Serdica J. Computing **7** (2013), No 1, 1–12

**Serdica**
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

# APPROXIMATE MODEL CHECKING OF REAL-TIME SYSTEMS FOR LINEAR DURATION INVARIANTS

Changil Choe , Hyong-Chol O, Song Han

Abstract. Real-time systems are usually modelled with timed automata and real-time requirements relating to the state durations of the system are often specifiable using Linear Duration Invariants, which is a decidable subclass of Duration Calculus formulas. Various algorithms have been developed to check timed automata or real-time automata for linear duration invariants, but each needs complicated preprocessing and exponential calculation. To the best of our knowledge, these algorithms have not been implemented.

In this paper, we present an approximate model checking technique based on a genetic algorithm to check real-time automata for linear durration invariants in reasonable times. Genetic algorithm is a good optimization method when a problem needs massive computation and it works particularly well in our case because the fitness function which is derived from the linear duration invariant is linear.

**1. Introduction.** *Model checking* (or *property checking*) in computer science is the following problem: Given a model of a *system*, exhaustively and

*ACM Computing Classification System* (1998): D.2.4, C.3.

*Key words:* Approximate Model Checking, Verification, Real-time System, Linear Duration Invariant, Genetic Algorithm.

automatically check whether this model meets a given *specification*. Here the systems can be hardware or software systems. The specifications can contain safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking is a methodology for automatically verifying correctness properties of finite-state systems. In order to solve such a problem algorithmically, we require that both the system model and the specification are formulated in some precise mathematical language. It is often formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. When two descriptions are not functionally equivalent, we use property checking. The implementation or design is regarded a model of the circuit whereas the specifications are properties that the model must satisfy [15].

We often apply model checking methods to hardware designs. For software, because of undecidability the approach cannot be fully algorithmic. We often give the structure as a source code description in an industrial hardware description language or a special-purpose language. Such a program corresponds to a finite state machine. A *finite state machine* is a directed graph with nodes and edges. A set of atomic propositions is associated with each node. The nodes represent states of a system whereas the edges represent possible transitions that may change the state, while the atomic propositions represent the basic properties that hold at a point of execution.

Formally, the problem can be stated as follows:

*given a desired property, expressed as a temporal logic formula p, and a structure M with initial state s, decide if* $M, s| = p$.

If $M$ is finite, as it is in hardware, model checking reduces to a graph search. Model checking tools were initially developed to reason about the logical correctness of *discrete state* systems, but have since been extended to deal with *real-time systems*.

*Duration Calculus* (DC) is an *interval logic* for real-time systems. It was originally introduced in [1] and then developed further in many other works. (See [2].) DC is mainly useful at the requirements level of the software development process for real-time systems. Validity in DC is generally undecidable, and depending on the choice of authors who sometimes use DC for the *subset* of DC that is captured by their particular technical results, model-checking may be deemed decidable or not. See e.g., [3] for an early collection of results, and the more recent related work [18, 19, 20] by Goranko, Montanari, Schiavico, Dario della Monica et al. for related work on ITL.

LDI (Linear duration invariant) which is a subclass of chop-free formulas is very useful to specify duration constraints of states of real-time systems [4]. For

this reason, many researchers devoted their works to develop efficient algorithm checking timed automata or real-time automata for LDIs under some restrictions on automata or LDIs in most cases [4∼11].

In our view, these algorithms can be classified into three methods. The first is to solve linear or integer programming problems generated from an automaton and LDI by applying a regular expression technique [4]. The second is to check an integer timed region graph generated from an original automaton and LDI using an exhaustive search technique [8]. The third is to check an untimed automaton for a CTL formula using a CTL model checking tool, which are derived from the original automaton and LDI respectively [9].

These methods are valuable because they allow complete verification of the satisfaction relation between automata and LDIs. However, from a practical view, all these algorithms need complicated preprocessing and mainly for this reason the total execution time is exponential.

On the other hand, a lot of work has been done to develop a model checking tool for DC formulas [12-14]. Nevertheless, compared with other temporal logic model checking, DC model checking is still not widely used by engineers though several model checking tools for some decidable class of DC formulas are available.

It seems necessary to consider the problem at a different view. We note the fact that computationally efficient model checking algorithm for LDI has not been developed yet, although many researchers tried to develop it since DC was first introduced 20 years ago.

There is another problem concerning DC model checking in our view. DC model checking only deals with a small decidable class of formulas. However, many undecidable classes of formulas of DC, e.g., chop formulas, are really useful to specify various patterns of real-time requirements. We think that approximate information on whether the system model satisfies the requirement specification with undecidable classes of DC formula or not may be valuable for the verification of real-time systems.

This work aims to develop an approximate model checking technique for DC formulas, which is quite different from normal model checking. Approximate model checking, which was first introduced in temporal logic based verification, is achived by generating a large number of random paths through the model, evaluating the given properties on each run and using this information to get an approximately correct result. It is particularly useful when normal model checking is infeasible because of the very large amount of computation.

We use the genetic algorithm as a mathematical basis of our technique.

The genetic algorithm is very appropriate for the verification of real-time systems requiring exponential computation for model checking as it is a good approach to searching a near-optimal solution in the case when the problem is so complicated that seeking an optimal solution is practically impossible.

In this paper, we don't try to present a complete approximate model checking technique covering arbitrary timed automata and DC formulas. Instead, we focus on showing the main idea of our technique clearly. This is the reason we consider the problem of verifying real-time automata for LDIs, which is a typical case of DC model checking.

The remainder of this paper is organized as follows. In section 2 we construct a framework for approximate model checking of real-time automata against LDIs. In Section 3, we present our genetic algorithm approach for approximate model checking of real-time automata against LDIs based on the framework constructed in Section 2. A conclusion and a description of future work constitute Section 4.

## 2. A Framework for Approximate Model Checking of Real-time Automata against LDIs.

In this Section, we define a satisfaction relation between real-time automata and LDIs as a framework for checking real-time automata aginst LDIs using a genetic algorithm.

**Definition 2.1** (*Real-time Automata*). *A* ***real-time automaton*** $M$ *is a triple* $M = (S,\ T, L)$ *consisting of*

- *a finite set* $S$ *of states,*
- *a transition relation* $T \subseteq S \times I \times S$*,*
- *a labeling function* $L : S \to 2^{AP}$ *assigning a set of atomic propositions to each state* $s \in S$*.*

Here, $I$ is the set of closed intervals $[a, b]$ or semi-infinite intervals $[a, \infty)$ on $R^+$, the set of nonnegative real numbers. For convenience, we simply denote these intervals by $[*, *]$. Every state of a real-time automaton is both an initial state and an accepting state. $AP$ is the set of atomic propositions which is differently decided according to the system. Real-time Automata have one clock which is reset by every transition.

**Example 2.2** (*Gas burner*). A gas burner is a device to generate a flame to heat up products using a gaseous fuel [1]. If the flame fails to be on with the gas valve open, gas leaks. A sensor should detect a gas leak and close the gas valve within one second. Then the gas valve should not be open within 30 seconds

to prevent accumulation of gas. Gas may leak again without the flame being on at any time after the valve is open. Fig. 2.3 shows a real-time automaton model of a gas burner. *Leak* and *NLeak* are used to denote atoms of the gas burner.

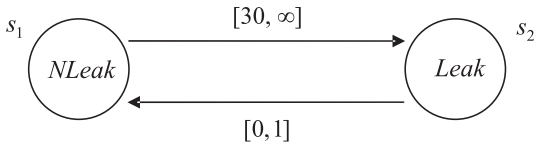


Fig. 2.3. Real-time automaton model of gas burner

For a transition $\rho = (s, [a, b], s')$ of $M$, notations $\overleftarrow{\rho} = s$ and $\overrightarrow{\rho} = s'$ are used. For a sequence $\rho_1\rho_2\cdots\rho_m$, $(\rho_1, t_1)(\rho_2, t_2)\cdots(\rho_m, t_m)$ is called a *time-stamped sequence*, where $\rho_i = (s_i, [a_i, b_i], s_i')$ and $t_i \in [a_i, b_i]$ for all $i\,(1 \le i \le m)$. $Seq$ and $TSeq$ are used to denote a sequence and time-stamped sequence respectively.

If a sequence $\rho_1\rho_2\cdots\rho_m$ satisfies $\overrightarrow{\rho_i} = \overleftarrow{\rho_{i+1}}$ for all $i\,(1 \le i \le m-1)$, it is called a *behavior* and denoted by $Beh = \rho_1\rho_2\cdots\rho_m$. If a *time-stamped sequence* $(\rho_1, t_1)(\rho_2, t_2)\cdots(\rho_m, t_m)$ satisfies $\overrightarrow{\rho_i} = \overleftarrow{\rho_{i+1}}$ for all $i\,(1 \le i \le m-1)$, it is called a *time-stamped behavior* and denoted by $TBeh = (\rho_1, t_1)(\rho_2, t_2)\cdots(\rho_m, t_m)$.

**Definition 2.4** (*Linear Duration Invariant*, shortly LDI)**.** *Let $M$ be a real-time automaton. A DC formula of the form*

$$A \le \ell \le B \rightarrow \textstyle\sum_{i=1}^{n} c_i \cdot \int P_i \le C$$

*is called a* ***linear duration invariant*** *over $M$* [4].

*Here, each $P_i$ $(1 \le i \le n)$ is an atomic proposition of $AP$ of $M$. $A$ and $B$ are nonnegative real numbers, $B$ could be $\infty$, $c_i$ $(1 \le i \le m)$ and $C$ are real numbers.*

LDI formalize a class of real-time properties that if the length of observation time interval over $M$ is between $A$ and $B$, the total duration $\int P_i$ of $P_i$-states for each $i$ $(1 \le i \le n)$ on this interval satisfies the linear constraint $\sum_{i=1}^{n} c_i \cdot \int P_i \le C$. Here, $P_i$-state means the state in which $P_i$ is labelled.

**Example 2.5** (*Real-time requirement of gas burner*)**.** A fan is installed to protect the self-ignition of accumulated gas leakage for a gas burner. However, frequent gas leak may cause self-ignition as the capacity of the fan is limited. A desirable real-time requirement of the gas burner is as follows.

"The proportion of total gas leak time is no more than one twentieth of the elapsed time, if the system is observed for more than one minute".

This real-time requirement can be specified using LDI as follows.

$$\ell \geq 60 \rightarrow 19 \cdot \int Leak - \int NLeak \leq 0$$

Here, $19 \cdot \int Leak - \int NLeak \leq 0$ is derived from $\int Leak \leq (1/20) \cdot \ell$ by substituting $\ell = \int Leak + \int NLeak$.

Let $M$ be a real-time automaton and $TBEH$ be the set of all $TBeh = (\rho_1, t_1)(\rho_2, t_2) \cdots (\rho_m, t_m)$ of $M$. The function $L : TBEH \rightarrow R^+$ is defined as

$$L(TBeh) = \sum_{j=1}^{m} t_j.$$

For each atomic proposition $P(\in AP)$ of $M$, the function $\int P : TBEH \rightarrow R^+$ is defined as

$$\int P(TBeh) = \sum_{j=1}^{m} \left\{ \begin{array}{ll} t_j & P \in L(\overleftarrow{\rho_j}) \\ 0 & \text{otherwise} \end{array} \right\}.$$

$\int P(TBeh)$ calculates the total duration time of $P$-states on $TBeh$. For example, if $TBeh = (\rho_1, 3.1)(\rho_2, 2.0)(\rho_3, 1.5)$ and $P$ is labelled in states $\overleftarrow{\rho_1}$ and $\overleftarrow{\rho_3}$, then $\int P(TBeh) = 3.1 + 1.5 = 4.6$.

Let $D$ be a linear duration invariant $A \leq \ell \leq B \rightarrow \sum_{i=1}^{n} c_i \cdot \int P_i \leq C$ over $M$. The function $LF : TBEH \rightarrow R^+$ is defined as

$$LF(TBeh) = \sum_{i=1}^{n} c_i \cdot \int P_i(TBeh)$$

$LF$ is the fuction calculating the value of the linear term $\sum_{i=1}^{n} c_i \cdot \int P_i$ of $D$ over $TBeh$.

Based on the functions $L$ and $LF$, the *satisfaction relation* between a real-time automaton $M$and $D$ (an LDI) for approximate model checking is defined as follows.

**Definition 2.6** (*Satisfaction relation between a real-time automaton and a LDI*)**.** *Let $D$ be an LDI. $D$ is said to be* ***satisfied by*** *a real-time automaton $M$, denoted by $M| = D$, $iff$ $A \leq L(TBeh) \leq B$ implies $LF(TBeh) \leq C$ for all $TBeh$ of $M$.*

**Example 2.7** (*satisfaction realtion between a gas burner model and its requirement specification*)**.** For a real-time automaton medel $M$ of the gas burner

of example 2.2 and its real-time requirement specification $D$ of example 2.5, $M| = D$ means that the gas burner satisfies its real-time requirement.

**3. A genetic algorithm approach for checking real-time automata against LDIs.** Mathematically, checking a real-time automaton $M$ against $D$ (an LDI) is to solve the following optimization problem.

*Find the maximum value of the function* $LF$ *over* $\{TBeh|A \leq L(TBeh) \leq B\}$.

Solving this problem, if the maximum value of $LF$ is smaller than or equal to $C$, then the real-time automaton $M$ satisfies $D$.

Unlike the previous methods, we want to check the model approximately using a genetic algorithm familiar to engineers. The genetic algorithm works very well especially when the fitness function is linear and the problem of checking real-time automata for LDIs just conforms to this case. In this section, we describe our genetic algorithm technique that takes into consideration the characteristics of the real-time automaton and the LDI.

In the first subsection, for readers' convenience, we provide some elementary knowledge about genetic algorithms.

**3.1. Preliminaries on Genetic Algorithms.** Genetic algorithms are widely applied in computational science and other fields.

A *genetic algorithm* is a heuristic search that imitates the natural evolution process, which is used to generate useful solutions to optimization problems according to a routine [16]. A genetic algorithm is a kind of evolutionary algorithm. It gives a solution to optimization problems using techniques such as inheritance, mutation, selection and crossover.

Candidate solutions to an optimization problem in a genetic algorithm are often called *creatures*, *individuals* or *phenotypes* and the population of candidate solutions evolves toward better solutions. Each candidat solution has its *chromosomes* or *genotype*, which is a set of properties. Chromosomes can mutate and change. Often, solutions are expressed in binary as strings of 0s and 1s. [17].

A population of randomly given creatures is a starting point of the evolution, which is often an iterated process. The population of the creatures in every iteration is called a *generation*, in which we estimate the fitness of every individual in the population. In order to calculate the *fitness*, we often make an optimization problem and calculate the value of the objective function of it. We randomly select the more fit individuals from the current population. And

then, we recombine and possibly randomly mutate each individual's genome to form a new generation. We use the population of the new generation in the next iteration. In general, we end the algorithm when a satisfactory fitness level for the population is reached.

Generally, in order to solve some problem with a genetic algorithm, at first we must give a *genetic representation* of the solution domain and a *fitness function* to evaluate the solution domain. A standard representation of each candidate solution is as an array of bits [17]. Once the genetic representation and the fitness function are defined, a genetic algorithm proceeds to initialize a population of solutions and then to improve it through repeated application of the mutation, crossover, inversion and selection operators.

**3.2. Fitness function and genetic operations.** Let $M = (S, T, L)$ be a real-time automaton and let $D$ be an LDI

$$D := A \leq \ell \leq B \rightarrow \sum_{i=1}^{n} c_i \cdot \int P_i \leq C.$$

For each transition $\rho = (s, [a, b], s')$ of $M$, the time-stamped transition $(\rho, t)$ where $t \in [a, b]$ is a *gene*. Then we use the individual $(\rho_1, t_1)(\rho_2, t_2) \cdots (\rho_m, t_m)$, i.e. time-stamped behavior, directly as *chromosome*. That is, we use *variable length encoding*.

• ***Fitness function.*** $LF$ defined in Section 2 is used as the fitness function. $LF$ calculates the value of linear constraint $\sum_{i=1}^{n} c_i \cdot \int P_i$ of $D$on each individual $(\rho_1, t_1)(\rho_2, t_2) \cdots (\rho_m, t_m)$.

• ***Initialization.*** The set of behaviors of $M$ can be expressed as the union of regular expressions consisting of concatenation and Kleene star on the alphabet $T$ of $M$. For example, the set of behaviors of the gas burner of Example 2.2 can be expressed as follows.

$$\rho_1(\rho_2\rho_1)^* \cup \rho_2(\rho_1\rho_2)^* \cup \rho_1(\rho_2\rho_1)^*\rho_2 \cup \rho_2(\rho_1\rho_2)^*\rho_1$$

Here, $\rho_1 = (s_1, [30, \infty], s_2)$ and $\rho_2 = (s_2, [0, 1], s_1)$.

This feature of the real-time automaton is useful when we generate the initial population and replace some individuals for the generation of the new population. It's better to generate individuals randomly but uniformly from each component of the union to expand the search space quickly and uniformly in the feasible set.

• ***Selection operation.*** Elitist preserving selection is used. That is, the best individuals of the current generation are retained in the next generation.

***Mutation operation.*** Mutation is realized by altering a selected gene $(\rho, t)$ with $(\rho, t')$ where $\rho = (s, [a, b], s')$ and $t, t' \in [a, b]$. Multi-point mutation can be used for relatively long individuals. Application of mutation operation expands the breadth of search space.

• ***Cut and splice operation.*** Cut and splice is applied to two parents having same gene. Swapping each suffix beyond the selected gene produces two childrens. Application of the cut and splice operation expands the depth of search space.

**3.3. The main procedure.** Based on the operation specified above, a genetic algorithm for checking a real-time automaton against a LDI is composed as follows.

Step 1

Generate the initial population $P(0)$ consisting of $N$ individuals satisfying $A \leq \ell(TBeh) \leq B$ using the initialization method described above.

Step 2

Evaluate the fitness of each individual. If $LF(TBeh) > C$ for a individual $TBeh$, stop the procedure and output $M| \not\models D$.

Step 3

Maintain the fittest individuals in the current population $P(n)$.

Step 4

Generate population $Q$ by applying the mutation operation and cut and splice operation to $P(n)$. Remove all individuals not satisfying $A \leq \ell(TBeh) \leq B$ from $Q$ and add new individuals satisfying $A \leq \ell(TBeh) \leq B$ as many as the number of removed individuals.

Step 5

Evaluate the fitness of all individuals of $Q$ and generate the new population $P(n+1)$ by replacing the least-fit individuals of $Q$ with the best-fit individuals of $P(n)$.

Step 6

Repeat step 2-5 until the best-fitness value of populations is settled or the maximum number of generation $n$ is reached.

**3.4. Experiments.** We applied the genetic algorithm described above to check the model of Example 2.2 against LDI $\ell \geq 60 \rightarrow 19 \cdot \int Leak - \int NLeak \leq 0$ of Example 2.5. The encoding and fitness function were designed according to the method of Subsection 3.1. The initial population is generated by choosing individuals randomly but uniformly in $\rho_1(\rho_2\rho_1)^*$, $\rho_2(\rho_1\rho_2)^*$, $\rho_1(\rho_2\rho_1)^*\rho_2$ and $\rho_2(\rho_1\rho_2)^*\rho_1$ respectively.

We executed our genetic algorithm 10 times by varying parameter $N$ between 40–80, $P_m$ between 0.1–0.3 and $P_d$ between 0.4–0.6, where $P_m$ was the probability of mutation and $P_d$ is the probability of cut and splice. The termination condition was $n$=100. The best fitness reached $-3$ or nearly -3 in each execution. From this, we could estimate the maximum value of $19 \cdot \int Leak - \int NLeak$ is $-3$ which was much smaller than $C$=0 and therefore definitely conclude that the gas burner model of Example 2.2 satisfies LDI $\ell \geq 60 \rightarrow 19 \cdot \int Leak - \int NLeak \leq 0$.

**4. Conclusion and future work.** While the DC model checking of real-time systems has been focussed on checking timed automata or real-time automata against LDIs for more than 20 years, an acceptable model checking algorithm is still not available as some researchers have noted in their papers. We changed the approach to the verification of real-time systems from normal model checking to approximate model checking by considering the structural feature of the automaton and the LDI.

Our approach is based on a genetic algorithm familiar to engineers, does not need any complicated preprocessing, and generates useful information making it possible to estimate the satisfaction relation between model and specification in reasonable times. In case that the maximum of $LF$ is different from $C$, it certainly demonstrates the same effect as normal model checking. The opposite case needs more executions of the algorithm to get enough information between the maximum of $LF$ and $C$.

In the future, we want to extend our approach to more general cases such as extremely complicated probabiliistic DC model checking. Especially we want to develop an approximate model checking technique for useful undecidable classes of DC formulas.

**Acknowledgement.** The authors profoundly thank Professor Peter Stanchev and the anonymous reviewers for their help and useful advice.

## REFERENCES

[1] Chaochen Z., C. A. R. Hoare, Anders P. Ravn.; calculus of durations. *Information Processing Letters*, **40** (1991), No **5**, 269–276.

[2] Chaochen Z., M. R. Hansen. Duration Calculus: A Formal Approach to Real-Time Systems. Monographs in Theoretical Computer Science, An EATCS Series, Springer-Verlag, 2004.

[3] Chaochen Z., M. R. Hansen, P. Sestoft. Decidability and Undecidability Results for Duration Calculus. In: Proceedings of the STACS '93. 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, LNCS, Vol. **665**, Springer- Verlag, Feb. 1993, 58–68.

[4] Chaochen Z., Z. Jingzhong, Y. Lu, L. Xiaoshan. Linear Duration Invariants. In: Formal Techniques in Real-Time and Fault-Tolerant systems, LNCS, Vol. **863**, 1994, 86–109.

[5] Xuandong L., D. Van Hung. Checking linear duration invariants by linear programming. In: Concurrency and Paralellism, Programming, Networking, and Security, LNCS, Vol. **1179**, Springer- Verlag, 1996, 321–332.

[6] Kesten Y., A. Pnueli, J. Sifakis, S. Yovine. Integration graphs: a class of decidable hybrid systems. In: Hybrid Systems, LNCS, Vol. **736**, Springer-Verlag, June 1992, 179–208.

[7] Changil C., D. Van Hung. On Verification of Linear Occurrence Properties of Real-time Systems. In: Electronic Notes on Theoretical Computer Science, Vol. **207**, 2008, 107–120.

[8] Thai P. H., D. Van Hung. Verifying Linear Duration Constraints of Timed Automata. In: Proceedings of the ICTAC 2004, LNCS, Vol. **3407**, Springer-Verlag, 2005, 295–309.

[9] Zhang M., D. Van Hung, Z. Liu. Verification of Linear Duration Invariants by Model Checking CTL Properties. In: Theoretical Aspects of Computing – ICTAC 2008, LNCS, Vol. **5160**, Springer-Verlag, 2008, 395–409.

[10] Jianhua Z., D. Van Hung. Checking Timed Automata for Some Discretisable Duration Properties. *Journal of Computer Science and Technology*, **15** (2000), No **5**, 423–429.

[11] Hansen M. R. Model-checking discrete duration calculus. *Formal Aspects of Computing*, **6** (1994), No 1, 826–846.

[12] Pandya P. K. Interval Duration Logic: Expressiveness and Decidability. *Electr. Notes Theor. Comput. Sci.,* **65** (2002), No 6, 254–272.

[13] Meyer R., J. Faber, A. Rybalchenko. Model Checking Duration Calculus: A Practical Approach. *Formal Aspects of Computing,* **20** (2008), 481–505.

[14] Fränzle M., M. R. Hansen. Deciding an Interval Logic with Accumulated Durations. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Vol. **4424**, Springer-Verlag, 2007, 201–215.

[15] Lam, W. K. Hardware Design Verification: Simulation and Formal Method-based Approaches. Prentice Hall Computer, 2008.

[16] Mitchell M. An Introduction to Genetic Algorithms. Cambridge, MA, MIT Press, 1996.

[17] Whitley D. A genetic algorithm tutorial. *Statistics and Computing*, **4** (1994), No 2, 65–85. DOI:10.1007/BF00175354.

[18] Monica D. D., V. Goranko, A. Montanari, G. Sciavicco. Interval Temporal Logics: A Journey. *Bulletin of the European Association for Theoretical Computer Science*, **105** (2011), 73–99.

[19] Goranko V., A. Montanari, G. Sciavicco. Road Map of Interval Temporal Logics and Duration Calculi. *Journal of Applied Non-Classical Logics*, **14** (2004), No 1-2, 9–54. DOI: 10.3166/jancl.14.9-54

[20] Goranko V. F., A. Montanari. Interval temporal logics. Lecture slides, Hamburg, ESSLLI, August 2008.

*Faculty of mathematics*
*Kim Il Sung University*
*D.P.R. Korea*
*e-mail:* ohyongchol@yahoo.com

*Received October 3, 2012*
*Final Accepted June 11, 2013*