Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2015-IC-003

2015-IC-003

CSP Bounded Model Checking of Preprocessed CTL extended with Events using Answer Set Programming

Lingyun Situ, Lingzhong Zhao

Asia-Pacific Software Engineering Conference 2015

CSP Bounded Model Checking of Preprocessed CTL extended with Events using Answer Set Programming

Lingyun Situ*†‡, Lingzhong Zhao§¶
*State Key Laboratory of Novel Computer Software Technology, Nanjing University, Nanjing 210023, China
†Jiangsu Novel Software Technology and Industrialization, Nanjing 210023, China
‡Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China
§Guangxi Key Laboratory of Trusted Software, Guilin 541004, China
¶School of Computer Science and Engineering, Guilin University of Electronic Technology, Guilin 541004, China
Email: situlingyun@seg.nju.edu.cn, zhaolingzhong@guet.edu.cn

***Abstract*—Model checking is a mainstream method for formal verification of communicating sequential processes (CSP). Existing CSP Model Checkers are incapable of verifying multiple properties concurrently in one run of a model checker. In addition, the properties to be verified are described with CSP in most model checkers, which is good for refinement checking, but leads to limited description power and weak generality. In order to tackle the two problems, answer set programming(ASP), which is completely free of sequential dependencies, is used to construct a CSP bounded model checking framework, where the CSP model checking problem is turned into a computation problem of answer sets. CTL is extended with events to describe the properties to be verified. In addition, preprocessing technique of properties is proposed for the sake of reducing the expense spending on replicated verification of the same sub formulas. An ASP based description system is constructed for complete description of various CSP processes and automatic generation of parallel processes. We integrated all the methods into a CSP model checker - ACSPChecker. The feasibility and efficiency of our methods are illustrated by the experiments with a classic concurrency problem - dining philosophers problem.**

***Keywords*-Communicating Sequential Processes, Bounded Model Checking, CTL, Preprocess, Answer Set Programming.**

I. Introduction

Model checking [1] [2] is one of the most important and powerful formal verification techniques. It has been widely and successfully used in analysis and verification of various domains including computer hardware, communication protocol and concurrent system. Main stream model checking methods include bounded model checking [3] [4], compositional checking [5], assume-guarantee reasoning [6] and module model checking [7] etc.

Communicating sequential processes (CSP) [8] [9], is one of the two original process algebras. It allows the precise description and analysis of event-based concurrent systems. Some CSP model checking tools have been developed, such as Failure Divergence Refinement(FDR) [10] [12] [13], Process Analysis Toolkit(PAT) [14], SymFDR [15], ProB [16] [17], ARC [19] and so on.

However, there are two common shortages of all the existing CSP model checkers. The first is inability to verify multiple properties concurrently in one run of a model checker. Multiple runs of a model checker cause massive overhead in context switching. Moreover, the problem of repeated verification of the same sub property is not considered, which limits verification efficiency since the same subformula in the specification of different properties must be handled respectively. The second is that most of CSP model checkers use CSP process as the specification language of properties. This description method is good for refinement checking, but is not as common as temporal logic formulas. And it is not suitable for expressing liveness properties [11].

This paper reports our attempts to apply answer set programming(ASP) [26] [27] to CSP model checking. ASP is a declarative logic programming paradigm for solving combinational search problems with the feature of completely free of sequential dependencies. In order to achieve the ability to verify multiple properties concurrently in one run of a model checker, an ASP based CSP bounded model checking framework is established, which turns the CSP model checking problem into the computation problem of answer sets. The properties to be verified are specified by computation tree logic(CTL) [32] extended with events, which is much more common and expressive. Furthermore, a preprocessing procedure for a set of properties formulas is proposed for the sake of avoiding the overhead of replicated validation of the same sub formulas, thus improving the verification efficiency. An ASP based description system of CSP processes is constructed. It is not only to be used to describe various forms of basic processes, but also allows several processes to be composed automatically to generate a new parallel process. All the methods are integrated into a CSP model checker - ACSPChecker. The feasibility and efficiency of our methods are illustrated by the experiments with a classic concurrency problem - dining philosophers problem.

1530-1362/15 $31.00 © 2015 IEEE
DOI 10.1109/APSEC.2015.16

II. Related Works

FDR [10] [12] is a model checking tool for CSP developed by A.W.Roscoe and C.A.R.Hoare of Oxford University. A hybrid high/low level approach is adopted to compute the operational semantic of a process in FDR. It builds up a system gradually with the hierarchical state space compression technique. The compression technique makes FDR could check systems such as a network of 10^{20} states [13] .

Recently, efficient SAT solvers have greatly broadened the horizons of symbolic model checking [18], and bounded model checking [3] [4] has been proved to be an extremely powerful technique. Some works on symbolic model checking of CSP have been done, including OBDD-based refinement checker ARC [19], SAT based bounded refinement checker SymFDR [15] and so on. ARC outperforms FDR in a few cases, but the OBDD encoding is complex and especially sensitive to the variable ordering. SymFDR integrate SAT-based BMC and temporal k-induction into FDR focusing on the CSP trace model, which is sufficient to verify safety properties. Experiments indicate that in some complex combinatorial problems, SymFDR significantly outperforms FDR.

Refinement checking [20] is the major verification method embedded in FDR, SymFDR and ARC. A limit of the method is that it is not good for description and verification of liveness properties [21]. In [22], some evidences are also presented to show that refinement-based approach does not seem to be suited for verifying certain temporal properties. Moreover, the properties to be verified in these model checkers are specified as CSP processes, which is not as common as temporal logic formulas, such as Linear Temporal Logic(LTL) and Computation Tree Logic(CTL) [32].

PAT [14] is another CSP model checker. Different from the tools mentioned above, it supports automated refinement checking and model checking of LTL extended with events. An on-the-fly approach is adopted to reduce the computation expense of normalization, and PAT outperforms FDR in some cases. In the paper, we apply another commonly used property description language- CTL to specify the properties to be verified.

In literature, there already exists some work on applying declarative programming to the verification of software systems [29], the main motivation is to show that the declarative programming is a convenient language to model other languages and is a competitive tool for validation.

Literature [23] summarizes Michael Leuschels ten years of experience using declarative programming (Prolog in particular) for developing tools to validate high-level formal specification languages, ranging from process algebras such as CSP to model-based specifications such as Z and B [25]. It shows that the logic programming is a good foundation for developing and animating new specification languages [24]. ProB [16] [17] is a new animation and model checking tool for CSP developed by Michael Leuschel using Prolog technology. Similarly, a more declarative programming language - ASP [26] [27] is used in the implementation of our method.

E. Ternovska presents an ASP based BMC method for verifying abstract state machine(ASM) models [28]. It illustrated how to translate an ASM and a temporal property into a logic program and to solve the BMC problem by computing an answer set for the logic program. K. Heljanko proposes symbolic model checking as a promising application area for answer set programming in [29] [30]. Additionally, E. De Angelis introduces a framework based on ASP for the synthesis of concurrent programs satisfying given behavioral and structural properties [31]. This method gives us hint on automatically combining several processes to generate a new parallel process using ASP rules.

III. Preliminaries

A. CSP

Let Σ be a finite alphabet of visible events with $\tau,\surd \in \Sigma$. τ denotes the invisible silent action and $\surd$ is a successful termination of a process-a special action which is visible but uncontrollable from outside and can only occur at last. For a given process P, we denotethe set of all visible events that P can perform [8] [9] by $\alpha P \in \Sigma^{\surd}$.

Definition 1: A CSP process is defined recursively via the following grammar:

$$P \equiv STOP|SKIP|CHAOS|DIV|X : A \to P(x)|\mu X : A \bullet F(X)|P1 \sqcap P2|P1|P2|P1 \parallel P2$$

$STOP$ represents a deadlocked process, which is not capable of communicating any visible or τ actions. The process $SKIP$ denotes successful termination and is willing to perform $\surd$ at any time. $CHAOS$ is a process that may non-deterministically perform events from A. It may as well refuse to do anything at all. DIV denotes a livelock, a process that is engaged in performing an infinite loop of internal τ actions without ever communicating with the external environment. The prefix process $X : A \to P(x)$ initially offers the environment to perform any event a from A and subsequently behaves like $P(a)$. $\mu X : A \bullet F(X)$ denotes a recursive process.$P1 \sqcap P2$ and $P1|P2$ denote, respectively, external and internal choice of $P1$ and $P2$. Parallel composition of processes $P1$ and $P2$ is written as $P1 \parallel P2$, where shared events must be synchronized by both processes whose alphabet contains the events.

Definition 2: **(Process P with bounded k: P^k)** Let P be a process, then P^k is a process where the parallel step of all parallel operators in the process is bounded to k.

Example 1: $P = a \to b \to c \to d \to STOP||a \to e \to b \to d \to STOP = a \to e \to b \to c \to d \to STOP$. If the parallel step k is set equal to 3, then $P^3 = a \to e \to b$.

B. ASP

Let A be an atom, a literal takes the form A or $\sim A$, where A is a positive literal and $\sim A$ is a negative literal; A and $\sim A$ are called complementary literals[3]. An extended disjunctive logic program is a set of rules, and each rule r is of the form:

$$L_1 \vee ... \vee L_k :- L_{k+1}, ..., L_m, not\ L_{m+1}, ..., not\ L_n$$

where $n \geq m \geq k \geq 0$, each L_i is a literal, and notice the negation as failure (NAF), head(r)=$L_1, ..., L_k$ is the head of r, pos(r)=$L_1, ..., L_m$ and neg(r)=$L_{m+1}, ..., L_n$ are the positive and negative literals present in body respectively. In particular, a rule without head is called a constraint.

The computation of answer sets corresponding to ASP logic programs is performed by ASP solvers, such as DLV [33], Smodel [34] and Cmodel [35].

C. BMC

Bounded model checking [3] [4] is a sound but generally incomplete technique that focuses on searching for counterexamples of bounded length only. The underlying idea is to fix a bound k and unwind the implementation model for k steps, thus considering behaviours and counterexamples of length at most k. In practice, BMC is conducted iteratively by progressively increasing k until one of the following happens: (1) a counterexample is detected. (2) k reaches a pre-computed threshold which indicates that the model satisfies the specification. (3) the model checking instance becomes intractable.

IV. ASP based Framework of CSP Bounded Model Checking

The structure of ASP based CSP bounded model checking framework is shown as Fig.1.

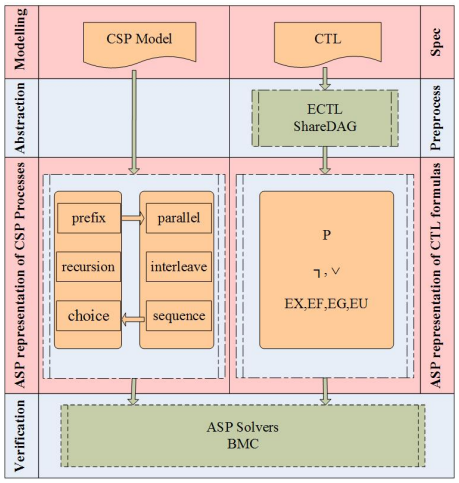

Figure 1. ASP based CSP BMC framework

As illustrated in Fig.1, the framework consists of six modules The modeling module describes the system to be verified with CSP. The specification module specifies the properties with CTL extened with events. A sharedDAG describing multiple properties will be constructed through the preprocessing module. The module for ASP representation of CSP processes is not only used to represent various basic CSP processes, but also be used to generate the compositional processes (parallel processes in particular) automatically. The ASP representation of Existential Computation Tree Logic(ECTL) formulas encodes every ECTL formulas as ASP predicates and rules. In the verification module, BMC is adopted to alleviate the state explosion problem. At last, ASP encoded CSP model and properties formulas are input into an ASP solver to complete the verification.

Our framework for CSP model checking has some features as follows:

1. *Model Checking of CTL extended with events*. Our framework supports model checking of CTL extended with events, which allows us to specify and validate properties based on states and events.

2. *Verification of multiple properties concurrently in one run of the model checker*. ASP is completely free of sequential dependencies. Thanks to the feature, our method can verify multiple properties concurrently in one run of our model checker.

3. *Preprocessing of properties formulas to avoid repeated verification of same sub-properties.* The preprocessing technique is proposed to eliminate redundant information of multiple properties to be verified, and thus avoiding overhead of repeated verification of same sub properties.

In the framework, CSP model checking problem is turned into a computation problem of answer sets. The core verification idea is searching the state space represented by ASP to validate if there exists an assignment in the state space satisfying the given property. If it does, the label corresponding to the property will occur in the answer sets, otherwise it will not. For example, if the property φ holds for process P, then $\varphi(P)$ must occur in the answer set finally.

V. Preprocessing and ASP Encoding of CTL Formulas extended with Events

CTL is extended with events for the sake of description of properties based on states and events in this section.

Definition 3: The CTL formulas extended with events are as follows:

$$\varphi ::= e|s|\neg\psi|(\psi_1 \wedge \psi_2)|(\psi_1 \vee \psi_2)|(\psi_1 \rightarrow \psi_2)|$$
$$AX\psi|EX\psi|AF\psi|EF\psi|AG\psi|EG\psi|A[\psi_1 U \psi_2]|E[\psi_1 U \psi_2]$$

where e ranges over $\sum^{\surd}$. s represent the state, A represent all the paths, E represents exist one path at least, X represent the next state, F represent some state in the future, G represent all the state in the following and U represent until.

Let P be a process, then P^k is a process where each execution behavior of P is bounded to k steps. Thus

$\pi = \{P_0, e_0, P_1, e_1, \cdots, e_{i-1}, P_i, e_i, \cdots P_k\}$ be an execution path of process P^k, P_i is the ith subsequent process of $P^k, 0 \leq i \leq k$

The satisfiability relationship $\models$ of process P^k and extended CTL formulas φ is defined inductively as follows:

$P^k \models e \quad \Leftrightarrow \exists \pi, e_i = e(0 \leq i \leq k), e_i \in \pi.$
$P^k \models s \quad \Leftrightarrow \exists \pi, P_i = s(0 \leq i \leq k), P_i \in \pi.$
$P^k \models \neg\psi \quad \Leftrightarrow P \not\models p.$
$P^k \models \psi_1 \wedge \psi_2 \quad \Leftrightarrow P^k \models \psi_1$ and $P^k \models \psi_2.$
$P^k \models \psi_1 \vee \psi_2 \quad \Leftrightarrow P^k \models \psi_1$ or $P^k \models \psi_2.$
$P^k \models \psi_1 \rightarrow \psi_2 \quad \Leftrightarrow P^k \not\models \psi_1$ or $P^k \models \psi_2.$
$P^k \models AX\psi \quad \Leftrightarrow \forall \pi, (P_1 \models \psi, k \geq 1).$
$P^k \models AG\psi \quad \Leftrightarrow \forall \pi, (\forall j \geq 0 \wedge j \leq k, P_j \models \psi)$.
$P^k \models AF\psi \quad \Leftrightarrow \forall \pi, (\exists j \geq 0 \wedge j \leq k, P_j \models \psi).$
$P^k \models A[\psi_1 U \psi_2] \quad \Leftrightarrow \forall \pi, (\exists m \geq 0 \wedge m \leq k,$
$(P_m \models \psi_2 \wedge \forall j < m, (P_j \models \psi_1))).$
$P^k \models EX\psi \quad \Leftrightarrow \exists \pi, (P_1 \models \psi, k \geq 1).$
$P^k \models EG\psi \quad \Leftrightarrow \exists \pi, (\forall j \geq 0 \wedge j \leq k, P_j \models \psi).$
$P^k \models EF\psi \quad \Leftrightarrow \exists \pi, (\exists j \geq 0 \wedge j \leq k, P_j \models \psi).$
$P^k \models E[\psi_1 U \psi_2] \quad \Leftrightarrow \exists \pi, (\exists m \geq 0 \wedge m \leq k,$
$(P_m \models \psi_2 \wedge \forall j < m, (P_j \models \psi_1))).$

In order to facilitate subsequent handling, we translate the CTL formulas into semantically equivalent ECTL formulas.

Definition 4: **(ECTL Formula)** A CTL formula is a ECTL formula iff any symbol γ in CTL formula, $\gamma \in \{\neg, \vee, EX, EF, EG, EU\}$.

Definition 5: **(Sub formula)** A ECTL formula ψ is the sub formula of φ, if and only if the syntactic tree of ψ is the sub tree of syntactic tree of φ.

In order to avoid repeated verification of same sub formulas in a set of ECTL formulas, all the syntactic trees of ECTL formulas will be integrated into a shared syntactic tree (i.e. sharedDAG) by merging the same sub formulas of different syntactic trees.

Definition 6: **(SharedDAG)** A sharedDAG is a directed acyclic graph $G = (V, E)$, where V is a set of nodes and E is a set of edges:

(1) The out degree of each node is no larger than 2;

(2) The ECTL formulas represented by any two nodes are different.

Given a set U of ECTL formulas, the sharedDAG could be obtained by applying the following rules:

Rule 1(remove duplicate leaf nodes) If a syntax tree has the same leaf nodes $n_1, n_2, ..., n_k$, then remove all the same leaf nodes except n_k, and redirect all arcs into the $n_i (1 \leq i \leq k)$ to n_k.

Rule 2 (remove duplicate internal nodes) Let u and v be internal nodes, the label of u and v are the same, remove the sub tree with the root u and redirect all arcs into u to v if conditions below are satisfied, otherwise, remove v.

(1) Both u and v have two sub-trees, and the sub-trees of u and v are the same.

(2) Node u has one sub tree, v has two sub trees, and the sub tree of u is the same as one of the sub trees of v.

(3) Both nodes u and v have only one sub tree, and the sub trees of u and v are the same.

Example 2: Consider following properties of dining philosopher problems:

(1)Pro 1: Any fork can not be picked up by two philosophers at the same time.

(2)Pro 2: Any philosopher will eat if he has picked up his forks.

The two properties above are described with extended CTL formulas as follows:

(1) $AG\neg(i.pickFork.i \wedge i \bigoplus 1.pickFork.i)$

(2) $AG(i.pickFork.i \rightarrow AF(i.eat)$

The formulas are input into the preprocessing procedure, and then a sharedDAG is obtained as Fig.2(To be simple, set i=2).

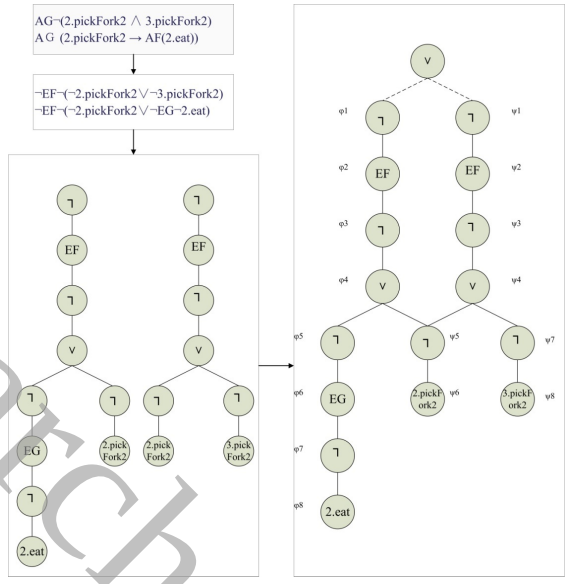

Figure 2. shareDAG

Intuitively, in this example, if we verify the two properties using classical labeling algorithm, $\neg 2.pickfrok2$ will be verified twice, but only once after preprocessing. Our preprocessing procedure will reduce the repeated verification overhead and improve verification efficiency especially when the number of properties is large.

Next to give the ASP encoding of ECTL formulas. In the following, by $\varphi(P)$ it is meant that φ is satisfied in process P, and let $\pi = \{P_0, e_0, P_1, e_1, e_{i-1}, P_i, e_i, ..., P_k\}$ be an execution path of process P.

Formula 1: $\varphi \equiv e$. φ is true when e occurs in P. $\varphi(P)$ is true when $prefix(e, P^{'}, p)$ is true for some e and $P^{'}$, i.e. p is a subsequent process of P.

$\varphi(P) : -prefix(e, \_ , P_j), subsequent(P_j, P).$

Formula 2: $\varphi \equiv s$. $\varphi(P)$ is true when s is the subsequent process of P.

$\varphi(P) : -subsequent(s, P).$

Formula 3: $\varphi \equiv \neg\psi$. $\varphi(P)$ is true if $\psi(P)$ is false.

$\varphi(P) : -not\ \psi(P).$

Formula 4: $\varphi \equiv \psi_1 \vee \psi_2 . \varphi(P)$ is true if $\psi_1(P)$ is true or $\psi_2(P)$ is true.

$\varphi(P) : -\psi_1(P) . \varphi(P) : -\psi_2(P)$.

Formula 5: $\varphi \equiv EX\psi$. $\varphi(P)$ is true if there exists a path $= \{P_0, e_0, P_1, e_1, ...e_{i-1}, P_i, e_i, ..., P_k\}$ of P such that $\psi(P_1)$ is true.

$\varphi(P) : -\psi_1(P_1), next(P_1, P)$.

Formula 6: $\varphi \equiv EF\psi$. $\varphi(P)$ is true if there exists a path π of P such that $\psi(P_i)$ is true.

$\varphi(P) : -\psi_1(P_i), subsequent(P_i, P)$.

Formula 7: $\varphi \equiv EG\psi$. $\varphi(P)$ is true if there exists a path π of P such that $\psi(P_i)$ is always true .

$\varphi(P) : -not\ f. f : -not\ \psi(P_j, subsequent(P_j, P)$.

Formula 8: $\varphi \equiv E[\psi_1 \cup \psi_2]$. $\varphi(P)$ is true if there exists a path π of P such that $\psi_2(P_i)$ is always true until $\psi_1(P_j)$ is true.

$\varphi(P) : -\psi_2(P_j, subsequent(P_j, P_k), \psi_1(P_0...P_{j-1})$.

VI. ASP Encoding of CSP Processes

An ASP based description system for various forms of CSP processes is constructed in this section.

A. Prefix Process

Definition 7: **(Prefix process)** Let x be an event, P and Q are processes and $x \in \alpha Q, \alpha Q = \alpha(x \to P)$, then $Q = x \to P$ describes the process Q which first engages in the event x and then behaves exactly as described by P. We call the x prefix, and $x \to P$ prefix expression.

Some predicates are defined in Table I for the sake of describing prefix processes conveniently.

Table I
PREDICATES DEFINED FOR DESCRIBING PREFIX PROCESS

Predicate	Meaning
prefix(X,P,Q)	Process Q is a prefix process which first engages in the event X and then behaves as process P.
event(X,P)	event X is a event of Process P, i.e. $X \in \alpha P$.
process(P)	P is a process.
next(P,Q)	Process P is the direct subsequent process of Q.
subsequent(P,Q)	Process P is the subsequent process of Q.
startEvent(X,P)	X is the initial event of process P.
endEvent(X,P)	X is the last event of process P.

The semantic rules between the above defined predicates are presented in Table II.

Relying on the rules 1-11 in Table II, given a set of prefix processes, process information including the alphabets, sub processes, the initial events and end events of any processes and so on could be produced.

Example 3: Let a philosopher sit down in his chair, pick up and put down the forks twice and then stand up to leave. The philosopher could be described as a CSP process Ph:

$Ph = sit \to pickFork \to downFork \to pickFork \to downFork \to up \to STOP$

It is easy to describe the above prefix process Ph with $prefix(X, P, Q)$ as the following set of atoms:

Table II
SEMANTIC RULES FOR PREFIX PROCESSES

ASP rules	Name
event(X,Q):- prefix(X,P,Q).	Rule 1
event(X,P):- prefix(X,P,Q).	Rule 2
process(P):- prefix(X,P,Q).	Rule 3
process(Q):- prefix(X,P,Q).	Rule 4
next(P,Q) :- prefix(X,P,Q)	Rule 5
subsequent(P,Q):- next(P,Q).	Rule 6
subsequent(R,Q):- subsequent(P,Q),next(R,Q).	Rule 7
startEvent(X,Q):- prefix(X,P,Q).	Rule 8
endEvent(deadlock,P):- prefix(deadlock,stop,P).	Rule 9
endEvent(succStop,P):- prefix(succStop,skip,P).	Rule 10
endEvent(X,P1) :- endEvent(X,P2),subsequent(P2,P1).	Rule 11

$\{prefix(pickFork, p0, ph).prefix(downFork, p1, p0)$
$.prefix(pickFork, p2, p1).prefix(downFork, p3, p2).prefix(up, stop, p3).prefix(deadlock, stop, stop).\}$

where $prefix(deadlock, stop, stop)$ represents the special process $STOP$, $prefix(succStop, skip, skip)$ represents the $SKIP$ process, and the relationship between process $p0, ..., p4$ can be illustrated as the following Fig.3

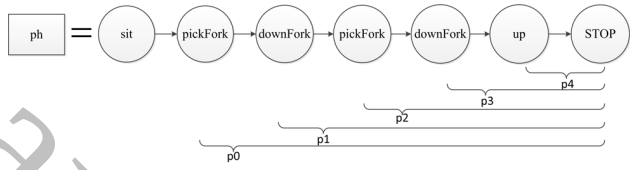

Figure 3. The relationship between process p0-p4

B. Recursion Process

Definition 8: **(Recursion Process)** The equation X=F(X) is said to be a recursion process, if F(X) is a guarded expression containing the process name X. The expression X=F(X) has a unique solution with alphabet A, where A=αX. which could be denoted by expression $\mu X : A \bullet F(X)$.

Simply, process $P = x \to P$ or $P = \mu X : \{x\}(x \to X)$ represent a recursion process which performs event x endlessly. By the definition of a recursion process, it can be described with the ASP predicate $prefix(X, P, Q)$. A simple example is to encode process $P = a \to P$ as $prefix(a, P, P)$.

C. Choice Process

Choice process is used to describe the situation where an object may have many possible streams of behaviors.

Definition 9: **(Choice process)** Let x and y be different events, P and Q be distinct processes, then $R = (x \to P | y \to Q)$ describes an object which initially engages in either of the events x and y, and then behaves as P if the first event was x, or as Q if the first event was y, where $\alpha(x \to P | y \to Q) = \alpha P = \alpha Q$

Example 4: A change-giving machine which offers a choice of two combinations of change for 10p, that is two

5p or one 5p, one 1p, two 2p. The choice is exercised by the customer of the machine. The object is described as a CSP choice process as below:

$CH10D = in10p \rightarrow (out5p \rightarrow out5p \rightarrow CH10D | out2p \rightarrow out1p \rightarrow out5p \rightarrow out1p \rightarrow CH10D)$

It is easy to describe the above prefix process CH10D with $prefix(X, P, Q)$ as the following set of atoms.

$\{prefix(in10p, p, ch10D).prefix(out5p, m1, p).prefix(out5p, ch10D, m1).prefix(out2p, n1, p).prefix(out1p, n2, n1).prefix(out5p, n3, n2).prefix(out2p, ch10D, n3)\}$.

The relationship between process $p, m1, n1, n2, n3$ can be illustrated as the following Fig.4.

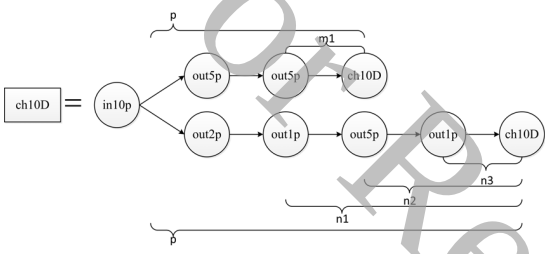

Figure 4. The relationship between process $p, m1, n1, ..., n3$

VII. ASP Encoding of Parallelization

The composition of CSP processes provides a way for generating more complex processes. Parallelization is one of the most important composition ways.

Definition 10: **(Parallelization)** Let P and Q be processes, $\alpha P \neq \alpha Q$, the notation $P \parallel Q$ denotes that process P and Q are assembled to run concurrently, events in both of their alphabets require simultaneous participation of both P and Q; however, events in the only alphabet of P or Q may occur independently. Obviously, $\alpha(P \parallel Q) = \alpha P \cup \alpha Q$.

Let $a \in (\alpha P - \alpha Q), b \in (\alpha Q - \alpha P), c, d \in (\alpha Q \cap \alpha P)$, we have the following concurrency laws governing the behavior of $P \parallel Q$:

(1) $P \parallel STOP = STOP$

(2) $(c \rightarrow P) \parallel (c \rightarrow Q) = (c \rightarrow (P \parallel Q))$

(3) $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP \qquad c \neq d$

(4) $(a \rightarrow P) \parallel (c \rightarrow Q) = (a \rightarrow (P \parallel (c \rightarrow Q)))$

(5) $(c \rightarrow P) \parallel (b \rightarrow Q) = (b \rightarrow ((c \rightarrow P) \parallel Q)))$

(6) $(a \rightarrow P) \parallel (b \rightarrow Q) = (a \rightarrow (P \parallel (b \rightarrow Q)) | b \rightarrow ((a \rightarrow P) \parallel Q))$

ASP predicate $concurrent(P, Q, R)$ is introduced for the sake of description of parallelization. It means that process p and Q are composed to be process R through parallelization.

The ASP encoding of the concurrency laws is presented in Table III.

The ASP rules defined according to the concurrency laws of CSP make it possible that the processes encoded with ASP predicates could automatically composed to generate a new parallel process with the same structure. which allows us to describe various styles of concurrent systems of larger scale. Note that we could bound the steps of the parallelization of two recursion processes to control the states space.

Example 5: In dining philosophers problem, the behaviors of a philosopher while having dinner could be modelled as a CSP process simply as follows :

$Ph_i = sit \rightarrow pickFork_i \rightarrow pickFork_{i\oplus 1} \rightarrow eat \rightarrow downFork_i \rightarrow downFork_{i\oplus 1} \rightarrow up \rightarrow Ph_i$

Note that the $\oplus$ is a plus operator of mode N, represent the right neighbor of the ith philosopher.

For simplicity, we consider a system consists of 3 philosophers. The compositional process PHS could be obtained according the CSP algebraic laws:

$PHS = Ph_1 \parallel Ph_2 \parallel Ph_3$

$= \mu X.(sit \rightarrow pickFork_1 \rightarrow pickFork_2 \rightarrow pickFork_3 \rightarrow eat \rightarrow downFork_1 \rightarrow downFork_2 \rightarrow downFork_3 \rightarrow up \rightarrow X) \parallel Ph_3$

$= sit \rightarrow STOP$

The system described using ASP are a set of prefix and concurrent predicates:

$\{\%Ph_i$ i=1,2,3

$prefix(sit, ph_{i\_1}, ph_i).prefix(pickForki, ph_{i\_2}, ph_{i\_1}).prefix(pickForki\oplus 1, ph_{i\_3}, ph_{i\_2}).prefix(eat, ph_{i\_4}, ph_{i\_3}$
$prefix(downForki, ph_{i\_5}, ph_{i\_4}).prefix(downForki \oplus 1,$
$).ph_{i\_6}, ph_{i\_5}).prefix(up, ph_i, ph_{i\_6}).$

$\%parallelization$

$concurrent(ph_1, ph_2, 1).concurrent(1, ph_3, 11).\}$

Note that the "1" in $concurrent(ph_1, ph_2, 1)$ represents the resulting process $ph_1 \parallel ph_2$. The "11" in $concurrent(1, ph_2, 11)$ represent process $Ph_1 \parallel Ph_2 \parallel Ph_3$.

Put the predicates set and ASP rules 1-23 as the input of ASP solver , we can get the answer set below:

$\{prefix(sit, 2, 1).prefix(pickFork1, 3, 2).prefix(pickFork2, 4, 3).prefix(pickFork3, 5, 4).prefix(eat, 6, 5).prefix(downFork1, 7, 6).prefix(downFork2, 8, 7).prefix(downFork3, 9, 8).prefix(up, 1, 9).prefix(sit, 12, 11).prefix(deadlock, stop, 12).\}$

where $\{prefix(sit, 12, 11).prefix(deadlock, stop, 12)\}$ is the parallel process of Ph_1, Ph_2, and Ph_3.

VIII. Experimental Results

All the methods discussed above are integrated into a CSP model checker named ACSPChecker. The experimental results are presented as follows, where the configuration of computer is Intel Core(TM) i3-2100 CPU, 3.10GHz, 4GB(RAM), 64bit window 7.

Table IV is the situation for the model of 3 philosophers. It is shown that the preprocessing technique has great effect on improving the efficiency of verifying properties. The time of multi-properties verification concurrently is smaller than the sum of verification respectively. And the verification of liveness properties is affected by the bounded step k, because the k determine the size of states space of system.

Table III
ENCODING CONCURRENCY LAWS WITH ASP RULES

ASP rules	Laws	Name
prefix(deadlock,stop,T):-prefix(deadlock,stop,R),process(Q),concurrent(R,Q,T).	law (1)	rule 12
concurrent(M,N,T1):-prefix(X,M,P),prefix(Y,N,Q),X=Y,P!=Q,event(X,P),event(X,Q),event(Y,P),event(Y,Q) ,concurrent(P,Q,T),not #mod(T,10,0),#succ(T,T1).	law (2)	rule 13
prefix(X,T1,T):- prefix(X,M,P),prefix(Y,N,Q),concurrent(P,Q,T), concurrent(M,N,T1).		rule 14
prefix(deadlock,stop,T):-prefix(X,M,P),prefix(Y,N,Q),X!=Y,P!=Q,event(X,P),event(X,Q),event(Y,P) ,event(Y,Q),concurrent(P,Q,T).	law (3)	rule 15
concurrent(M,Q,T1):-prefix(X,M,P),prefix(Y,N,Q),X!=Y,P!=Q,event(X,P),not event(X,Q),event(Y,P) ,event(Y,Q).concurrent(P,Q,T),not # mod(T,10,0),#succ(T,T1).	law (4)	rule 16
prefix(X,T1,T):- prefix(X,M,P), prefix(Y,N,Q),concurrent(P,Q,T), concurrent(M,Q,T1).		rule 17
concurrent(P,N,T1):-prefix(X,M,P),prefix(Y,N,Q),X!=Y, P!=Q,event(X,P), event(X,Q),not event(Y,P) ,event(Y,Q),concurrent(P,Q,T),not #mod(T,10,0),#succ(T,T1).	law (5)	rule 18
prefix(Y,T1,T):- prefix(X,M,P),prefix(Y,N,Q),concurrent(P,Q,T), concurrent(P,N,T1).		rule 19
concurrent(M,Q,T1):-prefix(X,M,P),prefix(Y,N,Q),X!=Y,P!=Q,event(X,P), not event(X,Q),not event(Y,P) ,event(Y,Q),concurrent(P,Q,T),not #mod(T,10,0),#succ(T,T1).		rule 20
prefix(X,T1,T):- prefix(X,M,P),prefix(Y,N,Q),X!=Y,P!=Q, event(X,P), not event(X,Q) ,not event(Y,P), event(Y,Q),concurrent(P,Q,T),concurrent(M,Q,T1).		rule 21
concurrent(P,N,T1):-prefix(X,M,P),prefix(Y,N,Q),X!=Y,P!=Q,event(X,P),not event(X,Q),not event(Y,P) ,event(Y,Q),concurrent(P,Q,T),not #mod(T,10,0),#succ(T,T1).	law (6)	rule 22
prefix(Y,T1,T):- prefix(X,M,P),prefix(Y,N,Q),X!=Y,P!=Q,event(X,P),not event(X,Q),not event(Y,P) ,event(Y,Q),concurrent(P,Q,T),concurrent(P,N,T1).		rule 23

Table IV
RESULTS OF ACSPCHECKER WITH BOUNDED K

ACSPChecker	CSP Model			
(DLV)	Pro 1	Pro 2	Sum	All
Bounded k=5	0.09/T	0.15/F	0.24	0.11
Bounded k=10	1.32/T	1.38/T	2.70	1.34
Bounded k=15	1.41/T	3.44/T	4.85	3.45
Bounded k=20	6.67/T	6.68/T	13.35	7.39

Table V
RESULTS OF ACSPCHECKER WITH DIFFERENT SOLVERS

ACSPChecker Bounded k=10		3	5	7	8
DLV	Pro 1	1.32	11.42	37.94	85.69
	Pro 2	1.34	11.49	38.32	85.25
	All	1.35	11.55	38.50	86.46
Smodel	Pro 1	1.39	11.47	38.08	85.65
	Pro 2	1.41	11.54	38.41	85.22
	All	1.46	11.58	35.51	85.40
Cmodel	Pro 1	1.33	11.41	37.96	85.70
	Pro 2	1.33	11.52	38.29	85.27
	All	1.36	11.55	38.51	85.50

Table V collects the results by increasing numbers of philosophers, where bounded step k is set to be 10 and the inner solver are DLV, Smodel, Cmodel respectively. It is shown that the three ASP solvers can complete the verification effectively. The DLV and Cmodel's capacity are nearly the same, DLV is better than Smodel when dealing with system of small scale. With the increasing numbers of philosophers, smodel perform better and better.

IX. CONCLUSION

An ASP based CSP bounded model checking framework is established to achieve the verification of multiple properties concurrently in one run of a model checker relying on the feature that ASP is completely free of sequential dependencies. The framework turns the CSP model checking problem into a computation problem of answer sets. CTL is extented with events for specification of properties and preprocessing technique is proposed to reduce the overhead of replicated verification of same sub formulas. An ASP description system is constructed, which can be used to describe various forms of CSP processes, and it allows several processes to be composed automatically to generate a new parallel process. Finally all the methods are integrated into a new CSP model checking tool - ACSPChecker. The experiments with the classic dining philosopher's problem show the feasibility and efficiency of our methods.

Although preprocessing of multi-properties is helpful of improving the efficiency of verification in our method, but the efficiency gap with mainstream model checker such as FDR,PAT is still big. Reducing the state space by abstraction techniques and furthermore improving the efficiency of our verification method will be the research emphasis in the future.

ACKNOWLEDGMENT

The work is supported by National Natural Science Foundation of China (No.61472179, No.61262008, No.61562015). Guanxi Natural Science Foundation of China (2015GXNSFAA139307, 2015GXNSFDA139038). The High Level of Innovation Team of Colleges and Universities in Guanxi, Outstanding Scholars Program Funding and Program for Innovative Research Team of Guilin University of Electronic Technology.

REFERENCES

[1] Clarke E M, Grumberg O, Peled D. Model checking[M]. MIT press, 1999.

[2] Edmund M. Clarke: The Birth of Model Checking. 25 Years of Model Checking 2008: 1-26

[3] Biere A, Cimatti A, Clarke E M, et al. Bounded model checking[J]. Advances in computers, 2003, 58: 117-148.

[4] Sun J, Liu Y, Dong J S, et al. Bounded model checking of compositional processes[C]//Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on. IEEE, 2008: 23-30.

[5] Wehrheim H, Wonisch D. Compositional CSP Traces Refinement Checking. Electr. Notes Theor. Comput. Sci. 250(2): 135-151 (2009)

[6] Moffat N, Goldsmith M. Assumption-Commitment Support for CSP Model Checking. J. Autom. Reasoning 41(3-4): 365-398 (2008)

[7] Wen YJ, Wang J, Qi ZC. Compositional model checking and compositional refinement checking of concurrent reactive systems. Journal of Software, 2007,18(6):12701281.

[8] Hoare C A R. Communicating sequential processes[M]. Englewood Cliffs: Prentice-hall, 1985.

[9] Roscoe A W, Hoare C A R, Bird R. The theory and practice of concurrency[M]. Englewood Cliffs: Prentice Hall, 1998.

[10] Armstrong P, Goldsmith M, Lowe G, et al. Recent developments in FDR[C]//Computer Aided Verification. Springer Berlin Heidelberg, 2012: 699-704.

[11] Joel Ouaknine, Hristina Palikareva, A.W. Roscoe, James Worrell: Static Livelock Analysis in CSP. CONCUR 2011: 389-403

[12] Palikareva H, Ouaknine J, Roscoe B. Faster FDR counterexample generation using SAT-solving[J]. Electronic Communications of the EASST, 2009, 23.

[13] Roscoe A W, Gardiner P H B, Goldsmith M H, et al. Hierarchical compression for model-checking CSP or how to check 1020 dining philosophers for deadlock[M]//Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 1995: 133-152.

[14] Sun J, Liu Y, Dong J S. Model checking CSP revisited: Introducing a process analysis toolkit[M]//Leveraging Applications of Formal Methods, Verification and Validation. Springer Berlin Heidelberg, 2009: 307-322.

[15] Palikareva H, Ouaknine J, Roscoe A W. SAT-solving in CSP trace refinement[J]. Science of Computer Programming, 2012, 77(10): 1178-1197.

[16] Leuschel M, Fontaine M: Probing the Depths of CSP-M: A New fdr-Compliant Validation Tool. ICFEM 2008: 278-297

[17] Leuschel M, Butler M. ProB: an automated analysis toolset for the B method[J]. International Journal on Software Tools for Technology Transfer, 2008, 10(2): 185-203.

[18] Khanh N T, Sun J, Liu Y, et al. Symbolic Model-Checking of Stateful Timed CSP Using BDD and Digitization. ICFEM 2012: 398-413

[19] Parashkevov A N, Yantchev J. ARC-a tool for efficient refinement and equivalence checking for CSP[C]//Algorithms amp; Architectures for Parallel Processing, 1996. ICAPP 96. 1996 IEEE Second International Conference on. IEEE, 1996: 68-75.

[20] Bartels B, Kleine M. A csp-based framework for the specification, verification, and implementation of adaptive systems[C]//Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, 2011: 158-167.

[21] Murray T. On the limits of refinement-testing for model-checking CSP[J]. Formal Aspects of Computing, 2013, 25(2): 219-256.

[22] Leuschel M, Massart T, Currie A: How to Make FDR Spin LTL Model Checking of CSP by Refinement. FME 2001: 99-118

[23] Leuschel M: Declarative programming for verification: lessons and outlook. PPDP 2008: 1-7.

[24] Leuschel M: Advanced Techniques for Logic Program Specialisation. AI Commun. 10(2): 127-128 (1997)

[25] Plagge D, Leuschel M. Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more[J]. International journal on software tools for technology transfer, 2010, 12(1): 9-21.

[26] Baral C. Knowledge representation, reasoning and declarative problem solving[M]. Cambridge university press, 2003.

[27] Lifschitz V. What Is Answer Set Programming?[C]//AAAI. 2008, 8: 1594-1597.

[28] Tang C K F, Ternovska E. Model checking abstract state machines with answer set programming[J]. Fundamenta Informaticae, 2007, 77(1): 105-141.

[29] Heljanko K, Niemel I. Bounded LTL model checking with stable models[J]. arXiv preprint cs/0305040, 2003.

[30] Heljanko K, Niemel I. Answer Set Programming and Bounded Model Checking[C]//Answer Set Programming. 2001.

[31] De Angelis E, Pettorossi A, Proietti M. Synthesizing concurrent programs using answer set programming[J]. Fundamenta Informaticae, 2012, 120(3): 205-229.

[32] Huth M, Ryan M. Logic in Computer Science: modeling and Reasoning about systems[M],2005.04.

[33] Leone N, Pfeifer G, Faber W, et al. The DLV system for knowledge representation and reasoning[J]. ACM Transactions on Computational Logic (TOCL), 2006, 7(3): 499-562.

[34] Niemela I, Simons P, Syrjanen T. Smodels: a system for answer set programming[J]. arXiv preprint cs/0003033, 2000.

[35] Lierler Y, Maratea M. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs[M]//Logic programming and nonmonotonic reasoning. Springer Berlin Heidelberg, 2004: 346-350.