# Do you trust your model checker?*

Wolfgang Reif[1] and Jürgen Ruf[2] and Gerhard Schellhorn[1] and Tobias Vollmer[1]

[1] Universität Augsburg, Lehrstuhl für Softwaretechnik und Programmiersprachen, D-86135 Augsburg, Germany
email:{reif,schellhorn,vollmer}@informatik.uni-augsburg.de
[2] Wilhelm-Schickard-Institute, University of Tübingen, D-72076 Tübingen, Germany
email: ruf@informatik.uni-tuebingen.de

**Abstract.** In this paper we describe the formal specification and verification of the efficient algorithm for real-time model checking implemented in the model checker RAVEN. It was specified and proved using the KIV system. We demonstrate how to decompose the correctness proof into several independent subtasks and indicate the corresponding verification efforts. The formal verification revealed some errors, reduced the code size, and improved the efficiency of the implementation.

## 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 [9]. 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 specification and verification system KIV to formalize and prove the algorithms of RAVEN. To our knowledge, this is the first case study tackling 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 [10, 12] (second author), and the development group of KIV [8] (remaining authors).

* This work is supported by the DFG (Deutsche Forschungsgemeinschaft) under the priority programme "Integrating Software Specification Techniques for Engineering Applications"

The KIV case study described in this paper consists of four steps: first, we define a formal specification of the semantics of CCTL, the basic model checking algorithm and the optimizations. Second, we verify the correctness of the simple and the optimized algorithm. Third, we give an efficient implementation of the abstract algorithms based on bitvectors, and finally we prove the implementation correct. This implementation relies on a standard software package for extended characteristic functions [5] (based on multiterminal BDDs, MTBDDs [1, 3]). We formalized the interface to this package in KIV. The verification relies on the specification of this interface. Verifying [5] against the interface is an independent subtask which was not part of the case study.

Our case study shows that it is possible to give a modular specification, such that the correctness of the model checker can be split into several independent verification tasks. With the help of the correctness proofs we found some critical definition errors in the formal specification of the optimizations. After correcting them we proved that the basic algorithms and the optimizations using bitvectors are correctly implemented. Parts of the code were shrinked. One prediction function (see Sect. 5.3) worked too pessimistic and was optimized.

In Section 2, we will describe interval structures and the logic CCTL, which constitutes the basis of RAVEN. Section 3 discusses the used optimizations and the efficient implementation. Section 4 gives our approach to formalization and verification. An overview over the specifications and correctness proofs is presented in Section 5. Section 6 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 if a given state transition system satisfies a given property specified as a propositional temporal logic formula.

The approach we will examine is developed for timed systems and timed specifications. It is presented in [10, 12]. In this section, we will explain the main ideas behind the model checking verification technique which are necessary for the remaining part of the paper. First we will present the formal model and the temporal logic. Afterwards we will introduce the representation with extended characteristic functions and the main model checking procedure.

### 2.1 Interval Structures

Interval structures are finite state transition systems. The transitions are labeled with intervals of natural numbers to represent delay times. The structures use the notion of clocks to represent time: every structure contains exactly one clock working in a discrete time domain. A transition is enabled if the actual clock value is within the interval of an outgoing transition. The successor state as well as the delay time is chosen indeterministically w.r.t. the transition relation and the labeled delay intervals. The clock is reset if a transition fires.

**Definition 1.** *An interval structure $\mathcal{J} = (P, S, S_0, T, L)$ is a tuple with a set of atomic propositions $P$, a set of states $S$, a set of initial states $S_0$, a function $T : S \times S \to \wp^{\omega}(\mathbb{N}_0)$ that connects states with labeled transitions and a state labeling function $L : S \to \wp(P)$.*

Every state of an interval structure must be left after the maximal state time: $MaxTime(s) := \max\{\, v \mid \exists s'.v \in T(s, s')\}$

Besides the states, we now also have to consider the currently elapsed time to determine the transition behavior of the system. Hence, the actual configuration of a system is given by an interval structure state $s \in S$ and the actual clock value $v \in \mathbb{N}_0$. The set of all configurations of an interval structure is given by: $G_{\mathcal{J}} = \{\, (s, v) \mid s \in S \wedge v \leq MaxTime(s)\}$

The semantics of interval structures is defined over runs. A run is a sequence of configurations $r = (r_0, r_1, \ldots)$ with $r_i = (s_i, v_i) \in G_{\mathcal{J}}$ and for all $i \geq 0$ holds either

- $r_{i+1} = (s_i, v_i + 1)$ and $v_i < MaxTime(s_i)$ or
- $r_{i+1} = (s_{i+1}, 0)$ and $v_i \in T(s_i, s_{i+1})$

In the following, we call $(s_i, v_i)$ the local predecessor of $(s_i, v_i + 1)$, which corresponds to the first case of the definition. Similar, we call $(s_i, v_i)$ a global predecessor of $(s_{i+1}, 0)$ if $v_i \in T(s_i, s_{i+1})$. Note that for a set of configurations, only the computation of global predecessors depends on the transition relation T.

### 2.2 CCTL

CCTL (Clocked Computation Tree Logic) is a propositional temporal logic using quantitative time bounds for expressing real time properties (e.g. bounded liveness). The following definition describes the syntax of CCTL formulas.

**Definition 2.** *Given a set of atomic propositions $P$. The set of CCTL formulas $F_{CCTL}$ is defined to be the smallest set with*

- $P \subseteq F_{CCTL}$
- *if $m \in \mathbb{N}_0, n \in \mathbb{N}_0 \cup \{\infty\}$, $m \leq n$ and $\varphi, \psi \in F_{CCTL}$ then*
  $\neg\varphi$, $\varphi \wedge \psi$,
  $\mathsf{EX}_{[m]}\varphi$ (*Next*), $\mathsf{EF}_{[m,n]}\varphi$ (*Eventually*), $\mathsf{EG}_{[m,n]}\varphi$ (*Globally*),
  $\mathsf{E}(\varphi\mathsf{U}_{[m,n]}\varphi)$ (*Until*), $\mathsf{E}(\varphi\mathsf{S}_{[m]}\varphi)$ (*Successor*), $\mathsf{E}(\varphi\mathsf{C}_{[m]}\varphi)$ (*Conditional*)
  $\mathsf{AX}_{[m]}\varphi$, $\mathsf{AF}_{[m,n]}\varphi$, $\mathsf{AG}_{[m,n]}\varphi$, $\mathsf{A}(\varphi\mathsf{U}_{[m,n]}\varphi)$, $\mathsf{A}(\varphi\mathsf{S}_{[m]}\varphi)$, $\mathsf{A}(\varphi\mathsf{C}_{[m]}\varphi) \in F_{CCTL}$
  *The symbol $\infty$ is defined through: $\forall\, i \in \mathbb{N}_0 : i < \infty$.*

All interval operators can also be accompanied by a single time-bound only. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the $\mathsf{EX}$-operator has no time bound, it is implicitly set to one.

For this paper we will only define the semantics for the $\mathsf{EF}$-operator ("Eventually"), the semantics for the other operators may be found in [12]. The semantics of CCTL is given by a model relation ($\models$):

**Definition 3.** *Given the interval structure* $\mathcal{J} = (P, S, S_0, T, L)$*, a starting configuration* $r_0 \in G_{\mathcal{J}}$ *and the CCTL formula* $\varphi \in F_{CCTL}$*.*

$$\mathcal{J}, r_0 \models \mathsf{EF}_{[m,n]}\varphi :\Leftrightarrow \textit{there exists a run } r = (r_0, \ldots) \textit{ and an } i \in [m,n] \textit{ such that } \mathcal{J}, r_i \models \varphi$$

A formula $\varphi$ is valid in a model $\mathcal{J}$, iff $\mathcal{J}, (s, 0) \models \varphi$ for all initial states $s \in S_0$. The defined interval operator may be expressed by operators only carrying an upper time bound, e.g. $\mathsf{EF}_{[m,n]}\varphi := \mathsf{EX}_{[m]}\mathsf{EF}_{[n-m]}\varphi$.

### 2.3 The basic model checking algorithm

The main idea of model checking algorithms is the following: building the syntax graph of the formula to check, computing bottom-up sets of configurations representing sub formulas (called extension sets), checking if the set of initial states is a subset of the extension set of the complete formula.

The computation of the extension sets is done by a function *ext*. The extension sets of atomic formulas (i.e. the leaves of the syntax tree) are given by the labeling function and the possible clock values in the appropriate states of the interval structure $\mathcal{J}$. Boolean connections can be computed by applying the corresponding set operations on the extension sets. Finally, the computations of the extension sets of temporal logic operators are defined recursively. We will present them using the EF-operator as an example:

$$\text{ext}(\mathsf{EF}_{[0]}\varphi) := \text{ext}(\varphi)$$
$$\text{ext}(\mathsf{EF}_{[n+1]}\varphi) := \text{ext}(\varphi) \cup \text{ext}(\mathsf{EX}(\mathsf{EF}_{[n]}\varphi))$$

The operator EX ("Next") states, that a certain formula is fulfilled after the next step. The time bound $n$ for $\mathsf{EF}_{[n]}$ has to be finite. For $n = \infty$ we can show, that the extension sets reach a fixpoint, i.e. $\exists\ i.\ ext(\mathsf{EF}_{[i]}) = ext(\mathsf{EF}_{[i+1]})$. This means, that the recursion can be terminated if the $ext(\mathsf{EF}_{[i]})$ will not change anymore. The realization of the recursive definition of the EF-operator leads to the algorithm shown in Fig. 1.

The other CCTL operators can be implemented with similar algorithms. The operators EX, ES, EC, which do not reach a fixpoint during computation can not be computed for $n = \infty$.

Two questions remain: how are the sets of configurations represented in order to achieve efficient computations and what how is the EX-operator computed?

A main advance in the field of model checking was made with the introduction of symbolic representations of state spaces [2]. Instead of an explicit enumeration of sets of states, they are represented by characteristic functions. For our time extended model checking algorithm, we use an extension of characteristic functions (ECF) which maps interval structure states to sets of associated clock values [10]. The function $\Lambda_A : S \to \wp(\mathbb{N})$ represents a set $A \subseteq G_{\mathcal{J}}$ of configurations:

$$\Lambda_{\mathrm{A}}(\mathrm{s}) := \{\ \mathrm{v} \mid (\mathrm{s},\mathrm{v}) \in \mathrm{A}\ \}$$

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

**Fig. 1.** The basic EF-algorithm

In the following, we will use sets of configurations and the ECFs representing these sets synonymously, i.e. we will write $A$ instead of $\Lambda_A$. Since we aim at verifying an implementation of a model checker we will furthermore use the notation and intuition of MTBDDs (multi-terminal BDDs [1, 3]), which are used to implement extended characteristic functions.

MTBDDs representing sets of configurations code the state space in the decision diagram and associate a set of natural numbers representing the clock values with each state (i.e. each leaf of the decision diagram). An example MTBDD representing the configuration set $\{(a, 3), (a, 4), (b, 5), (ab, 2)\}$ is depicted in the dashed box in Fig. 2. Set operations can be implemented by performing the appropriate operations on the leaves of the MTBDD. The transition relation is represented in a similar way. Since we only need to compute predecessor configurations during model checking, it eases the computation if we store the set of predecessors for each state. This is done with two cascaded MTBDDs as shown in Fig. 2. The first MTBDD represents the state space of the model $\mathcal{J}$, the second represents the sets of predecessor configurations for the individual states.

The second open question is, how can the extension set of an EX-operator be computed? If the extension set of its argument formula is known, the extension set of the EX-operator is given by all predecessor configurations.

The local predecessors of a set of configurations may be computed by a function *local-pre* as follows. Using MTBDDs, the computation may be reduced to the leaves of the MTBDD where each clock value contained is decremented.

Global predecessors only exist for configurations with a zero clock value. The computation is done by a function *global-pre*$(C, T)$. For every state $s'$ with $(s', 0) \in C$, the set of predecessor configurations is looked up in the transition relation and the resulting configuration sets are combined to a new configuration set.

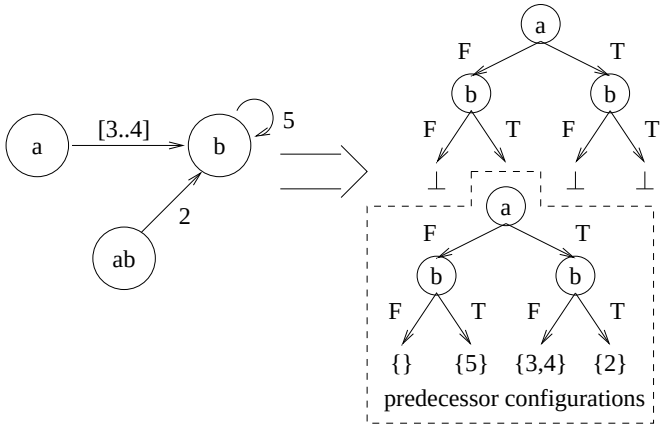


**Fig. 2.** Representation of the transition relation using two cascaded MTBDDs

## 3 The Central Idea of the RAVEN Model Checker

### 3.1 Time Prediction

If we analyze the predecessor computation we observe the following:

- The global predecessor computation is more expensive than the local predecessor computation since it takes the complete transition relation into account.
- The global predecessors often do not change between two predecessor computations.

We use a technique called time prediction to overcome the single step traversal and to avoid unnecessary global predecessor computations [10, 11]. The idea is to define a time prediction function that computes how many steps the global predecessors stay constant.

Again, we exemplarily discuss the EF-operator. Although the basic idea of time prediction can be applied to all operators, every temporal operator needs a separate prediction function.

The time prediction function *predict-EF* is computed locally, i.e. for each state separately by a function *local-pr-EF*. The minimum of the prediction times *mp* is the time span which can elapse without any change in the set of the global predecessors. Arguments of *local-pr-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:

$$\text{predict-EF(C,G)} = \min_{s \in S}\text{local-pr-EF(C(s),G(s))}$$

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

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 *mp* times. Analogous to the above, a function *apply-EF* is defined which performs the fixpoint iteration locally for every state by using the recursively defined function *local-EF*.

$$\text{apply-EF(C,G,mp)(s)} = \text{local-EF(C(s),G(s),mp)}$$

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

Putting together the above definitions and considerations, we obtain the optimized algorithm shown in Fig. 3.

```
confset EF'(confset C, natinfty n, transrel T)
begin
   confset old := ∅, R := C, G := ∅;
   natinfty p;
   while n > 0 ∧ old ≠ R do
      old := R;
      G := global-pre(R,T);
      p := predict-EF(R,G);
      if p > n then p := n;
      R := apply-EF(R,G,p);
      n := n − p;
   end
   return R;
end
```

**Fig. 3.** The optimized EF-algorithm

### 3.2 Time Jumps using Bitvectors

The local fixpoint iteration needs $O(mp)$ set operations for execution. We define a technique called time jumping which replaces this iterative execution by an efficient implementation using either bitvectors or interval lists.

Using interval lists has the advantage of little memory consumption but lacks the efficiency of the bitvector based algorithms. Hence, we now will concentrate on the bitvector based implementation.

The implementation *local-EF#* for the EF-operator using bitvectors is shown in Fig. 7. Bitvectors are used to represent sets of natural numbers. A number $n$ is contained in a set, iff the $n$th bit of the bitvector representation is 1. The basic idea of the implementation of *local-EF#* is to traverse bitvectors: in the $n$th step of the algorithm the $n$th bit of the input and an internal state of the

algorithm are used to compute the $n$th bit of the result. Hence, we only need a single specialized operation to compute *local-EF* instead of $O(mp)$ set operations when using the recursive definition.

## 4 Formal Specification and Verification Concept

Our case study consists of five parts:

1. specification of CCTL and its semantics. The left half of Fig. 4 illustrates this step.
2. specification of the basic model checking algorithm and verification, that the algorithm implements the semantics (cf. right half of Fig. 4).
3. specification of the optimized algorithms with time prediction for the temporal operators (e.g. $EF'$ as defined in Fig. 3) and verification that they yield the same results as the simple recursive version (e.g. $EF$ as given in Fig. 1). Figure 5 visualizes this step.
4. nonrecursive definition *local-EF'* of the local computations *local-EF* used in $EF'$ and verification, that both definitions are equivalent (cf. upper half of Fig. 6).
5. implementation of *local-EF'* based on bitvectors and correctness proof for the implementation. The lower half of Fig. 6 shows this part.

These five parts will be discussed in the five subsections of the next section. All five parts were specified using the structured, algebraic specifications of KIV, which are similar to the standard algebraic specification language CASL [4]. To do the correctness proofs, KIV offers a concept of modules, which describe a refinement relation between specifications. KIV automatically generates proof obligations that assure the correctness of modules.

An important goal in the design of the case study was to structure specifications, such that each of the four verification steps could be expressed as the correctness of one module. This has two advantages: First, each module can be verified independently of all others, which means that the correctness problem is decomposed into several orthogonal subproblems. Second, a general theory (see [6]) of modular systems (consisting of specifications and modules) assures, that a correct model checking algorithm (that implements the $\models$ predicate) can be "plugged" together by repeatedly replacing abstract definitions with more concrete versions: Starting with the unoptimized algorithm *modelcheck*, first $EF$ is replaced with $EF'$ (and similar for the other temporal operators), then the call to *local-EF* in $EF'$ is replaced with a call to *local-EF'*, and finally this call is replaced again with the bitvector implementation. The final algorithm is identical to the one used in RAVEN, except that it has an abstract interface of extended characteristic functions. Their implementation using actual MTBDD operations could be plugged in using another module (which could be separately verified).

Developing a modular system of specifications and modules, such that "plugging the algorithm together" and separate verification of the steps described

above became possible, was a major creative step in this case study. Two important design decisions were to use higher-order operations on ECFs (*apply* and *reduce*, see Sect. 5.3) to have an abstract interface to BDDs, and to use the intermediate nonrecursive definition *local-EF'* (see Sect. 5.4).

Technically, modular systems are developed in KIV using a graphical representation, called development graphs. Such a graph contains rectangular boxes for specifications and rhombic boxes for modules. Arrows represent the structure of specifications and implementation relations. The following section will show for each of the five steps some relevant part of the development graph and sketch the contents of the specifications. Putting all parts together, i.e. merging the development graphs of figures 4, 5 and 6 gives the full modular specification and implementation of the model checker. Full details on all specifications, modules and proofs can be found in [13].

## 5 Verification of Correctness

### 5.1 Specification of CCTL Semantics

The structure of the algebraic specification for CCTL and its semantics is shown in the left half of Fig. 4. The main predicate specified in the top-level specification is $\mathcal{J}$, $(s,v) \models \varphi$ ($\varphi$ holds in configuration $(s,v)$ over model $\mathcal{J} = (T,L)$[1]). Two typical axioms of this specification are

$$\begin{aligned} &\mathcal{J}, (\mathrm{s,v}) \models \mathsf{EF}_{[\mathrm{n}]}\ \varphi \\ \leftrightarrow\ &\exists\ \mathrm{r.\ run(r,T)} \wedge \mathrm{first(r)} = (\mathrm{s,v}) \wedge \exists\ \mathrm{i.\ i} \leq \mathrm{n} \wedge \mathcal{J}, \mathrm{r}_i \models \varphi \\ &(\mathrm{T,L}), (\mathrm{s,v}) \models \mathrm{p} \leftrightarrow \mathrm{p} \in \mathrm{L(s)} \end{aligned}$$

The definition is based on a specification of the data type of CCTL formulas, the specification *transrel* of the transition relation of an interval structure and (indirectly) on the specification *confsets* of configuration sets (in algebraic terms, the top-level specification is an enrichment of *transrel* and *confsets*). The transition relation is defined as a function $T : \mathit{state} \to (\mathit{state} \to \mathit{set}(\mathit{nat}))$. $T(s')(s)$ gives the possible delay times for a transition from state $s$ to state $s'$. Configuration sets are specified as functions $C : \mathit{state} \to \mathit{set}(\mathit{nat})$. A configuration set $C$ contains a configuration $(s,v)$, iff $v \in C(s)$. This representation corresponds to extended characteristic functions.

Both configuration sets and the transition relation are specified as actualizations of a generic datatype of extended characteristic functions $\mathit{ecf} : \mathit{state} \to \mathit{elem}$: the parameter type *elem* is instantiated by $\mathit{set}(\mathit{nat})$ and *confset* respectively. The use of generic ECFs allows us to be fully abstract in our correctness analysis of the model checking algorithms, while it is still possible to implement ECFs with MTBDDs (with *elem* being the type of the BDD leaves) and to verify this implementation separately.

[1] The set $P$ of atomic propositions, and the set $S$ of states are carrier sets in our algebraic specification. Therefore they need not be explicitly mentioned in the definition of a model

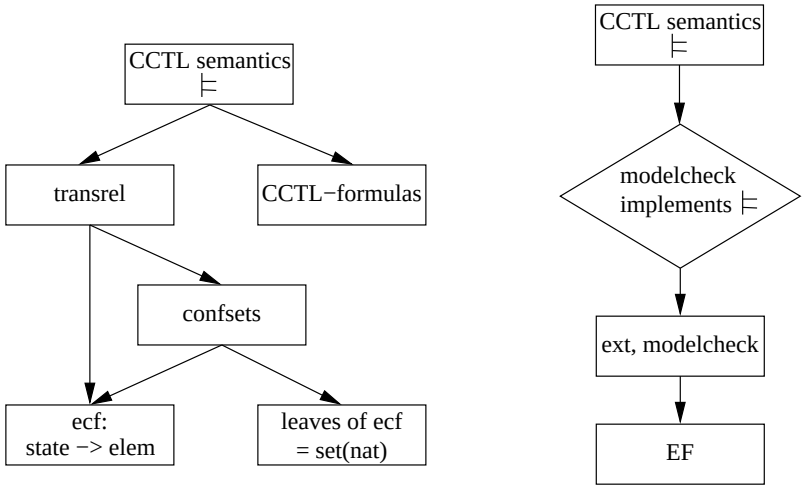


**Fig. 4.** Specification of CCTL Semantics

### 5.2 Correctness of Simple Model Checking

The right half of Fig. 4 shows the part of the KIV development graph that deals with the correctness proof of the simple model checking algorithm with respect to the semantics of CCTL (the structure of the specification of the CCTL semantics above the module has now been omitted).

*Specification of Simple Model Checking* The specification shown below the module in Fig. 4 contains the simple model check algorithm *modelcheck*. It is specified by the following axioms:

$$\text{modelcheck}(\mathcal{J}, \text{c}, \varphi) \leftrightarrow \text{c} \in \text{ext}(\varphi, \mathcal{J}),$$

$$\text{ext}(\mathsf{EF}_{[\text{n}]}\ \varphi,(\text{T},\text{L})) = \text{EF}(\text{ext}(\varphi),\text{n}, \text{T}),$$

$$\text{ext}(\neg\ \varphi,\mathcal{J}) = \text{G}_{\mathcal{J}} \setminus \text{ext}(\varphi,\mathcal{J}),$$

$$(\text{s},\text{v}) \in \text{ext}(\text{p},(\text{T},\text{L})) \leftrightarrow \text{p} \in \text{L}(\text{s}) \wedge (\text{s},\text{v}) \in \text{G}_{(\text{T},\text{L})},$$

$$\ldots$$

Again, the specification is based on configuration sets and the transition relation (not shown in the figure). It is also based on a subspecification which defines a tail-recursive function *EF*, which computes the extension set of the temporal operator $\mathsf{EF}_{[n]}$. The specification also contains similar functions for the other temporal operators, but like in the previous sections, we will now concentrate on the implementation of *EF*. We have preferred the tail-recursive version over the program given in Fig. 1 for two reasons: First, the specification remains independent of an implementation language. Second, proofs using the tail-recursive function are smaller compared to proofs using while-loops and invariants.

To define the computation of local and global predecessors, two generic higher-order operations *apply* and *reduce* on ECFs are used:

$$\text{local-pre(C)} = \text{apply(C, } -1\text{)}$$

$$\begin{aligned}&\text{global-pre(C,T)}\\ &= \text{reduce(T, } (\lambda\ \text{C}_0\text{,s}'.\ \text{if } 0 \in \text{C(s}') \text{ then } \text{C}_0 \text{ else } \emptyset\text{), } \cup\text{, } \emptyset\text{)}\end{aligned}$$

*apply*$(ecf, f)$ applies function $f$ on each leaf of *ecf*. *reduce*$(ecf, f, f', a)$ applies the function $f$ on each leaf of *ecf* and combines the results using function $f'$, starting with the value $a$. The function $-1$ used in the definition of *local-pre* decrements each number contained in a leaf of an ECF and drops zeros. The $\lambda$ expression contained in the definition of *global-pre* looks up the predecessors of all states that contain a zero clock value.

*Proof of Correctness* The KIV module shown in the development graph of Fig. 4 automatically generates proof obligations. We must show, that the simple model checking algorithm *modelcheck* satisfies all axioms of the predicate $\models$, i.e. that *modelcheck* implements the predicate $\models$.

$$\mathcal{J}\text{,c} \models \varphi \leftrightarrow \text{modelcheck}(\mathcal{J}, \text{ c}, \varphi)$$

Proving these proof obligations is straightforward using theorems that assure the existence of fixpoints for the operators EF, EG, etc. and takes only a few hours of verification time.

### 5.3 Time Jumps and Time Prediction

Figure 5 shows the part of the development graph that is relevant for the verification of the optimization step that introduces time prediction and time jumps.

*Specification of Time Prediction* Most parts of the specification of the optimized version *EF'* of the computation (cf. Fig. 3) can be adopted from the unoptimized algorithm. The functions required to specify time prediction and time jumps, *predict-EF* and *apply-EF* are defined using the functions *apply* and *reduce*:

$$\text{apply-EF(ecf)} = \text{apply(ecf,local-EF)}$$

$$\text{predict-EF(ecf)} = \text{reduce(ecf,local-pr-EF,min,}\infty\text{)}$$

Thus, the functions are reduced to functions *local-EF* and *local-pr-EF* which work on the leaves of the ECFs. The specification of the latter functions follows directly the definition in Sect. 3.

*Proof of Correctness* To prove correctness of the module, it must be shown, that all axioms of the simple algorithm are also satisfied by the optimized algorithm.

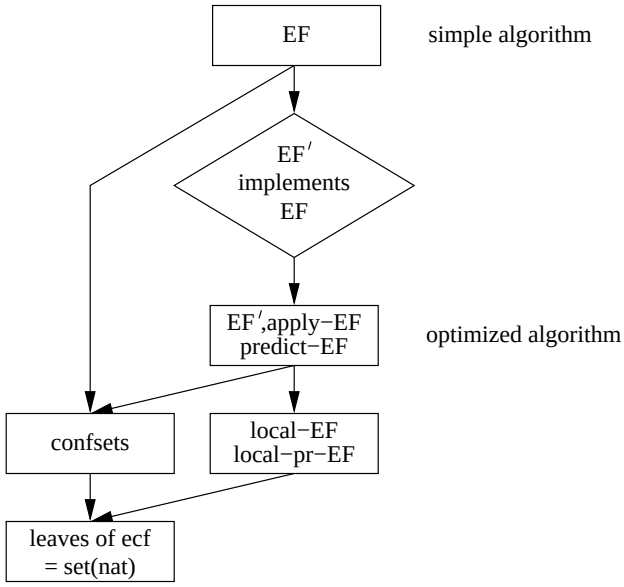


**Fig. 5.** Specification of time-prediction

The main theorem needed in the proof is:

$$\begin{aligned} & \text{n} < \text{predict-EF(C,global-pre(C,T))} \\ \rightarrow \quad & \text{global-pre(C,T)} \\ = \ & \text{global-pre(apply-EF(C,global-pre(C,T),n),T)} \end{aligned} \tag{1}$$

It expresses the central idea of the optimization step, that the global predecessors do not change during the computation of as many steps as the time prediction permits. Since *predict-EF* yields the minimum of the results of the predicted values for all leaves, the proof can be done by reducing the theorem to the leaves of the ECF considered and proving the analogous theorem for each single leaf:

$$\begin{aligned} & \text{p} = \text{local-pr-EF(C(s),G(s))} \wedge (\text{n} < \text{p} \vee \text{p} = \infty) \\ \rightarrow \ & (0 \in \text{local-EF(C(s),G(s),n)} \leftrightarrow 0 \in \text{C(s)}) \end{aligned} \tag{2}$$

Here, the term $0 \in$ *local-EF*() $\leftrightarrow \ldots$ states that the global predecessors of the leaf considered remain unchanged. For reasons explained in the next section, we assume property (2) as an axiom here and postpone its proof until then.

The proof obligation for the operator EF is proved by induction over the number of steps the operator computes. Two important theorems are needed in the induction step of the proof.
The first,

$$\text{EF}'(\text{C,n} + 1,\text{T}) = \text{EF}'(\text{EF}'(\text{C,n,T}),1,\text{T}) \tag{3}$$

ensures, that a single step can be split off from the computation of *EF'*. The proof is conducted by induction over $n$. Expanding the definition of a single recursion "computes" $p$ steps of the operator using function *apply-EF*. Since only one step has to be split off, a similar decomposition lemma is also needed for *apply-EF*. This lemma can be shown by proving the following, analogous lemma for the leaves of the ECF:

$$\text{local-EF(c,g,n + 1)} = \text{local-EF(local-EF(c,g,n),g,1)} \tag{4}$$

The second important theorem is required, because the recursion schemes of the simple and of the optimized version are slightly different. While the simple recursion $EF(C, n+1, T) = C \cup EX(EF(C, n, T), 1, T)$ always adds its argument $C$ to the interim result, the optimized version calls *apply-EF*$(R, p)$ with the interim result $R$ to compute $p$ steps. Hence, in any step of the algorithm, the result of the last major step is added to the configuration set. Since the operator considered increases monotonically, both recursion schemes produce the same results:

$$\begin{array}{l} \text{R} = \text{EF}'\text{(C,n,T)} \\ \rightarrow \text{R} \cup \text{EX(R,1,T)} = \text{C} \cup \text{EX(R,1,T)} \end{array} \tag{5}$$

*Results of Verification.* During verification, we found erroneous time prediction functions for two operators, *local-pr-EU* (for strong-until) and *local-pr-ES* (for the successor operator). In some cases, the original definition of *local-pr-EU*,

$$\text{local-pr-EU}(\text{c}_1,\text{c}_2,\text{g}) = \begin{cases} \text{n} & \text{if n} = \min(\text{c}_1,\text{g} + 1) \wedge [0, \text{n}] \subseteq \text{c}_2 \\ \infty & \text{otherwise} \end{cases}$$

yields too high values. Therefore the algorithm sometimes forgets to recompute the global predecessors, which leads to incorrect results. The corrected version of *local-pr-EU* (corrections **bold**) is

$$\text{local-pr-EU}(\text{c}_1,\text{c}_2,\text{g}) = \begin{cases} \text{n} & \text{if} \quad \neg\ \mathbf{0} \in \mathbf{c}_1 \wedge \text{n} = \min(\text{c}_1,\text{g} + 1) \\ & \qquad \wedge\ [0, \text{n} - \mathbf{1}] \subseteq \text{c}_2 \\ \infty & \text{otherwise} \end{cases}$$

Inspection of both implementations of RAVEN showed, that the bitvector based version behaves correctly. The implementation using interval lists contained the error and was subsequently corrected.

The time prediction *local-pr-ES* not only contained a case where too high results were computed, but also a too pessimistic case. Since the definition of *local-pr-ES* was rather complex (8 lines), we corrected it by reduction to the $\mathsf{EX}$ operator. This led to a much more compact definition using only 3 lines.

The verification also showed a critical point in the computation of $\mathsf{EX}_{[n]}\ \varphi$ (next operator), that was not detected during design and informal analysis of the algorithm. Normally, the $\mathsf{EX}$ operator, which does just computations of predecessors, never reaches a fixpoint. Nevertheless, cycles (i.e. $\mathsf{EX}_{[m]}\ \varphi = \mathsf{EX}_{[m+p]}\ \varphi$)

may occur in the computation. To stop the computation as early as possible, the simple version of $\mathsf{EX}$ (which is similar to the algorithm in Fig. 1) performs a fixpoint test after every step of the computation by comparing $\mathsf{EX}_{[m]}\ \varphi$ to $\mathsf{EX}_{[m+1]}\ \varphi$. The optimized version contains this fixpoint test, too. But since the optimized version computes $p$ steps at a time, $\mathsf{EX}_{[m]}\ \varphi$ is compared to $\mathsf{EX}_{[m+p]}\ \varphi$. Therefore, if the computation of $\mathsf{EX}_{[n]}\ \varphi$ with $n > m + p$ contained such a cycle of length $p$, the computation would stop too early.

However, we could show, that this situation can never occur, because the time-prediction function permits either an infinite number of steps or only one step at a time if a cyclic computation takes place:

$$\begin{aligned} &\text{apply-EX(C,G,n)} = \text{C} \wedge \text{p} = \text{predict-EX(C,G)} \wedge \text{n} \leq \text{p} \\ &\rightarrow \text{p} = 1 \vee \text{p} = \infty \end{aligned} \tag{6}$$

*Verification Effort.* All seven temporal operators implemented in RAVEN were proven correct. We found, that the proofs of all operators share a common pattern. This pattern consists of theorems (1), (2), (3), (4), (5) and some auxiliary theorems. Additionally, for operators, that do not reach a fixpoint during computation, a theorem like (6) was needed. Due to this pattern, the verification time required decreased from one week for the first operator to about 2 days. Although the complexity of the operators increased, the correctness proofs all have about the same length, because the growing experience helped to compensate the extra complexity with higher automation.

### 5.4 Nonrecursive Representation of Time Jumps

A look at the implementation of time jumps (cf. Fig. 7) and time prediction shows, that the computations of these programs do not fit to the recursive definition of *local-EF* very well. Therefore, even simple proofs using the recursive definition are technically very complex. Therefore we decided to introduce a nonrecursive function *local-EF′*, which describes the results of the operator $\mathsf{EF}$:

$$\begin{aligned} &\text{local-EF}'\text{(c,g,n)} \\ = \quad &\{\text{v} \mid \exists \text{v}_0 \in \text{c} \cap [\text{v}, \ldots, \text{v} + \text{n}]\} \\ \cup\ &\{\text{v} \mid \exists \text{v}_0 \in \text{g} \cap [\text{v}, \ldots, \text{v} + \text{n} - 1] \ \wedge\ \text{n} \neq 0\} \end{aligned} \tag{7}$$

Since we want to use this function instead of the recursive function *local-EF*, we first have to prove the equivalence of both representations. Again, this is done using a KIV module. The corresponding part of the development graph is depicted in the upper half of Fig. 6. The proof obligations generated for the module ensure, that the nonrecursive function *local-EF′* satisfies the axioms of *local-EF*.

An additional advantage of this approach is, that we can use the nonrecursive function *local-EF′* to prove the theorems which use *local-EF* and *local-pr-EF*. To do this, we added these theorems as axioms in the specification which contains *local-EF* and used them as assumption in the previous section.

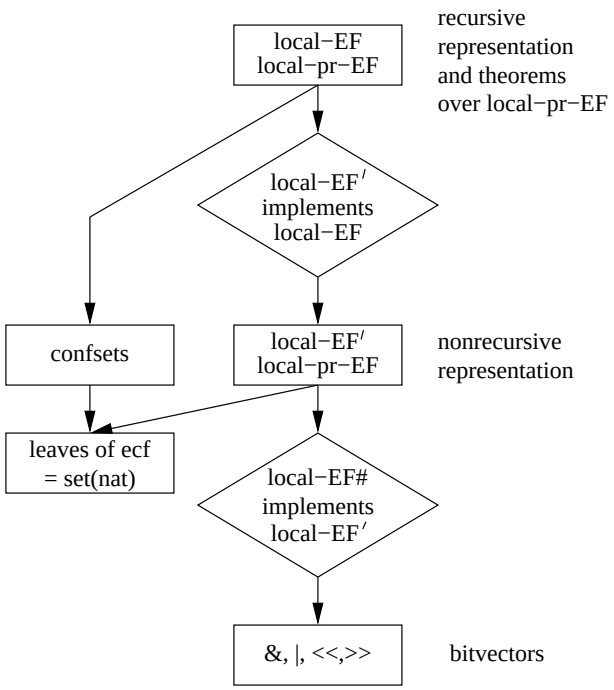


**Fig. 6.** Specification of explicit representation of time jumps

Now, we get these theorems as additional proof obligations for the nonrecursive definition *local-EF'* in the module.

The proofs of these theorems do not require any new ideas – they are typical for proofs over sets of natural numbers. Therefore, a discussion is omitted. The time required to do the correctness proofs was about two days per operator, including the proofs of the theorems assumed in the previous verification step.

### 5.5 Implementation using Bitvectors

The previous sections were concerned with deriving an efficient model checking algorithm on abstract data types. This section considers the efficient implementation of the optimized algorithm. RAVEN offers two representations. One that represents sets of natural numbers with bitvectors and one that uses interval lists. Since some of the algorithms used in the latter representation were already verified in an earlier case study with KIV [7] (unrelated to this project), and bitvectors are used as the default in RAVEN, we decided to verify this version.

The implementation verified was derived directly from the C++ source-code of RAVEN by omitting code concerned with memory allocation and partitioning of bitvectors into words.

Again, the verification was done using a KIV module, which is shown in the lower half of Fig. 6. In contrast to the previous sections, real programs are used

in the module. The programs *local-EF#* and *local-pr-EF#* depicted in Fig. 7 implement the functionality of *local-EF'* and *local-pr-EF* using bitvectors.

Bitvectors are defined to be strings of binary digits without leading zeros. In addition, the basic operations & (binary and), | (binary or), $\ll$ (logical shift left), $\gg$ (logical shift right), a length function # and a bit-selection function denoted by array-like subscripts are defined.

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

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

**Fig. 7.** Implementation of time-jump function *local-EF*

As proof obligations of the module it must be shown, that the implementation programs terminate and satisfy the axioms of *local-EF'* and *local-pr-EF* (for a general introduction to the theory of program modules see [6]; verification techniques for proof obligations in KIV are discussed in [7]).

To stay as close as possible to the implementation of RAVEN we decided to consider bitvectors without leading zeros only. This restriction is formalized as a predicate $r$. Additional proof obligations are generated by the KIV system to ensure that the programs terminate and keep the restriction invariant.

The correctness proofs for *local-EF#* and *local-pr-EF#* both require invariants for the while-loops. The one for *local-pr-EF#* was easy to obtain, since the current state of computation depends on few factors only. The invariant for *local-EF#* is shown in Fig. 8. It consists of two major parts. $INV_1$ states, that the postcondition is satisfied for the computations made so far. The main difficulty of the proof was to construct $INV_2$. It describes the variable *state*, which represents the "memory" of the algorithm. The construction of the invariants took several iterations and days, depending on the complexity of the operator and the number of "memory" variables. The size of the invariants ranges between 11

$$
\begin{aligned}
&\mathrm{INV}_{\mathrm{EF}} \equiv \mathrm{INV}_1 \wedge \mathrm{INV}_2 \\
&\mathrm{INV}_1 \equiv \forall\, n_1.\ \mathrm{pos} \le n_1 \\
&\qquad \rightarrow (\quad r[n_1] = 1 \\
&\qquad\quad \leftrightarrow \quad (\exists\, i.\ c[i] = 1 \wedge \neg\, i < n_1 \wedge \neg\, n_1 + n < i) \\
&\qquad\qquad \vee\, (\exists\, i.\ g[i] = 1 \wedge \neg\, i < n_1 \wedge \neg\, n_1 + n - 1 < i) \wedge n \neq 0) \\
&\mathrm{INV}_2 \equiv \quad r(r) \wedge \mathrm{state} \le n \\
&\qquad \wedge (\quad \mathrm{state} = 0 \vee c[\mathrm{pos} + n - \mathrm{state}] = 1 \\
&\qquad\quad \vee\, \mathrm{state} < n \wedge g[\mathrm{pos} + n - \mathrm{state} + 1] = 1) \\
&\qquad \wedge (\forall\, i.\ i < n - \mathrm{state} \rightarrow c[\mathrm{pos} + i] = 0) \\
&\qquad \wedge (\mathrm{state} < n \rightarrow (\forall\, i.\ i < n - \mathrm{state} + 1 \rightarrow g[\mathrm{pos} + i] = 0))
\end{aligned}
$$

**Fig. 8.** Invariant for while-loop of *local-EF*$\neq$-procedure

and 25 lines. Once the correct invariant was found, the proofs were large, but easy and automatic.

Summarizing, the effort taken to prove the correctness of the bitvector based implementation of RAVEN was about 2 weeks. On average, it took three iterations to find the correct invariant. A beneficial side effect of the verification was the discovery of inefficient and redundant code. The implementation of *local-EU* could be shortened from 73 to 18 lines of code.

## 6 Conclusion

In this paper we investigated the correctness of an optimized real-time model checking algorithm. We demonstrated that it is possible to develop the efficient implementation from the specification of the semantics in a series of refinement steps. Each step could be verified independently of the others.

During the verification, several errors were discovered in the time prediction functions, which constitute the heart of the optimization. Also, some inefficient code could be eliminated. We were able to define a common scheme for the correctness proofs of the different temporal operators, which reduced the verification effort significantly.

The time required to formalize and verify the model checking algorithm was about 4 months. Compared to the total time that was needed to develop and implement the optimizations, the extra effort was modest.

With the verification of the kernel algorithm we now feel justified to answer the question posed in the title of this paper with "yes", assuming the standard BDD package works correctly. Since model checkers are often used in safety critical applications, we hope our results encourage further research in their correctness.

## References


1. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 188–191, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.
2. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society Press.
3. E. Clarke, K.L. McMillian, X. Zhao, M. Fujita, and J.C.-Y. Yang. Spectral Transforms for large Boolean Functions with Application to Technologie Mapping. In *ACM/IEEE Design Automation Conference (DAC)*, pages 54–60, Dallas, TX, June 1993.
4. CoFI: The Common Framework Initiative. Casl — the CoFI algebraic specification language tentative design: Language summary, 1997. http://www.brics.dk/Projects/CoFI.
5. D. Long. Long-package sun release 4.1 overview of c-library functions, 1993.
6. 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.
7. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.
8. 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.
9. J. Ruf. RAVEN: Real-time analyzing and verification environment. Technical Report WSI 2000-3, University of Tübingen, Wilhelm-Schickard-Institute, January 2000.
10. 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.
11. Jürgen Ruf and Thomas Kropf. Using MTBDDs for Composition and Model Checking of Real-Time Systems. In *FMCAD 1998*. Springer, November 1998.
12. 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.
13. T. Vollmer. Korrekte Modellprüfung von Realzeitsystemen. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 2000. (in German).