KASU JOURNAL OF MATHEMATICAL SCIENCES (KJMS) VOL. 1, ISSUE 2, December 2020
ISSN 2734-3839 (Print), ISSN 2735-962X (E-Copy)
http:/www.journal.kasu.edu.ng/index.php/kjms

KASU JOURNAL OF MATHEMATICAL SCIENCES

A Hybrid Approach to Detecting Deadlock in STGs Unfoldings Using SAT

Abdullah Ismaila Jihad and Ahmad Muhammad Aminu

Department of Computer Science, Kaduna State University Kaduna, Nigeria
a.ismaila@kasu.edu.ng & muhdaminu@kasu.edu.ng

Abstract

Deadlock problem is often seen as one which consists of two parts: the detection, and the elimination of them. A number of techniques for solving the deadlock problem exist. Some of these methods operate directly on the STG level, but they restrict the class of the underlying Petri nets to only, *marked graphs*. While this purely state-based approach is relatively simple and well-studied, the issue of computational complexity for highly concurrent STGs is a serious one due to the *state space explosion* problem. To alleviate this problem, we chose to use methods that do not explicitly enumerate the input sequences, thus avoiding the state space explosion. Using a novel formulation of deadlock with some problem-specific optimization rules on Petri net unfoldings, we proposed a hybrid approach to detecting deadlock in asynchronous circuits. Experiment shows that the technique seems quite competitive on the example benchmark models available to us.

Keywords: asynchronous circuits, signal transition graphs - STG · Petri nets- nets unfolding· deadlock detection· SAT solver -partial order techniques.

1. Introduction

Digital circuits are usually classified into synchronous and asynchronous[1]. An asynchronous circuits, are a sequential digital logic circuits which are not governed by a clock circuit . Instead they often use signals that indicate completion of instructions and operations, specified by simple data transfer protocols as opposed to synchronous circuits, where the operation of the circuit is under the control of a global clock signal. Asynchronous circuits have the potential to be faster, and have advantages in lower power consumption, lower electromagnetic interference, and better modularity in large systems. Asynchronous circuits have become an active area of research in digital logic design[1,2].

Usually, the design of an asynchronous circuit, takes three distinct stages: *specification, implementation and validation*. At the specification stage, the designer first determines an unambiguous description of a circuit expected behaviour. There are various formalisms for specifying a circuit expected behaviour of which Signal Transition Graphs (STG) [3, 10] is one of the latest models [3]. After the desired specification is achieved, the corresponding digital logic that implements the circuit are obtained either manually or by using computer aided synthesis techniques or both, depending on which is more realistic. STG models,are supported by a well-developed public domain tool called PETRIFY for synthesizing them [1,5].

Finally, the obtained circuit implementation needs tobe validated against its specification to ensure its correctnessbefore it is passed to later design stages. Designers choose between techniques of circuit validation: *simulation or formal verification*.

Simulationcan be used to shed light on the correctfunctionality of the future behaviour of a circuit, and determine what can be done to influence that future behaviour. However, simulation is not always feasible since it requiresexamination of all allowable sequencesof actions of the environment, which quickly leads to state space explosion problem.

The formal verification of an asynchronous circuit appliesdifferent schemes to prove correctness. In contrast to simulation, the formal verification methods do not explicitlyenumerate the input sequences, thus avoiding the state spaceexplosion.It is therefore imperative to considerformal verification using models similar to those used forspecification.

Defect in circuits are responsible for the majority of problems found in the semiconductor industry. It is therefore imperative to detect and prevent them from reproducing. To identify defects in circuits a survey was conducted. This survey showed that one of the most frequently detected defects is deadlock. Deadlock problem is

often seen as one which consists of two parts: the detection, and the elimination of them.
There are a number of techniques published recently on the subject of formal detection for deadlock problem in asynchronous circuits (see [3], [12], [13]) for detailed descriptions). These detection approaches can be categorised by (i) the use of timing information ((timed vs untimed); (ii) representation of the circuit state space (reachability vs graph vs Binary vs Decision Diagram (BDD-based) vs unfolding prefix) (iii) circuit types, subdivided according to their functionality (control vs data path) or by size (small vs large controllers).

In [19, 22] we proposed an approach for one of the subproblems central to the formal verification of *Object Petri Nets.* There, wedefined the fundamental rules for transforming object nets into a 1-safe Petri net such that employing affordable resources can handle important questions regarding the verification of various properties using model checking. Also, we established the space upper bound complexity for the transformation. [21]. In essence, this approach consists in development of a software tool for automatic transformation of Object Petri net to obtain an equivalent 1-safe Petri net. We showed how deadlock detection can be characterized in terms of satisfiability formulae that can be given to a SAT solver. It is also worth mentioning that the approach allows one not only to find deadlocks, but also to derive execution paths leading to them without performing a reachability analysis.
In our experiments, we achieved significant speedups on benchmark models available to us using this method.
The exact nature of our detection approach in this paper, isthat we tackle here asynchronous control circuits modelled as STGs, and represented as a Petri netunfolding prefix. Hence, we proposed a hybrid technique for checking deadlock in controllers synthesized with STG using prefixes of unfoldingand SAT solver.
Basically, hybrid model checking with SAT via prefixes of unfolding of Petri nets, requires severalsteps before the prefixes can be fed to the external tools for verification. First, the prefixes are generated. Secondly, properties to be verified are translated and encoded as a *Booleansatisfiability formula*. Thirdly, these formulae, in Conjunctive Normal Form (CNF), popularly known as DIMACS format ([25]) are given as input to a SAT solver, to either obtain a satisfying assignment or to prove there is none. These steps are detailedin Section 3 of this article.
To be able to generate prefixes of unfolding, we chose to use an existing tool instead of developing this functionality from scratch. This tool is *PUNF*[6]it is a suite of tools including MPSAT, PCOMP and MP2DOT which is usedfor generating a finite and complete prefix of a safe Petri nets efficiently.
To obtain the DIMAC representation of these formulae, A software tool, called PrefixtoCNF [19], which supports the automatic translation of a property of STGs formalism that can be exploited for verification into SAT formulae has been developed by us.
To summarize, the main contributions described in this paper are: (i) the translations into Boolean satisfiability formulae of the problems of deadlock detection using finite prefix of STGs, (ii) development of a software workbench called PrefixtoCNF. This software, when implementedcan parse file containing the description of a finite prefix of STG models in particular and any bounded P/T net modelin general, (iii) generates encoding for deadlock problems in terms of satisfiability formulae , which can be checked by a SAT solver, (iv) we present experimental results that support the feasibility of our approach for the deadlock detection problem. The remaining of the paper is structured as follows: Section 2 contains a review of fundamentals from theories of Petri net, signal transition graph, unfolding and propositional satisfiability relevant for our use. Section 3 discusses how we characterized the deadlock detection problem as SAT formulae. Section 4 explains how the detection is done; Section 5 compares our approach to a previous prefix-based deadlock detection method. Section 6 summarizes our current contribution.

2.0 Fundamentals

The verification technique presented in this work rely on the Petri nets-based characterization of asynchronous systems. Thus, this section introduces the relevant background and basic definitions from theories of Petri nets and STGs. We further recall notions related to net unfolding and existing tools for unfolding Petri nets.Also presented here are notions related to Boolean satisfiability,

2.1 Fundamental of Petri nets

Petri nets [23], first introduced by Carl Adam Petri, are a simple yet quite powerful

formalism,suited for describing asynchronous systems. They allow tomodel, and study major aspect of behaviour of such systems, including concurrency, causality and conflict [3].As a general and accepted formal models of concurrency, they are option for studying the principles underlying different classes of systems. However, the size of the net requiredto model a system with complex behaviour can be very large. That is even a relatively small system specification can (and often does) yield a very large state space. To alleviate this problem, a designer can be presented with an implicit compact representation of the system states using finite prefixes of Petri net unfolding. [4].

Petri net, also called Place/Transition net (P/T-net), consists of places represented by circles, holding a varying number of indistinguishable tokens, and transitions represented by rectangles. Arcs are represented by arrows pointing from places to transitions or from transitions to places, and referred to as input and output arcs.

Definition 2.1*A place/transition net (P/T net for short) is a tuple* $N = (P, T, F, W)$ *where* P *is a finite non empty setplaces,* T *is a finite non empty set of transitions, disjoint from* P*,* $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation, between places and transition and* $W: F \longrightarrow \mathbb{N}\backslash\{0\}$ is the arc weight function.*The pair* $(x, y) \in F$ *is called an arc*.

If $W(x, y) = 1$ *for all arcs* $(x, y) \in F$ *we can omit* W *and simply write* (P, T, F)*.A P/T net marking is a function*$m: P \longrightarrow Z_+$*, where* $m(p)$*is called the number of tokens in a place* $p \in P$ *at a marking* m*.The set of places* $\bullet y = \{x \in P | (x, y) \in F\}$ *is called presetof transition* $t \in T$ and $t \bullet = \{x \in P | (y, x) \in F\}$*is called the postset oft,A transition* $t \in T$ *is enabled in a marking* m *iff* $m(p) \geq W(p, t)$ *for all* $p \in \bullet t$ *.A transition* $t \in T$*enabled at a marking* m *i.e.* $m[t >$*, may fire. The successor marking* m' *is defined as* $m'(p) = m(p) - W(p, t) + W(t, p)$ *. We denote this by* $m[t > m'$.

Definition 2.2:*A P/T net system* $\Sigma = (N, m_0)$ *is a net N=(P,T,F) together with an initial marking* m_0*. The zero marking with* $m(p) = 0$ *for all* $p \in P$ *is denoted by* **0**.

Checking whether a Petri net satisfies a certain property isvery important for the analysis of the system the net models. Inparticular, the notions of marking reachability and deadlock freedomare used throughout this paper.

Definition 2.3 The set of reachable markings of Σ is the smallest $(w.r.t. \subseteq)$ *set* $RM(\Sigma)$ *containing* $m_0 \in RM(\Sigma)$ *and such that* $m \in RM(\Sigma) and m[t\rangle m'$ *, for some* $t \in T$ *then* $m' \in RM(\Sigma)$*. A marking* m *is reachable if* $m \in RM(\Sigma)$.

P/T net is 1-safe if all arc weights are 1 and there is at most one token in each place in every reachable marking.

A marking m is deadlocked if at this marking no transitions
are enabled. A Petri net is deadlock-free if none of itsreachable markings is deadlocked.

2.2 Signal Transition Graphs

Here, we begin by giving an intuitive knowledge of Signal Transition Graphs using the features to a small example asynchronous circuit. Furthermore, we recapitulate the formal definition of them as presented in [5].

STGs are a form of labelled Petri nets, in which transitions are labelled with the names of rising and falling circuit signal of a timing diagram.In an STG the transitions are interpreted as signal transitions and the places and arcs capture the causal relations between the signal transitions. Figure 2.1,shows a C-element and a ‘well behaved’ dummy environment that maintains the input signals until the C-element has changed its outputs

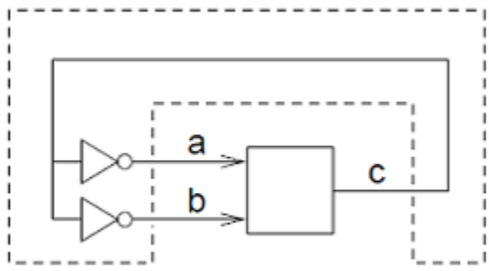


Figure 2.1 A C-element and dummy environment

The intended behaviour of the C-element could be expressed in the form of a timing diagram as shown in the Figure 2,2.

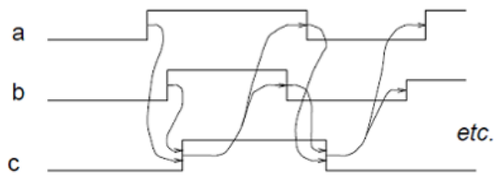


Figure 2.2 Timing diagram

ISSN 2734-3839 (Print), ISSN 2735-962X (E-Copy)
http:/www.journal.kasu.edu.ng/index.php/kjms

Figure 2.3 also shows the corresponding Petri net specification. The Petri net is marked with tokens on the input places to the a+, and b+ transitions, corresponding to state (a, b, c) = (0, 0, 0). The a+ and b+ transitions may fire in any order, and when they have both fired the c+ transition becomes enabledto fire, etc. Often, STGs are drawn in a simpler form where most places are omitted. Every arc that connects two transitions is then thought of as containing a place. Figure 2.4 shows the STG specification of the C-element.

Definition 2.4*: A Signal Transition Graph (STG) (see [5]) is a triple* $\Gamma \stackrel{\text{def}}{\cong} (\sum, Z, \boldsymbol{\lambda})$ *such that* $\sum = (N, M_0)$ *is a net system,Z is a finite set of signals, generating a finite alphabet* $Z^{\pm} = Z \times \{+, -\}$ *of signal transition labels, and* $\boldsymbol{\lambda} : T \to Z^{\pm}$ *is a labelling function.* The signal transition labels are of the form z^+ or z^-, and denote a transition of a signal z ∈ Z from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively.

Signal transitions are associated with the actions whichchange the valueof a particular signal.

The notation $z^{\pm}$ is use to denotea transition of signal z if we are not particularly interested in its direction.

Definition 2.5: A circuit ([24]) is a triple $C = (V, \mathcal{F}, s_0)$**,** where$V = \{v, v_1, \ldots, v_n\}$ isa set of signals. $\boldsymbol{\mathcal{F}}$ is a mapping $\mathcal{F}: f_{v_i}, v_i \in V$ where f_{v_i} corresponds to the function of logic gate that drives v_i and s_0 is the initial state of the circuit.

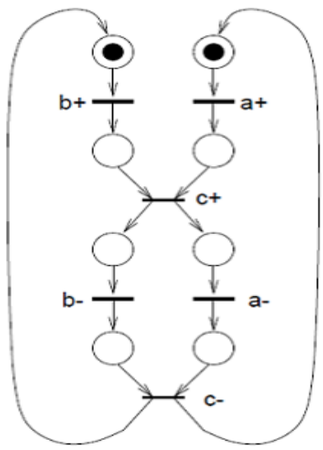


Figure 2.3: A Petri net

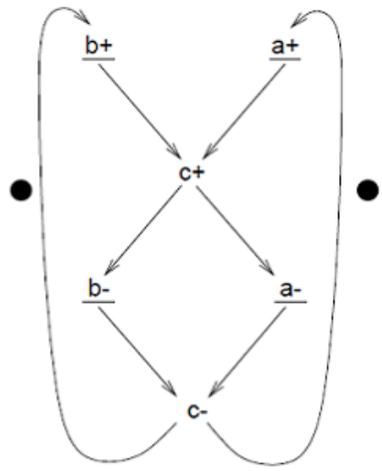


Figure 2.4 STG formalization of the timing diagram

Definition 2.6: A circuit Petri net Γ associated with a circuit C is a type of STG that satisfies the following properties:

1. For each signal z∈Z, a pair of complementary places, p_z^0 and p_z^1, Each z^+–labelled transition has p_z^0 in its preset and p_z^1 in its postset, and each z^-–labelled transition has p_z^1 in its preset and p_z^0 in its postset. Exactly one of the two places is marked at the initial state, accordingly to the initial state $s_0 \in c$ signal z.
2. at any reachable state of an STG augmented with such places, p_z^0 (respectively, p_z^1) is marked iff the value of z is 0 (respectively,1). Thus, if a transition labelled by z^+ (respectively, z^-) isenabled then the value of z is 0 (respectively, 1), which in turn guarantees the consistency
3. Γ is *consistent* if, for every reachable marking M, all firing sequences σ from M_0 to M have the same *encoding vector* Code(M) equal to $v^0 + v^\sigma$ and this vector is binary, i.e., Code(M) $\in \{0,1\}^{|z|}$. Such a property guarantees that, for every signal z ∈ Z, the STG satisfies the following two conditions: (i) the first occurrence of z in the labelling of any firing sequence of Γ starting from M_0 has the same sign (either rising or falling); and (ii) the transitions corresponding to the rising and falling edges of z alternate in any firing sequence of Γ. In this work it is assumed that all the STGs considered are consistent. We will denote by $Code_z(M)$ the component of

Code(M) corresponding to a signal z ∈Z..

Circuit synthesis with STG involves checking the necessary and sufficient condition for the STGs to be implementable as logic circuit. Thus, satisfying key properties:

- Normalcy: normalcy is a necessary condition for an STG to be implementable as a logic circuit built of gates whose characteristic functions are monotonic.
- positive normalcy (or p-normalcy) condition $w.r.t.$ an output signal $z \in Z_0$if for every pair of reachable states M' and M'', Code(M') $Code(M') \leq Code(M'')$ implies $Nxtz(M') \leq Nxtz(M'')$.
- Negative normalcy (or n-normalcy) condition $w.r.t.$ an output signal $z \in Z_0$ if for every pair of reachable states $M' \; and \; M'', \leq$ Implies *Nxtz(M') ≥ Nxtz(M'')*
- Finally, Γ is normal if it either p-normal or n-normal $w.r.t$. each output signals. It turns out that normalcy implies freedom from flaws ([9]).

STGs satisfying these properties can be implementable as a logic circuit.

2.3 Unfolding Petri net system

Reachability analysis is fundamental for checking whether a P/T net satisfies a certain dynamic property of the system the net models. To Determine if there exist a reachable marking satisfying certain properties the set of reachable markings $RM(\Sigma)$ must be computed. This, however, often suffers from state space explosion problem, and requires state space compression technique to be employed. Interestingly, such techniques exist in the form P/T net unfolding [4]. the finite prefix of a P/T net unfolding is usually able to represent all of the possible behaviour of the net in a very compact way.

Since its introduction [10], the unfolding-based approach has been extensively improved to be able to deal with more complex, and diverse applications, and there are existing tools for unfolding nets available online. Therefore, Petri net verification techniques, particularly of the tools based on unfolding theory cannot be ignored: recent research has shown that many verification problems for unfoldings can be formulated in terms of Boolean satisfiability and very efficiently dealt with by existing SAT solvers [9], [15].

In what follows, we give summaries to basic notions, definitions, and properties related to net unfoldings which we require for use in Section 3. More details can be found in [4], [8].

Definition 2.7 Occurrence net: An *occurrence net* is a net ON $\stackrel{def}{=}$ (B,E,G) where B is the set of *conditions* (places), E is the set of *events* (transitions) and G is a *flow relation*. Two nodes of ON, $y \; and \; y'$, are in *structural conflict*, denoted $y \# y'$,, if there are distinct events $e, e' \in ,E$ such that •e∩•e′ ≠∅and (e, y) and (e′, y′) are in the reflexive transitive closure of the flow relation G, denoted by ⪯. A node y is in *structural self-conflict* if $y \# y$. It is assumed that: ON is acyclic (i.e., ⪯ is a partial order); for every $b \in B, |\bullet b| \leq 1$; for every $b \in B \cup E, \neg(y \# y)$ and there are finitely many y', such that $y' \prec y$, where ≺ denotes the irreflexive transitive closure of G. $Min(ON)$ will denote the minimal w.r.t. ⪯ elements of $B \cup E$. The relation ≺ is the *causality relation*. Two nodes are *concurrent*, denoted $y \; co \; y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$.

A *homomorphism* from an ON to a net system Σ is a mapping $h: B \cup E \rightarrow P \cup T$ such that: $h(B) \subseteq P \; and \; h(E) \subseteq T$ (conditions are mapped to places, and events to transitions); for all $e \in E$, the restriction of h to •e is a bijection between •e and • $h(e)$ and the restriction $of \; h \; to \; e \bullet$ is a bijection between $e \bullet \; and \; h(e) \bullet$ (transition environments are preserved); the restriction of h to $Min(ON)$ is a bijection between Min(ON) and M_0 and for all e, $f \in E, if \bullet e = \bullet f \; and \; h(e) = h(f) \; then \; e = f$ (there is no redundancy).

Definition 2.8 A *branching process* of Σ is a quadruple $\pi \stackrel{def}{=} (B, E, G, h)$ such that(B, E, G) is an occurrence net and h is a homomorphism from it to Σ. A branching process $\pi' = (B', E', G', h')$ of $\sum$ is a *prefix* of a branching process $\pi = (B, E, G, h)$ of $\sum$, denoted $\pi' \sqsubseteq \pi$, if (B', E', G') is a subnet of (B, E, G) such that: B' contains the minimal $(w.r.t. \prec)$ conditions of π; if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and h' is the restriction of $h \; to \; B' \cup E'$.

Definition 2.9 A *configuration* of an occurrence net is a set of events C ⊆ E such that for all $e, f \; C, \neg(e \# f)$ and, for

$every\ e \in C, f \prec e$ implies $f \in C$.Intuitively, a configuration is a partial-order execution, where the order of firing of some of its transitions is not important.

Definition 2.10 A *cut* is a maximal (w.r.t.$\subset$) set of conditions B' such that $b\ co\ b'$, for all distinct $b, b' \in B'$. Every marking reachable from $Min(ON)$ is a cut. Let C be a finite configuration of a branching process π. Then $Cut(C) \stackrel{\text{def}}{=} (Min(ON) \cup C\bullet) \setminus \bullet C$ is a cut; moreover, the multiset $Mark(C) \stackrel{\text{def}}{=} h(Cut(C))$ of places is a reachable marking of Σ. A marking M of Σ is *represented* in π if the latter contains a finite configuration C such that $M = Mark(C)$. Every marking represented in π is reachable, and every reachable marking is represented in the unfolding of Σ.

So, for a given a Petri netΣ the *unfolding* of Σ aims at

building a labelled acyclic net π satisfying two key properties:

- For every reachable marking M of $\sum$, there exists a finite configuration $C\ of\ \pi$ such that $C \cap E_{cut} = \emptyset and\ M = Mark(C)$, and for each such C and every transition t enabled by M, there is an event $e \notin C$ in π.
- Finiteness The prefix is finite.

An unfolding satisfying these two properties can be used for model checking as a condensed representation of the state space of a system.

2.4 Boolean Satisfiability

Propositional Satisfiability (SAT) is one of the classical problems in computer science. Its popularity started when it was proven to be an NP-complete problem in 1971 [11] and since that time, many algorithms were designed to solve this problem efficiently. Being an NP-complete problem, other well-known problems such as graph coloring, vertex cover, Hamiltonian path, and traveling salesman problem, of this class, can be encoded into a SAT instance giving them simpler representation. That makes SAT problem have a wide range of practical real-world applications. This include but not limited to Scheduling [13], VLSI design [14] testing problems [16], automated formal verification of hardware/software design [17]) and a number of reasoning problems in artificial intelligence like planning [18].

The Boolean Satisfiability problem (SAT) consists in finding a satisfying assignment, that is, a mapping $A : Var_{\varphi} \rightarrow \{false; true\}$, defined on the set of variables Var , occurring in φ , a given Boolean formula φ, such that φ evaluates to true. This formula is often assumed to be given in the conjunctive normal form $(CNF)\ \bigwedge_{i=1}^{n} \bigvee_{l \in L_i} l$,, i.e., it is represented as a conjunction of *clauses*, which are disjunctions of *literals*, each literal l being either a Boolean variable or the negation of a Boolean variable. It is assumed that no two literals in the same clause correspond to the same variable. The size of a formula in CNF is defined as the total number of literals in all its clauses, i.e., $\sum_{i=0}^{n} |l_i|$.

The following corresponds to an instance of SAT:

$$\varphi = (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3 \vee x_4 \vee \overline{x_5}) \wedge (x_1 \vee \overline{x_3} \vee x_4). \quad (1)$$

A formula is said to be satisfiable if there is a truth assignment to its variables that makes it true. One possible assignment to the variables that will make the above formula satisfied is:

$$x_1 = true,\ x_2 = false, x_3 = tru,\ x_4 = false\ x_5 = false. \quad (2)$$

There are several algorithms, for SAT problems. Each of these algorithms has different strategy for reaching a solution and each, has two possible outcomes; either it gives answers for both satisfiable and unsatisfiable instances respectively, or it can give answer only for satisfiable instances. Consequently, SAT algorithms are categorized as complete and incomplete algorithms. Some of the leading SAT solvers, e.g., Minisat ([15]), can be used in the incremental mode, i.e., after the algorithm solves a particular SAT instance the user can modify it (e.g., by adding and/or removing a small number of clauses) and execute the solver again. Accordingly, we chose to use the MiniSAT for our experiments.

3 Deadlock detection using SAT

In this section we describe how the finite prefix can be used **as** an input to model checking algorithms, e.g., those searching for deadlocks.At first, we repeat results from [19], [20] regarding the translation of prefixes to SAT. Then we explained how the detection was done.

The translations for dead-lock freedom into SAT formulae φ, is built using the prefix, such that:

- φ will be unsatisfiable iff the system deadlock
- every satisfiable assignment of φ gives a violating configuration
- φ has the form CONF∧VIOL
- Some of the variable of φ are associated with events of the prefix

The principle of the translation is to construct a propositional formula of each configuration wherein there is an event in (direct or otherwise) conflict for any $cut - off$ event in the complete finite prefixes generated it then means that some dead markings are reachable from the reachable marking corresponding to the cut of such a configuration.

The procedure is illustrated using the branching process of the Dining Philosophers system Figure 3.1. The configurations are *partially ordered* traces: the order of execution of concurrent events does not matter and so should not be enforced hence we have to explore fewer runscompared to the interleaving semantics but only one configuration is used see Figure 3.3.
Figure 3.2 shows the net system unfolding respectively. At the initial state, the configuration is set to the minimal configuration of an event in conflict with a given cut-off event. In our example, the configuration is set to the minimal configuration [e] of e_6 (in conflict with the cut-off e_5) as shown in Figure 3.4. To complete the procedure, an event in conflict with the cut-off e_{10} must be introduced into the configuration. Since e_1 satisfies these properties and because the union of its minimal configuration with the already formed configuration also yields to a configuration (there is no event in conflict), the procedure leads to the construction of the final marking of the configuration $\{e_1. e_6\}$. This is shown in Figure 3.5. From the corresponding reachable marking $p_1 + p_7 + p_8 + p_9$ the dead marking $p_{10} + p_{11}$ is reachable.

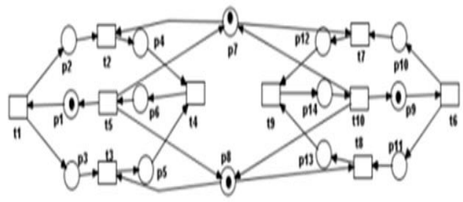


Figure 3.1 Dinning Philosopher PN

Configuration constraint

At the level of a branching process, a deadlock configuration constraint is represented for each event $e \in E \setminus E_{cut}$, e and its immediate predecessors are in the configuration. Therefore, the role of configuration constraints, which we represent as $CONF$, is to ensure that for each event in the set of all events in the prefix excluding *cut-off* events, and its immediate predecessors are in the configuration. If the event is executed, then

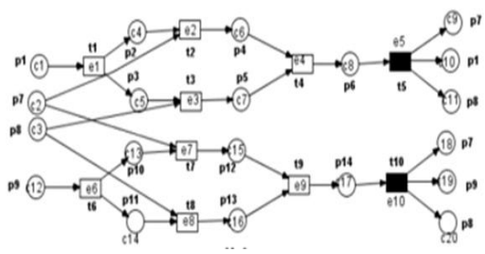


Figure 3.2 the unfolding

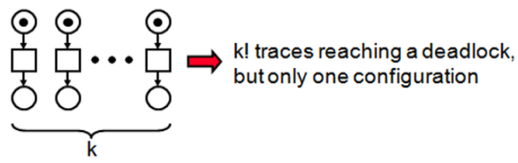


Figure 3.3: K! traces reaching a deadlock

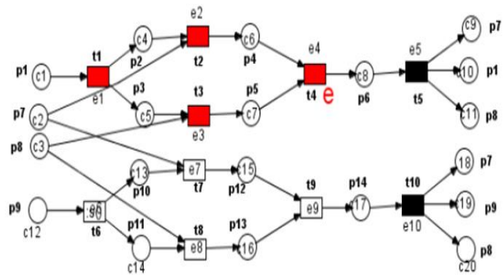


Figure 3.4: Minimal configuration

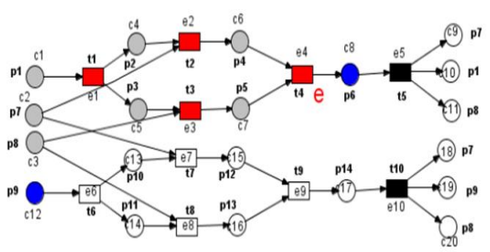


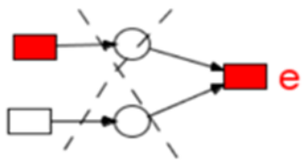
Figure 3.5: Final marking of the configuration

all its direct causal predecessors are also executed, else if the event is executed then no event in direct choice relationship with it can be executed. This is to ensure that it corresponds to the configurations C of the prefix (not just arbitrary sets of events). $CONF$ is formally defined as the conjunction of the formulae.

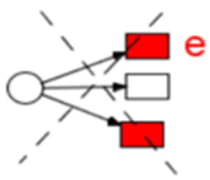
$$Conf = \wedge_e \wedge_{f \in e} (e \rightarrow f) \wedge \wedge_e \wedge_{f \in (e)^{\bullet} \setminus \{e\}} (\neg e \vee \neg f) \quad (3)$$

The above equation ensures that:

- if $e \in C$ then its immediate predecessors are also in C, i.e., C is downward closed $w.r.t. \prec$. Figure 3.6(a) illustrates this notion.
- If e is executed then all its [direct] causal predecessors are also executed and if e is executedthen no events in [direct] choice relationship with e can be executed (Figure 3.3(b)).



(a)



(b)

Figure 3.6 Configuration constraint

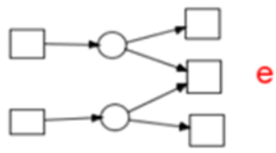
Deadlock violation constrain

The deadlock violation formula illustrated in Figure 3.7 below ensures that no event is enabled to fire, i.e. for every e. Figure 3.7 (a) presents this notion.

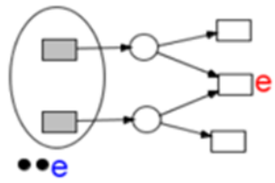
- Some [direct] predecessor of e has not fired: see Figure 3.7 (b) or
- An event in [direct] conflict with e or e itself has fired: see Figure 3.7 (c).

The violation constraint is defined formally as follows:

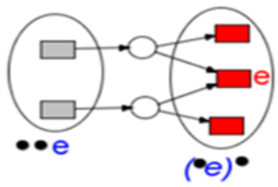
$$VOIL = \wedge_e (\vee_{f \in \bullet e} \neg f \quad \vee \vee_{f \in (\bullet e)^{\bullet}} f) \quad (4)$$



(a)



(b)



(c)

Figure 3.7 Violation constraint

Finally, the problem at hand can be formulated as the SAT problem

$$Deadlock \stackrel{\text{def}}{=} CONF \wedge VOIL \quad (5)$$

4. Deadlock Detection using SAT

We experimented the hybrid approach with our tool by implementing it along with Minisat solver to test out the ideas set forth in this manuscript. By doing so, we check for deadlock in each of thesame benchmarks used in [15, 20].

Tables 1, presents the running times in seconds for the some STGs benchmark used for the test. The running times have been measured using a Pentium® 2.30 GHz, 4.00 GB RAM.

The rows of the tables correspond to different classes of problems. The columns represent: problem statistics, net statistics, unfolding statistics, times in seconds and the marking - deadlock. The other fields of the figures are as follows: |P|: number of places, |T|: number of transitions, |B|: number of conditions, |E| numbers of events, $|\boldsymbol{E_{cut}}|$: number of cut-off events, $\boldsymbol{Var}$: number of variables, Lit : number of literals, Cl : number of clauses CPU: CPU time, SAT: time it takes the solver to parse the input file, DL: YES - the net system has deadlock. NO - net system hasn't deadlock

The marking $vm(n)$ notes that our tool ran out of virtual memory after n seconds when computing the number of literals for the problems, KEY (4) and MMGT (4) respectively.

ISSN 2734-3839 (Print), ISSN 2735-962X (E-Copy)
http:/www.journal.kasu.edu.ng/index.php/kjms

Using our tool along Minisat was able to produce answers for all the examples presented here. An observation that should be made is that the tool shows that classes of problems DAC, DP, ELEVETOR, HARTSTONE and MMGT have deadlock. And classes of problems DME, DPD, DPMF, DPH6, and OVER are without deadlocks.

5. A comparative analysis

We put this work in context with a previous prefix-based deadlock detection method [26]. That method, can translates the problems of deadlock and reachability checking into the problem of finding a stable model of a logic program using finite complete prefixes, in a tool called *mcsmodels-toolset.*

The comparative analysis technique applied common factors: benchmark models, time it takes each tool mcsmodel (resp. PrefixtoCNF) to perform the translation, and the propertyverified. The result shows how efficient the PrefictoCNF performed with respect to accuracy and speed.

Table 2 below, presents the running time in seconds for our approach and that in [26]. The rows of the table correspond to different problems. The columns represent: problem statistics, and tool statistics. The other fields of the table are as follows: DL: YES - means the net system has a deadlock while NO - means not deadlock, DCsmo: time for mcsmodels, average of five runs (The DCsmo columns also includes the logic program translation time, which was always under 10 seconds) DCSAT : time it takes for SAT solver to check if a problem deadlock or doesn't with average of five runs. Both approaches were able to produce answers for all the example models presented.Our approach was most times much faster, see for example some models that are deadlock-free like BDS, DME,DPH(5), DPH(6), FURNACE(1), FURNACE(2), GASQ(3), GAS(4), and for the once that deadlock: ELEVATOR(2, 3, 4), HART(50,75, 100), KEY(2, 3, 4), Q(1), RING(7), RING(9), and RW(12). An important observation is that our approach produces exact answer to whether a problem deadlock or not similar to the mcsmodels approach. This experiment shows that, on these problems set, the functionality of the tool PrefixtoCNF is accurate, speedy , quite robust and competitive.

6 Conclusion

A software tool which supports the automatic translation into Boolean satisfiability formulae of the problems of deadlock detection using finite prefix of unfolding has been developed by us and we proposed a hybrid technique for checking deadlock in controllers synthesized with STGs using the combination of unfolding and SAT solver. Experimental results and comparison with previously existing tool are demonstrated which support the feasibility of our approach for the deadlock detection problem.

Acknowledgements

The authors would like to express their gratitude to all researchers whose works were referenced in his this work, particularly Victor Khomenko who provided us with PUNF and the stg benchmark models used in our experiments. To family and friends.

Table 1 running time in seconds for some STGs:

Problem	Net		Prefix			Formula			Time [s]		DL
	$\|P\|$	$\|T\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	Var	Cl	Lit	CPU	SAT	
DAC(6)	42	34	90	52	0	53	160	447	0.031	0.03	YES
DP(6)	36	24	132	84	24	66	336	1038	0.00	0.00	YES
ELEVATOR3	327	783	4200	3860	1629	2266	36357	149973	1.076	0.09	YES
ELEVATOR4	736	1939	17980	16845	7337	9598	342665	1498034	144.4	0.39	YES
KEY(3)	129	133	8105	6961	2911	4057	55547	213433	5.023	0.06	YES
KEY(4)	164	174	17786	67939	32049	0	0	$Vm(13.0)$	0.015	0.00	YES
MMGT(3)	122	172	6619	5829	2520	3312	118389	462063	29.0	0.09	YES
MMGT(4)	158	232	51882	46886	20945	0	0	$Vm(13.0)$	0.00	0.00	YES
DME(2)	135	98	465	120	4	118	493	1628	0.00	0.00	NO
DPD(6)	54	54	2780	1884	495	1393	13080	46185	0.093	0.01	NO
DPME(8)	87	321	98	199	154	47	364	1720	0.00	0.00	NO
OVER(3)	52	53	270	185	52	134	584	1893	0.015	0.01	NO

Table 2 Deadlock detection running time in seconds

Problem(size)	**Mcsmodel**		**PrefixtoCNF**	
	DL	DC_{smo}	DL	DC_{SAT}
BDS(1)	NO	1.4	NO	0,01
DME(4)	NO	1.4	NO	0.03
DME(5)	NO	4.8	NO	0.07
DME(6)	NO	13.6	NO	0.18
ELEVATOR(2)	YES	0.2	YES	0.01
ELEVATOR(3)	YES	4.3	YES	1.18
ELEVATOR(4)	YES	85.1	YES	171.10
FTP(1)	NO	702.9	NO	2586.55
FURNACE(1)	NO	0.09	NO	0.01
FURNACE(2)	NO	0.20	NO	0.06
GAS(3)	NO	0.80	NO	0.08
GAS(4)	NO	51.5	NO	46.10
HART(50)	YES		YES	0.00
HART(75)	0.10		YES	0.01
HART(100)	YES		YES	0.02
	0.10			

	YES 0.20	
KEY(2)	YES	YES 0.03
KEY(3)	0.20	YES 7.77
KEY(4)	YES 17.80 YES 1287.20	YES 51.50
MMGT(3)	YES 6.1	YES 58.87
MMGT(4)	YES 523.4	YES 3440.05
Q(1)	YES 7.7	YES 4.73
RING(7)	YES 0.1	YES 0.03
RING(9)	YES 0.0	YES 0.06
RW(9)	YES 0.1	YES 0.10
RW(12)	YES 2.0	YES 0.01

References

[1] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev:"Logic Synthesis of Asynchronous Controllers and Interfaces". New York: Springer-Verlag, 2002

[2] S. M. Nowick and M. Singh: "Asynchronous design – part 1: overview and recent advances," IEEE Design & Test, vol. 32, no. 3, pp. 5–18, Jun. 2015.

[3] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, and A. Yakovlev" "Automated verification of asynchronous circuits using circuit petri nets," Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on, pp. 161–170, April 2008.

[4] J. Esparza and K. Heljanko. "Unfoldings : APartial-Order Approach to Model Checking". Springer-Verlag New York Inc, 2008.

[5] V. Khomenko:"Logic Decomposition of Asynchronous Circuits using STG Unfoldings". In Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pages 3–12. IEEE Computer Society Press, 2011.

[6] V. Khomenko: "Petri net Unfolder", http://homepages.cs.ncl.ac.uk/victor.khomenko/home.formal/tools/punf/851/punf_manual_851.pdf 2016

[7] V. Khomenko: Model Checking Based on Prefixes of Petri Net Unfoldings. PhD thesis, University of Newcastle upon Tyne, School of Computing Science 2003

[8] V. Khomenko, M. Koutny and A. Yakovlev: :Detecting State Coding Conflicts in STG Unfoldings Using SAT". Proc. of ICACSD'03, IEEE Comp. Soc. Press (2003) 51–60. Full version: to appear in Special Issue on Best Papers from ICACSD'2003, Fundamenta Informaticae.

[9] N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov, and A. Smirnov: "Towards Synthesis of Monotonic Asynchronous Circuits from Signal Transition Graphs". Proc. of International Conference on Application of Concurrency to Sysytem Design (ICACSD'2001), IEEE Computer Society Press (2001) 179–188.

[10] K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of International Conference on Computer Aided Verification (CAV'1992), G. von Bochmann and D. K.Probst (Eds.). Springer-Verlag, Lecture Notes in Computer Science 663 (1992) 164–174.

[11] ,S. A. Cook: "The complexity of theorem-proving procedures". In Proceedings of the third annual ACM symposium on Theory of computing, pages 151-158. ACM. 1971.

[12] K. Masukura, M. Tomisaka, and T. Yoneda "Verification of asynchronous circuits based on zero-suppressed BDDs". Systems and Computers in Japan, 32:43–54, 2001.

[13] J. M. Crawford, and A. B. Baker: "Experimental results on the application of satisfiability algorithms to scheduling problems". In AAAI, volume 2, pages 1092-1097 1994

[14] S. Devadas: "Optimal layout via Boolean satisfiability". Technical report, Massachusetts Inst of Tech Cambridge Microsystems Technology Labs. 1989.

[15] N. Sorensson, and N. Een: Minisat v1. 13-"A sat solver with conflict-clause minimization SAT, 2005(53):1-2. 2005.

[16] T. Larrabee: "Test pattern generation using Boolean satisfiability" IEEE Transactions 1992.

[17 R. E. Bryant., S. German, and M. N. Velev: "Microprocessor verification using efficient decision procedures for a logic of equality with uninterpreted functions". In International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, pages 1-13. Springer 1999.

[18] H. A Kautz, B. Selman, J. Hoffman: "Planning as satisfiability". In ECAI, volume 92, pages 359-3631992 1992.

[19] I. J. Abdullahi: "Automated Verification of Object Petri Nets based on Transformation, Unfoldings and SAT Solving". Student thesis: Doctoral Thesis. University of South Wales 2018

[20] I. J. Abdullahi: "Model Checking a Class of Object Petri nets Based on Transformation, Unfolding and SAT Solving". In Kaduna Journal of Postgraduate Research (KJPR) 1(1): 125-148 ISBN:2659-1197. 2018.

[21] I. J. Abdullahi: Space Upper bound Analysis for Transformation from Elementary Reference-net System to Low-level P/T nets: In Science World Journal 13(4), pp 100-107 *ISSN 1597-6343 Published by Faculty of Science, Kaduna State University*

[22] I. J. Abdullahi, and B. Muller: "Towards Efficient Verification of ElementaryObject Systems" Proc. CS&P, vol.247, pp.86-100, Humboldt**,** University and CEUR Oct, 2016.

[23] C. A. Petri: "Kommunikation mit Automaten". PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[24] O. Roig. Formal Verification and Testing of Asynchronous Circuits. PhD thesis, Universitat Politecnica de Catalunya, 1997.

[25] H. Zheng, C. Myers, D. Walter, S. Little, and T. Yoneda. Verification of timed circuits with failure directed abstractions. IEEE Transactions on Computer-Aided Design, 25(3):403–412, 2006.

[26] Heljanko, K. "Deadlock and Reachability Checking with Finite CompletePrefixes," http://citeseerx.ist.psu.edu/viewdoc/summary? 1999.