Journal of Software Engineering and Applications, 2015, 8, 43-50
Published Online February 2015 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2015.82006

Timed-Automata Based Model-Checking of a Multi-Agent System: A Case Study

Nadeem Akhtar, Muhammad Nauman

Department of Computer Science and IT, The Islamia University of Bahawalpur, Baghdad-ul-Jadeed Campus, Bahawalpur, Pakistan
Email: nadeem.akhtar@iub.edu.pk

Received 17 May 2014; accepted 10 February 2015; published 12 February 2015

Abstract

A multi-agent based transport system is modeled by timed automata model extended with clock variables. The correctness properties of safety and liveness of this model are verified by timed automata based UPPAAL. Agents have a degree of control on their own actions, have their own threads of control, and under some circumstances they are also able to take decisions. Therefore they are autonomous. The multi-agent system is modeled as a network of timed automata based agents supported by clock variables. The representation of agent requirements based on mathematics is helpful in precise and unambiguous specifications, thereby ensuring correctness. This formal representation of requirements provides a way for logical reasoning about the artifacts produced. We can be systematic and precise in assessing correctness by rigorously specifying the functional requirements.

Keywords

Software Correctness, Formal Verification, Model Checking, Timed-Automata, Multi-Agent System, Timed Computation Tree Logic (TCTL)

1. Introduction

The use of formal methods to define the requirements, design and architecture of a multi-agent system results in precise and unambiguous specifications. These formal specifications provide the basis for systematic, mathematically-proven, well-defined, and unambiguous software development phases of analysis, design, and implementation. Multi-agent systems are distributed, decentralized, consisting of autonomous computing entities known as agents. Correctness is a mathematical property that establishes the equivalence between software and its specifications. Software systems analyzed, designed, and implemented by using agents to offer significant challenges in ensuring their correctness. One of the methods of ensuring the correctness of safety and liveness properties of these agent-based systems is to use formal model checking methods based on timed automata.

2. State of the Art

2.1. Formal Methods

The primary objective of a formal approach is precise and unambiguous specification. A representation of the requirements based on mathematics aids in precise specification of the software, thereby ensuring that the correctness, completeness, and unambiguous properties of the system are preserved [1]. The formal representation of software requirements provides a method for logical reasoning about the artifacts produced. This achieves more precision in the description and allows for a stronger design that satisfies the required properties. Formal methods offer the ability to rigorously prove system correctness, *i.e.* that specifications are consistent with the stated objective; that code is consistent with specification; and that code produces a desired result and none other. To overcome the complexity problems in multi-agent systems and get significant results with formal specifications, we must cope with complexity at each phase: requirements, architecture to design and implementation. We can prove the correctness of a multi-agent system by formalizing critical components in the multi-agent development life-cycle.

The most important reasons to use formal methods in software engineering are: rigorous analysis of system properties; property-preserving transformations; error-free implementation; high quality of each phase of the development process; firm foundation for the adaptation and evolution process; continuous correctness as multi-agent systems are concurrent and often have dynamic environments; formal specification and modeling of a multi-agent system architecture which can change at run-time (*i.e.* dynamic architecture); specification according to the functional and non-functional properties; property-preserving step-by-step transformations from abstract to concrete concepts and then stepwise refinement to implementation code; improved documentation and understanding of specifications.

2.2. Agents and Multi-Agent Systems

An agent is a computer system that is capable of autonomous actions on behalf of its user or owner [2]. Agents are coarse-grained computational systems, each making use of significant computational resources [3]. An agent is a software entity that is able to conduct information-related tasks without human supervision [4]. Intelligent agents (*i.e.* referred to as rational agents) are systems that accomplish their goals by acting rationally. Rational agents can use reasoning to make decisions about their goals. A rational agent is an autonomous computing entity that can accomplish tasks autonomously on the behalf of its user. It can also interact and collaborate with other agents to accomplish its goals. It can also refuse an order or an action that is called from another agent. Intelligent agents must show some degree of autonomy, social ability, and combine proactive and reactive behavior.

- Autonomous: agents have a degree of control on their own actions, they own their thread of control and under some circumstances, they are also able to take decisions;
- Proactive: agents do not only react in response to external events (*i.e.* a remote method call), but they also exhibit a goal-directed behavior and where appropriate are able to take initiative;
- Social: agents are able to and need to interact with other agents in order to accomplish their task and achieve the complete goal of the system [5].

The environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources [6]. A generic environment program has been defined by [7]. This simple program gives the agents precepts and receives back their actions; it then updates the state of the environment based on the actions of the agents and other dynamic processes in the environment that are not considered to be agents. Demazeau [8] considers four essential building blocks for agent systems: agents (*i.e.*, the processing entities), interactions (*i.e.*, the elements for structuring internal interactions between entities), organizations (*i.e.*, elements for structuring sets of entities within the multi-agent system), and finally the environment that is defined as the domain-dependent elements for structuring external interactions between entities. The environment is an independent component of the multi-agent system that has its own responsibilities. These responsibilities are not dependent on agents. It provides the medium for agents to exchange messages, and all agent interactions are done through it.

A multi-agent system is composed with autonomous entities (*i.e.* agents) that interact with one another. Multiple agents are necessary to solve a problem, especially when the problem involves distributed data, knowledge, or control [9]. A multi-agent system is a collection of several interacting agents in which each agent has incomplete information or capabilities for solving the problem [10]. In a multi-agent system, agents are autonomous. There is no global system control, data is decentralized, and communication is asynchronous.

2.3. Correctness: Safety and Liveness Properties

A program is functionally correct if it behaves according to its stated functional requirements. Correctness is a mathematical property that establishes the equivalence between software and its specifications [11]. We can be systematic and precise in assessing correctness by rigorously specifying the functional requirements. Software systems provide critical services to users, *i.e.* process control systems in nuclear power plants or in chemical industry, radiation machines in hospitals, transport systems such as cars, trains and airplanes. In these types of systems, correctness is of vital importance.

Safety and liveness properties are correctness properties. The safety property is an invariant which asserts that something bad never happens, that an acceptable state of affairs is maintained. Magee and Kramer [12] have defined safety property S = {a1, a2 … an} as a deterministic process that asserts that any trace including actions in the alphabet of S is accepted by S. ERROR conditions are like exceptions which state what is not required, as in complex systems we specify safety properties by directly stating what is required. The liveness property asserts that something good happens, which describes the states of a system that an agent must bring about given certain conditions. These properties play a vital role in system verification. Both safety and liveness properties are complementary to each other, safety alone or liveness alone is not sufficient to ensure system correctness. Progress is a type of liveness property. Progress P = {a1, a2 ... an} defines a property P which asserts that in an infinite execution of a target system, at least one of the actions (a1, a2 ... an) will be executed infinitely [13]. We have the safety and liveness properties mathematically based on timed automata and are unambiguous.

2.4. Formal Verification

Formal verification is the mathematical demonstration of the correctness of a system. The basic idea is to construct a mathematical model of the system under investigation, a model which represents the possible behavior of the system. The correctness requirements are specified along with the other functional requirements that represent the desirable behavior of the system. Based on these specifications, we check formal proof whether the possible behavior agrees with the desired behavior. Verification process can be made precise by using formal methods. Formal verification leads to proving or disproving the correctness with respect to this formal correctness notion. Formal verification can achieve complete exhaustive coverage of the system thus ensuring that undetected failures in the behavior are excluded.

In summary, formal verification requires a model of the system consisting of:

1) A set of states incorporating information about values of variables program counters;
2) A transition relation that describes how the system can change from one state to another;
3) A specification method for expressing requirements in a formal way;
4) A set of proof rules to determine whether the model satisfies the stated requirements.

To obtain a more concrete feeling of what is meant, we consider the way in which sequential programs can be formally verified.

2.5. Model Checking

Model checking [14]-[19] is a method for automatic and algorithmic verification of finite state concurrent systems. It takes as input a finite state model of a system and a logical property, it then systematically checks whether this property holds for a given initial state in that model. Model checking is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. It uses temporal logic to specify correct system behavior. An efficient, flexible search procedure is used to find correct temporal patterns in the finite state graph of the concurrent system. The orientation of the method is to provide a practical verification method. The technical formulation of the Model checking is: Given structure M, state s, and TL formula f, does M, s |= f?. Clarke and Emerson [14] formulated the CTL (Computation Tree Logic) and proposed a CTL Model checking algorithm. They proposed that concurrent programs can be abstracted to finite state synchronization skeletons, suppressing behavior irrelevant to concurrency.

Model checking addresses finite systems but can be scaled up to a more complex system as a multi-agent system. Here, by complex we mean a system with a large number of independent interacting components, with non-linear aggregate activity, with concurrency between components and constant evolution. Model checking basic idea is to use algorithms executed by software tools to verify the correctness of the system. The user inputs a description of a model of the system, the possible behavior, and a description of the requirements specification, *i.e.* the desirable behavior, and leaves the verification up to the machine. If an error is recognized, the tool provides a counter-example showing under which circumstances the error can be generated. The counter-example consists of a scenario in which the model behaves in an undesired way. Thus the counter-example provides evidence that the model is faulty and needs to be revised. This allows the user to locate the error and to repair the model specification before continuing. If no errors are found, the user can refine its model description e.g. by taking more design decisions into account so that the model becomes more concrete and can restart the verification process.

2.6. Timed Automata

A timed automaton is a finite state automaton equipped with a finite set of real value clock variables called clocks, which are used to measure the elapse of time. Timed automata are used to model finite state real-time systems. A state of a timed automaton consists of the current location of the automaton plus the current values of all clock variables. Clocks can be initialized when the system makes a transition. Once initialized, they start incrementing their value implicitly. All clocks proceed at the same rate. The value of a clock thus denotes the amount of time that has been elapsed since it has been initialized. Conditions on the values of clocks are used as enabling conditions of transitions: only if the clock constraint is fulfilled, the transition is enabled, and can be taken; otherwise, the transition is blocked. Invariants on clocks are used to limit the amount of time that maybe spent in a location. Enabling conditions and invariants are constraints over clocks.

A timed automaton A is a tuple (***L, l_0, E, Label, C, clocks, guard, inv***) [21] with
- ***L***, a non-empty, finite set of locations with initial location $l_0 \in L$
- $E \subseteq L \times L$, a set of edges
- ***Label***: $L \rightarrow 2^{AP}$, a function that assigns to each location $l \in L$ a set Label(*l*) of atomic propositions
- ***C***, a finite set of clocks
- ***clocks***: $E \rightarrow 2^{C}$, a function that assigns to each edge $e \in E$ a set of clocks clocks(e)
  clocks: $E \rightarrow \psi(C)$, a function that assigns to each edge $e \in E$ a set of clocks clocks(e)
- ***guard***: $E \rightarrow \psi(C)$, a function that labels each edge $e \in E$ with a clock constraint guard(e) over C, and
- ***inv***: $L \rightarrow \psi(C)$, a function that assigns to each location an *invariant*.

3. UPPAAL

UPPAAL [20] is a toolkit for symbolic model checking of real-time systems developed at the University of Uppsala (Sweden) and Aalborg (Denmark). It provides model checking for verification of behavioral properties as well as simulation of timed automata. It also has some features to detect deadlocks. The model checking algorithms that are implemented in UPPAAL are based on sets of clock constraints, rather than on explicit sets of regions. By dealing with (disjoint) sets of clock constraints, a coarser partitioning of the infinite state space is obtained. A multi-agent system in UPPAAL is modeled as a network of timed automata, called processes. A clock variable evaluates to a real number and clocks progress synchronously. The fulfilled constraints for the clock values only enable state transitions but do not force them to be taken. A process is an instance of a parameterized template. A template can have local declared variables, functions, and labeled locations. State of the system is defined by locations of the automata, clocks, and variables values.

UPPAAL uses a query language (*i.e.* subset of TCTL) for defining requirements. The query language consists of state formulae and path formulae. State formulae describe individual states with regular expressions such as $x \geq 0$. State formulae can also be used to test whether a process is in a given location. Path formulae quantify over paths of the model and can be classified into reachability, safety, and liveness properties:

- Reachability properties are used to check whether a given state formula f can be satisfied by some reachable state. The syntax for writing this property is E <> f.

- Safety properties are used to verify that something bad will never happen. There are two path formulae for checking safety properties. A[] f expresses that a given state formula f should be true in all reachable states, and E[] f means that there should exist a path that is either infinite, or the last state has no outgoing transitions, called maximal path, such that f is always true.
- Liveness properties are used to verify that something eventually will hold, which is expressed as A <> f.

Processes communicate with each other through channels. Binary channels are declared as **chan x**. The sender **x!** can synchronize with the receiver **x?** through an edge. If there are multiple receivers **x?**, then a single receiver will be chosen non-deterministically. The sender **x!** will be blocked if there is no receiver. Broadcast channels are declared as broadcast **chan x**. The syntax for sender **x!** and receiver **x?** are the same as for binary channels. However, a broadcast channel sends a signal to all the receivers, and if there is no receiver, the sender will not be blocked. UPPAAL also supports arrays of channels. The syntax to declare them is **chan x [N]** or broadcast **chan x [N]**, and sending and receiving signals are specified as **x[id]!** and **x[id]?**. Processes cannot pass data through signals. If a process wants to send data to another process then the sender has to put the data in a shared variable before sending a signal and the receiver will get the data from shared variable after receiving the signal.

4. Case Study: A Multi-Agent Transport System

In this section we present a case study of multi-agent system. It is a system composed of transporting agents. The objective is to specify our system using timed automata and then verify the correctness properties of safety and liveness. The mission is to transport stock from one storehouse to another. They move in their environment which in this case is static, *i.e.* topology of the system does not evolve at run time. We have specified each and every part of the system, *i.e.* agents along with the environment in the form of LTS.

There are three types of agents:

1) Carrier agent: It transports stock from one storehouse to another, it can be loaded or unloaded and can move both forward and backward direction. Each road section is marked by a sign number and the carrier agent can read this number.

2) Loader/Un-loader agent: It receives/delivers stock from the storehouse, it can detect if a carrier is waiting (for loading or unloading) by reading the presence sensor, it ensures that the carrier waiting to be loaded is loaded and the carrier waiting to be unloaded is unloaded.

3) Store-manager agent: It manages the stock count in the storehouse and it also transports the stock between storehouse and loader/un-loader.

4.1. UPPAAL Model

The template of the Carrier agent has eight locations: *Safe*, *Appr Loaded Carrier*, *Parking Store House A*, *Start Loaded Carrier*, *Crossing*, *unload Carrier*, *Parking Store House B* and *Start Empty Carrier*. The templates for Carrier agent and path have been modeled as shown in **Figure 1**.

The Carrier Agent Template: The initial location is *Safe*, which corresponds to a carrier agent has not appeared on crossing loaded yet. The location has no invariant, which means that a carrier agent may stay in this location for an unlimited amount of time. When a carrier agent is approaching, it synchronizes with the controller. This is done by the channel synchronization appr! on the transition to *Appr Loaded Carrier*. The controller has a corresponding appr?. The clock x is reset and the parameterized variable *e* is set to the identity of this carrier agent. This variable is used by the queue and the controller to know which carrier agent is allowed to continue or which carrier agent must be stopped and later restarted.

The location *Appr Loaded Carrier* has the invariant $x \le 20$, which means that the carrier agent must leave the location within 20 time units. The two outgoing transitions are guarded by the constraints $x \le 10$ and $x \ge 10$, which corresponds to the two sections before the crossing: can be stopped and cannot be stopped. At exactly 10, both transitions are enabled, which allows us to take into account any race conditions if there is one. If the carrier agent can be stopped ($x \le 10$) then the transition to the location *Parking Store House A* is taken, otherwise the carrier agent goes to location *Crossing*. The transition to *Parking Store House A* is also guarded by the condition e == id and is synchronized with stop?

When the controller decides to stop a carrier agent, it decides which one (sets *e*) and synchronizes with stop!

The location *Parking Store House A* has no invariant: a carrier agent may be stopped for an unlimited amount of time. It waits for the *synchronization go*?. The guard e == id ensures that the right carrier agent is restarted.

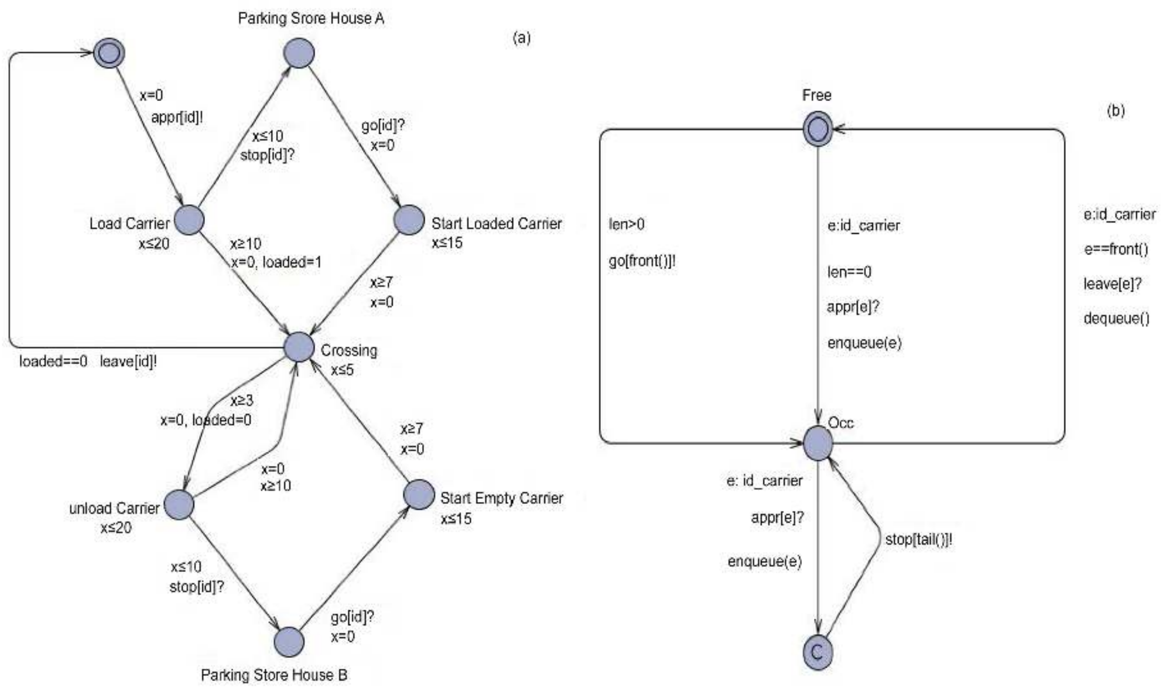

Figure 1. The template for the (a) Carrier agent; and (b) Carrier path.

We can assume that a carrier agent may receive a go? synchronization even when it is not stopped completely, which will give a non-deterministic restarting time. The location *Start Loaded Carrier* has the invariant $x \leq 15$ and its outgoing transition has the constraint $x \geq 7$.

This means that a carrier agent is restarted and reaches the crossing section between 7 and 15 time units non-deterministically. The location Crossing is similar to *Start Loaded Carrier* in the sense that it is left between 3 and 5 time units after entering it.

The Template of the Path: The path controller synchronizes with the Carrier agent. Some of its locations do not have names. Typically, they are committed locations (marked with a c). The controller starts in the *Free* location (*i.e.*, the path is free), where it tests the queue to see if it is empty or not. If the queue is empty then the controller waits for approaching carrier agent with the appr? synchronization. When a carrier agent is approaching, it is added to the queue with the add! synchronization. If the queue is not empty, then the first carrier agent on the queue is restarted with the go! synchronization.

In the *Occ* location, the controller essentially waits for the running carrier agent to leave the path (leave?). If other carrier agent is approaching (appr?), they are stopped (stop!) and added to the queue (add!). When a carrier agent leaves the path, the controller removes it from the queue.

Table 1 shows the UPPAAL code snippets for system declarations, global declarations, and carrier agent declarations. There is also the UPPAAL verification code which highlights the safety properties.

5. Conclusions and Future Work

A well-defined, precise, timed automaton based model of a multi-agent transport system is proposed. The correctness properties of safety and liveness of this proposed model are verified. The multi-agent system is modeled as a network of timed automata. A clock variable evaluates to a real number and clocks progress synchronously.

Our future work is the proposition, design and implementation of transformation rules for the translation of the current timed automata based formal model into a working system. This working system would have a formal foundation as it would be based on timed automata model.

Acknowledgements

We are grateful to The Worthy Vice Chancellor, The Islamia University of Bahawalpur for motivation and en-

Table 1. UPPAALspecifications (liveness and safety properties).

Declarations	UPPAAL Specifications
System declarations	//Template instantiations system Carrier_Agent,Path;

Global declarations	//Global declarations
	const int N = 2; //number of carrier agents
	typedef int [0,N-1] id_carrier;
	chanappr[N], stop[N], leave[N];
	urgent chan go[N];

Carrier agent declarations	id_carrier list[N+1];
	int[0,N] len;
	// Postcondition: Put an element at the end of the queue
	void enqueue(id_carrier element)
	{list[len++] = element;}
	// Postcondition: Removes the front element of the queue
	void dequeue()
	{
	int i = 0;
	len -= 1;
	while (i < len)
	{list[i] = list[i + 1];
	i++;}
	list[i] = 0;
	}
	// Postcondition: Returns the front element of the queue
	id_carrier front()
	{return list[0];}
	// Postcondition: Returns the last element of the queue
	id_carrier tail()
	{return list[len - 1];}

Formal verification using UPPAAL verifier	//Collion Detection on Crossing Area
	E<> Carrier_Agent(1).Crossing imply not Carrier_Agent(0).Crossing
	//Reachability
	E<> Carrier_Agent(1).Crossing
	//Check Deadlock For System
	A[] not deadlock

couragement. This work has been possible due to the support of The Department of Computer Science & IT, The Islamia University of Bahawalpur, Pakistan.

References

[1] George, V. and Vaughn, R. (2003) Application of Lightweight Formal Methods in Requirement Engineering. *Crosstalk*: *The Journal of Defense Software Engineering*, **16**, 30.

[2] Wooldridge, M. (2002) An Introduction to MultiAgent Systems. John Wiley and Sons, Chichester.

[3] Wooldridge, M., Jennings, N.R. and Kinny, D. (2000) The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, **3**, 285-312. http://dx.doi.org/10.1023/A:1010071910869

[4] Dogac, A. and Cingil, I. (2004) Agent Technology. In: *B2B E-Commerce Technology*: *Frameworks*, *Standards and Emerging Issues*, Addison-Wesley, Boston, 103-150.

[5] Wooldridge, M. and Jennings, N.R. (1995) Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, **10**, 115-152. http://dx.doi.org/10.1017/S0269888900008122

[6] Weyns, D., Omicini, A. and Odell, J. (2007) Environment as a First-Class Abstraction in Multi-Agent Systems. International Journal of *Autonomous Agents and Multi-Agent Systems*, **14**, 5-30.

[7] Russell, S. and Norvig, P. (2002) Artificial Intelligence: A Modern Approach. 2nd Edition, Prentice Hall, Upper Saddle River.

[8] Demazeau, Y. (2003) Multi-Agent Systems Methodology. Franco-Mexican School on Cooperative and Distributed Systems (LAFMI).

[9] Ferber, J. (1999) An Introduction to Distributed Artificial Intelligence. Addison-Wesley, Boston.

[10] Jennings, N.R., Sycara, K. and Wooldridge, M. (1998) A Roadmap of Agent Research and Development. *International Autonomous Agents and Multi-Agent Systems*, **1**, 7-38. http://dx.doi.org/10.1023/A:1010090405266

[11] Ghezzi, C., Jazayeri, M. and Mandrioli, D. (2003) Fundamentals of Software Engineering. 2nd Edition, Prentice Hall, Upper Saddle River.

[12] Magee, J. and Kramer, J. (2006) Concurrency: State Models and Java Programs. 2nd Edition, John Wiley and Sons, Hoboken.

[13] Giannakopoulou, D., Magee, J. and Kramer, J. (1999) Fairness and Priority in Progress Property Analysis. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, London.

[14] Clarke, E.M. and Emerson, E.A. (1981) Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: Kozen, D., Ed., Logics of Programs, Volume 131, Springer-Verlag, New York, 52-71.

[15] Quielle, J.P. and Sifakis, J. (1982) Specification and Verification of Concurrent Systems in CESAR. *Proceedings of the 5th International Symposium on Programming*, Turin, 6-8 April 1982, 337-351. http://dx.doi.org/10.1007/3-540-11494-7_22

[16] Clarke, E., Grumberg, O. and Peled, D. (1999) Model Checking. MIT Press, Cambridge.

[17] Clarke, E.M., Emerson, E.A. and Sistla, A.P. (1986) Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, **8**, 244-263.

[18] Clarke, E.M., Grumberg, O. and Long, D.E. (1994) Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, **16**, 1512-1542. http://dx.doi.org/10.1145/186025.186051

[19] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. (2003) Counter Example-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, **50**, 752-794. http://dx.doi.org/10.1145/876638.876643

[20] Larsen, K.G., Pettersson, P. and Yi, W. (1997) UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, **1**, 134-152.

[21] Katoen, J.P. (1999) Concepts, Algorithms, and Tools for Model Checking. A Lecture Notes of the Course Mechanized Validation of Parallel Systems. For 1998/99 at Friedrich-Alexander Universitat, Erlangen-Nurnberg, 195.