On-the-Fly Model Checking of RCTL Formulas

Ilan Beer, Shoham Ben-David, Avner Landver

IBM Science and Technology
Haifa Research Laboratory
Haifa, Israel
landver@vnet.ibm.com

Abstract. The specification language $RCTL$, an extension of CTL, is defined by adding the power of regular expressions to CTL. In addition to being a more expressive and natural hardware specification language than CTL, a large family of $RCTL$ formulas can be verified on-the-fly (during symbolic reachability analysis). On-the-fly model checking, as a powerful verification paradigm, is especially efficient when the specification is false and extremely efficient when the computation needed to get to a failing state is short. It is suitable for the inherently gradual design process since it detects a multitude of bugs at the early verification stages, and paves the way towards finding the more complex errors as the design matures. It is shown that for every erroneous finite computation, there is an $RCTL$ formula that detects it and can be verified on-the-fly. On-the-fly verification of $RCTL$ formulas has moved model checking in IBM into a different class of designs inaccessible by prior techniques.

1 Introduction

CTL Model-Checking [CE81a] [CE81b] is the procedure of deciding whether a given model satisfies a given CTL formula (we use the term *model* to denote a finite, closed, non-deterministic state machine). The main problem of model checking in general is the state explosion problem. That is, the number of states in the model grows exponentially with the number of state variables and therefore, very quickly, models become too large to be model checked. Symbolic model checking, using BDD's, was suggested by McMillan who wrote the model checker SMV [McM93]. SMV has made model checking applicable to real life designs. Nevertheless, the state explosion problem is still the greatest concern of model checking.

Another major concern related to CTL model checking is the specification language. CTL is difficult to use for most users, and requires a new way of thinking about hardware design. The tree of computations is not a natural idea for most hardware designers which tend to think more in terms of a single computation. In addition, even the expert CTL user might have great difficulties expressing some properties in bare CTL.

In this paper we define the specification language $RCTL$ which is an extension of CTL. This extension was motivated by the need of the users for tools to express hardware properties that are difficult to express in CTL. A very common property template is the following: "at the end of every finite computation that agrees with a given <computation-description>, p must be true". In $RCTL$, a <computation-description> is represented

by a regular expression R and the syntax of the above template is $\{R\}(p)$. Using regular expressions to describe sets of computations is a powerful and intuitive way of thinking.

Even though the original motivation for introducing $RCTL$ was its expressive power and ease of use, it was also realized that $RCTL$ has a large subset that can be verified on-the-fly. (In this paper, the term on-the-fly is used for error detection during symbolic reachability analysis.) Let M be a model and let R be a regular expression that represents an erroneous computation of M (computation that contradicts the requirements on M). Let $\mathcal{A}(R)$ be a finite automaton that runs with M and enters the state $match_R$ only if the model M performs a computation that agrees with R. The specification "M has no computations that agree with R" is equivalent to

$$M \times \mathcal{A}(R) \models AG(\neg match_R)$$

This check can be performed on-the-fly as described in [Lon93] and in [EM95]. That is, in the process of reachable state space computation, one checks whether $\mathcal{A}(R)$ enters the state $match_R$. If it does, then model checking is stopped and the above specification fails, otherwise model checking is stopped when the entire reachable state space is computed and the specification passes. There is no need to compute the transition relation for the entire reachable state space, neither to apply model checking algorithms to verify the specification. This is an enormous saving, and the only price is the extra automaton $\mathcal{A}(R)$. Since the number of states in $\mathcal{A}(R)$ is linear in the length of R, and since $\mathcal{A}(R)$ does not influence M ($\mathcal{A}(R)$ is a satellite), in our experience this price is negligible with respect to the benefits of on-the-fly model checking. Our experience also shows that well over 80% of the formulas needed for a typical hardware design can be verified with the above on-the-fly method.

The results of this paper were implemented in 1995 in RuleBase [BBEL], which is an IBM model checker based on SMV. RuleBase reads formulas in $RCTL$ and decides whether it is possible to verify them on-the-fly. Formulas that can not be verified on-the-fly are evaluated using the original CTL model checking algorithm. A large number of errors were detected by RuleBase using on-the-fly verification, usually much faster than they would have been detected with the original algorithm. In many cases, errors were detected that would have been missed by original CTL model checking since that run would not have been completed due to memory explosion. On-the-fly verification of $RCTL$ formulas has moved model checking in IBM into a different class of designs inaccessible by prior techniques.

Translating formulas into state machines is not a new idea. Tableau construction for LTL model checking was given in [LP85]. A different algorithm was presented in [VW86] . In [CGH97], LTL model checking is performed using a tableau construction and running within SMV. Tableau construction for $ACTL$ is presented in [GL94]. In all these referenced works, the construction of the tableau is exponential in the length of the formula. In [CYF94] it is shown how to translate a specific CTL formula into an FSM in order to save run-time, on-the-fly verification is not mentioned there and no other CTL formulas are discussed. Using regular expressions for specifications is discussed in [Wol81] (for LTL) and in [IN97] (for CTL).

The rest of the paper is organized as follows. In the next section we define the specification language $RCTL$. In section 3, we introduce a significant subset of $RCTL$ and

show how its formulas can be verified on-the-fly. Section 4 includes some experimental results, and section 5 concludes the paper.

2 Definition of RCTL

In this section we define Regular CTL ($RCTL$) which is an extension of CTL. Let AP be a non-empty finite set of atomic propositions that includes all the signals of the model under discussion, and the constants $true$ and $false$. Let B be the collection of all boolean expressions over AP. Notice that, modulo logical equivalence, B is finite. For every regular expression R let $\mathcal{L}(R)$ be the language of R over the alphabet B. Let ϵ denote the empty word.

In $RCTL$ regular expressions are used to specify sets of non-empty finite computations. Let M be a model (finite, closed, non-deterministic state machine). We say that the computation $s_0, s_1 \ldots, s_n$ of M belongs to $\mathcal{L}(R)$ when there is a word $b = b_0 b_1 \ldots b_n$ over B such that $b \in \mathcal{L}(R)$ and for every $0 \leq i \leq n$, $(M, s_i) \models b_i$.

The following two definitions are needed before we define $RCTL$. First we define the operation $\sim$ between any two regular expressions. Informally, $Q^\sim R$ is a regular expression whose language consists of all words that are a result of taking a word from $\mathcal{L}(Q)$ and a word from $\mathcal{L}(R)$ and concatenating them with an overlap by "anding" the last letter of the former with the first letter of the latter.

Definition 1. Let Q, R, U, V, W be regular expressions and let $q, r \in B$. We define the regular expression $Q^\sim R$ by:

1. If $Q = \emptyset$ or $R = \emptyset$, then $Q^\sim R = \emptyset$
2. If $Q = Uq$ and $R = rV$, then $Q^\sim R = U(q \wedge r)V$
3. If $Q = UV^*$, then $Q^\sim R = UV^*V^\sim R + U^\sim R$
4. If $Q = U(V + W)$, then $Q^\sim R = UV^\sim R + UW^\sim R$
5. If $R = U^*V$, then $Q^\sim R = Q^\sim UU^*V + Q^\sim V$
6. If $R = (U + V)W$, then $Q^\sim R = Q^\sim UW + Q^\sim VW$

For example, if $Q = q$ and $R = p * P$, then

$$Q^\sim R = (q \wedge p)(p*)P + q^\sim P$$

Next, we define the operator $\mathcal{S}$ that determines, for every non-empty regular expression R, whether $\epsilon \in \mathcal{L}(R)$ or not. That is, $\mathcal{S}(R) = 1$ if and only if $\epsilon \in \mathcal{L}(R)$.

Definition 2. Let Q and R be non-empty regular expressions over B, and $p \in B$.

1. $\mathcal{S}(p) = 0$
2. $\mathcal{S}(QR) = \mathcal{S}(Q) \wedge \mathcal{S}(R)$
3. $\mathcal{S}(R^*) = 1$
4. $\mathcal{S}(Q + R) = \mathcal{S}(Q) \vee \mathcal{S}(R)$

We are now ready to define $RCTL$. The specification language $RCTL$ is an extension of CTL where to the usual CTL temporal operators ($AX, EX, AU, EU, \ldots$) we add infinitely many new temporal operators, one for each regular expression over B. Roughly, for every regular expression R we add the temporal operator $\{R\}()$ and the meaning of $\{R\}(\psi)$ is the following. For every computation $\pi \in \mathcal{L}(R)$, ψ is true in the last cycle of π.

Definition 3. $RCTL$ is the smallest superset of CTL that is closed under all boolean and temporal operators of CTL and in addition satisfies the following condition:

- If $\psi \in RCTL$ and R is a non-empty regular expression over B such that $\mathcal{S}(R) = 0$, then $\{R\}(\psi) \in RCTL$

To formally define the semantics of $RCTL$ we'll need the following. It is well known that for every regular expression R there exists a non deterministic finite automaton with no ϵ-transitions, $\mathcal{A}(R)$, such that $\mathcal{L}(R) = \mathcal{L}(\mathcal{A}(R))$. The number of states in $\mathcal{A}(R)$ is linear in the length of R. For the purposes of what is presented in this paper, $\mathcal{A}(R)$ has the following properties. The input to $\mathcal{A}(R)$ is a stream of elements of B which should be viewed as a computation of M. $\mathcal{A}(R)$ has a match state and a no match state denoted by $match_R$ and no_match_R. The only transitions out of $match_R$ and no_match_R are into no_match_R. The set of initial states is denoted by $\mathcal{I}_R$.

We also need a variant of $\mathcal{A}(R)$ denoted by $\mathcal{A}^s(R)$. It has an additional initial state $idle_R \in \mathcal{I}_R$. The only transition into $idle_R$ is from $idle_R$ itself and there are transitions out of $idle_R$ into every initial state. $\mathcal{A}^s(R)$ has the ability to start its action in the middle of the word (computation) and ignore the prefix of that word simply by staying in $idle_R$ for an arbitrary number of cycles.

Next, the semantics of $RCTL$ is defined in the following manner. For every $\psi \in RCTL$ we define $\mathcal{T}(\psi) \in CTL$ and $\mathcal{E}(\psi)$ which is a collection of automata such that for every model M

$$M \models \psi \iff M \times \Pi(\mathcal{E}(\psi)) \models \mathcal{T}(\psi)$$

We need $\mathcal{E}(\psi)$ because the expressive power of $RCTL$ is larger than that of CTL. This is demonstrated by the $RCTL$ formula

$$\{true((true)(true))^*\}(p)$$

which expresses the fact that p is true in every even cycle. It is well known that this fact can not be expressed in CTL. In the following definition, for every regular expression Q^* ($Q \notin B$) that appears in ψ we'll add the automaton $\mathcal{A}^s(Q)$ to $\mathcal{E}(\psi)$.

Let us define the CTL formula $\mathcal{T}(\psi)$ and the set $\mathcal{E}(\psi)$ for a given $RCTL$ formula ψ. This is done by the following recursive procedure.

Procedure 4 *Initialize $\mathcal{E}(\psi) = \emptyset$. Let R be a non-empty regular expression with $\mathcal{S}(R) = 0$. Let OP_1 be any of the usual unary CTL temporal operators and let OP_2 be any of the usual binary CTL temporal operators. Let $\varphi, \rho \in RCTL$.*

1. $\mathcal{T}(\neg\varphi) = \neg\mathcal{T}(\varphi)$ *and* $\mathcal{E}(\neg\varphi) = \mathcal{E}(\varphi)$
2. $\mathcal{T}(\varphi \vee \rho) = \mathcal{T}(\varphi) \vee \mathcal{T}(\rho)$ *and* $\mathcal{E}(\varphi \vee \rho) = \mathcal{E}(\varphi) \cup \mathcal{E}(\rho)$
3. $\mathcal{T}(OP_1(\varphi)) = OP_1(\mathcal{T}(\varphi))$ *and* $\mathcal{E}(OP_1(\varphi)) = \mathcal{E}(\varphi)$
4. $\mathcal{T}(OP_2(\varphi, \rho)) = OP_2(\mathcal{T}(\varphi), \mathcal{T}(\rho))$ *and* $\mathcal{E}(OP_2(\varphi, \rho)) = \mathcal{E}(\varphi) \cup \mathcal{E}(\rho)$
5. $\mathcal{T}(\{R\}(\varphi))$ *and* $\mathcal{E}(\{R\}(\varphi))$ *are given by the following definition*

Definition 5. Let P, Q and R be non-empty regular expressions, $\varphi \in RCTL$ and $p \in B$. Let $idle_P$ denote the statement "$\mathcal{A}^s(P)$ is in the state $idle_P$" and similarly for $match_P$.

1. (a) $\mathcal{T}(\{p\}(\varphi)) = \neg(p \wedge \neg\mathcal{T}(\varphi))$
   (b) $\mathcal{E}(\{p\}(\varphi)) = \mathcal{E}(\varphi)$
2. (a) $\mathcal{T}(\{pP\}(\varphi)) = \begin{cases} \neg(p \wedge EX(\neg\mathcal{T}(\{P\}(\varphi)))) & \text{if } \mathcal{S}(P) = 0 \\ \neg(p \wedge (\neg\mathcal{T}(\varphi) \vee EX(\neg\mathcal{T}(\{P\}(\varphi))))) & \text{otherwise} \end{cases}$
   (b) $\mathcal{E}(\{pP\}(\varphi)) = \mathcal{E}(\{P\}(\varphi))$
3. (a) $\mathcal{T}(\{p^*\}(\varphi)) = \neg(E[pUp \wedge \neg\mathcal{T}(\varphi)])$
   (b) $\mathcal{E}(\{p^*\}(\varphi)) = \mathcal{E}(\varphi)$
4. (a) $\mathcal{T}(\{p^*P\}(\varphi)) = \begin{cases} \neg(E[pU\neg\mathcal{T}(\{P\}(\varphi))])) & \text{if } \mathcal{S}(P) = 0 \\ \neg(E[pU((p \wedge (\neg\mathcal{T}(\varphi)) \vee (\neg\mathcal{T}(\{P\}(\varphi))))) & \text{otherwise} \end{cases}$
   (b) $\mathcal{E}(\{p^*P\}(\varphi)) = \mathcal{E}(\{P\}(\varphi))$
5. (a) $\mathcal{T}(\{P^*\}(\varphi)) = \mathcal{T}(\{P^{\sim idle_P}(\neg idle_P)^*(match_P \vee idle_P)^{\sim} P + P\}(\varphi)) \wedge \mathcal{T}(\varphi)$
   (b) $\mathcal{E}(\{P^*\}(\varphi)) = \{\mathcal{A}^s(P)\} \cup \mathcal{E}(\varphi)$
6. (a) $\mathcal{T}(\{P^*Q\}(\varphi)) = \mathcal{T}(\{P^{\sim idle_P}(\neg idle_P)^*(match_P \vee idle_P)^{\sim} Q + Q\}(\varphi))$
   (b) $\mathcal{E}(\{P^*Q\}(\varphi)) = \{\mathcal{A}^s(P)\} \cup \mathcal{E}(\{Q\}(\varphi))$
7. (a) $\mathcal{T}(\{P + Q\}(\varphi)) = \mathcal{T}(\{P\}(\varphi)) \wedge \mathcal{T}(\{Q\}(\varphi))$
   (b) $\mathcal{E}(\{P + Q\}(\varphi)) = \mathcal{E}(\{P\}(\varphi)) \cup \mathcal{E}(\{Q\}(\varphi))$
8. (a) $\mathcal{T}(\{(P + Q)R\}(\varphi)) = \mathcal{T}(\{PR\}(\varphi)) \wedge \mathcal{T}(\{QR\}(\varphi))$
   (b) $\mathcal{E}(\{(P + Q)R\}(\varphi)) = \mathcal{E}(\{PR\}(\varphi)) \cup \mathcal{E}(\{QR\}(\varphi))$

Let us emphasize that at the entrance to procedure 4 it is checked that all regular expressions R that appear in ψ satisfy $\mathcal{S}(R) = 0$ (otherwise it is an error). So, for example, $\{P^*\}(\varphi)$ can appear only during the recursion as a sub formula, but is not a legal $RCTL$ formula on its own.

Notice that item 5 and 6 should be invoked only when $P \notin B$, otherwise items 3 and 4 are sufficient. The subset of $RCTL$ that allows the $*$ operation to be applied only to boolean expressions is in fact equal to CTL. For every formula ψ in this subset, $\mathcal{E}(\psi) = \emptyset$ and $\mathcal{T}$ is a mapping into CTL such that for every model M

$$M \models \psi \iff M \models \mathcal{T}(\psi)$$

The main reason for preferring $RCTL$ over CTL as an hardware specification language is not its theoretical expressive power but rather its practical expressive power (i.e. ease of use).

The following formula is an example of $RCTL$ relative ease of use.

$$AG(\{wb^*a(v^*r + v^*wb^*r)\}(d))$$

The CTL version of this formula is

$$AG(\neg(w \wedge (EX(E[bU(a \wedge (EX(((E[vU(r \wedge \neg d)]) \wedge (E[vU(w \wedge (EX(E[bU(r \wedge \neg d)])))])))])))))$$

Sugar is the RuleBase specification language [BBEL]. Many useful Sugar operators are easily defined in $RCTL$. The formula φ *until* p (weak until) means that on all paths, φ is true until p is true, but p could be false forever (in which case φ stays true forever). In $RCTL$, φ *until* p is expressed by

$$\{\neg p^* \neg p\}(\varphi)$$

The $next\_event(p)(\varphi)$ operator states that on all paths, in the next cycle in which p is true, φ is also true. On the paths where p is never true, nothing is being claimed. In $RCTL$, $next\_event(p)(\varphi)$ is expressed by

$$\{\neg p^* p\}(\varphi)$$

Notice that by definition 5, both φ *until* p and $next\_event(p)(\varphi)$ can be expressed in CTL.

3 On-The-Fly Model Checking of RCTL formulas

In order to model check a CTL formula, SMV computes the transition relation of the model and then applies the CTL model checking algorithm to determine the truth value of the given CTL formula. Since the computation of the transition relation on the entire state space is often too costly, an option exists in SMV to first compute the reachable state space (this is an iterative process where at every iteration only the partial transition relation on the new states is needed), then compute the transition relation only on the full reachable state space, and finally apply the CTL model checking algorithm. According to our experience, this three stage SMV computation is by far more efficient for most hardware designs. In many examples, the tasks of computing the transition relation on the full reachable state space and applying the model checking algorithm are the bottlenecks of the whole process. In all examples they consume a significant part of the space and time resources that are needed for model checking.

If a CTL formula has the form $AG(p)$, where $p \in B$, a better technique can be used [Lon93, EM95]. Note that an $AG(p)$ formula states that p is true in every reachable state of the model. Therefore, to disprove this formula, it is sufficient to find one state in which p is false. Let S be the set of states where p is false.

All that one needs to do is to check, after every iteration of the reachable state space analysis, whether the intersection of S with the reachable state space computed so far is empty. If it is not empty, the process is stopped and $AG(p)$ is false, otherwise, the process continues and is terminated when the entire reachable state space has been computed, and in this case, the formula $AG(p)$ is true.

Thus, there is no need to compute the full transition relation, neither to apply the model checking algorithm. This saves significant space and time resources. Furthermore,

since this check is done "on-the-fly", in the cases where the formula fails, only a portion of the reachable states space is computed, saving even more space and time.

Experience shows that in the beginning stages of the design/verification process, most of the formulas that fail, fail quickly, and a short computation is needed to demonstrate the error in either the design or the specification. As the design process progresses, longer and longer computations are needed to reveal errors. This makes the on-the-fly approach very attractive. It finds a large number of easy bugs (in the specification or design) very quickly in the beginning and works harder, as the design/verification matures, to find the more complex errors. Unfortunately, in real life CTL model checking, most of the formulas do not have this desired form of $AG(p)$.

To overcome this limitation, and in order to apply the on-the-fly method to a larger class of formulas $\mathcal{F} \subset RCTL$ (formally defined below), we translate a formula $\psi \in \mathcal{F}$ into a CTL formula of the form $AG(p)$ and an automaton. We then verify the $AG(p)$ formula in a model slightly different from the original model.

$\mathcal{F}$ can be viewed as a generalization of the class of $AG(p)$ formulas. While disproving an $AG(p)$ formula is equivalent to finding a *single bad state*, disproving a formula $\psi \in \mathcal{F}$ is equivalent to finding a *single bad finite computation*.

Note that we do not include in $\mathcal{F}$ formulas such as $AX(\phi) \vee AX(\psi)$ since no single finite computation could demonstrate their failure.

Definition 6. $\mathcal{F}$ is the set of all formulas $\psi \in RCTL$ for which there exists a non empty regular expression R with $\mathcal{S}(R) = 0$ such that $\psi \equiv \{R\}(false)$

The statement $M \models \{R\}(false)$ simply states that the model M has no computations that belong to $\mathcal{L}(R)$.

Being in $RCTL$, formulas of the form $\{R\}(false)$ can be verified as described in Section 2. However, there is an alternative way to verify this type of formulas.

Theorem 7. *For every non-empty regular expression R with $\mathcal{S}(R) = 0$ and for every model M*

$$M \models \{R\}(false) \iff M \times \mathcal{A}(R) \models AG(\neg match_R)$$

This theorem reduces model checking of formulas from $\mathcal{F}$ to model checking of $AG(p)$ type formulas and hence allows one to check formulas from $\mathcal{F}$ on-the-fly with all the benefits that were described above. From theorem 7 it follows that for every erroneous finite computation, there is an $RCTL$ formula that detects it and can be verified on-the-fly.

The price one pays for running on-the-fly is that the model $M \times \mathcal{A}(R)$ is larger than M. However, the number of states in $\mathcal{A}(R)$ is linear in the length of R. In addition, $\mathcal{A}(R)$ has no influence on M (i.e. $\mathcal{A}(R)$ is a satellite). This makes the price of running on-the-fly negligible in light of the benefits one gets from this model checking technique.

In RuleBase, the user writes specifications in $RCTL$ and the tool tries to map into $\mathcal{F}$. If it succeeds, the run continues on-the-fly, otherwise it switches to normal CTL model checking as described in the previous section. The following definition describes a class of $RCTL$ formulas that can be mapped into $\mathcal{F}$.

Definition 8. $\mathcal{G}$, the subset of $RCTL$ that RuleBase verifies on-the-fly is defined recursively. Let $\varphi, \psi \in \mathcal{G}$, $p \in B$ and Q a non-empty regular expression with $\mathcal{S}(Q) = 0$.

1. $p \in \mathcal{G}$
2. $\varphi \wedge \psi \in \mathcal{G}$
3. $(p \rightarrow \varphi) \in \mathcal{G}$
4. $AX(\varphi) \in \mathcal{G}$
5. $AG(\varphi) \in \mathcal{G}$
6. $\{Q\}(\varphi) \in \mathcal{G}$

Many other useful Sugar operators, such as *until* (weak until) and *next_event*, can be translated into $\mathcal{G}$ (see Section 2). The following $RCTL$ formula belongs to $\mathcal{G}$ and hence, by the following theorem, can be verified on-the-fly.

$$AG(\{xy^*z\}(next\_event(x)(AX(next\_event(y)(\psi)))))$$

Theorem 9. $\mathcal{G} \subset \mathcal{F}$.

Proof. We translate every $\varphi \in \mathcal{G}$ into a regular expression $\mathcal{R}(\varphi)$ such that

$$\varphi \equiv \{\mathcal{R}(\varphi)\}(false) \qquad (*)$$

Let us define $\mathcal{R}$ recursively.

Definition 10. Let $\varphi, \psi \in \mathcal{G}, p \in B$ and Q a non-empty regular expression with $\mathcal{S}(Q) = 0$.

1. $\mathcal{R}(p) = \neg p$
2. $\mathcal{R}(\varphi \wedge \psi) = \mathcal{R}(\varphi) + \mathcal{R}(\psi)$
3. $\mathcal{R}(p \rightarrow \varphi) = p^{\sim}\mathcal{R}(\varphi)$
4. $\mathcal{R}(AX(\varphi)) = (true)\mathcal{R}(\varphi)$
5. $\mathcal{R}(AG(\varphi)) = (true)^*\mathcal{R}(\varphi)$
6. $\mathcal{R}(\{Q\}(\varphi)) = Q^{\sim}\mathcal{R}(\varphi)$

It is now straightforward to finish proving $(*)$, recursively on φ, going down the list of items in definition 8.

4 Experimental Results

As was mentioned before, according to our experience, a large portion (over 80%) of all practical formulas belong to the set $\mathcal{G}$ (see definition 8) and hence can be verified on-the-fly. In fact, RuleBase has two alternative modes for model checking formulas that belong to the set $\mathcal{G}$. That is, the original (normal) CTL model checking mode (presented in Section 2) on one hand, and the on-the-fly mode (presented in Section 3) on the other hand. In this section we present results comparing these two modes.

The results presented are of several blocks that belong to an IBM node bus adapter verified by RuleBase. Each of these blocks contains several thousands of state variables (5000 - 7000). After applying reductions techniques (both automatic and manual), the blocks were reduced to several hundreds of state variables each (see Table 1).

Table 1. The Models

Model Name	# of State Variables	# of Iterations	Reachable States
M1	244	202	1.69×10^{9}
M2	167	264	3.47×10^{11}
M3	419	281	3.33×10^{49}
M4	314	81	4.43×10^{15}

Table 2. Results M1

Specification	Mode	Status	Run Time Seconds	BDD Nodes
S1	normal	pass	7020	8.9×10^{5}
	on-the-fly	pass	4468	3.4×10^{5}
S2	normal	fail	2910	1.2×10^{5}
	on-the-fly	fail at iteration 51	911	4.7×10^{4}

On each of these examples (models) we ran several formulas. Each formula ran in both normal mode and on-the-fly mode, starting with the same initial BDD ordering. For the failures, we have indicated at which iteration the failure occurred in the on-the-fly mode. All examples were run on RS/6000 with up to 500 MB of memory.

Table 1 gives some information on these examples. The state variables include the variables of the environment. The number of iterations indicates the maximal number of cycles needed to reach to any given reachable state.

The rest of the tables (2 - 5) compare results of running formulas in both modes. Table 2 presents a slight advantage for the on-the-fly mode. In table 3, the results of formula T1 show that, in "pass" cases, the normal mode might be comparable to the on-the-fly mode though in most of the "pass" cases the advantage is still for the on-the-fly mode (see S1 and R1). It should not be surprising that in many "pass" cases the

Table 3. Results M2

Specification	Mode	Status	Run Time Seconds	BDD Nodes
T1	normal	pass	4590	2.1×10^{5}
	on-the-fly	pass	5408	2.4×10^{5}
T2	normal	fail	4695	2.6×10^{5}
	on-the-fly	fail at iteration 89	1136	1.7×10^{5}
T3	normal	fail	6325	2.5×10^{5}
	on-the-fly	fail at iteration 21	49	3.5×10^{4}

Table 4. Results M3

Specification	Mode	Status	Run Time Seconds	BDD Nodes
R1	normal	pass	17926	3.2×10^6
	on-the-fly	pass	7489	9.7×10^5
R2	normal	fail	18115	3.3×10^6
	on-the-fly	fail at iteration 46	208	8.6×10^4

Table 5. Results M4

Specification	Mode	Status	Run Time Seconds	BDD Nodes
Q1	normal	terminated	–	9.7×10^6
	on-the-fly	fail at iteration 25	770	3.2×10^5

on-the-fly mode requires less space. This happens because in the on-the-fly mode there is no need to compute the full transition relation on the entire reachable state space, only partial transition relations on the new states at every step of the reachability analysis are needed. In addition, the model checking algorithms to verify the specification are not applied in the on-the-fly mode which might save in space as well.

Formula T3 in table 3 shows the remarkable advantage of the on-the-fly mode in "fail" cases. A similar advantage is demonstrated by R2 (table 4).

The biggest advantage for the on-the-fly mode is sharply demonstrated in table 5. The formula Q1 could not run to completion in the normal mode. It successfully computed the reachable state space but it ran out of memory during the construction of the transition relation on the full reachable state space. On the other hand, in the on-the-fly mode it found an error at cycle 21 after less than 13 minutes.

5 Conclusion

We have introduced the specification language $RCTL$ with the motivation of narrowing the usability gap of CTL model checking. We identified a subset of $RCTL$ that can be verified on-the-fly, significantly reducing both space and time. The subset $\mathcal{F}$ of $RCTL$ that can be verified on-the-fly might look small from a theoretical point of view, but in practice consists of most of the formulas that are used in hardware specification (at least in our methodology). Verification of formulas that belong to this subset has been reduced to invariant checking, and model checking of these formulas has been reduced to reachability analysis.

Acknowledgements

We wish to thank Dana Fisman for her helpful feedback and for helping with the implementation, Gregory Ronin and Tali Yatzkar-Haham for their help with preparing the experimental results, and Yaron Wolfsthal for his insightful comments on the draft of the paper.

References

[BBEL] I. Beer, S. Ben-David, C. Eisner, A. Landver, "RuleBase: an Industry-Oriented Formal Verification Tool", in Proc. 33^{rd} Design Automation Conference 1996, pp. 655-660.

[CE81a] E.M. Clarke and E.A. Emerson, "Design and synthesis of synchronization skeletons using Branching Time Temporal Logic", in Proc. Workshop on Logics of Programs, Lecture Notes in Computer Science, Vol. 131 (Springer, Berlin, 1981) pp. 52-71.

[CE81b] E.M. Clark and E.A. Emerson, "Characterizing Properties of Parallel Programs as Fixed-point", in Seventh International Colloquium on Automata, Languages, and Programming, Volume 85 of LNCS, 1981.

[CGH97] E. Clark, O. Grumberg and K. Hamaguchi, "Another Look at LTL Model Checking", Formal Methods in System Design, Volume 10, Number 1, Feb. 1997.

[CYF94] B.Chen, M. Yamazaki, M. Fujita, "Bug Identification of a Real Chip Design by Symbolic Model Checking", Proc. European Design and Test Conference, 1994, pp. 132-136.

[EM95] A. Th. Eiriksson and K.L. McMillan, "Using Formal Verification/ Analysis Methods on the Critical Path in System Design: A Case Study", 7th International Conference, CAV '95, pp. 367-380.

[GL94] O. Grumberg and D.E. Long, "Model checking and modular verification", ACM Trans. on Programming Languages and Systems 16 (3), 1994.

[IN97] H. Iwashita and T. Nakata, "Forward Model Checking Techniques Oriented to Buggy Designs", International Conference on Computer Aided Design, ICCAD '97.

[LP85] O. Lichtenstein an A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification", Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages, Jan. 1985.

[Lon93] D. Long, "Model Checking, Abstraction and Compositional Verification", Ph.D. Thesis, CMU, 1993.

[McM93] K.L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[VW86] Y. Vardi and P. Wolper "An automatic theoretic approach to automatic program verification", Proceeding of the First Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, June 1986.

[Wol81] P. Wolper "Temporal Logic can be more expressive", 22nd Annual Symposium on Foundation of Computer Science, Oct. 1981.