Efficient CTL* Model Checking for Analysis of Rainbow Designs

W. Visser, H. Barringer, D. Fellows, G. Gough, A. Williams
Department of Computer Science
University of Manchester, Manchester M13 9PL, UK
phone: +44 (0) 161-275-6248 fax: +44 (0) 161-275-6211
email: {visserw,howard,donal,graham,alanw}@cs.man.ac.uk

Abstract

We describe an efficient implementation of a CTL* model-checking algorithm based on alternating automata. We use this to check properties of an asynchronous micropipeline design described in the Rainbow framework, which operates at the micropipeline level and leads to compact models of the hardware. We also use alternating automata to characterise the expressive power and model-checking complexity for sub-logics of CTL*.
Keywords: CTL* model-checking, alternating automata, asynchronous hardware micropipeline design.

1 INTRODUCTION

There is renewed interest in asynchronous hardware design, in particular at the VLSI chip level (Birtwistle & Davis 1995, Asy 1997), as an alternative to the (globally clocked) synchronous approach which has dominated the recent past. The asynchronous approach potentially offers low-power, high speed design, which is becoming increasingly difficult with synchronous systems for the sizes of design possible due to the advances in processing technology. This new interest has coincided with the emergence of new asynchronous design methodologies, in particular Sutherland's (1989) Micropipeline technique. This has been used successfully to develop significant designs, including asynchronous versions of the ARM microprocessor (Furber 1995, Furber et al. 1997) developed by the AMULET Group at the University of Manchester.

However, designers are hampered by the current lack of suitable specialised design representations for asynchronous systems, and support or analysis tools for checking properties such as equivalence or deadlock-freedom, the latter being of particular concern to the asynchronous design community.

We have developed the Rainbow asynchronous design framework (Barringer et al. 1996, Barringer et al. 1997) as a means of giving compact abstract descriptions of micropipeline designs. From these, we can generate state-based models for analysis using automated model-checking tools. Of course, model-checking suffers in general from some manifestation of the state-space explosion problem (even though this can sometimes be reduced by the use of special encodings, such as BDDs, and/or partial order techniques). This explosion is exacerbated by the large size of the design, and/or by the size and kind of property to be checked; for example, more expressiveness in the property language can lead to an explosion in size of the representing automaton.

Temporal logics are popular property-description languages since they can describe event orderings without having to introduce time explicitly. There are two main kinds of temporal logics: *linear* and *branching* (Lamport 1980). In linear temporal logics, each moment in time has a unique possible future, while in branching temporal logics, each moment in time has several possible futures. We adopt as our property language the branching time temporal logic CTL* that can express both linear and branching time properties.

Nondeterministic automata have traditionally been the automata of choice when translating temporal formulas to automata (Vardi & Wolper 1986*b*, Vardi & Wolper 1994). However, for both branching and linear time temporal logics, there is an exponential increase when going from formulas to nondeterministic automata. Although this may be acceptable in the linear case, where model checking for the propositional linear time temporal logic (LTL) is known to be PSPACE-complete (Sistla & Clarke 1985), it is not for branching time where the well-known logic CTL has a linear time model checking algorithm (Clarke et al. 1986). Recently it has been shown that *alternating* automata on infinite trees, first introduced in Muller et al. (1986), is a solution for efficient translation of branching time temporal logics to automata (Vardi 1995, Bernholtz et al. 1994, Bernholtz 1995, Bernholtz & Vardi 1995). In this paper we will describe an implementation of Bernholtz's (1995) translation of CTL* to alternating automata*. We will then use the alternating automata as the basis for novel and efficient CTL and CTL* model checking algorithms, which have also been implemented.

Our approach therefore aims to reduce the size of *both* the design and property models. Firstly, the abstract design representation using Rainbow reduces the size of the design model, and secondly we define sub-logics of CTL* for which linear sized alternating automata exist that also allow for efficient model checking. This will be illustrated using a simplified version of the AMULET1 address interface (Paver 1994); we show this to be deadlock-free, whereas an earlier version of the design contains a deadlock.

The rest of the paper is structured as follows. First we give the syntax and semantics of CTL* and review the theory of alternating automata. In sections 3.2(b) and 3.3 we discuss our efficient implementations for, respectively, CTL and CTL* model checkers, based on the algorithms in Bernholtz (1995). We then show how alternating automata can be used to determine sub-logics of CTL* for which linear model checking algorithms exist. Section 4 briefly describes the Rainbow framework and

*To the best of our knowledge this is the first such implementation to have been undertaken.

its underlying semantics. The alternating automata model checker is used to analyse the address interface design in section 4.2, and the results are compared with those obtained from Promela/SPIN (Holzmann 1991).

2 SYNTAX AND SEMANTICS OF CTL*

Since CTL* can express both linear and branching time properties, it is more expressive than the linear time logic LTL (Lichtenstein & Pnueli 1985) and the branching time logic CTL (Clarke et al. 1986). In fact, both these logics are sub-logics of CTL*. Here, only *positive* CTL* formulas will be used, i.e. formulas with negations only applied to atomic propositions. There are two types of formula in CTL*: formulas whose satisfaction is related to states, *state* formulas, and those whose satisfaction is related to paths, *path* formulas. Let $q \in Props$, where $Props$ is a set of atomic propositions. The syntax of CTL* state (S) and path formulas (P) are given by the following BNF rules:

$$
\begin{array}{lll}
S & ::= & true \mid false \mid q \mid \neg q \mid S \wedge S \mid S \vee S \mid AP \mid EP \\
P & ::= & S \mid P \wedge P \mid P \vee P \mid XP \mid P\ U\ P \mid P\ V\ P
\end{array}
$$

A and E are referred to as path quantifiers and X, U and V as path modalities. The sub-logics CTL and LTL are now defined as:

CTL Every path modality is immediately preceded by a path quantifier.
LTL Formulas AP where the only state sub-formulas of P are propositions.

The semantics of CTL* is defined with respect to a *Kripke* structure $K = (Props, S, R, s_0, L)$, where $Props$ is a set of atomic propositions, S is a set of states, $R \subseteq S \times S$ is a transition relation that must be total, s_0 is an initial state and $L : S \to 2^{Props}$ maps each state to the set of atomic propositions true in that state. A path in K is an infinite sequence of states, $\pi = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$ for $i \geq 0$. The suffix $s_i, s_{i+1}, \ldots$ of π is denoted by π^i. $K, s \models \varphi$ indicates that the state formula φ holds at state s and $K, \pi \models \psi$ indicates the path formula ψ holds at the path π of the Kripke structure K. $s \models \varphi$ and $\pi \models \psi$ are written when K is clear from the context. The relation $\models$ is inductively defined as:

- $\forall s$, $s \models true$ and $s \not\models false$
- $s \models p$ for $p \in Props$ iff $p \in L(s)$
- $s \models \neg p$ for $p \in Props$ iff $p \notin L(s)$
- $s \models \varphi_1 \wedge \varphi_2$ iff $s \models \varphi_1$ and $s \models \varphi_2$
- $s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$
- $s \models A\psi$ iff for every path $\pi = s_0, s_1, \ldots$, with $s_0 = s$, then $\pi \models \psi$
- $s \models E\psi$ iff there exists a path $\pi = s_0, s_1, \ldots$, with $s_0 = s$, then $\pi \models \psi$
- $\pi \models \varphi$ for a state formula φ, iff $s_0 \models \varphi$ where $\pi = s_0, s_1, \ldots$
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$
- $\pi \models X\psi$ iff $\pi^1 \models \psi$
- $\pi \models \psi_1\ U\ \psi_2$ iff $\exists i \geq 0$ such that $\pi^i \models \psi_2$ and $\forall j, 0 \leq j < i$, $\pi^j \models \psi_1$

- $\pi \models \psi_1\ V\ \psi_2$ iff $\forall i \geq 0$ such that $\pi^i \not\models \psi_2$ then $\exists j, 0 \leq j < i, \pi^j \models \psi_1$

Note that V is the dual of U and is only used to express succinct *positive* form CTL* formulas. In the rest of the paper the following two abbreviations will also be used: $F\varphi = true\ U\ \varphi$ and $G\varphi = false\ V\ \varphi$.

3 AUTOMATA-THEORETIC BRANCHING TIME MODEL CHECKING

Automata over infinite words have long been used as the basis for model checking LTL formulas (Vardi & Wolper 1986*a*, Vardi & Wolper 1994). With each LTL formula a nondeterministic finite automaton over infinite words, *Büchi* automaton, is associated that accepts exactly all the computations that satisfy the formula (Gerth et al. 1995). This translation is exponential in the size of the formula, but this matches the PSPACE complexity of LTL model checking. For branching time logics, a translation to automata over infinite trees is required. Here, however, the exponential blowup in size of the resulting nondeterministic automata (Vardi & Wolper 1986*a*) is unacceptable, since it is known that for some branching time logics a linear model checking algorithm exist (Clarke et al. 1986). Recently it has been shown that *alternating* tree automata are the automata-theoretic counterpart for branching time logics (Bernholtz 1995, Vardi 1995, Bernholtz et al. 1994). Not only does the translation into alternating tree automata allow matching of the linear time complexity of CTL model checking, but also of the space efficiency of CTL model checking. Note that the original CTL model checking algorithm described in Clarke et al. (1986) was a bottom-up algorithm that required the whole state space to be kept in memory throughout the procedure. An introduction to the theory of automata on infinite trees can be found in Thomas (1990). *Alternating* automata on infinite trees generalise nondeterministic tree automata and were first introduced in Muller et al. (1986). We now introduce alternating tree automata and show how efficient CTL and CTL* model checking can then be performed.

3.1 Alternating Tree Automata

For a given set X, let $B^+(X)$ be the set of positive Boolean formulas over X (i.e. boolean formulas built from elements in X using $\wedge$ and $\vee$), where the formulas $true$ and $false$ are also allowed. An *alternating tree automaton* A is a tuple $(\Sigma, D, S, \delta, s_0, F)$. Here Σ is a finite alphabet, $D \subset \mathbb{N}$ is a finite set of branching-degrees, S is a finite set of states, $s_0 \in S$ is the initial state, F is the acceptance condition (the type of condition depends on the type of alternating automata; two types are discussed below) and $\delta : S \times \Sigma \times D \rightarrow B^+(\mathbb{N} \times S)$ is a partial transition function, where $\delta(s,a,k) \in B^+(\{1,\ldots,k\} \times S)$ for each $s \in S$, $a \in \Sigma$ and $k \in D$ such that $\delta(s,a,k)$ is defined. For example, $\delta(s,a,2) = ((1,s_1) \vee (2,s_2)) \wedge ((1,s_3) \vee (2,s_1))$ can choose between four splittings:

- one copy proceeds in direction 1 in state s_1 and another in direction 1 in s_3.
- one copy proceeds in direction 1 in state s_1 and another in direction 2 in s_1.
- one copy proceeds in direction 2 in state s_2 and another in direction 1 in s_3.
- one copy proceeds in direction 2 in state s_2 and another in direction 2 in s_1.

A run r of an alternating automaton A on a tree T is a tree where the root is labelled by s_0 and every other node by an element of $\mathbb{N} \times S$. For example, taking the transition given above then the tree representing the run will have its root labelled by s_0 and the nodes at level 1 will include the label $(1,s_1)$ or $(2,s_2)$ and the label $(1,s_3)$ or $(2,s_1)$. Note that many nodes of r can represent the same node of T. The run is accepting if all its infinite paths satisfy the acceptance condition F. For example, let $inf(\pi)$ be the set of states that occur infinitely often on the infinite path π, then the Büchi acceptance condition $F \subseteq S$ will be satisfied on π iff $inf(\pi) \cap F \neq \emptyset$.

Muller et al. (1986) introduced *weak alternating automata* (WAA). A Büchi acceptance condition, $F \subseteq S$, is used for WAA. Furthermore, there exists a partition of S into disjoint sets, S_i, such that each S_i is either an *accepting set*, i.e. $S_i \subseteq F$, or is a *rejecting set*, i.e. $S_i \cap F = \emptyset$. A partial order $\leq$ can also be defined between the S_i sets, in the following manner: for every $s \in S_i$ and $s' \in S_j$ for which s' occurs in $\delta(s,a,k)$ for some $a \in \Sigma$ and $k \in D$, it follows that $S_j \leq S_i$. Thus, transitions from a S_i either lead to states in the same S_i or a lower one. An infinite path in the run of a WAA will therefore get trapped within some S_i; if this S_i is accepting then the path satisfies the acceptance condition.

WAA can be used to define linear time algorithms for model checking CTL (Bernholtz 1995). Bernholtz et al. (1994) defined *bounded alternation* WAA, that allows also space efficient CTL model checking. In fact, it was shown that CTL model checking is in NLOGSPACE in the size of the Kripke structure. Since the interest here is in CTL* model checking, *Hesitant alternating automata* (HAA) will be defined as in Bernholtz (1995). HAA have an even more restricted transition structure than WAA, but a more powerful acceptance condition. As with WAA, there exists a partial order between disjoint sets S_i of S. Furthermore, each set S_i is classified either as *transient*, *existential* or *universal*, such that for each S_i, and for all $s \in S_i$, $a \in \Sigma$ and $k \in D$ the following holds:

S_i transient: $\delta(s,a,k)$ contains no elements from S_i.
S_i existential: $\delta(s,a,k)$ only contains disjunctively related elements of S_i.
S_i universal: $\delta(s,a,k)$ only contains conjunctively related elements of S_i.

The acceptance condition is a pair of sets of states, (G,B). From the above restricted structure of HAA it follows that an infinite path, π, will either get trapped in an existential or universal set, S_i. The path then satisfies (G,B) iff either S_i is existential and $inf(\pi) \cap G \neq \emptyset$ or is universal and $inf(\pi) \cap B = \emptyset$.

3.2 CTL Model Checking with HAA

Let us first consider the general algorithm for model checking with alternating automata. Given a branching temporal formula φ and a Kripke structure K with degrees in D:

1. Construct the alternating automaton for the formula, $A_{D,\varphi}$.
2. Construct the product alternating automaton $A_{K,\varphi} = K \times A_{D,\varphi}$. This automaton simulates a run of $A_{D,\varphi}$ on the tree induced by the Kripke structure K.
3. If the language accepted by $A_{K,\varphi}$ is nonempty then φ holds for K.

Thus, a non-emptiness check for HAA is required to check CTL/CTL* properties in K. The general non-emptiness check for HAA cannot be done efficiently (Bernholtz 1995). Fortunately, taking the product with the Kripke structure K, results in a 1-letter HAA over words (i.e. a HAA with $|\Sigma| = 1$ and $D = \{1\}$), for which a non-emptiness check can be done in linear time (Bernholtz 1995). Let us now define this product automaton. Let $A_{D,\varphi} = (2^{Props}, D, Q_\varphi, \delta_\varphi, q_0, (G_\varphi, B_\varphi))$ be a HAA which accepts exactly all the D-trees that satisfy φ and let $K = (Props, S, R, s_0, L)$ be a Kripke structure with degrees in D. The product automaton is then an HAA word automaton $A_{K,\varphi} = (\{a\}, S \times Q_\varphi, \delta, (s^0, q_0), (S \times G_\varphi, S \times B_\varphi))$ where δ is defined as:

- Let $Q \in Q_\varphi$, $s \in S$, $succ_R(s) = (s_0, \ldots, s_{(d(s)-1)})$ and $\delta_\varphi(q, L(s), d(s)) = \alpha$. Then $\delta((s,q),a) = \alpha'$, where α' is obtained from α by replacing each atom (c, q') in α by (s_c, q').

(a) Translating CTL to HAA

First, the closure of a CTL (CTL*) formula φ, $cl(\varphi)$, is defined as all *state* subformulas of φ including φ but excluding true and false. Given a CTL formula φ and a set $D \subset \mathbb{N}$ a HAA $A_{D,\varphi} = (2^{Props}, D, cl(\varphi), \delta, \varphi, (G, B))$ can be constructed such that the language $A_{D,\varphi}$ recognised is the set of D-trees satisfying φ. The acceptance condition is (G, B), where G is the set of all EV formulas and B is the set of all AU formulas in $cl(\varphi)$. The transition function for all $a \in 2^{Props}$ and $k \in D$ is the following:

- $\delta(q,a,k)$ = true if $q \in a$
- $\delta(q,a,k)$ = false if $q \notin a$
- $\delta(\neg q,a,k)$ = true if $q \notin a$
- $\delta(\neg q,a,k)$ = false if $q \in a$
- $\delta(\psi_1 \wedge \psi_2, a, k) = \delta(\psi_1, a, k) \wedge \delta(\psi_2, a, k)$
- $\delta(\psi_1 \vee \psi_2, a, k) = \delta(\psi_1, a, k) \vee \delta(\psi_2, a, k)$
- $\delta(AX\psi, a, k) = \bigwedge_{c=0}^{k-1}(c, \psi)$
- $\delta(EX\psi, a, k) = \bigvee_{c=0}^{k-1}(c, \psi)$
- $\delta(A\psi_1\ U\ \psi_2, a, k) = \delta(\psi_2, a, k) \vee (\delta(\psi_1, a, k) \wedge \bigwedge_{c=0}^{k-1}(c, A\psi_1\ U\ \psi_2))$
- $\delta(E\psi_1\ U\ \psi_2, a, k) = \delta(\psi_2, a, k) \vee (\delta(\psi_1, a, k) \wedge \bigvee_{c=0}^{k-1}(c, E\psi_1\ U\ \psi_2))$

- $\delta(A\psi_1\ V\ \psi_2,a,k) = \delta(\psi_2,a,k) \wedge (\delta(\psi_1,a,k) \vee \bigwedge_{c=0}^{k-1}(c,A\psi_1\ V\ \psi_2))$
- $\delta(E\psi_1\ V\ \psi_2,a,k) = \delta(\psi_2,a,k) \wedge (\delta(\psi_1,a,k) \vee \bigvee_{c=0}^{k-1}(c,E\psi_1\ V\ \psi_2))$

The special structure of the HAA follows from the fact that each formula ψ in $cl(\varphi)$ constitutes a singleton set $\{\psi\}$ in the partition and the partial order is defined by $\{\psi_1\} \leq \{\psi_2\}$ iff $\psi_1 \in cl(\psi_2)$. Note how the existential set is formed by the rules for EU and EV and the universal set is formed by AU and AV. Furthermore, since each transition from a state ψ leads to states in $cl(\psi)$ it must follow that an infinite run will get trapped either in an existential or universal set.

Example 1: Consider the CTL formula $\varphi = AFAGp$. For every $D \subset \mathbb{N}$, the HAA for φ is $A_{D,\varphi} = (2^{\{p\}}, D, \{\varphi, AGp\}, \delta, \varphi, (\{\}, \varphi))$, where δ is given by:

q	$\delta(q,\emptyset,k)$	$\delta(q,\{p\},k)$
φ	$\bigwedge_{c=0}^{k-1}(c,\varphi)$	$\bigwedge_{c=0}^{k-1}(c,AGp) \vee \bigwedge_{c=0}^{k-1}(c,\varphi)$
AGp	$false$	$\bigwedge_{c=0}^{k-1}(c,AGp)$

In state φ and if p holds then the automaton can choose either to assume AGp holds or to postpone the eventuality of φ to the future. It would, however, not be able to postpone it forever, since this will violate the acceptance condition: φ must be seen only finitely often in the S_i associated with φ. In state AGp the automaton expects a tree in which on all paths p holds.

(b) Model Checking Algorithm

An NLOGSPACE-complete CTL model checking algorithm is given in Bernholtz et al. (1994). This algorithm, however, is not of linear running time. Here we will present a less space efficient algorithm, that is linear in the size of the Kripke structure times the size of the HAA (number of states in the HAA). Our algorithm is goal directed, that is to say, only the part of the state space required to satisfy (or invalidate) the formula is visited. Also, the state space is generated "on-the-fly" and in a depth-first manner, similar to the LTL model checker SPIN (Holzmann 1991). A stack is kept for all the states in the current path and a hash table for all the states that are no longer on the current path, but have been visited before. Due to the depth-first nature of the state generation each state in the hash table also contains a field indicating its truth-value. The use of the stack is two-fold: firstly, the state at the top of stack is the current state being evaluated, therefore when a state is labelled with either true or false it is removed from the stack and stored in the hash table and the previous state on the stack becomes the current state; secondly, when a new state is generated the stack is used to see if this state can be reached from itself (i.e. it is in a loop). This algorithm is more space efficient than the original CTL model checking algorithm of Clarke et al. (1986), which required the whole state space to be stored.

The model checking algorithm evaluates the boolean expression in the current state of the HAA by recursively labelling each successor state. When the truth-value of an expression is determined, it is stored in the hash table. This result can then be reused when this state is revisited during the evaluation of another expression. This is the major difference between our algorithm and the one in Bernholtz et al. (1994): there information is not stored and can therefore not be reused; the work needs to be redone, and hence the algorithm is not linear. The key point of our efficient CTL model checking algorithm is that the partially ordered sets S_i of the product automaton are distinguishable by the singleton component of the alternating state, i.e. the state of $A_{D,\varphi}$ associated with the formula φ. For this reason, we refer to the states of the product HAA as either transient, existential or universal, depending on whether the part of the state from $A_{D,\varphi}$ is transient, existential or universal. This allows efficient checking of the acceptance condition: when a new state closes a loop (i.e. is on the stack) then the acceptance condition is violated if the state is universal and in B; or its satisfied if the state is existential and in G. In essence therefore, we exploit the structure of HAA (namely that infinite paths get trapped in a set S_i) and the fact that the sets S_i are singleton in the component of the formula's HAA. In the next section it will be shown that for CTL* formulas the sets S_i are no longer singleton and the algorithm therefore needs to be adjusted, making it less efficient.

Example 2: Let us now consider checking whether the formula $AFAGp$ holds in the start state (x, p) (a state is described by its name and the propositions true in the state) of the following Kripke structure K (where $s \to s'$ indicates a state transition):

$$(x,p) \to (x,p) \qquad (x,p) \to (y,\neg p) \qquad (y,\neg p) \to (z,p) \qquad (z,p) \to (z,p)$$

Recall the HAA for $AFAGp$ from Example 1. The initial state of $A_{D,\varphi}$ is φ and since p holds in state x the following expression must be evaluated: $\forall(AGp,x) \bigvee \forall(\varphi,x)$, where $\forall(\alpha, state_K)$ should be read as, evaluate α in all successors of $state_K$ in K. Let us consider the state y to be the first successor of x in which to evaluate AGp; since $\neg p$ holds in y a lookup in the transition table of $A_{D,\varphi}$ returns false. Since AGp is a universal state, this means $\forall(AGp,x)$ is false. Therefore, we now have, that φ holds in x iff $false \bigvee \forall(\varphi,x)$ is true. Now consider the state x itself to be the first successor of x in which to evaluate φ. The product state (φ,x) is however on the stack, and since it is a universal state and in B, the acceptance condition is violated. Hence $\forall(\varphi,x)$ is also false and so too then is AFAGp in x. When our model checker tackles this problem it evaluates more states, since in both cases it first evaluates state y as the first successor of x. Here follows the actual output of the model checker in its *VERBOSE* mode:

```
G = {} B = {AF}
<AF,(x,{p})> = (ForAll<AG,x> OR ForAll<AF,x>)
Succ(<AG,x>): <AG,y> =  FALSE
Succ(<AF,x>): <AF,(y,{})> = ForAll<AF,y>
Succ(<AF,y>): <AF,(z,{p})> = (ForAll<AG,z> OR ForAll<AF,z>)
Succ(<AG,z>): In stack <AG,z>: <AG,z> =  TRUE
```

```
<AF,y> =  TRUE
In stack <AF,x>:
  Invalid
```

3.3 CTL* Model Checking with HAA

Here we will only discuss informally the translation of CTL* formulas into HAA. The interested reader is referred to Bernholtz (1995) for a detailed analysis of this translation. First, *maximal* state sub-formulas of a formula φ, $max(\varphi)$, need to be defined: ψ is a maximal state sub-formula of φ if it is a state sub-formula and there are no state sub-formula of φ for which ψ is also a state sub-formula. For example, let φ = AF(Xq V AFAGp), then $max(\varphi) = \{q, AFAGp\}$. Secondly, observe that complementing a HAA $A = (\Sigma, D, Q, \delta, q_0, (G, B))$ is $\overline{A} = (\Sigma, D, Q, \overline{\delta}, q_0, (B, G))$, where $\overline{\delta}$ is defined as switching all the true and false values and the $\wedge$ and $\vee$ symbols. For example, if $\delta(q, a, k) = p \vee (true \wedge g)$, then $\overline{\delta} = p \wedge (false \vee g)$.

The first step in the translation of φ is to build the HAA for all the formulas in max(φ). The formula φ is now rewritten, as φ', with all the formulas in max(φ) replaced by atomic propositions. The formula φ' now consists of path modalities preceded by a path quantifier (A or E) and only has propositions as state formula, i.e. linear time formula preceded by an A or E. Let us consider the case where we have $\alpha = E\psi$, where ψ is a linear time formula. Informally, the idea is to build a HAA for $E\psi$ over the alphabet $\Sigma' = 2^{max(\alpha)}$ and then to expand it over the alphabet Σ by using the HAA for the formulas in $max(\alpha)$. For the formula $A\psi$, a HAA for $E\neg\psi$ is built in the above fashion and then complemented. In the construction of the HAA for $E\psi$ the crucial point is the construction of a nondeterministic Büchi word automaton that accepts all the infinite words recognised by ψ. A simple translation exists from this automaton to the HAA for $E\psi$ (see Bernholtz (1995)). Unfortunately, the Büchi word automaton is exponential in the size of the ψ (Vardi & Wolper 1994, Gerth et al. 1995). This results in the complete translation from a CTL* formula into a HAA being exponential. Note that we do not translate the linear time formula into an alternating Büchi word automaton, even though this translation is known to be linear (see section 3.4); this is because the reduction to a 1-letter non-emptiness problem is impossible for alternating Büchi word automata, but valid for nondeterministic Büchi word automata (Bernholtz 1995).

Example 3: Consider the CTL* formula $\varphi = AFGp$. Since it is of the form $A\psi$ we need to negate and complement the HAA for $EGF\neg p$. Note that we do not need to construct a HAA for the maximal formula $(\neg p)$ since it is already a proposition. The nondeterministic Büchi word automaton for $GF\neg p$ has the following transition relation M:

$$M(q_0, \{\neg p\}) = M(q_1, \{\neg p\}) = q_1 \qquad M(q_0, \{p\}) = M(q_1, \{p\}) = q_0$$

with the accepting set $\{q_1\}$ and the initial state q_0. We implemented an optimised version of the algorithm given in Gerth et al. (1995) to create Büchi word automata from linear time formulas. From this we construct the HAA for $EGF\neg p$:

q	$\delta(q,\emptyset,k)$	$\delta(q,\{p\},k)$
q_0	$\bigvee_{c=0}^{k-1}(c,q_1)$	$\bigvee_{c=0}^{k-1}(c,q_0)$
q_1	$\bigvee_{c=0}^{k-1}(c,q_1)$	$\bigvee_{c=0}^{k-1}(c,q_0)$

with acceptance condition $(\{q_1\},\{\})$ and initial state q_0. Since this HAA is already defined over the alphabet $2^{Props} = \{\emptyset, p\}$, all that remains is to complement it to get the HAA for φ (call this HAA A_{AFGp}):

q	$\delta(q,\emptyset,k)$	$\delta(q,\{p\},k)$
q_0	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_0)$
q_1	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_0)$

with acceptance condition $(\{\},(\{q_1\})$ and initial state q_0.

This last example was chosen specifically because the formula $AFGp$ is not expressible in CTL: in Muller et al. (1988) it is shown that CTL formulas can be translated into WAA, but here we show that $AFGp$ cannot be translated into a WAA. Both the states q_0 and q_1 in A_{AFGp} are in the same universal set, but only one of them (q_1) is in the acceptance condition. In a WAA all partially ordered sets must either be accepting or rejecting, and this is clearly impossible here, since the universal set is neither accepting nor rejecting with regards to the Büchi acceptance condition $F = \{q_1\}$. A consequence of this is of course that the model checking algorithm for CTL, given in the previous section, must be extended to handle CTL* formulas. Specifically, it is no longer sufficient only to consider states in B or G that close loops to be of interest when deciding on the satisfaction of the acceptance condition. We now need to know whether a state in B and in a universal set is reachable from itself (in which case the acceptance fails) or whether a state in G and in an existential set is reachable from itself (in which case the acceptance condition is satisfied). We adopted the algorithm proposed in Courcoubetis et al. (1992) and implemented in SPIN (Holzmann et al. 1996), where a second depth-first search is started whenever a state in a universal (existential) set and in B (G) is backtracked, to see if this state is reachable from itself. Note that this second search does not increase the memory required a great deal (only a small amount for every state to distinguish between the states reached in the first and second search), but the time may double in the worst-case (when no cycle is found and all reachable states are visited in the second search). Here follows the output of the model checker when the formula $AFGp$ is checked in the Kripke structure of Example 2:

```
G = {} B = {q1}
<q_0,(x,{p})> = ForAll<q_0,x>
Succ(<q_0,x>): <q_0,(y,{})> = ForAll<q_1,y>
Succ(<q_1,y>): <q_1,(z,{p})> = ForAll<q_0,z>
Succ(<q_0,z>): In stack <q_0,z>: <q_0,z> =  TRUE
<q_1,z> =  TRUE
<q_0,y> =  TRUE
In stack <q_0,x>:
 Satisfied
```

3.4 Model Checking Sub-Logics of CTL*

In the previous section it was mentioned that model checking for full CTL* is exponential in the size of the formula, since the translation of the linear fragments of the formula into a nondeterministic Büchi word automaton is exponential. It was also mentioned that translating a linear time formula to alternating Büchi word automata is linear, but the non-emptiness problem for these automata cannot be reduced to the 1-letter non-emptiness problem required for efficient model checking. The question is therefore: is there a sub-logic of LTL for which the non-emptiness problem of the associated alternating word automata can be reduced to the 1-letter non-emptiness problem? Here we show that such a sub-logic does exist and can be used to define a sub-logic of CTL* for which model checking can be done in linear time. The construction of this logic is based on the observation that nondeterministic Büchi word automata can only express existential choice, whereas alternating Büchi word automata can express both universal and existential choice. Let us first consider the translation of LTL formula into alternating Büchi word automata (Vardi 1995).

Given an LTL formula φ, an alternating Büchi word automaton $A_\varphi = (2^{Props}, S, s_0, \delta, F)$ can be constructed such that the language it accepts is exactly the set of infinite words satisfying φ. The set S is all the sub-formulas of φ and the acceptance condition F includes all sub-formulas of the form $\psi_1\ V\ \psi_2$. The transition relation, δ, is defined as follows:

- $\delta(q,a)$ = true if $q \in a$
- $\delta(q,a)$ = false if $q \notin a$
- $\delta(\neg q,a)$ = true if $q \notin a$
- $\delta(\neg q,a)$ = false if $q \in a$
- $\delta(\psi_1 \wedge \psi_2, a) = \delta(\psi_1, a) \wedge \delta(\psi_2, a)$
- $\delta(\psi_1 \vee \psi_2, a) = \delta(\psi_1, a) \vee \delta(\psi_2, a)$
- $\delta(X\psi, a) = \psi$
- $\delta(\psi_1\ U\ \psi_2, a) = \delta(\psi_2, a) \vee (\delta(\psi_1, a) \wedge \psi_1\ U\ \psi_2)$
- $\delta(\psi_1\ V\ \psi_2, a) = \delta(\psi_2, a) \wedge (\delta(\psi_1, a) \vee \psi_1\ V\ \psi_2)$

The universal and existential choice can be seen here in the fact that both $\wedge$ (universal) and $\vee$ (existential) connectives appear in the transition relation. In Vardi (1995) it is proved that translating alternating Büchi word automata to nondeterministic Büchi word automata is exponential. We therefore need to restrict the form of the LTL formulas such that only existential choice can be expressed, i.e. $\wedge$ connectives are not to be allowed in the transition relation. This would mean we create nondeterministic Büchi word automata for free when applying the above translation from the restricted LTL formulas to alternating Büchi word automata. The translation rules for the following three path modalities can produce $\wedge$ connectives: $\psi_1 \wedge \psi_2$, $\psi_1\ U\ \psi_2$ and $\psi_1\ V\ \psi_2$. From the translation rules for these we can see the restricted form should be: $prop \wedge \psi$, $prop\ U\ \psi$ and $\psi\ V\ prop$, where $prop$ is either a proposition or the negation of a proposition. The following sub-logic of CTL*, called LinearCTL* here, can therefore be translated to HAA in linear time:

$$
\begin{array}{lll}
S & ::= & true \mid false \mid q \mid \neg q \mid S \wedge S \mid S \vee S \mid AP_A \mid EP_E \\
P_E & ::= & S \mid S \wedge P_E \mid P_E \vee P_E \mid XP_E \mid S\ U\ P_E \mid P_E\ V\ S \\
P_A & ::= & S \mid P_A \wedge P_A \mid S \vee P_A \mid XP_A \mid P_A\ U\ S \mid S\ V\ P_A
\end{array}
$$

Note also that the HAA created from LinearCTL* formulas has the same special structure we exploited for the efficient checking of acceptance conditions for CTL (section 3.2(b)). Namely, all existential and universal sets are singleton sets. This allows our CTL model checker to check formulas of LinearCTL*. It is also interesting to note that LinearCTL* is a sub-logic of LeftCTL* defined in Schneider (1997). There the idea was to find sub-logics of CTL* for which a translation exists to CTL. LeftCTL* is such a logic, but with a potential exponential blowup in the size of the resulting CTL formula. LinearCTL*is the sub-logic of LeftCTL* for which this translation is linear. The difference between LinearCTL* and LeftCTL* is that the latter includes $P_E \wedge P_E$ and $P_A \vee P_A$ formulas, which explains why the translation to CTL will be exponential.

4 RAINBOW DESIGNS

We can now apply the above model-checking approach to asynchronous hardware designs, checking that they have the required behaviour, expressed as a CTL* formula, and that they do not suffer from problems such as deadlock. We first outline the Rainbow framework for giving compact design representations, and then illustrate how this can lead to a reduction in the size of model generated.

The Rainbow asynchronous design framework offers a suite of unified design and modelling languages for giving mixed-view hierarchical descriptions of asynchronous micropipeline systems. Rainbow includes a control-flow sequential language, called Yellow, that is similar to occam (Burns 1988), but uses an Ada-like rendezvous. A (static) dataflow-style language called Green uses micropipeline communication as primitive, thus hiding lower-level handshaking control components — we will concentrate on the latter here.

Micropipeline communication between sender and receiver components involves a 3-part handshake, with a *request* control signal from the sender indicating that it is ready to offer some data, the receiver reading the data, and then giving an *acknowledge* signal to the sender once this is complete. In Rainbow, the micropipeline communication is atomic, both at the user-language level — only a single (data) channel and the data transfer activity on the channel is seen by the designer — and at the semantics level. In other description methods, such as CCS (Milner 1989), occam or Promela, then the full handshake must be explicitly encoded, by the user and in the semantics, because the communication primitive is simply a synchronisation.

Green includes a (state-full) buffer node which can either read a value on its input channel when empty, or output a value when full. Other nodes are stateless; once the required values are present on their inputs, then the outputs can be generated, but the inputs are not released until all the outputs have been consumed. There is no implicit buffering between nodes.

A common underlying formal semantics for the component languages is given via a process term language called APA using SOS-style transition rules. The semantics ensures that full interworking between components is supported at the micropipeline level, and provides the basis for formal analysis of designs. This will be used below to construct the automaton representing the address interface example.

4.1 Example: Address Interface

Figure 1 shows the top-level hierarchical Green dataflow description of a simplified version of the AMULET1 microprocessor address interface (Paver 1994). We first outline its behaviour: it performs several functions, combined (so that only one memory address port MemOut is required on the processor). It generates sequences of increasing PC addresses, by cycling values through PC–MemAddr–Inc; these are used to fetch sequential instructions from memory. It may also receive external requests (from the execute unit in the processor) modelled by the EXE_Input node — in these cases, Arb suspends the normal PC value generation. This may introduce data read/write address values, which need to be passed to MemOut but do not enter the loop. Also, branch addresses can be input, so that the old PC value is replaced by the new new branch value — it is therefore necessary to identify 'old' and 'new' PC values, and this is achieved by tagging them with a 'colour', toggling this for every new value. The Fork node only passes on to MemAddr those PC values which match the current colour, and discards those that do not. The Lock node ensures that a new branch PC value can only be introduced when the old value has been discarded. In this model, we are interested mainly in the control and flow of address values in the interface, and so we can ignore the actual values given for each address, simply rep-

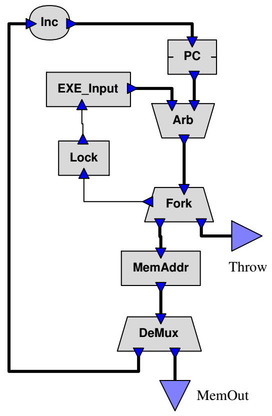

Figure 1 Green Description of Address Interface

resenting each one by its type (PC address or Data address) and its colour (red or green).

The design has the following state-full components: it has three data-buffers — 2 in the PC node (PC1, PC2), and MemAddr. Nodes Fork and EXE_Input store copies of the current colour and the Lock token is used to control branch address generation.

This version with 2 buffers in PC will be shown below to be deadlock-free. However an earlier '1-buffer' version can be shown to deadlock.

4.2 Analysis

A state machine representing the address interface is extracted from the Green description. This contains only the states of the buffers indicated above. The outputs MemOut and Throw are also modelled as buffers that immediately empty.

The alternating automaton model-checker (AltMC) can now be used to analyse the properties of the design. The results are shown in Table 4.2, giving the number of states that need to be stored for a selection of formulae for both the 2-buffer and 1-buffer versions of the interface.

The Green version of the interface only has 32 reachable states*. In this example, deadlock-freedom can be checked by '`AG(EF init)`', where `init` is the initial state (row 1 in Table 4.2). A second useful property to be checked is that if there is a value in PC, then either it has the wrong colour and will be discarded by Fork (i.e. it will reach Throw) or it will again reach PC (row 2). However, this is invalid, since Arb may

*The model-checker can be utilised to determine the number of states simply by having it search the whole state-space for a proposition that is known to be false, for example '`EF(pf)`', where `pf` $\notin L(s)$, and s are all the states in the model.

	Design	Formula	AltMC	SPIN	
1	2-buf	AG(EF init)	53	7615 (deadlock)	valid
2	2-buf	AG(PC $\Rightarrow$ AXAF(Throw $\vee$ PC))	37	375	invalid
3	2-buf	AG(PC $\Rightarrow$ A(GF(DataAddr) $\vee$ XF(Throw $\vee$ PC)))	42	15710	valid
4	1-buf	AG(EF init)	19	51 (deadlock)	invalid

Table 1 Analysis Results

(unfairly) choose forever to accept data address values (DataAddr) from EXE_Input. When this possibility is allowed for in the formula (`GF(DataAddr)` in row 3) then the property is valid*. Finally, the deadlock in the 1-buffer version of the address interface is found (row 4).

4.3 Using Promela/SPIN to Check Properties

The address interface was also modelled in Promela, the input language of the LTL model checker SPIN. Promela simply uses synchronisation as the atomic communication between channels. This exposes the handshaking protocol between the components that was hidden in the Green model; consequently the address interface model much larger (7615 states). Since it is impossible to express the formula AG(EF init) in LTL, we use the built-in deadlock detection function in SPIN. The other two formulas checked are translated into LTL by simply removing the '`A`' path quantifier. The results from SPIN are also shown in Table 4.2. It would also be interesting to compare results with those obtained from a CCS model of the address interface, using the approach advocated in Liu (1995).

5 CONCLUSION

We have presented an implementation of an efficient automata-theoretic CTL* model checker (which to the best of our knowledge is also the first such implementation). We have also shown how this model checker can be used to check properties of an abstraction of an address-interface from a real-world asynchronous processor. No LTL or CTL model checker could have been used for this task, since one of the formulas (of the form $AGEFp$) is not expressible in LTL, and another (of the form $AG(p => A(GFq \vee XFr))$) is not expressible in CTL.

The theory of alternating automata is also shown to be a useful basis for determining

*Note that strong fairness cannot be expressed in CTL and so property 3 is not expressible in CTL.

the expressive power and model checking complexity of sub-logics of CTL*. For example, we show that there exists a sub-logic of CTL*, LinearCTL*, that has a linear translation to alternating automata and can be model checked by the algorithm for CTL formulas, by examining the alternating automata produced by the translation of CTL* formulas in general. One of the advantages of using alternating automata is that we do not need syntax-directed translations between sub-logics in CTL* and CTL formulas, as used in (Schneider 1997, Bernholtz & Grumberg 1994), to show that efficient model checking algorithms exist for these sub-logics, but rather we simply translate them into alternating automata and check these to see if they conform to the requirements for efficient CTL model checking. This approach was taken to show that CTL^2 can be checked by a CTL model checker: all the translations given in Bernholtz & Grumberg (1994) were shown to hold by simply generating the alternating automata for the CTL* formula and translating these automata back to CTL, thus avoiding the cumbersome proofs required in (Bernholtz & Grumberg 1994) to show the translations to be valid. The only formula (AFGp) for which the translation back to CTL did not work (since it cannot be expressed by CTL) is shown to have an efficient model checking algorithm of linear complexity (Bernholtz & Grumberg 1994). Future work in this area will be concerned with finding even more expressive logics with efficient model checking algorithms, by exploiting the theory of alternating automata.

We have demonstrated that the Rainbow design framework is capable of producing compact models directly from asynchronous hardware descriptions given in its user-level languages. This shows the potential benefits of using an abstract application-specific language/framework for modelling micropipelines, instead of employing existing general-purpose languages and explicitly encoding the micropipeline handshake.

6 ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their useful comments. This work was supported by FRD (South Africa), ORS (UK), and EPSRC (UK) research grant GR/K42073.

7 REFERENCES

Asy (1997), *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, Eindhoven, The Netherlands.

Barringer, H., Fellows, D., Gough, G., Jinks, P., Marsden, B. & Williams, A. (1996), Design and simulation in rainbow: A framework for asynchronous micropipeline circuits., *in* A. Bruzzone & U. Kerckhoffs, eds, 'Proceedings of the European Simulation Symposium (ESS'96)', Vol. 2, Society for Computer Simulation International, Genoa, Italy, pp. 567–571. See also the Rainbow Project web pages, URL: `http://www.cs.man.ac.uk/fmethods/projects/AHV-PROJECT/ahv-project.html`.

Barringer, H., Fellows, D., Gough, G. & Williams, A. (1997), Abstract Modelling of Asynchronous Micropipeline Systems using Rainbow, *in* Kloos & Cerny (1997).

Bernholtz, O. (1995), Model Checking for Branching Time Temporal Logics, PhD thesis, The Technion, Haifa, Israel.

Bernholtz, O. & Grumberg, O. (1994), Buy One, Get One Free !!!, *in* 'ICTL '94 : 1st International Conference on Temporal Logic', Vol. 827 of *Lecture Notes in Artificial Intelligence*.

Bernholtz, O. & Vardi, M. (1995), On the Complexity of Branching Modular Model Checking, *in* 'CONCUR '95: 6th International Conference on Concurrency Theory', Vol. 962 of *Lecture Notes in Computer Science*.

Bernholtz, O., Vardi, M. & Wolper, P. (1994), An Automata-Theoretic Approach to Branching-Time Model Checking, *in* 'CAV '94 : 6th International Conference on Computer Aided Verification', Vol. 818 of *Lecture Notes in Computer Science*.

Birtwistle, G. & Davis, A., eds (1995), *Asynchronous Digital Circuit Design*, Springer.

Burns, A. (1988), *Programming in Occam 2*, Addison-Wesley.

Clarke, E., Emerson, E. & Sistla, A. (1986), 'Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications', *ACM Transactions on Programming Languages and Systems* **8**(2), 244–263.

Courcoubetis, C., Vardi, M., Wolper, P. & Yannakakis, M. (1992), 'Memory-Efficient Algorithms for the Verification of Temporal Properties', *Formal Methods in System Design* **1**, 275–288.

Furber, S. (1995), Computing Without Clocks: Micropipelining the ARM Processor, *in* Birtwistle & Davis (1995), pp. 211–262.

Furber, S., Garside, J., Temple, S. & Liu, J. (1997), AMULET2e: An Asynchronous Embedded Controller, *in* Asy (1997).

Gerth, R., Peled, D., Vardi, M. & Wolper, P. (1995), Simple on-the-fly automatic verification of linear temporal logic, *in* 'Protocol Specification Testing and Verification', Chapman & Hall, Warsaw, Poland, pp. 3–18.

Holzmann, G. (1991), *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, New Jersey.

Holzmann, G., Peled, D. & Yannakakis, M. (1996), On Nested Depth First Search, *in* J.-C. Gregoire, G. J. Holzmann & D. Peled, eds, 'Proceedings of the Second Workshop in the SPIN Verification System', American Mathematical Society, DIMACS/39.

Kloos, C. D. & Cerny, E., eds (1997), *Hardware Description Languages and their Applications (CHDL'97)*, Chapman and Hall, Toledo, Spain.

Lamport, L. (1980), 'Sometimes is sometimes "not never" — on the Temporal Logic of Programs', *Proceedings 7th ACM Symposium on Principles of Programming Languages* pp. 174–185.

Lichtenstein, O. & Pnueli, A. (1985), 'Checking That Finite State Concurrent Programs Satisfy Their Linear Specification', *Proceedings 12th ACM Symposium on Principles of Programming Languages* pp. 97–107.

Liu, Y. (1995), AMULET1: Specification and Verification in CCS, PhD thesis, Department of Computer Science, University of Calgary.

Milner, R. (1989), *Communication and Concurrency*, Prentice Hall, Hemel Hempstead, Herts, England.

Muller, D., Saoudi, A. & Schupp, P. (1986), Alternating Automata, the Weak Monadic Therory of the Tree and its Complexity, *in* '13th International Colloquium on Automata, Languages and Programming', Vol. 226 of *Lecture Notes in Computer Science*.

Muller, D., Saoudi, A. & Schupp, P. (1988), Weak Alternating Automata give a simple Explanation of why Temporal and Dynamic Logics are Decidable in Exponential Time, *in* 'Third Symposium on Logic in Computer Science', pp. 422–427.

Paver, N. (1994), The Design and Implementation of an Asynchronous Microprocessor, PhD thesis, Department of Computer Science, University of Manchester.

Schneider, K. (1997), CTL and Equivalent Sublanguages of CTL*, *in* Kloos & Cerny (1997).

Sistla, A. & Clarke, E. (1985), 'The Complexity of Propositional Linear Temporal Logics', *Journal of the ACM* **32**(3), 733–749.

Sutherland, I. (1989), 'Micropipelines', *Communications of the ACM* **32**(6).

Thomas, W. (1990), 'Automata on Infinite Objects', *Handbook of Theoretical Computer Science* pp. 165–191.

Vardi, M. (1995), Alternating Automata and Program Verification, *in* 'Computer Science Today. Recent Trends and Developments.', Vol. 1000 of *Lecture Notes in Computer Science*.

Vardi, M. & Wolper, P. (1986*a*), An Automata Theoretic Approach to Automatic Program Verification, *in* 'First Symposium on Logic in Computer Science', pp. 322–331.

Vardi, M. & Wolper, P. (1986*b*), 'Automata-theoretic Techniques for Modal Logics of Programs', *Journal of Computer and System Science* **32**(5).

Vardi, M. & Wolper, P. (1994), 'Reasoning about Infinite Computations', *Information and Computation* **115**(1).