Theoretical and Applied Informatics
ISSN 1896–5334
Vol.23 (2011), no. 3-4
pp. 229–243
DOI: 10.2478/v10179-011-0016-6

Modelling CTMC with a standard programming language and using conventions from computer networking

ARTUR RATAJ, MATEUSZ NOWAK, PIOTR PECKA

Institute of Theoretical and Applied Informatics
of the Polish Academy of Sciences

Received 7 September 2011, Revised 1 December 2011, Accepted 5 December 2011

Abstract: Continuous time Markov chains (CTMC) are one of the formalisms for building models. This paper discusses expressing these models in a standard programming language – Java. Using such a language as a model description allows for a partially common implementation of the production software and of the description of the model, for a greater flexibility in comparison to model–checker specific languages that often do not employ features of an object–oriented programming. Using Java also makes the parsing of models relatively fast, using optimised Java runtime environment.

Our approach aims at using typical mechanisms of the Java language when implementing the model, and at the same time, following closely the concepts from computer networking and from formalisms based on it, like the queueing systems. These assumption result in techniques like plain object fields constituting the state vector, or negotiation between nodes to decide if an event happens.

Keywords: Markov chain, continuous time, model checking, Java

1. Introduction

Along with model checkers or other software that analyses continuous–time Markov chain models (Parzen 1999) there is a host of languages for representing these models. These languages, usually being at the fringe of the effort focused on analysis of the models themselves, typically lack expressiveness and employ outdated programming paradigms.

A typical example is the Prism model checker (Kwiatkowska et al., 2005, 2009), which employs state–of–the–art algorithms for solving the models, but at the same time supports only a simple language that lacks basic features like intermediate variables outside of the state vector, and uses text–level preprocessing techniques instead of more modern methods like object instantiation. On the positive side, the language is built from ground up to represents models (Alur and Henzinger, 1999), having in effect advantages like the great readability of its dedicated grammar.

There are model checkers, though, that are able to embed a standard language like C into their model description, for example the SPIN model checker (Holzmann, 2004). This adds a lot of flexibility in comparison to the Prism language, but still, the Promela language, into which the C code is embedded, represents a simple procedural approach to programming.

It could be concluded, that the best way of improving the simple languages would be extending them with features like a standard library with complex data types or a support for object–oriented programming. Such approach would not compromise the readability of these languages. Yet, an user would need to learn a new, advanced objective language, what in certain cases might make the approach impractical. We base this supposition by looking at the typical traits of languages that are flexible and expressive, or in other words, include a complex set of grammar rules and a large standard library.

For starters, standard, general computer languages:

- are commonly used for production applications – that possibly allows for using the same implementation for both production and model checking, what can save on development and reduce the number of possible issues caused by the incoherence between two implementations – one of a system, and one of its model; that obviously may lead to checking of not only traits of the model, but also of the correctness of its implementation;
- complex languages naturally require a long training process to master due to their complexity – it is not surprising, thus, that the popular ones typically offer back a lot to reward the learning effort: *generality*; in contrast, a dedicated language for representing models usually offers nothing beyond that;
- they offer an excellent documentation – there are hundreds of books on Java or C++, and hundreds of discussion groups devoted to exchanging experiences on these dialects; this pairs well with the complexity of the languages;
- they can have a very fast runtime environment – for example, current versions of the Java Virtual Machine have a very effective just–in–time compiler (Inc., 2007); this can have a direct effect on the time of analysis of a model description;
- the discussed languages require a non–trivial maintenance – for example, there is a large number of people maintaining the Java language and its implementation, and still, they deal with hundreds of not yet resolved issues;

This raises a question about an alternative solution in the task of using an expressive language for describing models – instead of building a complex dedicated language, which would try to repeat the advantages given above, why not just design a library on top of an existing complex language. Naturally, some of the advantages of model languages like a dedicated grammar would be lost, but the discussed alternative might still have its niche.

This paper proposes such an approach. We developed `OLIMP2` – a library on top of Java for representing models in that language and for translating these models into transition matrices. The main reason for developing it was, that the Prism language makes it very hard or impossible to implement certain models cleanly and optimally.

We did not want, though, to, in a sense, copy the Prism language or a similar one into Java, by reusing as closely as possible the conventions for, say, defining guards or synchronising actions, that are commonly obeyed in many model languages. While we find such 'copying into a standard language' a possibly viable approach, especially that the effect might resemble more accepted formalisms like that in (Alur and Henzinger, 1999), we were instead interested in reusing natural mechanisms of an imperative language and of the formalism of computer networks, as the latter are commonly modelled using Markov chains.

Due to the potentially very large sizes of a model's transition matrix, its generation using `OLIMP2` is parallelised, and the matrix can be compressed on-the-fly using a dedicated compression based on finite-state automata.

The paper is organised as follows. The next section discusses a similar work. Section 3 illustrates the basic ideas behind our method. Sec. 4 shows a simple example. The last section concludes the paper.

2. Similar work

We are not aware of any CTMC library for a C–like language, that offers a similar functionality, beside `OLIMP` (Pecka, 2002), from which stems `OLIMP2`. There are, though, some projects that treat a Java application as a process with a *discrete* time, like JavaPathFinder (Khurshidetal., 2004), or Java2TADD (Wozna and Zbrzezny, 2008; Rataj et al., 2008; Rataj, 2009). The former is a kind of the Java Virtual Machine (Yellin and Lindholm, 1999), that is able to test complex production applications, but it does not translate these applications into transition matrices. The latter is, in contrast, limited to a small subset of Java grammar and requires the analysed application to be instrumented, but translates it to a form, that can be easily transformed to a transition matrix, which in turn can be reused by a lot of model checking software.

3. Basic concepts

The basic assumption is to use the style and concepts found in the Java language and in the formalism of queueing systems, that describe computer networks or similar systems. The former stems obviously from choosing Java as the model representation, the latter from the fact, that various kinds of networks are commonly modelled and checked using CTMC.

To realise the assumption, we aimed at making the model representation similar to an implementation of a computer network in Java.

3.1. Nodes and connections

Let a *node* be an entity, that, on basis of the current state vector, decides on a fragment or a whole of the new state vector, and on its contribution to the respective transition rate to that new state vector.

A node is thus a generalisation of a queueing theory's server, in the sense that there can be an arbitrary number of elements of the input state vector and in the output state vector, that are respectively read or written to by a node.

In a computer network, an event is typically a transfer of a packet from one node to another. The sender must decide that it sends a packet, and the receiver must accept that decision. Similarly, in a queueing model, a sending server needs a non–empty buffer, and the receiving server needs a non–full buffer to realise a transfer of a task. We follow that closely.

Let a *connection* in `OLIMP2` be a way, though which a *token* can be sent. per a connection, there is a single sender node, and a number of receiver nodes. A token, to be successfully transferred, must be sent by the sender and accepted by the receivers. Sending a token is the only possible way of an event to occur in `OLIMP2`.

Thus, there is a negotiation needed between a number of nodes for every event to occur. In a degenerated case, a single node sends a token to itself to change its own internal state, what also is a case of a negotiation.

A connection is a generalisation of a connection in the queueing theory, in the sense, that the token must not necessarily be a task, and its transfer may be an equivalent of a number of arbitrary transitions, as opposed to only changing servers' buffer sizes.

3.2. A node is a class

A node in `OLIMP2` is a Java class, that directly of indirectly extends `AbstractNode`. A node can contain a number of *state fields*. All state fields in all instantiated nodes create together the state vector. A state field is an integer value or an array of integer values, that can be read or modified by the nodes, what is an equivalent of, respectively, reading the input vector state and creating the output vector state.

Initialising the state fields while constructing the nodes determines the initial state of the model.

3.3. Negotiation details

`AbstractNode` needs two of its methods to be defined by subclasses – `transit` and `receive`. The former method is called by `OLIMP2` as a way of querying a node, if, for a given input state vector and a given connection c, the node can asynchronously initiate an event. A node, within `transit`'s code, reads the input state and then, on basis of the values read, may try to send a token through c, by calling via `OLIMP2` the `receive` methods of the receiver nodes assigned to the other, receiving end of c.

A `receive` method defines token acceptance and, if the token is accepted, a contribution to the rate and the output vector of a respective transition. The token sender, again within `transit`, on basis of these return values of the called `receive` methods, determines the resulting transition rate and returns it through `transit`'s return value to `OLIMP2`. If the token has been accepted, the return rate is non–zero, otherwise, it is zero.

If the rate is non–zero, `OLIMP2` reads the output state from node's state fields and completes the transition matrix.

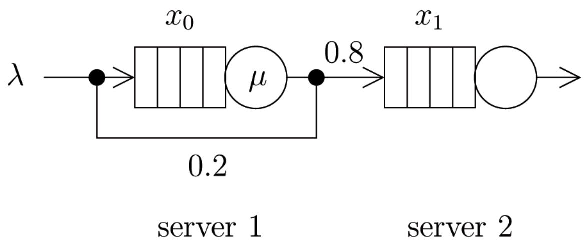

Fig. 1. An example queueing model

The example model in Fig. 1 illustrates a negotiation between two nodes – for a task to be send from the server 1 to the server 2, the following sequence is required:

1. server 1's `transit` reads from the input state, that $x_0 \neq 0$;
2. thus, it tries to send a token, by calling server 2's `receive`;
3. server 2 reads from the input state, that $x_1 \neq$ its maximum buffer size;
4. thus, it accepts the token sent by server 1, and increases x_1 by 1;
5. server 1 sees, that `receive`'s return value is non–zero, thus, the token has been accepted, so it decreases x_0 by 1, and returns a non–zero transition rate in `transit`'s return value.

3.4. Adhering to Markov's property

A CTMC should be memoryless, and can have races. Both qualities have a direct impact on the allowable behaviour of the nodes.

For each combination of a reachable input state and a connection, called further a base combination, for which some node is a possible sender because it is that connection's sender, a node's `transit` is called only once, unless some receiver node declares it wants to be queried again for a given base combination but with a different *race number*, because the node known that alternate output vectors are also possible. All in all, the following should together completely determine, because of the required memorylessness, the behaviour of a node, whose `transit` or `receive` has been called by `OLIMP2`:

- input state vector;
- a connection c;
- in the case of the sender, return values of called `receive` methods;
- in the case of a receiver, the mentioned race number.

While the base combination is the same for the sender and for the receivers of c, the race numbers are individual for each of the receivers. Initially, for each base combination, each receiver i of c has its race number $r = 0$. Any of these receivers, though, may request a successive race number as discussed.

The idea is, that all *combinations* of requested race numbers are queried. So, for example, if there are two receivers in some connection, the first one requested race numbers $0 \ldots N_1 - 1$, and the other, respectively, $0 \ldots N_2 - 1$, a total number of queries of the sender for a given base combination will be $N_1 N_2$.

To request a next race number, a node changes the state according to the current race number, and then modifies `receive`'s rate contribution by using a special function `neg`, to inform `OLIMP2`.

Note, that the sender must behave sensibly when the receivers request successive race numbers. It would be suspicious, if a receiver would request another race number, but the sender would not try to send a token for the given base combination any more, thus, the receiver would never be called with the race number it has requested. Such suspicious behaviour is forbidden by the already discussed policy, because when the sender decides or not to negotiate sending of a token, it does not yet know `receive`'s return values, and thus, only the base combination can determine its behaviour.

Support for the race number implies, that there are two ways of a race to occur:

- a usual, directly declared one – if there is more than a single connection, for which some node is a sender;
- if one or more receiver nodes requested more than a single race number.

The former is a typically defined race, as in the example illustrated in Fig. 1. As can be seen, the server with the service time μ can send a task with the probability of 0.8 to the next server, or return the task to its own buffer with the probability of 0.2. The node representing that server would thus be asked twice for each reachable state – one time for each of the two connections, to which it can send tokens.

The latter is a so called in–connection race. We will discuss that kind of race in detail in Sec. 3.6.

3.5. Transition rate

The basis for a rate of the transition from an input state to an output state is r_s, that is defined by the sender in `transit`'s return value. For example, if the sender is a exponential service server in the formalism of queueing systems, the value returned might simply be its service rate.

We call it a basis, and not the transition rate, as the sender may return a number of r_s values for a single input state vector, if the sender sent accepted tokens to multiple connections or if there is a in–connection race. If, for some of these values, there is not only the input state the same, but also the output state is the same, then the respective values r_s will sum into a single transition rate.

A sender, as discussed in in Sec. 3.4., may compute r_s on basis of the return values of the $receive$ methods it called. These return values are combined in a single value r_r by `OLIMP2` though, and the sender sees only that single value. A token is accepted if $r_r \neq 0$. The exact formula for computing r_r is dependent on c's *mode*.

One of the factors in that formula is mode–independent – it is a connection–specific constant value r_c. By convention, that value reflects the probability of choosing the connection. If a node can send tokens to a set of connections s, then the sum of r_c within s, if that convention is followed, is 1. In the example in Fig. 1, the server 1 would send tokens to two connections whose r_c values would be respectively 0.8 for the connection going to the next server and 0.2 for the returning connection.

Beside r_c, there are the mentioned values specified by receivers in `receive`'s return value. Let there be a connection whose receivers are a set of nodes t_i, $i = 0, \ldots T - 1$. Let the respective factors be r_i^t.

In the default mode `PRODUCT`, `receive`'s return values are factors:

$$r_r = r_c \prod_{i=0}^{T-1} r_i^t \tag{1}$$

This mode is similar to that used by other model checkers when combining rates of synchronised commands, and allows for multiplying rates by probabilities. In the mode, for a token to be accepted, all receivers must return a non–zero r_i^t.

There is also another mode possible – `SUM`:

$$r_r = r_c \sum_{i=0}^{T-1} r_i^t \tag{2}$$

An example application: receiving nodes are pumps, and r_r is the flow speed.

3.6. In–connection races

A connection has two purposes – it connects a sender to the receivers, but it also synchronises these receivers. It simply stems from the fact, that a given token is sent to all of these receivers at once. The contribution of the sender of a connection c and of all receivers of c to the state variables merges to form the output state.

By using the race numbers discussed in the previous section, a race within a single base combination is possible. Let there be a connection c with a sender node s and a number of N receivers t_i, $i = 0 \ldots N-1$. If $N = 1$, and t_0 requests race numbers $0 \ldots M-1$, then the resulting in–connection race could obviously be equivalently defined by M parallel connections between s and t_0. The `receive` method of t_0 would in such an equivalent definition be sensitive to the connection number, instead of the race number. So, an in–connection race would in such a case have no benefits – both implementations would be similar.

But, if $N > 1$, the in–connection races allow for a very convenient definition of certain complex races.

For example, let s be a clock, that ticks with the times between these ticks given by, say, an Erlang distribution. Let t_i be switches synchronised by a common tick from s. On a tick from s, each switch releases a packet from its buffer to some network with a probability $p = 0.2$, and the release is independent from possible releases made by other switches.

Let it translates in the model description to each switch deciding about a part of the output vector not overlapping with the respective parts of the vectors modified by the other switches.

Clearly, a tick may produce 2^N output states from a single input state. Were there no in–connection races, then even for a small N a great number of connections would be needed to which s would send the tick, one connection per output state. Instead, an in–connection race can be used. It requires only a single connection c, through which a tick token is sent from s. Each receiver t_i of c requests two race numbers – one for releasing the packet, and one for not releasing it. In both cases, the token is accepted, but the rate contributions r_i^t, $i = 0 \ldots N$, are respectively 0.2 and 0.8. The c's mode must be `PRODUCT`, so that the probabilities of the independent events , in this case being equal to the values r_i^t, are correctly combined into r_r.

3.7. Tokens

A sender can send various types of tokens through the connections. It does not give any more possibilities for node behaviour beside these discussed in Sec. 3.4., but can ease the implementation. For example, a sender node may parse the input vector to find out, which action should be taken, and then it can send a token that defines that action. This way, a receiver does not need to parse the input vector again, but it just looks at the token type. It not only may remove redundancy from the implementation, but also make nodes more modular – returning to the example, the receiver might be a model–independent module, if it looks only at the token type, as opposed to divulging into the model's input state.

3.8. A library

`OLIMP2` comes with a library of standard components. The library extensively uses the objective programming paradigm. For example, an abstract class of a server `AbstractBufferedServer` is extended by a subclass `ErlangBufferedServer`.

3.9. Further implementation details

In this section, some further implementation details are discussed.

3.9.1. Explicit state variables

As discussed, a node can contain various fields, but it explicitly lists to `OLIMP2` the subset of these fields, that are a part of the state vector. It does that using a field annotation. The annotation supports two parameters `min` and `max`, that define the range of values, into which fits the state variable. If not defined, these limits default to $\langle -32768, 32767 \rangle$. Example of a state variable:

```
@State{min = 0, max = MAX_BUFFER_SIZE}
int packetsNum = 0;
```

A node's initial state is determined while its construction. In the example, an initial value of 0 is assigned to the part `packetsNum` of the state vector.

3.9.2. Refreshing the state variables

`OLIMP2` must set the state variables to the input state before calling `transit`. Some non–public state variables might be not set, as discussed in the next section. Anyway, `OLIMP2` might refresh these variables before each receiver's `receive` is called, but to reduce the computational complexity, there is a instead rule, that the sender is not allowed to modify the state variables within `transit` before trying to send a token. If there is more than a single receiver in a connection, though, the state variables are refreshed between two successive calls to `receive`, along with saving of the modifications to the vector state made by the methods, of course. This is substantiated by the peer nature of the receivers – there is no asymmetry like the sender/receiver asymmetry, so all receivers equally receive the input vector in the state variables. The sender should not rely on the state variables after a token is accepted, as their values are undefined. After the acceptance, the sender may only write to the state variables, to specify the output state.

3.9.3. Public vs non–public state variables

With further discussed exception, a node can not access in any way a non–public state variable of *another* node – it can not either read it nor set the output state by writing to it. It is because `OLIMP2` optimises computations by possibly not updating such variables.

The exception is, that a node is allowed to *read* the non–public state variables of all senders and receivers of the connection, that is a part of the base combination, These not–public fields can not still, though, be written to by a node to which they do not belong.

A user that does not want to follow these somewhat complicated rules from this section, may simply declare all state variables as public, but it may come with some impact on the computation speed, which is typically very minor if the state vector is not very large.

3.9.4. Sending a token

There is a method `accept` that serves as `OLIMP2`'s layer between the sender and the receivers. The method calls the `receive` methods in turn – a sender should never directly call `receive`.

A node can send a token to itself. It can do so using a looped connection, but it can also do so using a so called internal token transfer.

To support th case of a looped connection, the `receive` methods have an additional parameter `me`, that represents, if the token to accept was send by the same node. This way, a node knows, that it may yet modify the output vector in `transit`, after `receive` returns. That knowledge is sometimes crucial, what is illustrated by the example described in Sec. 4.

An internal token transfer can be used, when using a looped connection would be unnatural. For example, a server having the service time given by the Erlang distribution, whose example is given in Sec. 4., changes its phase before the serviced task is released. Due to `OLIMP2` rules, such a change must involve a token transfer, just as any other event. But, in queueing terminology, servers with Erlang service time do not possess a looped connection just for changing the phase. An internal token transfer could thus be used instead. That type of transfer is realised by by a variant of the method `accept`.

The method `accept` also computes the value r_r. The return value of that method is specifically $r_r r_a$, where r_a is the method's argument decided by the sender. This eases computing r_s, as it is typical that the sender multiplies the 'acceptance factor' r_r by a given factor to create r_s. An example of it is shown in Sec. 4., where r_a is the rate of reaching a new phase by a server.

3.9.5. Caveats of parallel execution

`OLIMP2` is able to work concurrently. As the model is not reentrant, it is copied into a number of clones, accessed concurrently.

If the models happens to have references to the other nodes in a model, then these references must obviously be changed in the clone models..

To update the references, `OLIMP2` scans all fields in the nodes for types being `AbstractNode` or its subclass, and replaces these fields with references to cloned nodes. If such references are hidden outside the node fields, `OLIMP2` is not able to reach and replace them. Such hidden references should thus not be used.

3.9.6. Adding a receiver to a source

All sources in the library implement an interface `SourceInterface`, which has a method for adding new receivers. The method adds the receivers to a single connection, instead of adding a separate connection for each receiver.

The effect is, that a token generated by the source goes to all receivers at once. If this is not the desired behaviour, a number of separate sources should be declared, what will effect in independent tokens being sent to each receiver.

3.9.7. Free use of features of the Java language

`OLIMP2` is only a library, the user is thus free to use all of the language features.

For example, let us consider again the model with pumps, mentioned in Sec. 3.5. The receivers return a total flow speed, which is a scalar, and that in turn is all what can be returned in `receive`'s return value. So, if the pumps should also return the electricity consumed, that value can be returned in another way, for example, through a custom field in a token. The sender may then, say, quantise that value and increase a respective part of the output state vector by the quantised amount.

3.9.8. Compression of the transition matrix

Because the nodes are currently a black-box to `OLIMP2`, the transition matrix is not stored in a symbolic form. This might make it very large. To reduce memory requirements, a dedicated compression of the matrix using finite state automatons is used.

4. Example – a buffered server

Let us analyse an example – a simple buffered server with a service time given by the Erlang distribution. In `OLIMP2`, there is a class `ErlangBufferedServer`, that defines the service time only, as the buffer is defined in a superclass `AbstractBufferedServer`. Let us look into the superclass first. The buffer only receives packets, it never sends them over the network – the latter is done by the server itself. The packet reception is modelled straightforwardly by a token reception – `AbstractBufferedServer` defines the method `receive` for that end:

```
@Override
public double receive(boolean me, AbstractToken token,
        int num) {
    if(queueSize < maxQueueSize + (me ? 1 : 0)) {
        ++queueSize;
        return 1.0;
    } else
        return 0.0;
}
```

Note that the method is sensitive to the parameter `me`. It is because if the server transfers the packet to itself, the variable `queueSize` in `transit` and `receive` is exactly the same. A node, as discussed, can never change its state before calling `accept`. But, if `accept` returns a non–zero value, it means that the queue size should be decreased. Thus, if the token was sent by the same node, `queueSize` will yet be modified after `receive` exits. If the final value of `queueSize`, that is, the value of that variable in the output, and not in the input state, is important for the acceptance of the token, the method must predict the final value. It does so by looking at the parameter `me`. If it is false, it means, that the value of `queueSize` set in `receive` is already the final one. Otherwise, if the parameter is true, the node knows it can accept the token even that the *seems* to be already full, as in reality the acceptance of the token means, that `queueSize` will be decreased in `transit`, that called `receive`. So, if the packet is going out of the server and arriving to the same server's buffer again, all within a single transition, then the queue size will be the same in the input and in the output vector, thus, the respective token should be accepted for any buffer size.

The subclass `ExpBufferedServer` defines service time. It simply means, that it declares sending a single packet from its buffer elsewhere with some rate μ. As sending a packet is represented by the server's token being accepted by a node that agrees to receive that packet, the subclass needs only to declare an appropriate `transit` method:

```
@Override
public double transit(Connection t, int num) {
    double rate;
    if(queueSize > 0) {
        if(phase < phasesNum - 1) {
            rate = model.accept(this, phaseMu, null);
            ++phase;
        } else {
            if((rate = model.accept(t, phaseMu, getToken()))
                    != 0) {
                --queueSize;
                phase = 0;
            }
        }
    } else
        rate = 0.0;
    return rate;
}
```

As can be seen, the class changes the phase of processing the packet several times using internal token transfers, until finally the packet is sent to the network. Transitions that represent the change to the next phase have a rate $phaseMu$, and the transition that represents sending the already processed token out of the server has that rate as well.

5. Conclusion and future work

`OLIMP2` uses a different approach for defining CTMC models in comparison to that of most model checkers – the model description language is a standard computer language, that is oriented towards negotiation of transfers, and not guards and synchronising labels. The approach aims at filling a niche beside the other languages for representing model,s especially of computer networks.

`OLIMP2` is planned to be enhanced so that it is able to find out, that a `transit` method, along with any methods that are recursively called, reads only a subset b of the input vector state. In such a case, `OLIMP2` would decide to call that `transit` method only for all reachable combinations of b, instead of all reachable states. That would lead to the symbolic representation of the transition matrix, as used by other checkers like Prism (Kwiatkowska et al., 2002), whose languages are not ‘black boxes’. Such a representation might radically reduce the time and memory requirements needed by `OLIMP2`. We are going to adapt the Java2TADD translator to realise that.

We plan to implement several ways, beside the existing one, of accessing the state variables by the code that describes the model. For example, one where the changes to the state vector in `transit` are allowed before calling `receive`, and visible in the latter. It would make it more natural to define certain models, for example when a state variable expresses a number of packets, which circulate between the nodes.

References

[1] R. Alur and T. Henzinger: Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[2] G.J. Holzmann: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004. ISBN ISBN 0-321-22862-6.

[3] Kogent Solution Inc. *Java 6 Programm ing Black Book*. Dreamtech Press, 2007. ISBN ISBN 9788177227369.

[4] S. Khurshid, W. Visser, and C.S. Pasareanu: Test input generation with Java Path Finder. In Proceedings of the ACM/SIGSOFT International Symposiumon Software Testing and Analysis. ACM Press, 2004. ISBN ISBN 1-58113-820-2.

[5] M. Kwiatkowska, G. Norman, and D. Parker: Probabilistic symbolic model check-ing with Prism: A hybrid approach. *International Journalon Software Tools for Technology Transfer*, pages 52–66, 2002.

[6] M. Kwiatkowska, G. Norman, and D. Parker: Quantitative analysis with the probabilistic model checker PRISM. *Electronic Notesin Theoretical Computer Science*, 153(2):5–31, 2005.

[7] M. Kwiatkowska, G. Norman, and D. Parker: Prism: Probabilistic model checking for performance and reliability analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.

[8] E. Parzen: Stochastic processes. Classics in applied mathematics. Society for Industrial and Applied Mathematics, 1999. ISBN ISBN 9780898714418.

[9] P. Pecka: *Obiektowo zorientowany wielowatkowy system do modelowania stanow nieustalonych w sieciach komputerowych za pomoca lancuchow Markowa*, PhD thesis, IITiS PAN, Gliwice, 2002.

[10] A. Rataj: More flexible models using a new version of the translator of Java sources to times automatons J2TADD. *Theoretical and Applied Informatics*, 21(2):107–114, 2009.

[11] A. Rataj, B. Wozna, and A. Zbrzezny: A translator of Java programs to TADDs. In *Concurrency, Specifcation and Programming (CS & P2008)*, pages 524–535, Gross Vaeternear Berlin, Germany, 2008.

[12] B. Wozna and A. Zbrzezny: Towards verification of Java programsin VerICS. *Fundamenta Informaticae*, 85(1-4):533–548, 2008.

[13] F. Yellin and T. Lindholm: *Java Virtual Machine Specifcation*. Prentice Hall, 1999. ISBN ISBN 978-0201432947.

Modelowanie łańcuchów Markowa z czasem ciągłym przy użyciu standardowego języka programowania i z zastosowaniem konwencji z dziedziny sieci komputerowych

Streszczenie

Łańcuchy Markowa czasu rzeczywistego są jednym z formalizmów używanych do budowy modeli. Artykuł ten omawia wyrażanie takich modeli w standardowym języku programowania – Javie. Użycie takiego języka umożliwia częściowo wspólną implementację oprogramowania użytkowego i opisu modelu, większą elastyczność w porównaniu do często nie używających obiektowych konwencji programistycznych języków stosowanych przez oprogramowanie weryfikujące, oraz szybką budowę modelu z użyciem zoptymalizowanego środowiska czasu wykonania Javy.

Nasze podejście miało na celu wykorzystanie typowych mechanizmów języka Java przy opisie modelu i jednoczesnie trzymanie się konwencji z dziedziny sieci komputerowych i pokrewnych formalizmóm typu systemy kolejkowe. Dlatego używamy technik takich jak zastosowanie pól obiektów jako elementów wektora stanu czy negocjacja pomiędzy węzłami, czy dane zdarzenie ma mieć miejsce.