?x : T \to P(x)))$;
$P_1 = P(c.v)$;
$P' = c.v \cup P_1$;
$\mathrm{Tr}(P') = c.v \cup \mathrm{Tr}(P_1)$;
$X(P') = X(P_1)$;
return $\begin{pmatrix} \mathrm{Sat}_1(P', \mathrm{Tr}(P'), X(P')) \text{ or} \\ \mathrm{Sat}P_1(\mathrm{mdd}\ P') \text{ or } \mathrm{Sat}P_2(\mathrm{mdd}P') \end{pmatrix}$;

Sat(SKIP)
if Tr(SKIP)=$\langle\rangle$ **then**
check **if** $\surd \notin X$ **then** return true;
elseif Tr(SKIP)=$\langle\surd\rangle$ **then**
check **if** $X \subseteq \Sigma^{\surd}$ **then** return true;

$\mathrm{Sat}(P_1 \Box P_2)$
if $\mathrm{Tr}(P_1 \Box P_2) = \langle\rangle$**then**
check **if** $X \subseteq X(P_1) \cap X(P_2)$ and **then** return true;
elseif $\mathrm{head}(\mathrm{Tr}(P_1 \Box P_2)) = \mathrm{head}(\mathrm{Tr}(P_1))$ **then**
return $\begin{pmatrix} \mathrm{Sat}_1(P_1, \mathrm{Tr}(P_1), X(P_1)) \text{ or} \\ \mathrm{Sat}P_1(\mathrm{mdd}\ P_1) \text{ or } \mathrm{Sat}P_2(\mathrm{mdd}\ P_1) \end{pmatrix}$;
else **if** $\mathrm{head}(\mathrm{Tr}(P_1 \Box P_2)) = \mathrm{head}(\mathrm{Tr}(P_2))$ **then**
return $\begin{pmatrix} \mathrm{Sat}_1(P_2, \mathrm{Tr}(P_2), X(P_2)) \text{ or} \\ \mathrm{Sat}P_1(\mathrm{mdd}\ P_2) \text{ or } \mathrm{Sat}P_2(\mathrm{mdd}\ P_2) \end{pmatrix}$;

$\mathrm{Sat}(P_1 \sqcap P_2)$
if $\mathrm{head}(\mathrm{Tr}(P_1 \sqcap P_2)) = \mathrm{head}(\mathrm{Tr}(P_1))$ **then**
return $\begin{pmatrix} \mathrm{Sat}_1(P_1, \mathrm{Tr}(P_1), X(P_1)) \text{ or} \\ \mathrm{Sat}P_1(\mathrm{mdd}\ P_1) \text{ or } \mathrm{Sat}P_2(\mathrm{mdd}\ P_1) \end{pmatrix}$;
elseif $\mathrm{head}(\mathrm{Tr}(P_1 \sqcap P_2)) = \mathrm{head}(\mathrm{Tr}(P_2))$ **then**
return $\begin{pmatrix} \mathrm{Sat}_1(P_2, \mathrm{Tr}(P_1), X(P_2)) \text{ or} \\ \mathrm{Sat}P_1(\mathrm{mdd}\ P_2) \text{ or } \mathrm{Sat}P_2(\mathrm{mdd}\ P_2) \end{pmatrix}$;

$\mathrm{Sat}(P_1 {}_A||_B P_2)$
set
$A_1 = A^{\surd} \backslash A^{\surd} \cap B^{\surd}$;
$B_1 = B^{\surd} \backslash A^{\surd} \cap B^{\surd}$;
$\mathrm{Tr}(P_1') = A_1 \cap \mathrm{Tr}(P_1)$;
$\mathrm{Tr}(P_2') = B_1 \cap \mathrm{Tr}(P_2)$;
Keep the sequence of $\mathrm{Tr}(P_1')$, $\mathrm{Tr}(P_2')$ unchanged
set
n_1' is the number of $\mathrm{Tr}(P_1')$;
n_2' is the number of $\mathrm{Tr}(P_2')$;
$X(P_1') = X(P_1)$;
$X(P_2') = X(P_2)$;
$P_{\cap^1} = \left(A^{\surd} \cap B^{\surd}\right) \cap \mathrm{Tr}(P_1)$;
$P_{\cap^2} = \left(A^{\surd} \cap B^{\surd}\right) \cap \mathrm{Tr}(P_2)$;
$X(P_{\cap^1}) = A^{\surd} \cap X(P_1)$;
$X(P_{\cap^2}) = B \cap X(P_2)$;

Keep the sequence of $\mathrm{Tr}(P_{\cap^1})$, $\mathrm{Tr}(P_{\cap^2})$ unchanged
return $\begin{pmatrix}\mathrm{Sat}_1(P_{1'},\mathrm{Tr}(P_{1'}),X(P_{1'})) \quad \text{or}\\ \mathrm{Sat}_2(P_{1'},\mathrm{Tr}(P_{1'}),X(P_{1'}))\end{pmatrix}$
$\cup \begin{pmatrix}\mathrm{Sat}_1(P_{2'},\mathrm{Tr}(P_{2'}),X(P_{2'})) \quad \text{or}\\ \mathrm{Sat}_2(P_{2'},\mathrm{Tr}(P_{2'}),X(P_{2'}))\end{pmatrix}$
$\cup \left(\begin{pmatrix}\mathrm{Sat}_1(P_{\cap^1},\mathrm{Tr}(P_{\cap^1}),X(P_{\cap^1})) \quad \text{or}\\ \mathrm{Sat}_2(P_{\cap^1},\mathrm{Tr}(P_{\cap^1}),X(P_{\cap^1}))\end{pmatrix}\right.$
$\left.\cap \begin{pmatrix}\mathrm{Sat}_1(P_{\cap^2},\mathrm{Tr}(P_{\cap^2}),X(P_{\cap^2})) \quad \text{or}\\ \mathrm{Sat}_2(P_{\cap^2},\mathrm{Tr}(P_{\cap^2}),X(P_{\cap^2}))\end{pmatrix}\right)$;

Sat $(P_1|||P_2)$
Insert $\mathrm{Tr}(P_2)$ into $\mathrm{Tr}(P_1)$ in order to get $\mathrm{Tr}(P')$
Insert $\mathrm{Tr}(P_1)$ into $\mathrm{Tr}(P_2)$ in order to get $\mathrm{Tr}(P')$
$\mathrm{Tr}(P_1)$, $\mathrm{Tr}(P_2)$ all appear in the $\mathrm{Tr}(P')$, and keep the order of $\mathrm{Tr}(P_1)$ and $\mathrm{Tr}(P_2)$ unchanged.
$\mathrm{Tr}(P')$ interleaves $\mathrm{Tr}(P_1)$, $\mathrm{Tr}(P_2)$
set $X(P') = (X(P_1) \cup X(P_2)) \backslash \surd$
return $\begin{pmatrix}\mathrm{Sat}_1(P',\mathrm{Tr}(P'),X(P')) \quad \text{or}\\ \mathrm{Sat}P_1(\text{mdd } P') \text{ or } \mathrm{Sat}P_2(\text{mdd } P')\end{pmatrix}$;

$\mathrm{Sat}(P_1 \underset{A}{\|} P_2)$
insert the array of $\mathrm{Tr}(P_1^2)$ into the array of $\mathrm{Tr}(P_1^1)$ in order to get $\mathrm{Tr}(P_1)$
set
$\mathrm{Tr}(P_1^1) = Tr(P_1) \cap A$;
$\mathrm{Tr}(P_1^2) = Tr(P_2) \cap A$;
$X(P_1^1) = X(P_1) \cap A$;
$X(P_1^2) = X(P_2) \cap A$;
$X(P_\cap) = X(P_1^1) \cap X(P_1^2)$;
$\mathrm{Tr}(P')$ interleaves $\mathrm{Tr}(P_1 \setminus A^{\surd})$, $\mathrm{Tr}(P_2 \setminus B^{\surd})$
$\mathrm{Tr}(P_1 \setminus A^{\surd})$ and $Tr(P_2 \setminus B^{\surd})$ all appear in the $\mathrm{Tr}(P')$, and keep the order unchanged.
$X(P') = X(P_1 \setminus A^{\surd}) \cup X(P_2 \setminus B^{\surd})$
return $\begin{pmatrix}\mathrm{Sat}_1(P',\mathrm{Tr}(P'),X(P')) \quad \text{or}\\ \mathrm{Sat}_2(P',\mathrm{Tr}(P'),X(P'))\end{pmatrix}$
$\cup \begin{pmatrix}\mathrm{Sat}_1(P_\cap,\mathrm{Tr}(P_\cap),X(P_\cap) \quad \text{or}\\ \mathrm{Sat}_2(P_\cap,\mathrm{Tr}(P_\cap),X(P_\cap))\end{pmatrix}$;

$\mathrm{Sat}(P \setminus A)$
set $\mathrm{Tr}(P')$=$\mathrm{Tr}(P) \setminus A$, $X(P') = X(P) \setminus A$
return $\begin{pmatrix}\mathrm{Sat}_1(P',\mathrm{Tr}(P'),X(P')) \quad \text{or}\\ \mathrm{Sat}P_1(\text{mdd } P') \text{ or } \mathrm{Sat}P_2(\text{mdd } P')\end{pmatrix}$;

$\mathrm{Sat}(P_1; P_2)$
if $\mathrm{Tr}(P_1; P_2) = Tr(P_1)$ and $\mathrm{Tr}(P_1) \cap \surd = \phi$ **then**
set $P' = P_1$;
$X(P') = X(P_1) \cup \{\surd\}$;
return $\begin{pmatrix}\mathrm{Sat}_1(P',\mathrm{Tr}(P'),X(P')) \quad \text{or}\\ \mathrm{Sat}P_1(\text{mdd } P') \text{ or } \mathrm{Sat}P_2(\text{mdd } P')\end{pmatrix}$;
elseif $\mathrm{Tr}(P_1; P_2) = \mathrm{Tr}(P_1) \cap \langle\surd\rangle \cap \mathrm{Tr}(P_2)$ **then**
set $P' = P_1; P_2$;
$X(P') = X(P_1) \cup X(P_2)$;
return $\begin{pmatrix}\mathrm{Sat}_1(P',\mathrm{Tr}(P'),X(P')) \quad \text{or}\\ \mathrm{Sat}P_1(\text{mdd } P') \text{ or } \mathrm{Sat}P_2(\text{mdd } P')\end{pmatrix}$;

Sat $(P_1 \Delta P_2)$
if $\mathrm{Tr}(P_1 \Delta P_2) = \mathrm{Tr}(P_1)$ and $\{\surd\} \cap \mathrm{Tr}(P_1) = \phi$ **then**
set
$P' = P_1 \Delta P_2$;
$X(P') = X(P_1) \cup X(P_2)$;
return $\begin{pmatrix}\mathrm{Sat}_1(P',\mathrm{Tr}(P'),X(P')) \quad \text{or}\\ \mathrm{Sat}P_1(\text{mdd } P') \text{ or } \mathrm{Sat}P_2(\text{mdd } P')\end{pmatrix}$;
elseif $\mathrm{Tr}(P_1 \Delta P_2) = \mathrm{Tr}(P_1)$ and $\{\surd\} \subset \mathrm{Tr}(P_1)$ **then**

set $P' = P_1 \Delta P_2$;
$X(P') = X(P_1)$;
return $\begin{pmatrix} \text{Sat}_1(P', \text{Tr}(P'), X(P')) & \text{or} \\ \text{Sat}P_1(\text{mdd } P^{'}) \text{ or } \text{Sat}P_2(\text{mdd } P^{'}) \end{pmatrix}$;
elseif $\text{Tr}(P_1 \Delta P_2)$ =former part $\text{Tr}(P_1) \cap \text{Tr}(P_2)$;
then
set
$P' = P_1 \Delta P_2$;
$X(P') = X(P_1) \cup X(P_2)$;
return $\begin{pmatrix} \text{Sat}_1(P', \text{Tr}(P'), X(P')) & \text{or} \\ \text{Sat}P_1(\text{mdd } P^{'}) \text{ or } \text{Sat}P_2(\text{mdd } P^{'}) \end{pmatrix}$;

Algorithm 1-3 generally refer to any satisfiability approaches for process P. In this article, we call these algorithms of P in the process of stable failure. It can be seen that our method can find out the set of states that meet specific conditions based on event locality. At each calling we find out the reachable states by firing events satisfying property represented in CSP and add to MDD encoding the currently-known set of states.

6 Bacterial chemo-taxis modeling and Its Saturation checking

In previous sections above, we have given a complete framework for the CSP based saturation checking. In this section, we study its application in bacterial chemo-taxis system, which is a typical system. When the amount of signals are produced, they cause the reactions to emerge by external stimuli. The whole procedure is called Bacterial Chemo-taxis. There are two types of actions for escherichia coli: moving towards attractants and away from repellents. We discuss the all sensary pathways and model its conduct. Different proteins suffix with different letters. Because of repellent, these proteins interact to make the bacteria tumbling. Trans-membrane sensory receptors call methylacepting chemo-taxis proteins (MCPs) can detect chemo-tactic signals. Then the chemo-taxis pathway is started. CheW, as an important adaptor protein, is playing the bridge and link role in the process of MCP-CheW-CheA. In the model, the two response regulators-CheY and CheB have been linked together as response consequences by varying degrees of autophosphorylation. During the process, CheA has great significance. A complex interaction between CheY and Film can cause different reactions directions, which cause adverse results for bacteria. The model involves a number of events that are representative of actions in separate directions in pathways. In the whole procedure, we view various biological significant functions, most of them involved with phosphorylation, dephosphorylation, methylation and demethylation.

Considering the system as a closed box, then repellent events and tumble events are external events outside the interface. Different arrows in the diagram indicate different directions of actions.

The total system is described as:

$$SYSTEM' = repellent \rightarrow SYSTEM \rightarrow tumble$$

where $SYSTEM$ is internal iteration system.

The multiple subsystems are described as follows:
$MCP = a \rightarrow c \rightarrow ... \rightarrow h \rightarrow ... \rightarrow i \rightarrow MCP$
$CheW = b \rightarrow i \rightarrow ... \rightarrow a \rightarrow CheW$
$CheA = b \rightarrow c \rightarrow d \rightarrow CheA$
$CheR = j \rightarrow ... \rightarrow a$
$CheY = d \rightarrow g \rightarrow ... \rightarrow CheY$
$CheZ = f \rightarrow d \rightarrow e \rightarrow tumble$
For example, we denote simple properties to verify:

$$\#(s \upharpoonright \{repellent\}) \geq \#(s \upharpoonright \{tumble\})$$

It is easy to see that there are more repellents than tumbles. The saturation-based computation procedure is:

Algorithm 5 The saturation-based computation procedure

$Sat(\#(s \upharpoonright \{repellent\}) \geq (\#(s \upharpoonright \{tumble\})$
declare χ, χ_1, χ_2 : set of states
divide the set χ into two parts:χ_1, χ_2: $\chi_1 \cup \chi_2 = \chi$
set $x = x_f$, $counter_r = counter_t = 0$
if $N^{-1}(x) \cap x_s = \phi$ **then** continue;
classify states $(\alpha_s, x_s, \chi_1, \chi_2)$;
repeat
  $\chi_1 \leftarrow \chi_{\ repellent}$
  if Saturate (χ_1, α_s) **then** $counter_r++$;
  $\chi_2 \leftarrow \chi_{tumble}$
  if Saturate (χ_2, α_s) **then** $counter_t++$;
  $\chi \leftarrow N^{-1}(\chi)$
until $\chi = \chi_0$
if $counter_r \geq counter_t$
then return true
else return false

Based on above property, we want to guarantee that no rumble is refused. The whole property is

$$\#(s \upharpoonright \{repellent\}) \geq \#(s \upharpoonright \{tumble\}) \Rightarrow tumble \notin X.$$

So we want to perform the process

$$Sat(\#(s \upharpoonright \{repellent\}) \geq \#(s \upharpoonright \{tumble\}) \Rightarrow tumble \notin X).$$

Because the equation $a \rightarrow b \Leftrightarrow \neg b \rightarrow \neg a$ holds, we equivalently check

$$tumble \in X \Rightarrow \neg((\#(s \upharpoonright \{repellent\}) \geq \#(s \upharpoonright \{tumble\})).$$

The brief description of the above process is:

$$\textbf{if } N^{-1}(\chi) \cap \chi_s = \phi \text{ then continue; else break;}$$
$$\textbf{if } tumble \in \chi_s \text{ then check } \neg Sat(\#(s \upharpoonright \{repellent\}) \geq \#(s \upharpoonright \{tumble\}))$$

In the process, we call the previous saturation-based algorithms, namely the universal process's saturation checking.

7 Experimental studies

In this section, we compare three approaches:the traditional saturation algorithm, Saturation algorithm 1 ($SatP_1$) and Saturation algorithms 2 ($SatP_2$) by doing experiments. We implemented the proposed approach in FDR and reported on a set of experiments running on an Intel(R) i5-6300HQ 2. 3GHz workstation with 8. 00 GB RAM under windows 10. The state space size for each model is expressed as the number of MDD nodes. Then we record the final memery, peak memory, and the time of CPU. The results are shown in Table 1. The column items is used for the number of iterations.

It can be seen from Table 1 that our algorithms are much faster, as well as require less memory than traditional method. The process is under a constraint, the process P is determinant and reduces the complexity that built next-event function greatly since it starts from candidate events. While our method saturating the MDD nodes, the events not in the process P are gradually eliminated. We discuss diverse manners running in the typical systems and compare the outcomes of experiment. The application of algorithms can be further extended to the whole framework of CSP.

Model	Satisfied property	Number of process	Iters	MDD nodes	Approaches	Memory (Final)	Memory (Peak)	CPU (Second)
Bacterial chemo-taxis modeling	$\#(S \upharpoonright \{repellent\}) \geq \#(S \upharpoonright \{tumble\}) \Rightarrow tumble \notin X$	5	2	10	Traditional saturation approach	1998	3112	2. 1
					SatP_1	1032	1775	1. 8
					SatP_2	603	738	1. 2
Railway crossing system	$\neg\Diamond(enter.crossing \Rightarrow x.t \leq x + 10 \wedge \neg down \wedge \neg up)$	4	2	8	Traditional saturation approach	1456	2378	1. 9
					SatP_1	838	865	1. 5
					SatP_2	485	408	0.9
Railway signal model	$startObseved.t_1.t_2 \rightarrow informN?n \rightarrow STOP$	28	20	636	Traditional saturation approach	45600	49600	7. 03
					SatP_1	15888	19878	2. 3
					SatP_2	7289	7788	1. 1

Table 1: Results for the saturation computation

8 Conclusions

Summarizing, we present a novel approach for saturation based model checking in the framework of CSP in this paper. The sets of states are stored as MDDs equipped with event locality, that is, the advanced fixed-point iteration policy. For biological systems, we use CSP's theories to model the policy in a refinement methodology. Therefore, the saturation algorithm for CSP is also applicable to biological system. Finally, we apply theory to practice and show the application in the context of bacterial chemo-taxis. We discuss desirable properties of our algorithms and analyze the checking procedure. The checking procedure which starts from next-event function, namely layering iteration, provides a efficient testbed for our ideas since it eliminates unreachable system parts. The results remain unchanged if answer is negative. In the experimental studies, we compare the performances of our novel approaches, using the tool FDR, with previous saturation algorithm. Just as important, the new saturation algorithms tend to use less peak and final memory and less time. The future work is to check the processes that are not limited and a number of different processes directly.

References

[1] K. C. K. Cheng and R. H. C. Yap. "An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints". In: Constraints 15. 2 (2010), 265-304.

[2] A. A. Cire and W. Jan van Hoeve. "MDD Propagation for Disjunctive Scheduling". In: ICAPS. 2012.

[3] A. A. Cire and W. Jan van Hoeve. "Multivalued decision diagrams for sequencing problems". In: Operations Research 61. 6 (2013), 1411-1428.

[4] T. Kam, T. Villa, R. K. Brayton and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. Multiple-Valued Logic, 4(1-2): 9-62, 1998.

[5] G. Ciardo, G. Lüttgen and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In ICATPN 2000, vol. 1639 of LNCS, 103-122. Springer-Verlag, 2000.

[6] G. Ciardo, G. Lüttgen and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In M. Nielsen and D. Simpson, editors, Proc. 21th Int. Conf. on Applications and Theory of Petri Nets, LNCS 1825, 103-122, Aarhus, Denmark, June 2000. Springer-Verlag.

[7] G. Ciardo, G. Lüttgen and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2031, 328-342, Gen-ova, Italy, Apr. 2001. Springer-Verlag.

[8] G. Ciardo, R. Marmorstein and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2619, 379-393, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[9] A. S. Miner and G. Ciardo. Efficient reach-ability set generation and storage using decision diagrams. In H. Kleijnand S. Donatelli, editors, Proc. 20th Int. Conf. on Applications and Theory of Petri Nets, LNCS 1639, 6-25, Williams burg, VA, USA, June 1999. Springer-Verlag.

[10] G. Ciardo, G. Lüettgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. Formal Methods in System Design, 31: 63-100, 2007.

[11] Ciardo, R. Marmorstein and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2619, 379-393, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[12] G. Ciardo, R. Marmorstein and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. Software Tools for Technology Transfer, 8(1):4- 25, Feb. 2006.

[13] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In W. Hunt, Jr. and F. Somenzi, editors, Computer Aided Verification (CAV'03), LNCS 2725, 40-53, Boulder, CO, USA, July 2003. Springer-Verlag.

[14] M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In M. Nielsen et al. , editors, Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM), LNCS 5404, 582-594, Spindleruv Mlyn, Czech Republic, Feb. 2009. Springer-Verlag.

[15] A. S. Miner and G. Ciardo. Efficient reach-ability set generation and storage using decision diagrams. In ICATPN '99, vol. 1639 of LNCS, 6-25. Springer-Verlag, 1999.