*Journal of Universal Computer Science, vol. 7, no. 2 (2001), 194-209*
*submitted: 1/9/00, accepted: 13/10/00, appeared: 28/2/01 © Springer Pub. Co.*

# Correctness of Efficient Real-Time Model Checking

**Wolfgang Reif**
(University of Augsburg, Germany
reif@informatik.uni-augsburg.de)

**Gerhard Schellhorn**
(University of Augsburg, Germany
schellhorn@informatik.uni-augsburg.de)

**Tobias Vollmer**
(University of Augsburg, Germany
vollmer@informatik.uni-augsburg.de)

**Jürgen Ruf**
(University of Tübingen, Germany
ruf@informatik.uni-tuebingen.de)

**Abstract:** In this paper we describe the formal specification and verification of an efficient algorithm based on bitvectors for real-time model checking with the KIV system. We demonstrate that the verification captures the essentials of the C++ algorithm as implemented in the RAVEN model checker. Verification revealed several possibilities to reduce the size of the code and to improve its efficiency.

**Categories:** D.2.1 – Requirements/Specifications, D.2.4 – Software/Program Verification, F.3.1 – Specifying and Verifying and Reasoning about Programs, F.3.2 – Semantics of Programming Languages, F.4.1 – Mathematical Logic

**Keywords:** Model Checking, Interactive Theorem Proving, Structured Algebraic Specifications, Correctness Proofs, Program Verification, Optimization Techniques, Invariants, Temporal Logic

## 1 Introduction

Model checking is an important technique to detect errors or to prove their absence in safety critical soft- and hardware systems. Model checking automatically verifies properties of state based systems. For efficiency, it is usually implemented using highly optimized data structures and algorithms. On the other hand, when a property can be shown, the only result we usually get from a model checker is a "yes". The absence of a comprehensible proof raises the question: can the model checker be trusted?

In this paper, we will answer this question for the case of the real-time model checker RAVEN [RK97, RK99, Ruf00a]. RAVEN uses time-extended finite state machines (interval structures) to describe systems and a timed version of CTL (clocked CTL, CCTL) to describe their properties. Optimized algorithms based on extended characteristic functions are used to compute the extension sets in the model checker.

Our solution consists in the application of formal methods to ensure the correctness of formal methods: We apply the interactive verification system KIV [RSSB98] to formalize and prove the algorithms of RAVEN.

To our knowledge, our case study is the first to tackle formal verification of a state-of-the-art real-time model checker. This paper is the result of the cooperation of two groups, in the context of a research programme on formal methods for engineering applications: the developer of RAVEN (last author), and the development group of KIV (remaining authors).

The verification of RAVEN in KIV consists of three parts. First, it is shown that a simple, unoptimized model checking algorithm correctly implements the semantics of CCTL. Second, the simple algorithm is optimized using so-called time-jumps, and third the optimized algorithm is implemented using bitvectors. An overview over the verification steps can be found in [RRSV00], which focuses on the modularization of the proof and the verification of the second optimization step, where some errors were found.

This paper is about the implementation correctness of the bitvector implementation. We will compare the C++ code of the RAVEN implementation with the imperative programs defined in KIV, and demonstrate that the latter are suitable to capture the essentials of the former: verification revealed several possibilities to reduce size and improve the efficiency of the RAVEN implementation.

Our paper is organized as follows: Section 2 sketches interval structures, CCTL formulas, and the basic model checking algorithm informally, using an example. Section 3 defines the optimized algorithm. Both the simple and the optimized algorithm use abstract sets of natural numbers to represent time points.

The main part of the paper is Sect. 4 which gives details on the representation of sets by bitvectors and its verification in KIV. It also contains a comparison to the real C++ code. Section 5 concludes the paper.

## 2 Real-Time Model Checking

Model checking is a well established method for the automatic verification of finite state systems. It checks whether a given state transition system satisfies a given property specified as a propositional temporal logic formula.

The approach we examine is developed for timed systems and timed specifications. In this section, we give an informal introduction to the main ideas using an example. This is sufficient for the purpose of the paper. Formal definitions may be found in [RK97, RK99] and in [Ruf00b].

The structures representing real-time systems are interval structures. An interval structure is a state transition system with labeled transitions. The labels are intervals of delay times. Figure 1 shows an interval structure, which we will use as a running example. The initial states are doubly circled. States are marked with the values of propositional variables (so in state $\mathrm{a}\overline{\mathrm{b}}$, $a$ is true, $b$ is false). Each structure has one discrete clock. This clock is reset to zero in the initial states and also if a state change is performed. Possible transitions are displayed as arrows in Fig. 1. If the actual clock value reaches some value labeled at an outgoing transition (i.e. the clock value is an element of an interval) the structure may change to the connected successor state and the clock is reset to zero. To avoid, that the described system rests infinitely long in a state, a state must be left, when the maximal delay time of an outgoing transition is reached. Otherwise,

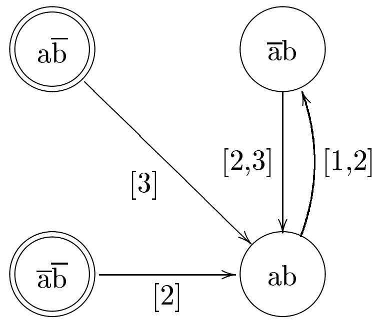


**Figure 1:** Interval structure

the system may choose indeterministically between the possible delay times of an outgoing transition as well as between possible successor states.

To explain the model checking algorithms, we introduce configurations of interval structures. A configuration is a pair of a state and a clock value. For configurations there exist two different kinds of successor configurations, local and global successors.

- The local successor configuration stays in the same state, i.e. the clock has to be increased: $(s, n) \rightarrow (s, n+1)$ if $n+1$ is not the maximal delay time of the state $s$.
- A global successor is one of the connected states together with a zero time: $(s, n) \rightarrow (t, 0)$ if there exists a transition between $s$ and $t$ and the transition is labelled with the delay time $n+1$.

A configuration $c$ is a local (global) predecessor of configuration $c'$, iff $c'$ is a local (global) successor of $c$.

The properties which have to be checked over a given interval structure are formulated in CCTL, an extension of CTL [CES83] with quantitative timed operators. In addition to the usual operators of CTL like $EF\ \varphi$ (there is a path on which eventually $\varphi$ holds) CCTL also has time-bounded operators $EF_n\ \varphi$ (there is a path such that within $n$ steps $\varphi$ holds). For the full set of operators and their formal semantics see [RK99]. We consider an example where only $EF_n\ \varphi$ and $EX\ \varphi$ (there is a step after which $\varphi$ holds) are used. The example property with respect to Fig. 1 is $EF_3(a \wedge b)$. Its meaning is "There is a path such that within 3 steps a state is reached for which $a \wedge b$ holds". The property holds for an interval structure if it holds in all initial states.

The basic idea of model checking is a recursive computation of extension sets. For a formula $\varphi$, the extension set *ext(*$\varphi$*)* is computed as the set of configurations satisfying the formula, i.e.

$$\text{ext}(\varphi) := \{\ (s, n) \mid \text{the configuration } (s, n) \text{ satisfies the formula } \varphi\}.$$

The algorithm starts in the leaves of the syntax tree of the formula and finally determines the extension set for the complete property. Finally the model checking algorithm tests whether the initial states are contained in the extension set for the full formula or not, i.e. whether the structure is a model of the specification or not.

The computation of extension sets is simple for atomic propositions: collect all configurations with a state labeled with the proposition. In our example:

$$\text{ext(a)} = \{(\text{a}\overline{\text{b}},0),(\text{a}\overline{\text{b}},1),(\text{a}\overline{\text{b}},2),(\text{ab},0),(\text{ab},1)\}$$

$$\text{ext(b)} = \{(\overline{\text{a}}\text{b},0),(\overline{\text{a}}\text{b},1),(\overline{\text{a}}\text{b},2),(\text{ab},0),(\text{ab},1)\}.$$

The extension set of propositional operations can be computed by applying the corresponding set operations to the arguments, e.g. the conjunction is realized by intersection, the negation is computed by the complement set with respect to the full set of configurations. Applied to our model checking example we obtain:

$$\text{ext(a} \wedge \text{b)} = \text{ext(a)} \cap \text{ext(b)} = \{(\text{ab},0),(\text{ab},1)\}$$

The temporal operators may be defined recursively (for n $= \infty$ the recursion is unfolded until it becomes stable and computes the least fixpoint). For instance, the EF-operator is formally defined through

$$\text{EF}_n(\varphi) = \varphi \vee \text{EX}(\text{EF}_{n-1}(\varphi)) \tag{1}$$

This shows that the temporal operators may be unrolled by using the EX-operator. Therefore, the computation of extension sets for the EX-operator is very important. The extension set $ext(EX(\varphi))$ contains all predecessor configurations of those in $ext(\varphi)$. We can define a function $EX(C, T)$ which performs $n$ predecessor computations for a given set $C := ext(\varphi)$ and the transition relation $T$. The predecessor configuration computation for our example is shown in the following equation:

$$\text{EX}(\text{ext}(a \wedge b), T) = \{\underbrace{(\overline{\text{a}}\overline{\text{b}},1),(\overline{\text{a}}\text{b},1),(\overline{\text{a}}\text{b},2),(\text{a}\overline{\text{b}},2)}_{\text{predecessors of (ab,0) and}},\underbrace{(\text{ab},0)}_{\text{(ab,1)}}\}$$

Then a call $EF(C, n, T)$ to the program in Fig. 2 computes $ext(EF_n\ \varphi)$ when given $C := ext(\varphi)$. The program implements the tail recursion of formula (1) by a while loop. The implementation of the other temporal operators is similar. Putting things together we have in the example

$$\begin{aligned}
&\text{EF(ext(a} \wedge \text{b),1,T)} = \text{ext(a} \wedge \text{b)} \cup \text{EX(ext(a} \wedge \text{b),T)}\\
= &\{(\text{ab},0),(\text{ab},1),(\overline{\text{a}}\overline{\text{b}},1),(\overline{\text{a}}\text{b},1),(\overline{\text{a}}\text{b},2),(\text{a}\overline{\text{b}},2)\}
\end{aligned}$$

$$\begin{aligned}
&\text{EX(EF(ext(a} \wedge \text{b),1,T),T)}\\
= &\{(\text{ab},0),(\overline{\text{a}}\overline{\text{b}},0),(\overline{\text{a}}\overline{\text{b}},1),(\overline{\text{a}}\text{b},0),(\overline{\text{a}}\text{b},1),(\overline{\text{a}}\text{b},2),(\text{a}\overline{\text{b}},1),(\text{a}\overline{\text{b}},2)\}
\end{aligned}$$

$$\begin{aligned}
&\text{EF(ext(a} \wedge \text{b),2,T)} = \text{ext(a} \wedge \text{b)} \cup \text{EX(EF(ext(a} \wedge \text{b),1,T),T)}\\
= &\{(\text{ab},0),(\text{ab},1),(\overline{\text{a}}\overline{\text{b}},0),(\overline{\text{a}}\overline{\text{b}},1),(\overline{\text{a}}\text{b},0),(\overline{\text{a}}\text{b},1),(\overline{\text{a}}\text{b},2),(\text{a}\overline{\text{b}},1),(\text{a}\overline{\text{b}},2)\}
\end{aligned}$$

```
confset EF(confset C, natinfty n, transrel T)
begin
   confset old := ∅, R := C;
   while n > 0 ∧ old ≠ R do
   begin
      old := R;
      R := C ∪ EX(R,T);
      n := n − 1;
   end
   return R;
end
```

**Figure 2:** The basic EF-algorithm

and obtain

$$\begin{aligned}&\mathrm{ext}(\mathrm{EF}_3(\mathrm{a} \wedge \mathrm{b})) = \mathrm{EF}(\mathrm{ext}(\mathrm{a} \wedge \mathrm{b}),3,\mathrm{T})\\ = &\{(\mathrm{ab},0),(\mathrm{ab},1),(\overline{\mathrm{a}}\overline{\mathrm{b}},0),(\overline{\mathrm{a}}\overline{\mathrm{b}},1),(\overline{\mathrm{a}}\mathrm{b},0),(\overline{\mathrm{a}}\mathrm{b},1),(\overline{\mathrm{a}}\mathrm{b},2),(\mathrm{a}\overline{\mathrm{b}},0),(\mathrm{a}\overline{\mathrm{b}},1),(\mathrm{a}\overline{\mathrm{b}},2)\}\end{aligned}$$

as final result. $ext(EF_3(a \wedge b))$ is (by accident) the full set of all possible configurations, and since the initial states $(\overline{a}b, 0)$ and $(\overline{a}\overline{b}, 0)$ are contained in it, the property $EF_3(a \wedge b)$ holds in our example.

For efficiency the implementation of the model checking algorithms uses MTBDDs to represent configuration sets. For instance, the set $\{(ab, 0), (ab, 1)\}$ is represented as shown in Fig. 3 in the dashed box on the left.

The transition relation $T$ of the interval-structure in Fig. 1 can be represented as the cascaded BDD of Fig. 3 that gives the predecessor configurations for every configuration $(s, 0)$. Please note, that the clock values contained in the leaves correspond to the values attached to the transitions, each decremented by one.

In [RRSV00] we have verified that the basic model checking algorithm correctly implements the semantics of CCTL formulas.

## 3 Time Prediction

If we analyze the computation of predecessor states in the function $EX$ we find two cases: For a configuration $(s, 0)$ global predecessors must be computed by looking at the transition relation, since only a state-changing transition resets the clock to zero. On the other hand, a configuration $(s, n)$ with $n \neq 0$ has only one local predecessor $(s, n-1)$. Of course computing it is much cheaper than computing global predecessors. We also observe, that the global predecessors computed by the calls to $EX$ in Fig. 2 often do not change between iterations of the while loop. As an example the first two calls to $EX$ with $ext(a \wedge b)$ and $EF(ext(a \wedge b), T)$ both compute global predecessors for the same set {(ab,0)} of configurations. Therefore, we use a technique called time prediction to overcome

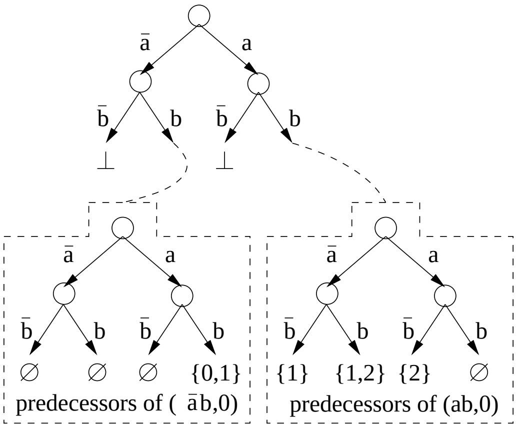


**Figure 3:** BDD representation of the transition relation

the single step iteration and to avoid unnecessary global predecessor computations [RK97, RK98]. The idea is to define a time prediction function that predicts how many steps the global predecessors remain unchanged. This time prediction operation for EF (other operators work similarly, but need a different prediction function) can be computed locally, i.e. for each state separately by a function *local-predict-EF* defined below. The minimum $p$ of the computed prediction times is the time span which can elapse without any change in the set of the global predecessors. In our example the first time prediction is $p = 2$, and one computation of global predecessors is saved. Arguments of *local-predict-EF* are the sets of clock values $c \subseteq \mathbb{N}_0$ and $g \subseteq \mathbb{N}_0$ which contain the last interim result of the computation and the results of the last computation of global predecessors.

$$\text{local-predict-EF}(c, g) := \begin{cases} \text{v} & \text{if v} = \min(\text{c,g} - 1) \wedge \text{v} > 0 \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

The set operation $g - 1$ decrements all members of $g$ by one.

After the prediction the fixpoint iteration of the temporal operators may be performed $p$ times. Analogous to the above, the computation of $p$ iterations with the same set $G$ of global predecessors can be broken down to applications of a function *local-EF* on every state which is recursively defined as

$$\text{local-EF(c,g,p)} = \begin{cases} \text{c} & \text{if p} = 0 \\ \text{local-EF(c,g,p} - 1) \cup \text{local-EF(c,g,p} - 1) - 1 \cup \text{g} \end{cases} \tag{3}$$

Putting together the above definitions and considerations, we obtain the optimized algorithm shown in Fig. 4. The notation *C(s)* used in the program interprets a set of configurations as an extended characteristic function from states to sets of natural numbers, i.e. *C(s)* abbreviates $\{n : (s,n) \in C\}$. Functions *local-EF* and *local-predict-EF* from equations 2 and 3 are called with $c := C(s)$ and $g :=$

```
confset EF(confset C, natinfty n, transrel T)
begin
  confset old := ∅, R := C, G := ∅;
  natinfty p;
  while n > 0 ∧ old ≠ R do
  begin
    old := R;
    G := global-pre(R,T);
    p := min_{s∈S} local-predict-EF(C(s),G(s));
    if p > n then p := n;
    R := ∪_{s∈S} local-EF(C(s),G(s),p);
    n := n − p;
  end
  return R;
end
```

**Figure 4:** The optimized EF-algorithm

*G(s)* for every state *s*. Correctness of the optimization was proven for all temporal operators with KIV. [RRSV00] gives a detailed account on the theorems which were necessary for the proof.

## 4 Time Jumps using Bitvectors

In the previous sections we first introduced a basic model checking algorithm computing on sets of configurations represented by MTBDDs. Then we sketched an optimization called time prediction. The optimized algorithm for each temporal logic operator uses two functions working on leaves of the MTBDDs (i.e. sets of natural numbers). We gave explicit definitions of the two resulting functions *local-EF* and *local-predict-EF* for the temporal operator *EF*. More details on the optimization, and on the verification, that both algorithms correctly reflect the semantics of the underlying logic CCTL, can be found in [RRSV00].

In this paper, we now give details on the verification of an efficient implementation of these functions based on bitvectors. The implementation will consist of two steps, and as in the previous section, we will give details for the EF operator.

The first step, called "time jumping", optimizes the definition of the evaluation function (here *local-EF*). Its definition and correctness proof are discussed in the first subsection. The second step implements the optimized definition of *local-EF* and the definition of *local-predict-EF*. It is defined in Sect. 4.3, using the specification of bitvectors as given in Sect. 4.2. Verification, using proof obligations is discussed in Sect. 4.4 and Sect. 4.5 gives some statistics on the correctness proofs. Finally Sect. 4.6 compares the imperative code verified with the actual C++ code used in RAVEN.

### 4.1 Time Jumping

The main idea of time jumping [Ruf00b] is to replace the recursive definition of *local-EF* by a single complex operation. Fig. 5 shows sample input sets $c = \{2, 17\}$ and $g = \{7, 20\}$ (first two lines of the picture) as well as the corresponding results of *local-EF*$(c, g, m)$ after up to 7 iterations (i.e. $m \in [1, 7]$). Each row in the picture represents a set of natural numbers. If the $n$th box is shaded, the clock value $n$ is contained in the set.

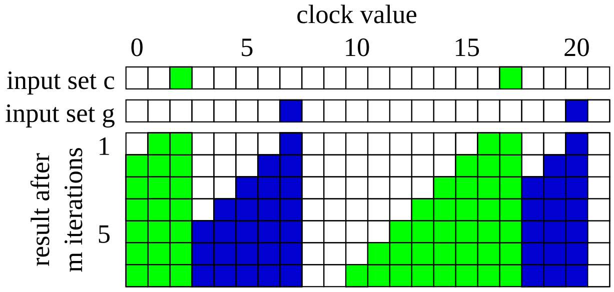


**Figure 5:** Results of computation of *local-EF*

A closer look shows that the result after $n$ iterations can be computed without performing intermediate steps. We may draw triangles starting at the inputs and take the $n$th row of the resulting picture as result of the computation. If we convert the picture into a formula, we get the following theorem:

**Theorem 1.** *The recursive definition (3) of* local-EF *is equivalent to the following, nonrecursive definition:*

$$\begin{array}{ll} & \mathrm{v} \in \text{local-EF}(\mathrm{c},\mathrm{g},\mathrm{p}) \\ = & \quad \exists \mathrm{v}_0 \in \mathrm{c} \ \cap \ [\mathrm{v}, \ldots, \mathrm{v}+\mathrm{p}] \\ & \vee \ \exists \mathrm{v}_0 \in \mathrm{g} \ \cap \ [\mathrm{v}, \ldots, \mathrm{v}+\mathrm{p}-1] \ \wedge \ \mathrm{p} \neq 0 \end{array} \tag{4}$$

The formal proof of each equivalence using KIV is straightforward and can be done within a single day. For *local-EF* it requires about 400 proof steps, over half of them being automatic. About two third of the proof effort lies in proving the case $m = \infty$, where the upper bound $v + m$ of definition (4) collapses – suitable lemmas are needed to show that the results of (4) are also contained in the recursive definition (3) of *local-EF*.

The rest of the proofs are typical proofs over sets of natural numbers. Since sets and natural numbers are contained in the KIV library, a large number of simplifier rules (about 250) were already available to automate the proofs. About 30 rules had to be added to automate proofs for naturals extended by an $\infty$-element. Proofs then were nearly fully automatic. Full automation could be achieved (saving some hours of work) for these proofs by implementing a suitable decision procedure.

Using the new definition of *local-EF*, it is now much simpler to derive a bitvector based implementation than by using the recursive definition. Also, proving correctness of the implementation is eased significantly by the use of the nonrecursive definition. Before discussing the implementation and its verification in detail, we will first introduce the specification of bitvectors.

### 4.2 Specification of Bitvectors

Since the leaves of the MTBDDs representing sets of configurations are just finite sets of natural numbers, bitvectors can be used to represent these sets. A bitvector has a one at position $i$, iff the set represented contains the clock value $i$. Using a bitvector representation bears the advantage, that most of the set operations can be performed very efficiently using hardware machine instructions.

In order to verify algorithms working on bitvectors, we first have to give formal definitions of bitvectors and their associated operations. Arbitrary length bitvectors are defined in KIV using a data type:

$$\text{bitvector} = 0 \mid 1 \mid\ .\ .0\ (\text{pop} : \text{bitvector}) \mid\ .\ .1\ (\text{pop} : \text{bitvector})$$

$$\text{size function}\ \# : \text{bitvector} \to \text{nat}$$

The formula states that a bitvector can be constructed starting with a single 0 or 1 by appending further zeros and ones at the end (operations .0 and .1, written postfix). KIV automatically generates the proper axioms for this data type definition, the length function # and the selector function *pop* which removes the least significant bit of the bitvector. An alternative to the definition of a new data type would have been to use lists of boolean values. Proofs would not have been very similar, except that an additional case, the empty bitvector would have to be considered. Since the original C++ code uses arrays to implement bitvectors (see Sect. 4.6), and there is no vector of zero length in C++, we preferred the definition of a new datatype over using lists of bits.

Based on this datatype, bit vector operations & (bitwise AND), | (OR), $\oplus$ (XOR), $\ll$ (shift left) and $\gg$ (shift right) can be specified. In addition, a selection function $\cdot[\cdot]$ for the $n$th bit of a bitvector is defined. $x[0]$ denotes the least significant bit. To guarantee a minimal representation of sets and therefore space efficiency of the implementation, a restriction predicate $r$ is defined which expresses that bitvectors contain (with the exception of the bitvector 0) no leading zeros. Later we will verify, that all operations (including *local-EF*) keep up this restriction, thereby minimizing the complexity of bit operations. Exemplarily, Fig. 6 shows the axioms for the restriction predicate $r$, the bitvector operations $\ll$ and & as well as the bit-selection function $\cdot[\cdot]$.

In the case of $\ll$ and &, the definition has to be split into several axioms in order to prevent the operation from introducing leading zeros. The other bitvector operations are defined similarly.

[1] $\alpha\ ?\ \beta : \gamma$ denotes $\beta$ if $\alpha$ is true and $\gamma$ otherwise. This corresponds to the semantics of the conditional operator in C++.

| | |
|---|---|
| $\mathrm{r(x)} \leftrightarrow \mathrm{x} = 0 \lor \mathrm{r_h(x)}$ | $\mathrm{r_h(x)} \leftrightarrow \mathrm{x} = 1 \lor \mathrm{x} \neq 0 \land \mathrm{r_h(pop(x))}$ |
| $\mathrm{x} \ll 0 = \mathrm{x}$ | $\mathrm{x} \neq 0 \rightarrow \mathrm{x} \ll \mathrm{n} + 1 = (\mathrm{x} \ll \mathrm{n})\ .0$ |
| $\mathrm{x} = 0 \rightarrow \mathrm{x} \ll \mathrm{n} + 1 = 0$ | $\mathrm{x[n]} = (\mathrm{x}\ \&\ (1 \ll \mathrm{n}) = 0\ ?\ 0 : 1)^1$ |
| $1\ \&\ \mathrm{y} = \mathrm{lsb(y)}$ | $\mathrm{x}\ .0\ \&\ \mathrm{y}\ .0 = ((\mathrm{x}\ \&\ \mathrm{y}) = 0\ ?\ 0 : (\mathrm{x}\ \&\ \mathrm{y}).0)$ |
| $\mathrm{x}\ \&\ 0 = 0$ | $\mathrm{x}\ .1\ \&\ \mathrm{y}\ .0 = ((\mathrm{x}\ \&\ \mathrm{y}) = 0\ ?\ 0 : (\mathrm{x}\ \&\ \mathrm{y}).0)$ |
| $\mathrm{x}\ \&\ 1 = \mathrm{lsb(x)}$ | $\mathrm{x}\ .0\ \&\ \mathrm{y}\ .1 = ((\mathrm{x}\ \&\ \mathrm{y}) = 0\ ?\ 0 : (\mathrm{x}\ \&\ \mathrm{y}).0)$ |
| $0\ \&\ \mathrm{y} = 0$ | $\mathrm{x}\ .1\ \&\ \mathrm{y}\ .1 = ((\mathrm{x}\ \&\ \mathrm{y}) = 0\ ?\ 1 : (\mathrm{x}\ \&\ \mathrm{y}).1)$ |

**Figure 6:** Axioms for bitvector operations

In order to gain automation in proofs concerned with bitvectors, we also had to add and prove some 120 simplifier rules which are automatically applied by the KIV system to solve goals concerned with bitvectors automatically.
A typical simplifier rule used for conditional rewriting looks like

$$\mathrm{r(x)} \land \mathrm{x} \neq 0 \rightarrow \mathrm{x}[\#(x)] = 1$$

It states that every bitvector (with the exception of the bitvector 0) free of leading zeros (the term $r(x)$) has a one at its top position. Simplifier rules were defined and proved as needed. Almost all simplifier rules had very easy proofs, the total effort to prove them was only a few hours.

### 4.3 Implementation of Time Jumps using Bitvectors

Fig. 7 shows the implementation of both, the time-jump function *local-EF* and the time prediction function *local-predict-EF* which were taken from the RAVEN C++ source-code.

The idea of the implementation of *local-EF* (left part of Fig. 7) is to traverse both input bitvectors starting at the most significant bit. After examining the $i$th bit of the inputs, the $i$th bit of the result is computed: A look at the nonrecursive definition of *local-EF* shows, that if a one in the bitvector $g$ is encountered, the next $n$ bits have to be set. The variable *state* memorizes the number of ones that remain to be set. In each step a one is output, iff $state \geq 1$ holds.

The time-prediction (right part of Fig. 7) algorithm works similarly. In order to compute the term $min(c, g + 1)$ contained in the formal definition of *local-predict-EF*, the inputs $c$ and $g$ are traversed starting at the least significant bit. When the first "one" is encountered, its position (or the position +1 in the case of $g$) is returned. Two special cases have to be handled by extra conditionals in the implementation of Fig. 7: The case of both sets being empty would cause nontermination of the **while**-loop and the case of $c[0] = 1$ where the predicted value is $\infty$.

```
bitvec local-EF(bitvec c, g, nat n)
begin
   nat m := 0, state := 0,
        pos := max(#(c),#(g)) +1;
   bitvec r := 0;
   if n = ∞ then m := pos;
   else m := n;
   while pos ≠ 0 do
   begin
      pos := pos − 1;
      if g[pos] = 1 then state := m;
      if c[pos] = 1 then state := m +1;
      if state ≥ 1 then
      begin
         r := (1 ≪ pos) | r;
         state := state − 1;
      end
   end
   return r;
end
```

```
nat local-predict-EF(bitvec c, g)
begin
   nat n := 0;
   if c = 0 ∧ g = 0 then n := ∞;
   else if c[0] = 1 then n := ∞;
   else
   begin
      nat pos = 0;
      while n = 0 do
      begin
         if c[pos] = 1 then n := pos;
         else if g[pos] = 1 then
                  n := pos + 1;
         pos := pos + 1;
      end
   end
   return n;
end
```

**Figure 7:** Implementation of time-jumps and time-prediction functions

### 4.4 Verification

To prove correctness of both implementations, the KIV concept of program refinements was used. The basic idea of this concept is to prove that an axiomatic definition (like the one of *local-EF*) may be replaced by a program, which has the same functionality. In our case, all set operations as well as the functions *local-EF* and *local-predict-EF* have to be implemented using bitvectors and natural numbers only.

KIV automatically generates the appropriate proof obligations: We have to show termination of all programs, that the restriction is kept invariant and that the implementations satisfy the axiomatic definitions of the functions *local-EF* and *local-predict-EF*. A general theorem [Rei92] assures that correctness of these proof obligations implies:

**Theorem 2.** *For the temporal operator* EF*, the algorithms given in Fig. 7 are a correct bitvector implementation of the operations* local-EF *and* local-predict-EF *as defined in (4) and (2). The implementations for the other temporal operators given in [Vol00] are correct too.*

The two proof obligations generated for the time-jump function *local-EF* are:

$$\mathrm{r(c)} \wedge \mathrm{r(g)} \rightarrow \langle \mathrm{res} := \text{local-EF(c,g,m)} \rangle\ \mathrm{r(res)} \tag{5}$$

$$\mathrm{r(c)} \wedge \mathrm{r(g)} \rightarrow \langle \mathrm{res} := \text{local-EF(c,g,m)} \rangle\ \varphi \tag{6}$$

where

$$\begin{aligned}\varphi \equiv \quad & \mathrm{res[n]} = 1 \\ \leftrightarrow \quad & (\exists\, \mathrm{n_0}.\ \mathrm{c[n_0]} = 1 \wedge \mathrm{n} \leq \mathrm{n_0} \wedge \mathrm{n_0} \leq \mathrm{n} + \mathrm{m}) \\ \vee \quad & \mathrm{m} = 0 \wedge (\exists\, \mathrm{n_0}.\ \mathrm{g[n_0]} = 1 \wedge \mathrm{n} \leq \mathrm{n_0} \wedge \mathrm{n_0} \leq \mathrm{n} + \mathrm{m} - 1)\end{aligned}$$

The formula $\langle res{:=}local\text{-}EF(c,g,m)\rangle r(res)$ is of Dynamic Logic [Har79] and states that the function $local\text{-}EF(c,g,m)$ terminates and the restriction $r$ holds on the results. The second proof obligation corresponds to definition (4) of *local-EF* where all functions working on sets are replaced by bitvector operations ($n \in s$ becomes $res[n] = 1$ for the bitvector *res* implementing set $s$).

### 4.4.1 Proof of Correctness

Both, proof obligations for time jumps and time-prediction are proved by induction over the number of remaining iterations of the **while**-loop contained in the algorithms. We will now concentrate on the basic ideas, since sometimes rather large formulae of over 100 lines occur in the proofs. To keep proofs feasible, efficient handling of several hundred rewriting rules is essential.

A first attempt to prove obligation (6) is to unfold the function *local-EF* and to apply structural induction on the formula ($\varphi$ as above)

$$\begin{aligned}&\mathrm{pos} = \max(\#(\mathrm{c}),\#(\mathrm{g})) \wedge \mathrm{state} = 0 \\ \rightarrow\ &\langle \textbf{while}\ \mathrm{pos} \neq 0\ \textbf{do}\ \ldots;\ \mathrm{state} := \ldots;\ \mathrm{pos} := \mathrm{pos} - 1;\ \textbf{end}\rangle\ \varphi\end{aligned}$$

afterwards. Since the bitvector is traversed downwards, variable *pos* can be used for induction. But obviously, the induction hypotheses is never applicable, since *pos* and *state* change during the computation. Therefore, it is necessary to generalize the preconditions in order to ensure applicability of the induction hypotheses.

For *pos*, it suffices to generalize the formula $pos = max(\#(c), \#(g))$ to $pos \leq max(\#(c), \#(g))$. The precondition $state = 0$ cannot be treated analogously, since it represents the "memory" of the algorithm and captures information about earlier in- and outputs. Therefore, an invariant capturing the current state of computation is required.

For the time-jump algorithm depicted in the left part of Fig. 7, the proper invariant is shown in Fig. 8.

The invariant consists of two parts. The first, $INV_1$, states that the intermediate result after the next iteration satisfies the proof obligation. The second

$$
\begin{aligned}
&\mathrm{INV_{EF}} \equiv \mathrm{INV_1} \wedge \mathrm{INV_2}\\
&\mathrm{INV_1} \equiv \forall\, \mathrm{n_1}.\ \mathrm{pos} \le \mathrm{n_1}\\
&\qquad\quad \to (\quad \mathrm{r[n_1]} = 1\\
&\qquad\qquad \leftrightarrow \quad (\exists\, \mathrm{i.\ c[i]} = 1 \wedge \neg\, \mathrm{i} < \mathrm{n_1} \wedge \neg\, \mathrm{n_1} + \mathrm{n} < \mathrm{i})\\
&\qquad\qquad\quad \vee \quad (\exists\, \mathrm{i.\ g[i]} = 1 \wedge \neg\, \mathrm{i} < \mathrm{n_1} \wedge \neg\, \mathrm{n_1} + \mathrm{n} - 1 < \mathrm{i})\\
&\qquad\qquad\qquad \wedge\ \mathrm{n} \ne 0)\\
&\mathrm{INV_2} \equiv \quad \mathrm{r(r)} \wedge \mathrm{state} \le \mathrm{n}\\
&\qquad \wedge (\quad \mathrm{state} = 0 \vee \mathrm{c[pos + n - state]} = 1\\
&\qquad\quad \vee\ \mathrm{state} < \mathrm{n} \wedge \mathrm{g[pos + n - state + 1]} = 1)\\
&\qquad \wedge (\forall\, \mathrm{i.\ i} < \mathrm{n} - \mathrm{state} \to \mathrm{c[pos + i]} = 0)\\
&\qquad \wedge (\mathrm{state} < \mathrm{n} \to (\forall\, \mathrm{i.\ i} < \mathrm{n} - \mathrm{state} + 1 \to \mathrm{g[pos + i]} = 0))
\end{aligned}
$$

**Figure 8:** Invariant for while-loop of *local-EF*-procedure

part $INV_2$ describes the structure of previous inputs depending on the "memory" variable *state*. Finding this formula was the major creative step in the verification of the bitvector implementation.

In contrast to the time jumps, the correctness proof of time prediction is much simpler, since only the trivial part of the invariant, $pos \le max(\#(c), \#(g))$ is required.

### 4.5 Results and Statistics

Proving the above proof obligations for all operators implemented in RAVEN took about one month of work. Together, all proofs sum up to about 4000 proof steps. Over three quarter of these steps were performed automatically by the KIV system. In addition, a theory of bitvectors containing a large number of simplification rules was constructed.

For each of the operators, four proof obligations (two for *local-EF* and two for *local-predict-EF*) had to be proven. Since no large invariant is needed, the correctness proof for the time-prediction program *local-predict-EF* and proof obligation (5) required only a few hours of work.

The proof of (6) is more difficult due to several reasons. First, formulas are much larger due to the invariant $\mathrm{INV}_{EF}$. Second, the first-order part of the proof which shows that $\mathrm{INV}_{EF}$ is indeed invariant, requires a lot of case distinctions. These have to be done manually to avoid unnecessary duplications of parts of the proof. Third, several proof attempts are necessary to find the correct invariant, and a lot of effort goes into the analysis why a certain invariant is not sufficient for the proof to carry through. Here, the counter examples generated by the algorithm implemented in KIV [RST00] gave valuable hints how to improve the invariant.

Proving correctness of *local-EF* required about 450 proof steps and two days. During the verification of the other four operators we made the following observations:

- Operator EF has one subformula. Some other operators (like until) have two subformulas, which almost doubles both the invariants and the proofs. The largest invariant consists of 25 lines of formula, the associated proof has more than 700 proof steps.
- Operators having several "memory" variables (like *state*) also require larger invariants. The size of $INV_2$ increases proportionally with the number of "memory" variables, the size of the proofs almost scales equally.

Summarizing, the verification of the bitvector implementation showed some inefficiencies in the RAVEN code. Fixing these led to both, shorter and faster code. For example, the RAVEN implementation of the until operator could be shortened from 73 to 18 lines of code.

### 4.6 Comparison with the RAVEN source code

Although the implementations discussed so far were directly taken from the RAVEN C++ source code, some aspects were suppressed during this process. In this section, these omissions are illustrated taking the implementation of the time prediction function *local-predict-EF* depicted in Fig. 9 as example.
The most important difference is due to the fact, that RAVEN uses the following, more detailed data structure to represent bitvectors:

```
class BVEC {
    CONTAINER *bv;
    ULONG length;
}
```

The first component holds a field of machine words which together constitute the bitvector, the second component holds the number of machine words stored in *bv*. Due to this partitioned representation of bitvectors some administrative effort has to be taken in the implementations. The corresponding code section is marked by a vertical bar in Fig. 9. Therein, the next input data words are retrieved each time all bits of the previous word of both inputs have been inspected. If one of the bitvectors is shorter than the other, it is padded with zero words. This code section is uniformly contained in all implementations of time-jump and time-prediction functions.

The KIV implementation is somewhat simpler, since it used unpartitioned bitvectors. To verify an implementation with partitioning, the restriction would have to require that the bitvector has no leading zero words. This would add some technical complexity, but the proofs would remain essentially the same. The verified code in KIV shown in Fig. 7 can be viewed as the special case, where all machine words just contain one bit.

Also, the KIV implementation performs padding of bitvectors and memory allocation implicitly by the selection function and variable declarations respectively.

```
ULONG BVEC::local-predict-EF(BVEC &c_bv, BVEC &g_bv) {
  ULONG res,i,j,len;
  CONTAINER *c, *g, mask, c_wrd, g_wrd;
  if((c_bv.length == 0) && (g_bv.length == 0)) return infinity;
  c = ptr(c_bv→bv);
  g = ptr(g_bv→bv);
  len = i = mask = 0;
  if(c[0] & 1) return infinity;
  else { // find first one-bit in c or g
    while(1) {
      if(! mask) {
        mask = 1;
        if(i < c_bv.length) c_wrd = c[i];
        else c_wrd = 0;
        if(i < g_bv.length) g_wrd = g[i];
        else g_wrd = 0;
        i++; }
      if(c_wrd & mask) return len;
      else if(g_wrd & mask) return len+1;
      len++;
      mask = (mask ≪ 1); }}}
```

**Figure 9:** C++ implementation of time-prediction function *local-predict-EF*

A technical difference is that our Pascal-like programs use a result variable $n$ in the **while** loop (and an exit condition $n = 0$), while the C++ code directly returns the result from within the loop.

## 5 Conclusion

In this paper we investigated the correctness of bitvector based efficient real-time model checking algorithms used in RAVEN.

The implementation consists of 5 pairs of programs on bitvectors for the different temporal operators (similar to *local-EF* and *local-predict-EF* for the EF operator). The algorithms were taken directly from the actual C++ implementation in RAVEN, omitting only the technical issues of bitvector partitioning and memory allocation. All programs could be successfully verified with the interactive theorem prover KIV within one month.

Verification revealed several possibilities to improve the code, both in size and in efficiency. We found that the additional verification effort was low compared to the development time of the full C++ code, so we propose to develop new model checking algorithms hand in hand with verification. We hope that our results encourage further research in the correctness of model checkers.

**Acknowledgements**

This work was partly sponsored by the German Research Foundation (DFG).

## References

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1983.

[Har79] D. Harel. *First Order Dynamic Logic.* LNCS 68. Springer, Berlin, 1979.

[Rei92] W. Reif. Correctness of Generic Modules. In Nerode and Taitslin, editors, *Symposium on Logical Foundations of Computer Science*, LNCS 620, Berlin, 1992. Logic at Tver, Tver, Russia, Springer.

[RK97] Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D.K. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 146–166, Montreal, Canada, October 1997. IFIP WG 10.5, Chapman and Hall.

[RK98] Jürgen Ruf and Thomas Kropf. Using MTBDDs for Composition and Model Checking of Real-Time Systems. In *FMCAD 1998.* Springer, November 1998.

[RK99] Jürgen Ruf and Thomas Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *Correct Hardware Design and Verification Methods (CHARME 99)*, pages 265–279. IFIP WG 10.5, Springer, September 1999.

[RRSV00] W. Reif, J. Ruf, G. Schellhorn, and T. Vollmer. Do you trust your model checker? In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design (FMCAD).* Springer LNCS 1954, November 2000.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications.* Kluwer Academic Publishers, Dordrecht, 1998.

[RST00] W. Reif, G. Schellhorn, and A. Thums. Fehlersuche in formalen Spezifikationen. Technischer Bericht 2000-06, Fakultät für Informatik, Universität Ulm, Germany, 2000. (in German).

[Ruf00a] J. Ruf. RAVEN: Real-time analyzing and verification environment. Technical Report WSI 2000-3, University of Tübingen, Wilhelm-Schickard-Institute, January 2000.

[Ruf00b] J. Ruf. Raven: Real-time analyzing and verification environment. *Journal of Universal Computer Science (J.UCS)*, 2000. (to appear in this volume).

[Vol00] T. Vollmer. Korrekte Modellprüfung von Realzeitsystemen. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 2000. (in German).