

---

# Checking Interval Based Properties for Reactive Systems

---

Pei Yu and Xu Qiwen

## UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST home page: <http://www.iist.unu.edu>, if you would like to know more about UNU/IIST and its report series.

P.O. Box 3058  
Macau

---

# Checking Interval Based Properties for Reactive Systems

---

Pei Yu and Xu Qiwen

## Abstract

A reactive system does not terminate and its behaviors are typically defined as a set of infinite sequences of states. In formal verification, a requirement is usually expressed in a logic, and when the models of the logic are also defined as infinite sequences, such as the case for LTL, the satisfaction relation is simply defined by the containment between the set of system behaviors and that of logic models. However, this satisfaction relation does not work for interval temporal logics, where the models can be considered as a set of finite sequences. In this paper, we observe that for different interval based properties, different satisfaction relations are sensible. Two classes of properties are discussed, and accordingly two satisfaction relations are defined, and they are subsequently unified by a more general definition. A tool is developed based on the Spin model checking system to verify the proposed general satisfaction relation for a decidable subset of Discrete Time Duration Calculus.

**Keywords:** model checking, finitary property, reactive system, interval temporal logic

Pei Yu is a fellow of UNU/IIST. He is also a Ph.D. student at the Department of Computer Science and Technology in Nanjing University, China. His research interests include real-time system, concurrent Java programming, model checking and formal methods application. His email address is [pyu@iist.unu.edu](mailto:pyu@iist.unu.edu)

Xu Qiwen is an Assistant Professor of Faculty of Science and Technology, University of Macau. His research interests are formal methods, including verification of concurrent programs, specification and verification of real time systems, temporal logics, and tools. His email is [qwxu@umac.mo](mailto:qwxu@umac.mo)

## Contents

<b>List of Figures</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Quantified RDC</b>	<b>3</b>
2.1 Syntax . . . . .	3
2.2 Semantics . . . . .	3
2.3 Segments as Models . . . . .	5
<b>3 Finitary Safety Property and Eventual Persistence Property</b>	<b>6</b>
3.1 Finitary Safety Property . . . . .	6
3.2 Eventual Persistence Property . . . . .	6
3.3 Recognizing FS and EP Properties . . . . .	7
<b>4 Finitary Safety Formulas and Eventual Persistence Formulas in DQRDC</b>	<b>9</b>
<b>5 Satisfaction Relations</b>	<b>11</b>
<b>6 Model Checking</b>	<b>12</b>
<b>7 Case Study</b>	<b>14</b>
7.1 Peterson's Mutual Exclusion Algorithm . . . . .	14
7.2 Peterson's Leader Election Algorithm . . . . .	16
<b>8 Discussion</b>	<b>18</b>



---

## List of Figures

1	Recognition algorithm for FS property. . . . .	8
2	Recognition algorithm for EP property. . . . .	8
3	Peterson's mutual exclusion algorithm . . . . .	15
4	Temporal claim for $Req_1$ . . . . .	15
5	Temporal claim for $Req_2$ . . . . .	16
6	Peterson's algorithm for leader election. . . . .	17
7	Temporal claim for $Req_3$ . . . . .	17



## 1 Introduction

As the first step towards establishing a formal verification method, we need to define the meaning of a system (semantics), a mechanism to express requirements (specification language, often a logic) and what it means for a system to satisfy a requirement (satisfaction relation). A reactive system does not terminate and its semantics is typically defined as the set of infinite sequences of states generated by the execution of the system. In formal verification, a requirement is usually expressed in a logic. A popular specification logic for reactive systems is LTL, and it is not by chance that a model of an LTL formula is also an infinite sequence of states. In this setting, the satisfaction relation is simply the containment between the set of system behaviors and that of logic models.

Another class of temporal logic is interval based. Although interval logics are less widely used compared to LTL, there are properties that are easier to express in such logics [16]. In almost all the interval based logics, intervals are finite ones, corresponding to finite sequences of states, which makes set containment no longer an appropriate satisfaction relation when these logics are used as the specification logic. One possible definition for this satisfaction relation is that a reactive system satisfies such a property iff all the finite prefixes of all the infinite behaviors satisfy the property, i.e. all the prefixes must be models of the requirement formula. However, while this definition makes sense in some cases, it does not in some other cases. This depends on the kind of properties. Take a mutual exclusion algorithm as an example, there are two well-known desirable properties: no two processes can be in critical sections at the same time, and a process should be allowed to enter the critical section eventually. An attempt to express these two properties in the Interval Temporal Logic (ITL) [11] or one of its variants may result in the following two formulas:  $\Box\neg(ain \wedge bin)$  and  $\Diamond ain$ . Although the syntax is same as LTL for these formulas, the interpretation for them as ITL formulas is over finite intervals:

- $\Box\neg(ain \wedge bin)$ : a model of it is a finite interval such that  $ain$  and  $bin$  are never true together in any of the sub-interval;
- $\Diamond ain$ : a model of it is a finite interval such that  $ain$  is true in one state.

For  $\Box\neg(ain \wedge bin)$ , all the execution prefixes of a mutual exclusion algorithm should be its models, and if all the execution prefixes are models of the formula, intuitively the first requirement is satisfied. This is due to the fact that the property is a *safety property* [8]. Therefore, for safety properties, the above mentioned satisfaction relation is appropriate. On the other hand, for  $\Diamond ain$ , this satisfaction relation is obviously not appropriate - clearly we cannot expect for all the execution prefixes of a mutual exclusion algorithm, a particular process is in critical section. The second property is a *liveness property* [8], and it is often said that ITL (with finite intervals) cannot express liveness.

We have used two simple and familiar properties to illustrate the issue. These properties can be easily expressed in LTL, and in fact in basically the same syntactical forms. There are certainly properties which have different forms in LTL and ITL, and which are easier to express in ITL.

For example, the following two formulas in Duration Calculus [20] (DC), an extension of ITL present the same problem as the previous two ITL formulas.

- $\int ain < 3$ : a model of it is a finite interval, over which  $ain$  occurs in less than 3 states;
- $\int ain \geq 3$ : a model of it is a finite interval, over which  $ain$  occurs in at least 3 states.

There are some efforts to develop logics over infinite intervals [12, 18, 21]. With such logics, it is possible to use set containment as the satisfaction relation. However, logics over infinite intervals have not been widely accepted due to several reasons. Firstly, length of interval is an important concept in ITL, but for an infinite interval, this leads to an infinite number, which many people do not feel comfortable to deal with. Secondly, ITL can be decided by using finite automata when it is over finite intervals [6, 13], but obviously needs automata over infinite words when the logic is defined over infinite intervals. Even worse, effective automaton constructing techniques for LTL do not seem to apply to interval logics as observed by Wolper [19].

In this paper, we propose to use the usual interval logic over finite intervals as the specification logic, and redefine the satisfaction relation. We identify two classes of properties, and accordingly define two satisfaction relations. The two relations are unified by a more general definition, and a method to automatically check whether a reactive system satisfies properties in interval logics according to this new definition is investigated. This turns out to be checking containment between the set of (infinite) behaviors of the reactive system and the language of a Büchi automaton which is the same as the finite automaton for the interval logic formula except the final states of the finite automaton are now the accepting states of the Büchi automaton.

As a particular application of our method, we have chosen a decidable subset of Discrete Time Duration Calculus as the specification logic and implemented an extension of the classical algorithm [6] for converting the formulas to finite automata by using the Grail+ package [5]. The output of the automata is transformed to Büchi automata in the form accepted by the Spin model checking system which subsequently is used for model checking.

The rest of the paper is organized as follows. In the next section, we give the syntax and the semantics of the logic, and the construction that converts a formula to a regular language representing the models of the formula. In Section 3, two kinds of properties, finitary safety property and eventual persistence property, are defined, followed by methods to recognize them using finite automata. Formulas expressing finitary safety and eventual persistence properties are characterized syntactically in Section 4. Satisfaction relations for finitary safety property and eventual persistence property are first studied separately and then unified in Section 5. A method to model check a reactive system with the proposed satisfaction relation is shown in Section 6. In Section 7, two well-known algorithms are model checked as case studies. The paper ends with a discussion.

## 2 Quantified RDC

Quantified RDC(QRDC) is an extension of a decidable subset of Duration Calculus by introducing quantifiers on state variables.

We use the following notations throughout the paper. For a finite set  $\Sigma$ ,  $\Sigma^+$  denotes the set of all non-empty finite sequences over  $\Sigma$  and  $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$ , where  $\varepsilon$  is the unique empty sequence. We use  $\Sigma^\omega$  to denote the set of infinite sequences, i.e.  $\omega$ -sequences, over  $\Sigma$ .  $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ . For a sequence  $\alpha$  over  $\Sigma$ ,  $|\alpha|$  denotes the length of  $\alpha$ . Specially,  $|\alpha| = \infty$  for  $\alpha \in \Sigma^\omega$  and  $|\varepsilon| = 0$ . For two sequences  $\alpha_1 \in \Sigma^*$  and  $\alpha_2 \in \Sigma^\infty$ ,  $\alpha_1 \cdot \alpha_2$  denotes the sequence obtained by concatenating  $\alpha_2$  to the end of  $\alpha_1$ ;  $\alpha_1$  is called a prefix of  $\alpha_2$ , denoted as  $\alpha_1 \preceq \alpha_2$ , iff  $\exists \alpha_3 \in \Sigma^\infty : \alpha_1 \cdot \alpha_3 = \alpha_2$ . For two sets of finite sequences  $P_1$  and  $P_2$ ,  $P_1 \cdot P_2 = \{\alpha_1 \cdot \alpha_2 \mid \alpha_1 \in P_1 \wedge \alpha_2 \in P_2\}$ .

### 2.1 Syntax

We use  $p, p_1, \dots, p_n$  to denote state variables and  $SVar$  the finite set of all the state variables. State expressions and formulas are defined as follows:

$$\begin{aligned} S &::= 0 \mid 1 \mid p \mid \neg S_1 \mid S_1 \vee S_2 \\ \phi &::= \llbracket S \rrbracket \mid \neg \phi \mid \phi_1 \vee \phi_2 \mid \phi_1; \phi_2 \mid \exists p. \phi \end{aligned}$$

### 2.2 Semantics

The interpretation for state variables is given as a function

$$I \in SVar \rightarrow (\mathbb{T}ime \rightarrow \{0, 1\})$$

When we choose  $\mathbb{T}ime$  to be  $\mathbb{N}$ , we obtain QRDC in discrete time (DQRDC). The interpretation function  $I$  is extended to state expression  $S$  by induction on the structure of  $S$ :

1.  $I\llbracket 0 \rrbracket(t) = 0$
2.  $I\llbracket 1 \rrbracket(t) = 1$
3.  $I\llbracket p \rrbracket(t) = I(p)(t)$
4.  $I\llbracket \neg S \rrbracket(t) = 1 - I\llbracket S \rrbracket(t)$
5.  $I\llbracket S_1 \vee S_2 \rrbracket(t) = \max(I\llbracket S_1 \rrbracket(t), I\llbracket S_2 \rrbracket(t))$

Given an interpretation  $I$ , the semantics of a QRDC formula  $\phi$  is given by a function

$$I\llbracket \phi \rrbracket \in \mathbb{Intv} \rightarrow \{tt, ff\},$$

where the set of time intervals is defined as

$$\mathbb{Intv} \hat{=} \{[b, e] \mid b, e \in \mathbb{Time} \wedge b \leq e\}.$$

The function is defined as follows:

1.  $I[\llbracket S \rrbracket](b, e) = tt$  iff  $b < e$  and  $\int_b^e I[S](t) = e - b$  for dense time or  $b < e$  and  $\sum_{t=b}^{e-1} I[S](t) = e - b$  for discrete time
2.  $I[\llbracket \neg \phi \rrbracket](b, e) = tt$  iff  $I[\llbracket \phi \rrbracket](b, e) = ff$
3.  $I[\llbracket \phi_1 \vee \phi_2 \rrbracket](b, e) = tt$  iff  $I[\llbracket \phi_1 \rrbracket](b, e) = tt$  or  $I[\llbracket \phi_2 \rrbracket](b, e) = tt$
4.  $I[\llbracket \phi_1; \phi_2 \rrbracket](b, e) = tt$  iff  $I[\llbracket \phi_1 \rrbracket](b, m) = tt$  and  $I[\llbracket \phi_2 \rrbracket](m, e) = tt$ , for some  $m \in [b, e] \cap \mathbb{Time}$
5.  $I[\llbracket \exists p. \phi \rrbracket](b, e) = tt$  iff for some interpretation  $I'$ , which is  $p$ -equivalent to  $I$ ,  $I'[\llbracket \phi \rrbracket](b, e) = tt$ . An interpretation  $I'$  is  $p$ -equivalent to another interpretation  $I$  iff  $I(p_1)(t) = I'(p_1)(t)$  for all  $p_1 \neq p$  and  $t \in \mathbb{Time}$ .

Standard abbreviations from predicate logic, e.g. “ $\wedge$ ”, “ $\Rightarrow$ ” and “ $\Leftrightarrow$ ”, are used in this paper. Moreover, the following abbreviations for QRDC formulas are frequently used:

$$\begin{array}{ll} \llbracket \perp \rrbracket & \hat{=} \neg \llbracket \top \rrbracket & \llbracket S \rrbracket^* & \hat{=} \llbracket S \rrbracket \vee \llbracket \perp \rrbracket \\ false & \hat{=} \llbracket 0 \rrbracket & true & \hat{=} \neg false \\ \diamond \phi & \hat{=} true; \phi; true & \square \phi & \hat{=} \neg \diamond \neg \phi \\ \square_p \phi & \hat{=} \neg(\neg \phi; true) & l = 0 & \hat{=} \neg \llbracket \top \rrbracket \end{array}$$

For DQRDC, we also use the following abbreviations ( $k \in \mathbb{N}$ ):

$$\begin{array}{ll} l = 1 & \hat{=} \llbracket \top \rrbracket \wedge \neg(\llbracket \top \rrbracket; \llbracket \top \rrbracket) \\ \int S = 0 & \hat{=} \llbracket \neg S \rrbracket \vee l = 0 \\ \int S = 1 & \hat{=} (\int S = 0); (\llbracket S \rrbracket \wedge l = 1); (\int S = 0) \\ \int S = k+1 & \hat{=} (\int S = k); (\int S = 1) \quad (k \geq 1) \\ \int S \geq k & \hat{=} (\int S = k); true \\ \int S > k & \hat{=} \int S \geq k + 1 \\ \int S \leq k & \hat{=} \neg(\int S > k) \\ \int S < k & \hat{=} \int S \leq k - 1 \end{array}$$

where  $\int S$  denotes the accumulated time when  $S$  evaluates 1. The length  $l$  of current interval can be encoded as  $\int 1$ .

### 2.3 Segments as Models

We call each  $obs \in Obs = 2^{SVar}$  an observation. For a state expression  $S$ ,  $N(S) \subseteq Obs$  denotes the set of observations where  $S$  is evaluated to 1:

$$N(S) \hat{=} \{obs \mid (\bigwedge_{p_1 \in obs} p_1 \wedge \bigwedge_{p_2 \in (SVar - obs)} \neg p_2) \Rightarrow S\}.$$

For an interpretation  $I$  and  $t \in Time$ ,  $I(t)$  is identified with the following observation

$$I(t) \hat{=} \{p \mid p \in SVar \wedge I(p)(t) = 1\}$$

In DQRDC, given an interpretation  $I$  and an interval  $[b, e]$ , we use the pair  $(I, [b, e])$ , called a segment, to denote the finite sequence  $I(b), I(b+1), \dots, I(e-1)$  of observations. A segment  $(I, [b, e])$  satisfies a formula  $\phi$ , denoted as  $I, [b, e] \models \phi$ , iff  $I \llbracket \phi \rrbracket ([b, e]) = tt$ . For a formula  $\phi$ , we call each segment  $(I, [b, e])$  satisfying  $I, [b, e] \models \phi$  a model of  $\phi$ . The set of models of  $\phi$ , denoted as  $\mathcal{M} \llbracket \phi \rrbracket$ , can be constructed [6, 13] as the following regular language over the alphabet  $Obs$ :

$$\begin{aligned} \mathcal{M} \llbracket \llbracket S \rrbracket \rrbracket &= (N(S))^+ && \text{(positive closure)} \\ \mathcal{M} \llbracket \phi_1 \vee \phi_2 \rrbracket &= \mathcal{M} \llbracket \phi_1 \rrbracket \cup \mathcal{M} \llbracket \phi_2 \rrbracket && \text{(union)} \\ \mathcal{M} \llbracket \neg \phi \rrbracket &= Obs^* - \mathcal{M} \llbracket \phi \rrbracket && \text{(complementation)} \\ \mathcal{M} \llbracket \phi_1; \phi_2 \rrbracket &= \mathcal{M} \llbracket \phi_1 \rrbracket \cdot \mathcal{M} \llbracket \phi_2 \rrbracket && \text{(concatenation)} \\ \mathcal{M} \llbracket \exists p. \phi \rrbracket &= Equiv_p(\mathcal{M} \llbracket \phi \rrbracket) && \text{(equivalence closure)} \end{aligned}$$

where  $Equiv_p(\mathcal{M} \llbracket \phi \rrbracket)$  is defined as follows.

**Definition 2.1.** Given two finite sequences  $\alpha = s_0, s_1, \dots, s_n$  and  $\alpha' = s'_0, s'_1, \dots, s'_n$  of observations and a state variable  $p$ ,  $\alpha$  and  $\alpha'$  are  $p$ -equivalent iff  $s_i \setminus \{p\} = s'_i \setminus \{p\}$  for  $0 \leq i \leq n$ .

Given a state variable  $p$ , the equivalence closure of a set of observation sequences  $\mathcal{M}$ ,  $Equiv_p(\mathcal{M})$  is defined to be the set

$$\{\alpha \mid \text{if there exists } \beta \in \mathcal{M}, \text{ such that } \alpha \text{ and } \beta \text{ are } p\text{-equivalent}\}$$

**Lemma 2.2.** In DQRDC, for a formula  $\phi$  and a segment  $(I, [b, e])$ ,  $(I, [b, e]) \in \mathcal{M} \llbracket \phi \rrbracket$  iff  $I, [b, e] \models \phi$ .

*Proof.* The correctness of this lemma can be easily achieved by induction on the structure of  $\phi$ .  $\square$

In DQRDC, a formula  $\phi$  is valid iff  $\mathcal{M} \llbracket \phi \rrbracket = Obs^*$ , and it is satisfiable iff  $\mathcal{M} \llbracket \phi \rrbracket \neq \emptyset$ . It is obvious that  $\phi$  is valid iff  $\neg \phi$  is not satisfiable.

### 3 Finitary Safety Property and Eventual Persistence Property

Our work on classification of properties follows that of [8, 3]. However, in our setting, a property is a set of finite sequences of states, and therefore some modification is necessary. To indicate whether properties are formed by finite or infinite sequences of states, we call them finitary properties or infinitary properties.

#### 3.1 Finitary Safety Property

A safety property stipulates that “something (bad) never happens”. In [3], it is considered that if a “bad” thing happens in an infinite sequence, then it must do so after some finite prefix and must be irremediable. We follow this view, but since our property is over finite sequences, we call our property *finitary safety property* and it is formally defined as follows:

**Definition 3.1 (finitary safety property).** *For a finite set  $\Sigma$  and a property  $P \subseteq \Sigma^*$ ,  $P$  is a finitary safety property, or an FS property, iff:*

$$\forall \alpha \in \Sigma^* : (\alpha \notin P \Leftrightarrow \exists \beta \in \Sigma^* : (\beta \preceq \alpha \wedge \forall \gamma \in \Sigma^* : (\beta \preceq \gamma \Rightarrow \gamma \notin P))).$$

By taking negation on both sides, we have

$$\forall \alpha \in \Sigma^* : (\alpha \in P \Leftrightarrow \forall \beta \in \Sigma^* : (\beta \preceq \alpha \Rightarrow \exists \gamma \in \Sigma^* : (\beta \preceq \gamma \wedge \gamma \in P))).$$

This can be simplified, and we have the following theorem which says that a property is an FS property iff it is prefix-closed.

**Theorem 3.2.** *A non-empty property  $P \subseteq \Sigma^*$  is an FS property iff*

$$\forall \alpha \in \Sigma^* : (\alpha \in P \Leftrightarrow \forall \beta \in \Sigma^* : (\beta \preceq \alpha \Rightarrow \beta \in P)),$$

*i.e.  $Pref(P) = P$ , where  $Pref(P) = \{\beta \mid \beta \in \Sigma^* \wedge \exists \alpha \in P : \beta \preceq \alpha\}$ .*

#### 3.2 Eventual Persistence Property

An infinitary property is a liveness property [3] iff all finite executions can be extended to satisfy the property. Intuitively, this is to say a “good” thing always has the chance to take place no matter what has happened. Although in the general case, the “good thing” may take infinite number of actions to ensure, for example, something that happens infinitely often, some “good thing” can and in fact is only meaningful to occur after a finite number of steps. This is often called *eventuality*. Moreover, we are particularly interested in a class of properties such that if something (“good”) has happened, nothing later can undo it. This is known as *persistence*. Formally, eventuality and persistence properties are defined as:

**Definition 3.3 (Eventuality and Persistence).** For a finite set  $\Sigma$  and a property  $P \subseteq \Sigma^*$  ( $P \neq \emptyset$ ),

- $P$  is an eventuality property iff  $\forall \alpha \in \Sigma^* : \exists \beta \in \Sigma^* : (\alpha \preceq \beta \wedge \beta \in P)$ ;
- $P$  is a persistence property iff  $\forall \alpha \in P : \forall \beta \in \Sigma^* : (\alpha \preceq \beta \Rightarrow \beta \in P)$ .

$P$  is called an eventual persistence property, or an EP property, iff  $P$  is both an eventuality property and a persistence property.

The following theorem follows directly from the definition.

**Theorem 3.4.** A non-empty property  $P \subseteq \Sigma^*$  is an EP property iff  $\text{Pref}(P) = \Sigma^*$  and  $P \cdot \Sigma^* = P$ .

### 3.3 Recognizing FS and EP Properties

A finitary property can be specified by a finite automaton whose language is the set of the finite sequences of the property. In the following, similar to what has been done in [2] w.r.t. Büchi automata expressing safety properties and liveness properties, we give the recognition procedures for FS properties and EP properties specified using finite automata.

Formally, a finite automaton  $FA$  is defined as a five-tuple  $\langle \Sigma, Q, Q_0, F, \Delta \rangle$ , where the alphabet  $\Sigma$ , is a finite set of input symbols,  $Q$  is a finite set of states,  $Q_0 \subseteq Q$  is the set of initial states,  $F \subseteq Q$  is the set of final states, and  $\Delta : (Q \times \Sigma) \rightarrow 2^Q$  is the transition function. The language of  $FA$  is denoted by  $L(FA)$ .

An  $FA$  is reduced if, for any state  $q \in Q$ , there is a path from an initial state to  $q$  and there is a path from  $q$  to a final state. From an  $FA$ , a reduced  $FA$  accepting the same language can always be obtained by deleting those states either not reachable from any initial state or from which no final state can be reached. For a reduced  $FA$ , we define its closure  $cl(FA)$  as the result automaton by setting  $F = Q$ . From the construction of  $cl(FA)$ , we can easily get

**Lemma 3.5.** For a reduced  $FA$ ,  $L(cl(FA)) = \text{Pref}(L(FA))$ .

For FS properties, we have

**Theorem 3.6.** A reduced finite automaton  $FA$  specifies an FS property iff  $L(FA) = L(cl(FA))$ .

*Proof.* It follows from Theorem 3.2 and Lemma 3.5. □

Input: A finite automaton  $FA$ .  
Output: If  $FA$  specifies an FS property, return “yes”; Otherwise, return “no”.

Reduce and determinize  $FA$ , get automaton  $DFA$ ;  
If  $F = Q$  in  $DFA$  then  
    return yes;  
else  
    return no;

Figure 1: Recognition algorithm for FS property.

Input: A finite automaton  $FA$ .  
Output: If  $FA$  specifies an EP formula, return “yes”; Otherwise, return “no”.

Reduce and determinize  $FA$ , get automaton  $DFA = \langle \Sigma, Q, Q_0, F, \Delta \rangle$ ;  
Compute  $\Sigma' = \bigcap_{q \in F} \text{Acpt}(q)$ , where  $\text{Acpt}(q) = \{a \mid a \in \Sigma \wedge \exists q' \in F : q' \in \Delta(q, a)\}$ ;  
If  $\Sigma' \neq \Sigma$  then  
    return no;  
Construct the closure of  $DFA$ , get  $cl(DFA) = \langle \Sigma, Q, Q_0, Q, \Delta \rangle$ ;  
Compute the complementary automaton of  $cl(DFA)$ , get  $\overline{cl(DFA)}$ ;  
If  $L(\overline{cl(DFA)})$  is not empty then  
    return no;  
else  
    return yes;

Figure 2: Recognition algorithm for EP property.

From this Theorem, we immediately have the following algorithm(Fig. 1) for recognizing FS properties.

The termination and correctness of this algorithm is obvious and we omit the proof for it.

For EP properties, we have the following similar theorem.

**Theorem 3.7.** *A finite automaton  $FA$  specifies an EP property iff  $L(\overline{cl(FA)}) = \Sigma^*$  and  $L(FA) \cdot \Sigma^* = L(FA)$ .*

*Proof.* It follows from Theorem 3.4 and Lemma 3.5. □

We can recognize an EP property by the algorithm shown in Fig. 2.

**Lemma 3.8.** *Given a finite automaton  $FA$ , algorithm shown in Fig. 2 terminates and correctly judges whether  $FA$  specifies an EP property.*

*Proof.* Termination: The algorithm obviously terminates, since each step of it terminates.

Correctness: Since it is obvious that  $L(\overline{cl(DFA)}) = \emptyset$  iff  $L(DFA) = L(FA) = \Sigma^*$ , it suffices to show  $\Sigma' = \Sigma$  iff  $L(FA) \cdot \Sigma^* = L(FA)$ .

For the only if case, according to the construction of  $\Sigma'$ ,  $\Sigma' = \Sigma$  means from any final state of  $DFA$ , any input symbol will only lead to a final state, so  $L(DFA) \cdot \Sigma^* \subseteq L(DFA)$ . It is obvious  $L(DFA) \subseteq L(DFA) \cdot \Sigma^*$ . Therefore,  $L(DFA) = L(DFA) \cdot \Sigma^*$ . Because  $L(FA) = L(DFA)$ , the only if part is proved.

For the if case, we prove by contradiction. Assume  $L(FA) \cdot \Sigma^* = L(FA)$ , but there exists  $a \in \Sigma \setminus \Sigma'$ . From the computing process of  $\Sigma'$ , we can conclude there exists  $q \in F$  and  $\Delta(q, a) \cap F = \emptyset$ . Let  $\alpha \in L(FA)$  and  $\alpha$  puts  $DFA$  in state  $q$ . It follows that  $\alpha \cdot a \notin L(DFA)$ , and this contradicts  $L(FA) \cdot \Sigma^* = L(FA)$ . Thus, the if case is proved.  $\square$

## 4 Finitary Safety Formulas and Eventual Persistence Formulas in DQRDC

In DQRDC, each formula  $\phi$  defines a set of finite sequences over  $Obs$  by  $\mathcal{M}[\phi]$ . A DQRDC formula  $\phi$  is called a finitary safety formula (abbrev. FS formula) iff  $\mathcal{M}[\phi]$  is an FS property, and it is called an eventual persistence formula (abbrev. EP formula) iff  $\mathcal{M}[\phi]$  is an EP property. In this section, we will characterize these two kinds of formulas.

**Lemma 4.1.** *A DQRDC formula  $\phi$  is an FS formula iff  $\mathcal{M}[\Box_p \phi] = \mathcal{M}[\phi]$ .*

*Proof.* According to Theorem 3.2, it suffices to prove  $Pref(\mathcal{M}[\phi]) = \mathcal{M}[\phi]$  iff  $\mathcal{M}[\Box_p \phi] = \mathcal{M}[\phi]$ .

For the if part, it suffices to prove  $Pref(\mathcal{M}[\Box_p \phi]) \subseteq \mathcal{M}[\Box_p \phi]$ , since  $\mathcal{M}[\phi] = \mathcal{M}[\Box_p \phi]$  and it's obvious that  $\mathcal{M}[\Box_p \phi] \subseteq Pref(\mathcal{M}[\Box_p \phi])$ . For each  $\beta \in Pref(\mathcal{M}[\Box_p \phi])$ , there exists  $\alpha (\beta \preceq \alpha)$  satisfying  $\alpha \in \mathcal{M}[\Box_p \phi]$ . We can easily see that  $\beta \in \mathcal{M}[\Box_p \phi]$ , thus  $Pref(\mathcal{M}[\Box_p \phi]) \subseteq \mathcal{M}[\Box_p \phi]$  and the if part is proved.

For the only if part we only need to prove  $\mathcal{M}[\phi] \subseteq \mathcal{M}[\Box_p \phi]$  holds. For any  $\alpha \in \mathcal{M}[\phi]$ , we have for any  $\beta \preceq \alpha$ ,  $\beta \in Pref(\mathcal{M}[\phi]) = \mathcal{M}[\phi]$ . Thus, we can conclude  $\alpha \in \mathcal{M}[\Box_p \phi]$  and  $\mathcal{M}[\phi] \subseteq \mathcal{M}[\Box_p \phi]$  holds.  $\square$

**Theorem 4.2 (FS formula).** *Every DQRDC formula of the form  $\llbracket S \rrbracket^*$ ,  $\int S \leq k (k \in \mathbb{N})$ ,  $\Box_p \phi$  or  $\Box \phi$  is an FS formula and if  $\phi_1, \phi_2$  are FS formulas, then so are  $\phi_1 \wedge \phi_2$ ,  $\phi_1 \vee \phi_2$ ,  $\phi_1; \phi_2$  and  $\exists p. \phi_1$ .*

*Proof.* According to Lemma 4.1, it suffices to prove that, for formula  $\phi$ 's of these forms,  $\Box_p \phi = \phi$  holds. This is obviously true for  $\llbracket S \rrbracket^*, S \leq k (k \in \mathbb{N})$  and  $\Box_p \phi$ . Since  $\Box \phi \hat{=} \Box_p (\neg(true; \neg\phi))$ , this

also holds for  $\Box\phi$ . For  $\phi_1 \wedge \phi_2$ , we have

$$\begin{aligned}
\phi_1 \wedge \phi_2 &= \Box_p \phi_1 \wedge \Box_p \phi_2 \\
&= \neg((\neg\phi_1; true) \vee (\neg\phi_2; true)) \\
&= \neg((\neg\phi_1 \vee \neg\phi_2); true) \\
&= \neg(\neg(\phi_1 \wedge \phi_2); true) \\
&= \Box_p(\phi_1 \wedge \phi_2)
\end{aligned}$$

For formulas  $\phi$  with the form  $\phi_1 \vee \phi_2$  or  $\exists p.\phi_1$ , it is obvious  $\Box_p\phi \Rightarrow \phi$ , and we will next prove  $\phi \Rightarrow \Box_p\phi$  as follows,

$$\begin{aligned}
\phi_1 \vee \phi_2 &= \Box_p \phi_1 \vee \Box_p \phi_2 \\
&\Rightarrow \Box_p(\phi_1 \vee \phi_2); \\
\exists p.\phi_1 &= \exists p.(\Box_p \phi_1) \\
&\Rightarrow \Box_p(\exists p.\phi_1).
\end{aligned}$$

For  $\phi_1; \phi_2$ , we prove  $\mathcal{M}[\phi_1; \phi_2] \subseteq \mathcal{M}[\Box_p(\phi_1; \phi_2)]$  holds. For any  $\alpha \in \mathcal{M}[\phi_1; \phi_2]$ , there exist  $\alpha_1, \alpha_2 \in Obs^*$  satisfying  $\alpha = \alpha_1 \cdot \alpha_2$  and  $\alpha_1 \in \mathcal{M}[\phi_1] \wedge \alpha_2 \in \mathcal{M}[\phi_2]$ . Each prefix of  $\alpha$  can be represented as  $\alpha'_1$  or  $\alpha_1 \cdot \alpha'_2$ , where  $\alpha'_1 \preceq \alpha_1$  and  $\alpha'_2 \preceq \alpha_2$ , and we can easily get that  $\alpha'_1 \in \mathcal{M}[\phi_1]$  and  $\alpha'_2 \in \mathcal{M}[\phi_2]$ . For the first form, we have  $\alpha'_1 = \alpha'_1 \cdot \epsilon \in \mathcal{M}[\phi_1; \phi_2]$ . Obviously, for the second form we have  $\alpha_1 \cdot \alpha'_2 \in \mathcal{M}[\phi_1; \phi_2]$ . Since all the prefixes of  $\alpha$  are in  $\mathcal{M}[\phi_1; \phi_2]$ , we can conclude  $\alpha \in \mathcal{M}[\Box_p(\phi_1; \phi_2)]$ . Thus,  $\mathcal{M}[\phi_1; \phi_2] \subseteq \mathcal{M}[\Box_p(\phi_1; \phi_2)]$ .  $\square$

For EP formulas, it follows from Theorem 3.4 that:

**Lemma 4.3.** *For a DQRDC formula  $\phi$ ,  $\phi$  is an EP formula if  $\mathcal{M}[\phi] = \mathcal{M}[\Diamond\phi]$ .*

*Proof.* From  $\mathcal{M}[\phi] = \mathcal{M}[\Diamond\phi]$ , we have

$$\begin{aligned}
Pref(\mathcal{M}[\phi]) &= Pref(\mathcal{M}[\Diamond\phi]) = Pref(\mathcal{M}[true; \phi; true]) \\
&= Pref(\mathcal{M}[true] \cdot \mathcal{M}[\phi] \cdot \mathcal{M}[true]) \\
&= Pref(\Sigma^* \cdot \mathcal{M}[\phi] \cdot \Sigma^*) = \Sigma^*
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{M}[\phi] \cdot \Sigma^* &= \mathcal{M}[\Diamond\phi] \cdot \Sigma^* = \mathcal{M}[true; \phi; true] \cdot \Sigma^* \\
&= \mathcal{M}[true] \cdot \mathcal{M}[\phi] \cdot \mathcal{M}[true] \cdot \Sigma^* \\
&= \mathcal{M}[true] \cdot \mathcal{M}[\phi] \cdot \mathcal{M}[true] = \mathcal{M}[\phi].
\end{aligned}$$

Thus,  $\phi$  is an EP property.  $\square$

**Theorem 4.4 (EP formula).** *Every DQRDC formula of the form  $\int S \geq k (S \neq 0 \text{ and } k \in \mathbb{N})$  and  $\Diamond\phi$  is an EP formula and if  $\phi_1, \phi_2$  are EP formulas, then so are  $\phi_1 \wedge \phi_2, \phi_1 \vee \phi_2, \phi_1; \phi_2$  and  $\exists p.\phi_1$ .*

*Proof.* According to Lemma 4.3, it suffices to prove that, for all the formula  $\phi$  of these forms,  $\phi = \Diamond\phi$  holds. This is obviously true for  $\int S \geq k$  and  $\Diamond\phi$ . For  $\phi_1 \vee \phi_2$  and  $\phi_1; \phi_2$ , we have

$$\begin{aligned}
\Diamond(\phi_1 \vee \phi_2) &= true; (\phi_1 \vee \phi_2); true \\
&= (true; \phi_1; true) \vee (true; \phi_2; true) \\
&= \Diamond\phi_1 \vee \Diamond\phi_2 \\
&= \phi_1 \vee \phi_2 \\
\Diamond(\phi_1; \phi_2) &= true; \phi_1; \phi_2; true \\
&= true; \Diamond\phi_1; \Diamond\phi_2; true \\
&= true; (true; \phi_1; true); (true; \phi_2; true); true \\
&= true; \phi_1; true; true; \phi_2; true \\
&= \Diamond\phi_1; \Diamond\phi_2 \\
&= \phi_1; \phi_2
\end{aligned}$$

For formulas  $\phi$  of the form  $\phi_1 \wedge \phi_2$  or  $\exists p.\phi_1$ , since  $\phi \Rightarrow \Diamond\phi$  obviously holds, we prove  $\Diamond\phi \Rightarrow \phi$  also holds as follows:

$$\begin{aligned}
\Diamond(\phi_1 \wedge \phi_2) &\Rightarrow \Diamond\phi_1 \wedge \Diamond\phi_2 = \phi_1 \wedge \phi_2; \\
\Diamond(\exists p.\phi_1) &\Rightarrow \exists p.(\Diamond\phi_1) = \exists p.\phi_1.
\end{aligned}$$

Therefore, the lemma holds. □

These syntactical conditions are sufficient but not the necessary for judging FS/EP formulas. It is obvious that formulas with different syntactical forms may define the same property. For example, although the set of FS formulas is not closed under “ $\neg$ ”, some formulas containing “ $\neg$ ” could be FS formulas, e.g.  $\neg\neg\Box\phi$ . To semantically judge an FS/EP formula, we can first construct a finite automaton  $FA$  such that  $L(FA) = \mathcal{M}[\phi]$ , then apply the algorithm presented in Fig. 1 and Fig. 2.

## 5 Satisfaction Relations

A system behavior is modeled by a finite or infinite sequence of states. As defined earlier, a property is a set of finite sequences of states. A finite sequence  $\alpha$  satisfies a property  $P$ , denoted by  $\alpha \models P$ , iff  $\alpha \in P$ . For an infinite, more precisely  $\omega$ -sequence, we first look at the satisfaction relations for FS and EF properties separately.

For an FS property, an  $\omega$ -sequence satisfies the property iff all the finite prefixes of the sequence satisfy it.

**Definition 5.1.** For an  $\omega$ -sequence  $\sigma$  and an FS property  $P$  all defined over  $\Sigma$ ,  $\sigma$  satisfies  $P$ , denoted as  $\sigma \models_i P$ , iff  $\forall \alpha \in \Sigma^* : (\alpha \preceq \sigma \Rightarrow \alpha \in P)$ .

Intuitively, an  $\omega$ -sequence satisfies an EP property iff it has a finite prefix where the “good” thing has happened.

**Definition 5.2.** For an  $\omega$ -sequence  $\sigma$  and an EP property  $P$  all defined over  $\Sigma$ ,  $\sigma$  satisfies  $P$ , denoted as  $\sigma \models_i P$ , iff  $\exists \alpha \in \Sigma^* : (\alpha \preceq \sigma \wedge \alpha \in P)$ .

The two satisfaction relations can be unified by a more general one.

**Definition 5.3 (satisfaction of finitary property).** For an  $\omega$ -sequence  $\sigma$  and a finitary property  $P$ , both defined over  $\Sigma$ ,  $\sigma$  satisfies  $P$ , denoted as  $\sigma \models_i P$ , iff  $\exists \gamma \in \Sigma^* : (\gamma \preceq \sigma \wedge \forall \alpha \in \Sigma^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \models P))$ .

Intuitively, this says that an  $\omega$ -sequence satisfies a finitary property if all except a finite number of the finite prefixes satisfy the property.

**Lemma 5.4.** Definition 5.3 is equivalent to Definition 5.1 for FS properties and Definition 5.2 for EP properties.

*Proof.* For an FS property  $P$ , Theorem 3.2 says that  $P$  is prefix-closed, and it follows that for any  $\omega$ -sequence  $\sigma$  over  $\Sigma$

$$\forall \alpha \in \Sigma^* : (\alpha \preceq \sigma \Rightarrow \alpha \in P) \text{ iff } \exists \gamma \in \Sigma^* : (\gamma \preceq \sigma \wedge \forall \alpha \in \Sigma^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \in P)).$$

Namely, Definitions 5.1 and 5.3 are equivalent for FS properties.

For an EP property  $P$ , Definition 3.3 indicates that if a prefix satisfies  $P$ , then all its extensions satisfy  $P$  too. Therefore, for any  $\omega$ -sequence  $\sigma$  over  $\Sigma$

$$\exists \alpha \in \Sigma^* : (\alpha \preceq \sigma \wedge \alpha \in P) \text{ iff } \exists \gamma \in \Sigma^* : (\gamma \preceq \sigma \wedge \forall \alpha \in \Sigma^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \in P)).$$

Subsequently, Definitions 5.2 and 5.3 are equivalent for EP properties.  $\square$

## 6 Model Checking

Model checking [4] is a verification technique that involves constructing a finite model of a system and checking if a desired property holds in that model. In this section, we study model checking finitary properties specified in DQRDC.

For a system  $S$ ,  $\mathcal{H}[[S]]$  denotes the set of all behaviors, i.e.,  $\omega$ -sequences of system states, of  $S$ , and  $S$  satisfies a DQRDC formula  $\phi$  iff  $\forall \sigma \in \mathcal{H}[[S]] : \sigma \models_i \phi$ , i.e.

$$\forall \sigma \in \mathcal{H}[[S]] : (\exists \gamma \in Obs^* : (\gamma \preceq \sigma \wedge \forall \alpha \in Obs^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \models \phi))).$$

Subsequently,  $S$  does not satisfy  $\phi$  iff

$$\exists \sigma \in \mathcal{H}[[S]] : (\forall \gamma \in Obs^* : (\gamma \preceq \sigma \Rightarrow \exists \alpha \in Obs^* : (\gamma \preceq \alpha \preceq \sigma \wedge \alpha \not\models \phi))).$$

We can construct a finite automaton  $FA(\neg\phi)$ , which accepts precisely all the finite sequences not satisfying  $\phi$ , thus the above condition can be rewritten as

$$\exists \sigma \in \mathcal{H}[[S]] : (\forall \gamma \in Obs^* : (\gamma \preceq \sigma \Rightarrow \exists \alpha \in Obs^* : (\gamma \preceq \alpha \preceq \sigma \wedge \alpha \in L(FA(\neg\phi)))).$$

When  $FA(\neg\phi)$  is deterministic, referred to as  $DFA(\neg\phi)$ , we shall prove this condition is equivalent to

$$\exists \sigma \in \mathcal{H}[[S]] : \sigma \in L(DBA(\neg\phi)),$$

where  $DBA(\neg\phi)$  is the deterministic Büchi automaton [1] obtained by taking final states of  $DFA(\neg\phi)$  as accepting states.

Therefore, to determine whether  $S$  satisfies  $\phi$ , it suffices to check whether  $\mathcal{H}[[S]] \cap L(DBA(\neg\phi)) = \emptyset$ . If the intersection is empty,  $S$  satisfies  $\phi$ , and if not,  $S$  does not satisfy  $\phi$ .

**Lemma 6.1.** *For an  $\omega$ -sequence  $\sigma$  and a deterministic finite automaton  $DFA$ , let  $DBA$  be the deterministic Büchi automaton obtained by taking all the final states of  $DFA$  as accepting states, then*

$$\forall \alpha \in Obs^* : \alpha \preceq \sigma : (\exists \gamma \in Obs^* : \alpha \preceq \gamma \preceq \sigma : \alpha \in L(DFA)) \text{ iff } \sigma \in L(DBA).$$

*Proof.* For the if case, assume  $\sigma \in L(DBA)$ , then by the definition of Büchi automaton, the accepting states are passed infinitely many times. It follows that any prefix of  $\sigma$  has an (in fact, infinitely many) finite extension that is a prefix of  $\sigma$  and goes to an accepting state. Since the accepting state of the Büchi automaton is a final state of the finite automaton, the extension is a word of the finite automaton.

For the only if case, from the assumption, there exists an infinite series of finite sequences  $\alpha_1, \alpha_2, \dots$ , with  $\alpha_1 \preceq \alpha_2 \preceq \dots \preceq \sigma$ ,  $\alpha_i \neq \alpha_{i+1}$  and  $\alpha_i \in L(DFA)$  for all  $i \geq 1$ . Obviously,  $\alpha_1$  ends in a final state. Because the automaton is deterministic,  $\alpha_2$  will first go through the same states as  $\alpha_1$ , in particular, the first final state, and then ends in a (possibly different) final state. Therefore,  $\alpha_2$  goes through the final states which are the accepting states of the Büchi automaton at least twice. By induction, we know that  $\alpha_i$  goes through the accepting states of the Büchi automaton at least  $i$  times. Since there are infinite many such  $\alpha_i$ 's, some accepting states are gone through infinite number of times by  $\sigma$ , and therefore  $\sigma \in L(DBA)$ .  $\square$

Note in the above proof, the assumption that the automaton is deterministic is important in the only if part. However, this is not necessary for checking FS properties. If  $\phi$  is an FS property, in  $FA(\neg\phi)$ , any input symbol will lead a final state to another final state. Informally this is because for an FS property, if something “bad” has happened in a behavior, any extension of it is still a bad behavior. Therefore, if a prefix of  $\sigma$  has reached a final state, any further input

will lead to a final state, and subsequently  $\sigma$  is accepted by the corresponding Büchi automaton. Since determinization can be expensive, it is in general more efficient to use non-deterministic automata for model checking FS properties.

Spin [7] is a widely distributed software system for on-the-fly model checking. In Spin, the system is modelled using Promela, its input language, as a set of interacting processes. The negation of the desired property is encoded in a temporal claim, which also takes the form of a Promela program. Each process is translated into an automaton, and the global behavior of the system is obtained by computing the asynchronous product of these automata. The temporal claim is translated to a Büchi automaton. The synchronous product of this automaton and the automaton representing the global behavior of the system is computed. If the language accepted by the resulting Büchi automaton is empty, no behavior of the system satisfies the negation of the desired property, i.e. the system satisfies the property. Otherwise, the language contains some system behaviors violating the desired property.

We have implemented the algorithm which takes a DQRDC formula as input and generates the temporal claim in Promela. Therefore, it has the same goal as the LTL to Promela translation program in Spin. This allows us to model check properties studied in this paper when they are expressed as DQRDC formulas using Spin. Our implementation makes use of the C++ package for finite state machine, Grail+, developed by researchers at Department of Computer Science, University of Western Ontario, Canada. The compiler construction program Parser Generator from Bumble-Bee Software Ltd. is also used.

## 7 Case Study

In this section, we study model checking of two well known algorithms: Peterson's mutual exclusion algorithm and Peterson's leader election algorithm.

### 7.1 Peterson's Mutual Exclusion Algorithm

In Fig. 3, Peterson's mutual exclusion algorithm for the case of two processes is modeled in Promela.

```

1  #define true 1
2  #define false 0
3  #define aturn 1
4  #define bturn 0
5
6  bool areq, breq, turn;
7  bool ain;
8  bool bin;
9
10 proctype a()
11 {
12     atomic{ areq=true;
13             turn=bturn; }
14     (breq==false || turn==aturn);
15     ain=true;
16     /* critical section cs */
17     /* assert(bin==false); */
18     ain=false;
19     areq=false;
20 }
21
22 proctype b()

```

```

23 {
24     atomic{ breq=true;
25         turn=aturn; }
26     (areq==false || turn==bturn);
27     bin=true;
28     /* critical section */
29     /* assert(ain==false); */
30     bin=false;
31     breq=false;
32 }
33
34 init
35 {
36     run a(); run b();
37 }

```

Figure 3: Peterson's mutual exclusion algorithm

The algorithm uses three global variables `turn`, `areq` and `breq`. The statements on lines 14 and 26, which can only be executed if the boolean equations in them evaluate to true, synchronize the processes. If there is only one process that applies to enter the the critical section, the entry is immediate. Variable `turn` is used to decide which process to enter the critical section when both processes have requested. Variable `ain(bin)` is used here to facilitate property specification and it is set `true` iff process `a(b)` is in the critical section.

The essential property of a mutual exclusion algorithm is naturally mutual exclusion, which can be easily specified in DQRDC as

$$Req_1 \hat{=} \square(l > 1 \Rightarrow \llbracket \neg(ain \wedge bin) \rrbracket)$$

The generated corresponding Büchi automaton in Promela is as shown in Fig. 4.

```

1 never {
2 T0_init:
3     if
4     :: goto T1
5     fi;
6 T1:
7     if
8     :: ((true))->goto T1
9     :: ((!ain || !bin))->goto T2
10    :: ((ain && bin))->goto accept_5
11    fi;
12 T2:
13     if
14     :: ((!ain || !bin))->goto T2
15     :: ((ain && bin))->goto accept_5
16     fi;
17 accept_5:
18     if
19     :: ((true))->goto accept_5
20     fi;
21 }

```

Figure 4: Temporal claim for  $Req_1$ 

The mutual exclusion property is simple and can be expressed using assertions (as shown on lines 17 and 29) or a simple monitor process. Next we consider a more involved property,

*if one process has requested to enter its critical section when the other process is in the critical section, then the first process must have entered the critical section before the second one enters the critical section for another time.*

This property can be expressed as follows w.l.o.g., taking the mutual exclusion property into consideration:

$$Req_2 \hat{=} \square(\llbracket areq \wedge bin \rrbracket; \llbracket \neg bin \rrbracket; \llbracket bin \rrbracket \Rightarrow \diamond \llbracket ain \rrbracket)$$

The generated Büchi automaton in Promela is shown in Fig. 5.

```

1 never {
2   T0_init:
3   if
4   :: goto T0
5   fi;
6   T0:
7   if
8   :: ((true)) -> goto T0
9   :: ((areq && bin && !ain)) -> goto T2
10  fi;
11  T2:
12  if
13  :: ((!bin && !ain)) -> goto T6
14  :: ((areq && bin && !ain)) -> goto T2
15  fi;
16  T6:
17  if
18  :: ((!bin && !ain)) -> goto T6
19  :: ((bin && !ain)) -> goto accept_14
20  fi;
21  accept_14:
22  if
23  :: ((true)) -> goto accept_19
24  :: ((bin && !ain)) -> goto accept_14
25  fi;
26  accept_19:
27  if
28  :: ((true)) -> goto accept_19
29  fi;
30 }

```

Figure 5: Temporal claim for  $Req_2$

$Req_1$  and  $Req_2$  in this case study are both FS properties.

## 7.2 Peterson’s Leader Election Algorithm

Peterson proposed an algorithm for leader election in an unidirectional ring in [15]. This algorithm elects a certain process as the leader by comparing the temporal identifiers of the processes participated in the election. Both informal and formal descriptions of this algorithm can be found in [9]. Here we will only describe how it works briefly. Suppose messages can only be sent clockwise along the ring.

1. Initially, all processes are in “active” mode and the temporal identifier (abbrev. tID) of each process is the identifier of itself. At the beginning, each process sends its tID two steps clockwise, so each process receives the tIDs of its successor and its next-to-last successor).
2. Each process P makes a decision based on its current tID and the tIDs it has received. If P’s immediate neighbor’s tID is the greatest of the three, then P remains “active” and adopts its neighbor’s tID as its tID. Otherwise, P becomes a “relay”.
3. Each active process sends its tID to its next two active successors clockwise around the ring. Relay processes simply pass along each message they receive.
4. Steps 2 and 3 are repeated, each time with less active processes left, until a process finds that its own immediate successor is itself, in which case it is the only active process left and it declares itself the leader.

In Fig. 6, we model the algorithm in Promela, and to facilitate specifying properties, we add a variable NL to record the number of leaders elected.

```

1 #define N 4 /* nr of processes */
2 #define L 3 /* size of buffer */
3
4 mtype = { elect, report };
5 chan q[N] = [L] of { mtype, byte};
6 byte NL = 0;
7
8 proctype process (chan in, out; byte myid)
9 { bit Active = 1, know_winner = 0;
10  byte uid1 = myid, uid2 = 0, uid3=0, uidt;
11  xr in;
12  xs out;
13
14  out!elect(uid1);
15  do
16  ::Active->
17  if
18  :: uid2 == 0->in?elect(uid2);
19  if
20  ::uid2 == uid1-> Active = 0;
21  NL++;
22  out!report, myid;
23  ::else->out!elect, uid2;
24  fi
25  ::else ->
26  if
27  ::uid3==0->in?elect(uid3);
28  ::else ->
29  if
30  ::uid2>uid1 && uid2>uid3 ->
31  uid1=uid2;
32  uid2=0;
33  uid3=0;
34  out!elect, uid1;
35  ::else->Active = 0;
36  fi
37  fi
38  fi
39  ::!Active ->
40  if
41  ::in?elect(uidt)->out!elect,uidt;
42  ::in?report(uidt) ->
43  if
44  ::myid != uidt ->
45  out!report, uidt;
46  break;
47  ::else->break;
48  fi
49  fi
50  od
51  do
52  ::skip; /* other processing */
53  od
54  }
55
56 init {
57 atomic { run process (q[0], q[1], 1);
58          run process (q[1], q[2], 3);
59          run process (q[2], q[3], 2);
60          run process (q[3], q[0], 4);
61        }
62 }

```

Figure 6: Peterson's algorithm for leader election.

The algorithm should guarantee that a leader is eventually elected, and this can be expressed as follows

$$Req_3 \hat{=} \diamond \llbracket NL == 1 \rrbracket$$

It is obviously an EP property. The corresponding temporal claim is depicted in Fig. 7.

```

1 never {
2 T0_init:
3 if
4 :: goto accept_0
5 fi;
6 accept_0:
7 if
8 :: ((!(NL==1))) -> goto accept_0
9 fi;
10 }
11

```

Figure 7: Temporal claim for  $Req_3$ 

Taking the temporal claim and the system model as input, the Spin model checker shows that the property is satisfied.

## 8 Discussion

In this paper, we have studied the automatic verification of reactive systems against interval logic specifications. Although various interval logics have been developed for some time, there is little work with similar goal, which is somewhat unusual in view of extensive research regarding LTL. In fact, there have been no clear definitions about what it means for a reactive system to satisfy properties in interval logics. We have considered two classes of common properties and for them defined satisfaction relations that have natural meanings. The two satisfaction relations are unified by a more general one for which model checking support has been provided. We are not clear if the general relation is meaningful for properties outside the two classes and their simple combinations.

In this paper, we have studied the automatic verification of reactive systems against interval logic specifications. Compared to extensive research on model checking properties expressed in LTL, there is little work with the similar goal as ours. Our work is useful to people who prefer to use interval logics. Furthermore, it can be used even by people who primarily use LTL. In fact, both LTL and interval logics can be used together in specifying properties of a system. For example, in Peterson's mutual exclusion algorithm case study, one can use the interval logic to specify  $Req_2$  as we have done, since this property may not be easy to express in LTL, but use LTL to specify other properties for which LTL is better or just as good.

Although the algorithm converting the interval logic formulas to finite automata is simple and well-known, there have been few implementations. The early BDD-based implementation by Skakkebæk and Sestoft is integrated into a proof assistant [17] developed on an old version of PVS which has not been supported for years. The more recent implementation by Pandya [13] uses the Mona [10] system by mapping the interval logic to the language of Mona, a monadic second-order logic. Although the implementation of Mona is highly efficient, its sophistication also makes it difficult to handle. What we really need is a package on finite automata, and with Grail+, we have a better control of the implementation. For example, we can control whether to determinize the automaton, which is an expensive operation.

Pandya has also studied model checking reactive systems with interval logics. In his approach, the interval logic does not stand alone and is combined into other logics, for example, into CTL in [14]. Pandya has apparently considered combining the interval logic with LTL, and using Spin to check properties expressed in the new logic. The resulting logics are substantially different from existing ones and considerable work may be needed before they are accepted.

## References

- [1] J.R. Buchi. *On a Decision Method in Restricted Second Order Arithmetic*. In Nagel et al., editor, *Logic, Methodology and Philosophy of Science*. Stanford Univ. Press, 1960.
- [2] Bowen Alpern and Fred B. Schneider. *Recognizing Safety and Liveness*. TR 86-727,

- 1986.
- [3] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, 1985.
  - [4] Edmund M. Clarke, Orna Grumberg and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
  - [5] The Grail+ Project. Department of Computer Science, University of Western Ontario, Canada. <http://www.csd.uwo.ca/research/grail/>
  - [6] Michael R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9:283-330, 1997.
  - [7] G. Holzmann. The SPIN Model Checker. *IEEE Trans. on Software Engineering*, 23:279-295, 1997
  - [8] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125-143, March 1977.
  - [9] Nancy A. Lynch. *Distributed Algorithm*. Morgan Kauffman Publisher, Inc. San Francisco, California, 1996.
  - [10] J.G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic Second-order Logic in practice. In Proc. of TACAS'95, LNCS 1019, Springer-Verlag, 1996.
  - [11] B. Moszkowski. A Temporal Logic for Multilevel Reasoning about Hardware. *IEEE Computer*, 18(2):10-19, 1985.
  - [12] B. Moszkowski. Compositional reasoning about projected and infinite time. Proc. of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95), IEEE Computer Society Press, p238–245, 1995.
  - [13] Paritosh K. Pandya. Specifying and Deciding Quantified Discrete-Time Duration Calculus Formulae using DCVALID. Tcs00-pkp-1, Tata Institute of Fundamental Research, Homi Bhabha Road, Colaba, Mumbai, May 2000.
  - [14] P.K. Pandya. Model checking CTL[DC]. In Proc. of TACAS 2001, Genova, Italy. LNCS 2031, Springer-Verlag, 2001.
  - [15] G. Peterson. An  $O(n \log n)$  Algorithm for the Circular Extrema Problem. *ACM Trans. on Prog. Lang. and Systems*, 4(4):758-762, 1982.
  - [16] Y. S. Ramakrishna, P. M. Melliar-Smith, Louise E. Moser, Laura K. Dillon, G. Kuttys. Interval Logics and Their Decision Procedures, Part I: An Interval Logic. *Theoretical Computer Science* 166(1&2): 1-47, Elsevier (1996).
  - [17] J.U. Skakkebaek. A Verification Assistant for Real-time Logic. PhD. Thesis, Department of Computer Science, Technical University of Denmark, 1994.

- [18] Wang Hanpin and Xu Qiwen. Completeness of temporal logics over infinite intervals. Technical Report 158, UNU/IIST, Macau. 1999. Accepted by Applied Discrete Mathematics, Elsevier.
- [19] P. Wolper. Constructing automata from temporal logic formula: a tutorial. First EEF/Euro Summer School on Trends in Computer Science, LNCS 2090, pp. 261-277, Springer-Verlag, 2001.
- [20] Zhou Chaochen, C.A.R. Hoare and A.P. Ravn. A Calculus of Duration. *Information Processing Letters*, 40, 5, pp. 269-276, 1991.
- [21] Zhou Chaochen, Dang Van Hung and Li Xiaoshan. A Duration Calculus with Infinite Intervals, *UNU/IIST Report No. 40, LNCS 965*, pp. 16-41, February, 1995.