Theoretical and Applied Informatics
ISSN 1896–5334
Vol. 25 (2013), no. 3–4
pp. 157–182
DOI: 10.2478/thai-2013-0008

Translation of probabilistic games in J2TADD

ARTUR RATAJ,
IITiS PAN, Bałtycka 5, Gliwice, Poland

Received 28 March 2013, Revised 18 September 2013, Accepted 31 September 2013.

Abstract: A new version of J2TADD – a translator from Java to automatons– is described, which adds support for a translation of Markov processes with non–deterministic players, that can form coalitions, which in turn strive for different aims. In order to ease the definition of a probabilistic game using a plain Java application, several new constructs, and also a special library, are supported within the input language.

Ranges on variables or on expressions can be defined, what helps in checking the self–consistency of a model, and can also make the solving of the model faster.

Keywords: model checking, Java, probabilistic game

1. Introduction

J2TADD [6, 7] is a translator of models specified in the Java language into automatons supported by several model checkers like `UPPAAL` [1] or `Prism` [3]. The current version of `hc` supports a number of new major features:

- choices, which are plain Java constructs, yet they are translated by J2TADD into various forms of probabilistic and non–deterministic branches; on a JVM, an identical or similar behaviour is realised (Sec. 2.);
- ranges on variables and on expressions – their correctness can be analysed by both `hc` and a target model checker; also, ranges help in generating more optimal models for `Prism` (Sec. 3.);
- conditions on clocks; can also be specified globally by using different schedulers (Sec. 4.);
- translation of probabilistic games (also known as multi–agent systems) into `Prism` models (Sec. 5.);

Sec. 6. shows an example of a model for `hc`, and finally there is a discussion in Sec. 7.

2. Choices

Apart from the conditional and non-deterministic choices, existing also in previous version there are also probabilistic choices available. Let us begin, though, with the conditional choice, as it can now be ‘flattened’ into a single node.

2.1. Conditional choice

A conditional choice is a state with a number of transitions such that they have complementary guards, i.e. always one and only one is true. In `hc` such an automaton can be a result of a translation of one or more constructs like Java’s `if`, where the condition is an expression that can be reduced to boolean comparisons of a mix of variables and constants.

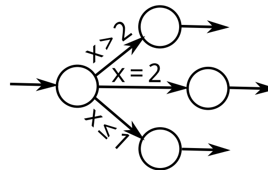

Fig. 1: A conditional choice made out of a number of binary branches.

As a volatile variable (understood within `hc` as one that can be modified by several threads), if evaluated multiple times, might yield a set of different values, a care is taken, that testing of such variables in the output automaton matches exactly the one in the source code. For non–volatile variables, though, optimisations can be applied, which collapse several evaluations into a single one. An example of such a collapse is illustrated in Fig. 1.

2.2. Probabilistic choice

A probabilistic choice is translated from conditional expressions that are of the form `if(Math.random() < `i`)`. A choice of that type is translated into a state with two transitions, each decorated with a probability value, and these two values sum to 1. For example, the mentioned construct would be translated as seen in Fig. 2.

A probabilistic choice defined using a single `if` has only two outgoing transitions. To overcome that limitation, there is also available a balanced n–ary probabilistic choice,

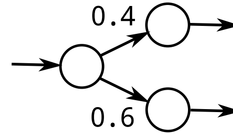

Fig. 2: An example probabilistic choice.

that is translated into n transitions outcoming from a single state, each labelled with a probability $1/n$. It is defined using an expression `(int)(Math.random()*`n`)`, which is typically assigned to a variable.

3. Ranges

An allowable range of values of a variable or a primary expression can be defined within a comment. Thanks to this, the model can still be a valid Java program. A range contains two expressions, representing respectively a minimum and a maximum allowed value. Such an expression can refer to any variable in a given scope, but `hc` must be able do evaluate the expression into a constant value, after the main thread ceases to modify the application. For that end, the compiler performs a number of optimisations in order to replace expressions with constants.

Ranges on variables are roughly translated into ranges on state variables in the output file – if `hc` can not verify by itself, that a given limit won't be violated, it widens a state variable's range, so that it can generate a PCTL property to check for any violations of that limit. Ranges on expressions, due to their locality, are translated into sub–automatons. An invalid value leads to an error state.

4. Clocks

For a better support of timed automatons [4], this version of `hc` brings a few enhancements to specification of clocks.

Let v be time in seconds. In previous versions of J2TADD, only a single delay construct was available `Thread.sleep(`$1 \cdot 10^3 v$`)`, that was effectively translated to two serial transitions: $x := 0$, $x >= v$. Currently, there are also methods `Sleep.exact(`v`)`, `Sleep.min(`v`)` and `Sleep.max(`v`)` that put different relational operators on the second transition.

Depending on the probability distribution of delay times, these methods can have a parameter having one the following forms:

- a constant distribution, $x = v$:
  `Sleep.exact(`v`);`
- a probabilistic negative exponential distribution[1], $x \in (0, \infty)$:
  `Sleep.exact(Dist.nxp(`λ`));`
- a probabilistic tabularised distribution[0], $x \in (v_1, v_N)$:
  `Sleep.exact(Dist.tab(`$\mathbf{v}, \boldsymbol{\rho}$`));`
- a non–determined, unknown distribution, $x = 0, 1, \ldots v - 1$:
  `Sleep.exact(Random.nextInt(`v`)).`

Wherever there is no clock condition or invariant on a state or on a transition, by default any duration time is assumed. That can be modified by *schedulers* – they may force, by adding clock conditions and invariants where they are absent, that the time of staying in some state is either constant, or given by a probability distribution, or that a state is left immediately, or is *hidden*. If an automaton A contains a hidden state S, then it means that that there is not a time instant, when A is in S – a series of hidden states merges with a first non–hidden state after that series, effectively forming a single update to A's state vector. A state with a synchronising label is never made hidden, from an obvious reason.

A formalism of hidden states is not supported by `Prism`, but tells `hc` that it can compact several transitions into a single one without affecting the model's meaningful behaviour. This is true e.g. in the case of turn based games, discussed in the next section.

As discussed, a scheduler by default affects only states or transitions, where no other clock conditions are defined. This can be changed by defining an *overriding* scheduler, that is applied to all states in a model.

5. Games

A Java thread, or effectively an automaton, can be declared as a player. These players can form coalitions. `Prism`'s rPATL queries can then be used to find out e.g. optimal strategies of players.

As Prism accepts only turn–based games now, only a single player can move at once, and that quality must be explicitly seen by Prism. It would be impossible to implement such a game by using Java's locks, as, due to the way `hc` works, there are always at least two automatons involved in a `wait()`/`notify()` operation [7], one of these is a special lock automaton. Thus, a different synchronising mechanism is used in turn–based games – a conditional barrier. It is directly translated to synchronising labels, and optional conditions on leaving a barrier are translated to guards of these labels.

[1] currently not supported by `hc`'s backends

For easier implementation of a game, a pure Java library for implementing games can be used, included with `hc`. It employs player synchronisation in a way compatible with Prism's requirements.

Using the *hidden* scheduling discussed in the previous section can greatly optimise a network of players, and does not break a basic system behaviour – when one of the players moves, the other players wait, and thus can't see what happens in the automaton of the moving player, in particular, that some of its states between synchronisation points have been merged.

6. Case study

In this study, an example model will be specified. Also, several detailed features of `hc` not discussed in the previous sections will be described if applicable.

Let us discuss a model of a distributed energy management system, as presented by [2]. It consists of a local source of electrical power, that forms a microgrid together with a number of nearby households. Similar schemes are becoming popular on energy markets, where small power providers, like wind turbines, perform a complementary service to global grid. The algorithm, if followed by the receivers of energy, smooths out peaks in demand.

There is a scheduler and N households. At the beginning of each time interval, the scheduler chooses one of the households at random, and then proposes it to buy load, i.e. a quantity of electric power at a given price. The price, within a single time step, is equal to the total number of loads generated by all households, including the additional load to be bought.

The probability, that a household needs to buy a load, is determined by a daily demand curve. If the household needs to buy a load, it is obliged to choose the microgrid if the price is below a given threshold c_{lim}. Otherwise, a household must perform a drawing, which with a constant probability P_{start} decides, that the load must be bought from the microgrid. Otherwise, that is with a probability $1 - P_{\mathrm{start}}$, the household is granted the right to back–off, e.g. by choosing a global grid instead. It is unknown, in which way a household will take advantage of that right.

The system can be modelled as a game, with $N + 1$ agents or players: the scheduler and each of the households. As a household has a choice of unknown characteristics, then it is non–deterministic – it has a strategy of picking of one of the possibilities, and the strategy is not specified in the model.

6.1. Source file

We will need to employ the mentioned non–determinism. In `hc`'s Java flavour of input files, a non–deterministic choice of one of N outcomes is modelled by the

method `Random.nextInt(N)`. Definitely, it is not due to the Java language's specs, which says, that the method draws a value using an uniform distribution. Yet, there is not an exact equivalent of what we need, and `Random.nextInt(N)` is a good default approximation, if the model is to be run on a JVM. For custom JDK–side implementations of strategies there is special a set of classes in `hc`'s library.

The discussed model is a *turn game* – only a single player can perform a move at a time. We could directly use a synchronisation method for that – a conditional barrier – but it is easier just to use a dedicated game package from `hc`'s library, which conveniently wraps that barrier.

The beginning of our model specification, that provides all of the necessary imports, is as follows:

Listing 1: Imports.

```
1 package example;
2
3 import java.util.Random;
4
5 import hc.*;
6 import hc.game.*;
```

Let us follow the rules of modular programming and divide the model into $N + 2$ entities – the microgrid itself, a scheduler, and N households. And so, we begin with the microgrid:

Listing 2: Class `Microgrid`.

```
1 /**
2  * A microgrid demand--side management game, as described
3  * in [1].
4  *
5  * [1] H. Hildmann and F. Saffre. Influence of variable
6  * supply and load flexibility on Demand-Side Management.
7  * In Proc. 8th International Conference on the European
8  * Energy Market (EEM'11), pages 63-68. 2011.
9  */
10 public class Microgrid
```

This is the only public class in the source file, and thus the class gives its name to the file. Also, the `Microgrid` class contains the start method or, in Java terms, the main method, and thus that class will give its name also to the output file.

The microgrid has a number of basic traits, that must be defined, like the number of households, or the demand curve. Let us define these:

Listing 3: Fields of `Micrograd`, part 1.

```
/**
 * Number of days.
 */
public static final int DAYS = 3;
/**
 * Number of intervals per day.
 */
public static final int INTERVALS = 16;
/**
 * Number of rounds.
 */
public static final int MAX_TIME = DAYS*INTERVALS;

/**
 * Demand curve.
 */
public static final double[] DEMAND = {
    0.0614, 0.0392, 0.0304, 0.0304,
    0.0355, 0.0518, 0.0651, 0.0643,
    0.0625, 0.0618, 0.0614, 0.0695,
    0.0887, 0.1013, 0.1005, 0.0762,
};
/**
 * Number of households.
 */
public static final int NUM_HOUSEHOLDS = 3;
```

Also, a few variables are needed to store the state of the grid.

Listing 4: Fields of `Microgrid`, part 2.

```
/**
 * Households.
 */
protected static Household[] households;
/**
 * Current time or interval.
 */
public static int/*@(0, MAX_TIME)*/ time;
/**
 * Current number of loads generated.
 */
public static int/*@(0, NUM_HOUSEHOLDS)*/ numJobs;
```

The microgrid must know, who its users are (line 4). It also remembers the current time (line 8), and can store the total number of loads being executed (line 12).

There are *ranges* on the last two fields – they guard a self–consistency of the model,

as hc is able to produce descriptions containing rigorous checks of the ranges. This helps in testing the specification for invalid assumptions and implementation bugs. The ranges also help in making the model definition more self–explaining, and may also help Prism in faster performing of computations.

The compiler can analyse expressions with ranged operands, in order to estimate the range of the expression's result. Thus, sometimes even a few scattered ranges may help hc iteratively determine the missing ranges. If not found, a default range is assumed, which, for the type int, is $\langle -2^{15}+1, 2^{15}-2\rangle$.

Operations within a model specification can be executed by either a single *main thread*, or in *automaton threads*. The main one is interpreted within hc, before any output is generated by the compiler. That thread initialises variables and creates instances of objects, including thread objects. Any such thread object, created by the main thread and started, is an automaton thread. In hc's output file, an automaton thread translates to a single automaton. The main thread is not translated into any automaton, and only effects of its actions are seen, in e.g. initial values of state variables, in the generated model properties, and, of course, in the very presence of the automatons, as it was the main thread, that initialised and started them.

Like in a regular Java application, the two types of threads may share code, by e.g. calling the same methods. Yet, there are operations that only one of the thread types supports. The divide is mostly intuitive – for example, the main thread is not able to synchronise with other threads, as when it is alive, it is all alone, the automaton threads exists as definitions only. An automaton thread, in turn, can not in general create objects, to fit into some rigid automaton formalism enforced by model checkers.

Note that the main thread is executed by hc's internal interpreter, and so it is not very fast. It is typically speedy enough to create complex topologies of automatons, but might be way too slow for performing intensive computations.

After the main thread ends (or in certain models almost ends, we will return to that later), hc begins to analyse the state of the translated application, including searching for all running automaton threads, which, even that started, did not perform a single operation yet. They will do so, in a sense, only in simulators or in analytic engines of model checkers.

Let us see, what the main thread does, in order to create the microgrid. The main thread always starts with the main method:

Listing 5: Main method of `Microgrid`.

```
1  public static void main(String[] args) {
2      // player ids at the table: 0 scheduler,
3      // 1..NUM_HOUSEHOLDS households
4      TurnGame table = new TurnGame();
5      Model.name(table);
6      // create an array of households
7      households = new Household[NUM_HOUSEHOLDS];
8      for(int i = 0; i < NUM_HOUSEHOLDS; ++i) {
9          Household h = new Household(table, 1 + i);
10         h.start();
11         Model.name(h, "", "" + (i + 1));
12         Model.player(h, "h" + (i + 1));
13         households[i] = h;
14     }
15     // create the scheduler
16     Scheduler s = new Scheduler(table);
17     s.start();
18     Model.name(s);
19     Model.player(s, "scheduler");
20     // reset time
21     time = 0;
22     // players consist of the scheduler and of the
23     // households
24     table.start(1 + NUM_HOUSEHOLDS);
25     for(int t = 1; t <= MAX_TIME/4; ++t)
26         Model.check("at time " + t,
27             "<<1, 2, 3>> R{\"value123\"}max=? " +
28             "[F time=" + t + "]");
29     Model.waitFinish();
30     // anything below won't be executed by hc, but
31     // will within JVM
32     System.out.println("time = " + time);
33     // releases all threads still waiting on the
34     // barrier
35     households[0].interrupt();
36 }
```

In line 4, a helper object from a game package from `hc`'s library is created. Thanks to using that package, some boilerplate code will be avoided in the model. In the next line, one of the methods of the `Model` class is called. The class gives some control over the output file generated by `hc`. In this particular case, it is specified that any field variables belonging to `table` should not be decorated with any prefixes or suffixes in the output files, raw names only. The method `Model.name(Object, String prefix, String suffix)`, and the convenience methods `Model.name(Object, String suffix)`, `Model.name(Object)` make names of state variables in the

output file shorter and predictable by adding a prefix and a suffix to the field name. If such a custom naming scheme is used, though, then the user must arrange the naming so that there are no resulting name conflicts. `Object` in the methods discussed can also be an array reference, in such a case concatenated prefix and suffix alone refer to the referenced array (not to the field).

Local variables, as opposed to non-static fields, do not fall into the naming scheme, and thus have the default, conflict–free yet verbose and unpredictable names. This is usually not a problem as locals are assumed to be generally not used within the properties checked anyway, also because of the way an optimiser may treat a local (e.g. replace it with a constant, specific for some method call). Designate a field if you want to test its values.

The loop in lines 8–14 creates households. The class `Household` is a thread, and thus its creation and starting causes an automaton to be generated. In the loop, there is a yet not discussed method `Model.player()`, that makes an automaton thread a named participant in the game.

There is a number of households, so we append numbers to their field names – otherwise `hc` would complain about duplicates. There is only a single scheduler, so raw field names are enough in its case.

In line 21, a field, and in effect a state variable, `time`, is initialised. In Java, integer fields are initialised with 0 anyway, so the instruction is not needed in fact, but serves as an example of a fragment of the definition of an initial state.

The next operation uses the helper `table` object to start the game. There are $N + 1$ players declared – `hc` will check, if it is a correct value. The value is in fact the number of objects, that use a conditional barrier, which synchronises players – but in the case of a `TurnGame` the value is the same as the number of players.

The statement is the first one executed by the main thread, that would actually made the automaton threads active (in the sense of 'not only waiting on a barrier') if on a JVM. This is why the initial state could still be specified (line 21) despite that the automaton threads had already been started using `Thread.start()`. Because of the discussed assumption within `hc`, that automaton threads never become active before the main thread ceases to be active, a situation where both types of threads *concurrently* interact (by e.g. modifying a variable by a thread of one type and reading that variable by a thread of another type) might result in a disrepancy between the behaviour of the model expected by `hc`, and an actual behaviour of the model on a JVM.

The next thing to do in the main thread is to specify the properties to check. These will be put into a respective file, unless `hc` works in the range–checking mode, in which special range–checking properties are generated instead. As can be seen, `hc`'s interpreter may be a help in generating long sequences of PCTL or rPATL properties.

The statement `Model.waitFinish()` in line 29 waits until some automaton

thread encounters an operation `Model.finish()`. This may seem contradictory to the way `hc` works – as has been said, the compiler exits without interpreting any automaton thread. So, it would need to wait forever in `Model.waitFinish()`. Instead, `hc`'s interpreter treats the statement as a signal to immediately finish interpreting of the main thread – any statement in lines 30–34 is invisible to `hc`. These are only of use if the model is run on a JVM – you may thus specify code, that runs once the automaton threads declare, that the simulation ends – yet note, that consequently `Model.finish()` can only be executed if the automaton threads will no longer modify any variable.

In our case, we use `Model.waitFinish()` to wait until the final simulation time is displayed. Then, we would like to terminate all automaton threads, in order to cleanly end a process. The scheduler ends by itself, but households wait on the barrier. They can be terminated at once by taking advantage of that barrier's property, which says, that if any thread waiting on it is interrupted, then all of these threads enter the interrupted state, as specified in the Java standard, and then leave the barrier. The statement at line 35 starts just that process. A counterpart statement in the definition of a household checks, if the household's automaton thread is in an interrupted state, and if yes, then leaves the loop immediately (Listing 14, lines 5–6).

Because `Microgrid` declares a number of fields that characterise the microgrid in general, let us also put some specifically microgrid–related functions into the class as well. All of the methods, as opposed to the main one, will in our case be only called by the automaton threads.

Let these methods be static. What is static and what is not is often the matter of programming style, not to be discussed here. Let us just mention, that these method were made class ones, because they are utility methods belonging to the main class `Microgrid`, which does not itself have any instances – there is only a single microgrid created. Were there more grids in a single network of automatons, we would make an instance of appropriately rewritten `Microgrid` for each, and `getNumJobs()` would not be static anymore. Were there separate demand curves for each microgrid, the array `DEMAND` would not be static, and in effect, `getDemand()` were an object method as well.

Let us now look at the contents of several methods, that report the microgrid's state.

Listing 6: Other methods of `Microgrid`, part 1.

```
1 /**
2  * Returns the current load demand.
3  */
4 public static double getDemand() {
5     return DEMAND[time%INTERVALS];
6 }
7 /**
```

```
8  * How many households currently generate a load.
9  */
10 public static int getNumJobs() {
11     int/*@(0, NUM_HOUSEHOLDS)*/ numJobs = 0;
12     for(int i = 0; i < NUM_HOUSEHOLDS; ++i)
13         if(households[i].job > 0)
14             ++numJobs;
15     return numJobs;
16 }
17 /**
18  * A price of a load to be bought.
19  */
20 public static double getPrice() {
21     return getNumJobs() + 1.0;
22 }
```

See that the method `getDemand()` not only uses an array, but also returns a variable floating–point value. Both arrays and non–integer variables are not supported in some model checkers, e.g. in `Prism`.

Yet, `hc` has a number of translation techniques, like streamlining the code or emulating arrays, that attempt to remove or replace unsupported elements. The compiler does not always manage, though, to optimise out all `double` variables, partly because there are e.g. strict rules on the order of accessing fields so to not inadvertently modify a model's behaviour.

The next method, `getNumJobs()`, returns how many households generate a load at the current time. There is a range definition on the local `numJobs` – obviously, the number of jobs can never be larger than the number of households. We could also put the same range on the return value of the method, yet that would not change a thing – `hc` finds out by itself, that `getNumJobs()`'s range results from the one of `numJobs`. Yet putting the range *only* on the return value would not be propagated backward to `numJobs` – `hc` does not know, if `numJobs` is allowed to be outside $\langle 0, N \rangle$, before flow of control reaches the `return` statement.

In the next listing, we have two methods related to the end of a single time interval. The comment of the first method describes it precisely. The method will be used by the scheduler to decide, if to end its activity.

Listing 7: Other methods of `Microgrid`, part 2.

```
1 /**
2  * If <code>MAX_TIME</code> has been reached.
3  *
4  * @return if the simulation is finished
5  */
6 public static boolean isFinished() {
7     return time == MAX_TIME;
```

```
8  }
9  /**
10  * Called by the scheduler at the end of each interval.
11  *
12  */
13 public static void endInterval() {
14     numJobs = getNumJobs();
15     Model.stateAnd("measure");
16     numJobs = 0;
17     // reduce job counters
18     for(int i = 0; i < Microgrid.NUM_HOUSEHOLDS; ++i)
19          households[i].tick();
20     ++time;
21 }
```

The second method, `endInterval()`, does several things needed at the end of each time interval. Firstly, it puts the number of jobs i.e. loads of electricity, into a field. It does so only to measure that value by a model checker – this is why `Model.state()` saves a position within an automaton after the field (a fragment of the state vector) already has the current value, assigning a label to reference that position. Another purpose of the method `endInterval()` is to decrease by one the counters, that represents lengths of time that are left each load.

Let us browse the thread classes of each player, beginning with the scheduler.

Listing 8: Class `Scheduler`.

```
1 class Scheduler extends TurnPlayer
```

`Scheduler` is a subclass of `TurnPlayer`, which is provided by the game package, and which itself is a subclass of `Thread`.

Listing 9: Fields of `Scheduler`.

```
1 protected final static boolean NON_DETERMINISTIC = false;
2
3 protected final Random strategy;
```

The scheduler (not to be confused with `hc`'s schedulers) is normally probabilistic, but it also has a modifier, that can make it non–deterministic instead – if `NON_DETERMINISTIC` is true, a strategy is decided by a `Random` object. `Random` is the simplest case of declaring strategy. Special wrappers for writing custom choice methods, or for declaring a common strategy, exist in `hc`'s library.

The scheduler has also a constructor, called by the main thread:

Listing 10: Constructor of `Scheduler`.

```
1 public Scheduler(TurnGame table) {
2     super(table, 0);
3     strategy = new Random();
4 }
```

The statement in line 2 passes to the super-class `TurnPlayer` a reference to the game in which a player participates, and also the player's number. The number is required by the logic rPATL. Here, the scheduler is assigned 0. The households declare in their constructors subsequent numbers $1, 2, \dots N$.

There is yet another method in `Scheduler` – `run()`, which is a top method of a thread, and thus also a top 'method' of the respective automaton:

Listing 11: Methods of `Scheduler`.

```
1  @Override
2  public void run() {
3      do {
4          // select a household
5          int address;
6          if(NON_DETERMINISTIC)
7              // non--deterministic choice
8              address = strategy.nextInt(Microgrid.
9                  NUM_HOUSEHOLDS);
10         else
11             // probabilistic choice
12             address = (int)
13                 (Math.random()*Microgrid.NUM_HOUSEHOLDS);
14         // contact the selected household
15         turnNext(1 + address);
16         turnWait();
17         // end of the current round
18         Microgrid.endInterval();
19     } while(!Microgrid.isFinished());
20     Model.finish();
21 }
```

The constant `NON_DETERMINISTIC` is set within the main thread, so at the time of generating this automaton, it is exactly known which branch of the choice in lines 6–13 is unconditionally executed, and which branch is dead. The compiler takes use of that knowledge and removes completely the condition at line 6 and one of the branches from the automaton.

There is a probabilistic choice in lines 12–13. As discussed, the exact behaviour of that kind of choice, as opposed to the non–deterministic one, is known – in this case, draw a natural number in the range $0, 1, \dots N - 1$, using an uniform probability

distribution. Thus, if `NON_DETERMINISTIC` is false, the scheduler, even that it is a player, has only a single, hard–wired strategy, defined in the very model.

After a household is selected by the scheduler by setting the local variable `address`, that exact household has a next move. The next player to move is declared by the player that currently moves, using a super-class' method `turnNext(`i`)`, where i is the player's number (line 15). After the scheduler selects the next player to move, it waits for its own move using `turnWait()` (line 16). When a time of its move comes again, the discussed `Microgrid.endInterval()` is called.

The scheduler's loop's condition in line 19 calls a method `Microgrid.isFinished()`, which returns, if to end the simulation. If yes, then the already mentioned method `Model.finish()` is called and the scheduler ends. For a model checker it means, that the scheduler's automaton will stay in some final state forever. For a JVM it means, that the scheduler's thread will return from the method `run()`, and thus will terminate.

The class of a household should now be self–explaining. A number of constants, directly related to a household, is declared here, and not in `Microgrid`:

Listing 12: Fields of `Household`.

```
1  /**
2   * Maximum time of running a single job, in intervals.
3   */
4  protected final static int MAX_JOB_TIME = 4;
5  /**
6   * Expected number of jobs per day.
7   */
8  protected final static int EXPECTED_JOBS = 9;
9  /**
10  * Price limit, above which this household may
11  * back--off.
12  */
13 protected final static double PRICE_LIMIT = 1.5;
14 /**
15  * Probability of starting a task independently of the cost.
16  */
17 protected final static double P_OVER_LIMIT = 0.8;
18 /**
19  * A running job, in intervals.
20  */
21 public int/*@(0, MAX_JOB_TIME)*/ job = 0;
22
23 protected final Random strategy;
```

In particular, the constant c_{lim} is represented by `PRICE_LIMIT`, and similarly, P_{start} is implemented as `P_OVER_LIMIT`. There is also `job` variable. If zero, the

household does not generate a load. If non–zero, then the field expresses the time left of generating the load bought, in intervals.

A household has a constructor, executed by the main thread, like it was in the case of `Scheduler`:

Listing 13: Constructor of `Household`.

```
1 public Household(TurnGame table, int playerNum) {
2     super(table, playerNum);
3     strategy = new Random();
4 }
```

and, as `TurnGame` is a thread, the `run()` method:

Listing 14: Run method of `Household`.

```
1  @Override
2  public void run() {
3      while(true) {
4          turnWait();
5          if(isInterrupted())
6              break;
7          if(job == 0 && Math.random() < Microgrid.getDemand()*
8                EXPECTED_JOBS) {
9              // decide if to generate a load
10             boolean lowPrice = Microgrid.getPrice() < PRICE_LIMIT;
11             if(lowPrice || Math.random() < P_OVER_LIMIT ||
12                     // a right to back--off is granted; decide,
13                     // if to generate a load anyway
14                     strategy.nextInt(2) == 0)
15                 job = 1 + (int)(Math.random()*MAX_JOB_TIME);
16         }
17         turnNext(0);
18     }
19 }
```

In lines 5–6, the already discussed loop exit statements are seen. They have no effect on an automaton generated by `hc`, as neither `hc`'s interpreter, nor automatons themselves, can cause an interrupt. The sole reason of these statements is to take part in stopping a Java application on the JVM side.

In line 7, there is another variant of a probabilistic choice – `Math.random() < Microgrid.getDemand()*EXPECTED_JOBS`. This time a boolean condition is evaluated, by comparing a random value, drawn using an uniform distribution, to a variable value, that here translates to a probability of this `if`'s sub-condition becoming true.

There is yet a third, rather self–explaining method in `Household`, called by `Microgrid.endInterval()` (the call statement is located in Listing 7, line 19)

on each household whenever a time interval ends.

Listing 15: The other method of `Household`.

```
1 /**
2  * Called by the scheduler after each turn.
3  */
4 public void tick() {
5     if(job > 0)
6       --job;
7 }
```

The source code we studied is enough to generate a model. In fact, it also generates a series of properties in a probabilistic logic. But model checkers may have also other features, not explicitly supported by `hc`. A *verbatim* section may help in using these – its contents is simply appended to the output model file as is, if the output format allows:

Listing 16: Appending to the output model file.

```
1  /*@modelAppend(
2
3  rewards "value1"
4          measure & job1>0 : 1/numJobs;
5  endrewards
6
7  rewards "value12"
8          measure & job1>0 : 1/numJobs;
9          measure & job2>0 : 1/numJobs;
10 endrewards
11
12 rewards "value123"
13         measure & job1>0 : 1/numJobs;
14         measure & job2>0 : 1/numJobs;
15         measure & job3>0 : 1/numJobs;
16 endrewards
17
18 )*/
```

Here, reward structures for `Prism` are seen. The term `measure` refers to a formula generated by `hc`, thanks to the statement in line 15 of Listing 7. We will return to that formula later.

So, let us generate `Prism`–compliant files, but before, there is the question which *scheduler* should be chosen by `hc` (that scheduler has nothing to do with the class `Scheduler`).

Schedulers, as mentioned, define time invariants on states, but in effect also, what kind of order of operation is possible within a network of automatons. There is the default scheduler that leaves an automaton as is – with that scheduler applied, any state,

by default, e.g. without delay operations like `Thread.sleep()`, may last any time from 0 to infinite, unless some *fairness* conditions are imposed by a model checker. This is a good scheduler for finding bugs in a concurrent code – as opposed to what a real operating system does, any possible order of execution is taken into account. We could use such a scheduler for our model, but it is more than we need here – the model is a turn game, and so there is no non–determinism confined within the order of activity of the automatons, that could be altered if e. g. some states would be merged between synchronisation points. We can give a hint to `hc` about that, by using a scheduler that makes each non–synchronising state *implicitly hidden* – implicitly, because there is no extra information put into the output model, like time invariants, and only `hc` knows, that a certain interference is impossible, and that in turn grants more freedom to the compiler's optimiser, which may be more aggressive in performing a partial order reduction [5], what in turn may result in smaller, more compact automatons.

Let us run `hc`:

```
$ hc -sh -i -op -v0 Microgrid.java
```

In the case of a `Prism` model, there are two files generated: `Microgrig.nm` contains a model of a `Prism`'s type `smg`, because players were defined, and `Microgrig.pctl` contains the properties to check.

The output model file will contain formulas that represent states saved with `Model.state()`, and also all final static fields, extracted from the model. These can be, thus, used in properties or in the appended verbatim section.

Listing 17: Constants in the model file.

```
1  formula measure = (s3=8);
2
3  const int DAYS = 3;
4  const int EXPECTED_JOBS = 9;
5  const int INTERVALS = 16;
6  const int MAX_JOB_TIME = 4;
7  const int MAX_TIME = 48;
8  const int NUM_HOUSEHOLDS = 3;
9  const double PRICE_LIMIT = 1.5;
10 const double P_OVER_LIMIT = 0.8;
```

The formula `measure` is added thanks to the call at line 15 in Listing 7. As `turnEnd()` is called by the scheduler, the formula points to the local variable of the automaton `Scheduler`.

Naming conventions are not applied to the constants, even that they are fields, as these fields are static, an thus are not owned by any object passed to `Model.name()`. Instead, a field's name is directly used, and in the case of a name conflict, the name is prefixed with that of a respective class.

In the case of an output for Prism, the compiler generates yet another file, a one with properties, whose initial fragment is as follows:

Listing 18: Properties – a fragment.

```
1 // at time 1
2 <<1, 2, 3>> R{"value123"}max=? [F time=1]
3 // at time 2
4 <<1, 2, 3>> R{"value123"}max=? [F time=2]
5 // at time 3
6 <<1, 2, 3>> R{"value123"}max=? [F time=3]
7 // at time 4
8 <<1, 2, 3>> R{"value123"}max=? [F time=4]
```

As seen, the expressions make use of player numbers declared in constructors of automaton threads, of a reward structure appended to the model as seen in Listing 16, and of a state variable time, declared in Listing 3. A property that checks for a total expected value of a household after the simulation ends might contain time=MAX_TIME.

6.2. Example of range checking

Let us shorten the array with the demand curve by one element, so that its new size is 15, and thus not as much as the number of time intervals per day.

Listing 19: An array one element too short.

```
1 public static final double[] DEMAND = {
2     0.0614, 0.0392, 0.0304, 0.0304,
3     0.0355, 0.0518, 0.0651, 0.0643,
4     0.0625, 0.0618, 0.0614, 0.0695,
5     0.0887, 0.1013, 0.1005, /* 0.0762, */
6 };
```

The array is indexed with time % INTERVALS, as seen in line 5 of Listing 6. The variable time can be as large as MAX_TIME = 48, and that maximum value (amongst some other values), modulo INTERVALS, is greater than the new array size. Clearly, indexing of DEMAND would be incorrect for certain time intervals. The compiler, though, is not able to prove in this case, if time *actually* has values, at the time of indexing the array, that make the indexing invalid – it is not a model checker anyway. It concludes, however, that its limited proving algorithm is not able to rule out such a possibility. Thus, it gets suspicious enough to decorate the transition, at which the indexing occurs, with an additional guard. Because of the same reason, hc also creates an additional transition, that is triggered in the case of an invalid indexing. The following fragments of the output model file show these two transitions:

Listing 20: Accessing an array one element too short.

```
1 [] (s0=2) & ((mod(time, 16))<=14) ->
2     ((DEMAND_index_nested__lmod_ltime_C_w16_r_r) * 9):(s0'=3) &
3         (Household114runMicrogrid_getNumJobs_numJobs' = 0) +
4     (1.0 - ((DEMAND_index_nested__lmod_ltime_C_w16_r_r) * 9)):
5         (s0'=11);
6 [] (s0=2) & ((mod(time, 16))>14) -> (error'=1);
```

Let us include, for completeness, the formula generated to emulate indexing of the array:

Listing 21: Example formula for accessing the array.

```
1  formula DEMAND_index_nested__lmod_ltime_C_w16_r_r =
2      (mod(time, 16))=0 ? 0.0614 :
3      ((mod(time, 16))=1 ? 0.0392 :
4      ((mod(time, 16))=2 ? 0.0304 :
5      ((mod(time, 16))=3 ? 0.0304 :
6      ((mod(time, 16))=4 ? 0.0355 :
7      ((mod(time, 16))=5 ? 0.0518 :
8      ((mod(time, 16))=6 ? 0.0651 :
9      ((mod(time, 16))=7 ? 0.0643 :
10     ((mod(time, 16))=8 ? 0.0625 :
11     ((mod(time, 16))=9 ? 0.0618 :
12     ((mod(time, 16))=10 ? 0.0614 :
13     ((mod(time, 16))=11 ? 0.0695 :
14     ((mod(time, 16))=12 ? 0.0887 :
15     ((mod(time, 16))=13 ? 0.1013 :
16     0.1005))))))))))))));
```

Note that the expression `time % INTERVALS` is embedded within the formula. It is an example of `hc`'s attempts at simplifying the vector state when generating to `Prism`.

6.3. Locals and fields

The compiler does not treat local variables very seriously, as it assumes, that only the field variables are generally checked. Yet, it does not mean that locals are always optimised out. For example, the source model might store the contents of a field in a local, and then the local might be read multiple times. `hc` could get rid of the local and make the generated automaton read the field multiple times instead. Yet, the compiler is careful in such situations – it does not know, if the field is modified by another automaton in the meantime. `hc` would not even join field accesses from two serial transitions into a single transition, as the merged transitions would mean *accessed at the same time instant*.

The compiler can be much more lax in the case of locals. See the update in the following listing.

Listing 22: A reset of a local variable.

```
1 [] (s0=6) & (1.5 <= (Household114runMicrogrid_getNumJobs_numJobs
2     + 1.0)) -> (s0'=7) &
3     // reset
4     (Household114runMicrogrid_getNumJobs_numJobs' = 0);
```

The long name indicates a local variable. The name contains an instance of an object, and also a trace of subsequent method calls. See that the variable is read, and then immediately *reset* to its initial value. This is because `hc` found out, that the variable does not transport a meaningful value any longer. The resets may improve the performance of `Prism` or other checkers, because assigning a single constant value to a variable may limit state explosion. But the resets also cause, that if you want to check for a local in a model property, then beside determining the variable's long name, you should also analyse the output file. Especially that `hc` may merge a set of local variables into a single one.

As field variables are never reset by `hc`, it is sometimes advantageous to do that by hand. See Listing 7, line 16, for an example of a source–level reset.

6.4. Analytic and statistical versions

Models that work best with `Prism`'s analytic engines may differ from these that work better with simulators. This is why `hc` provides a method `Model.isStatistical()` in its library. The method's default return value is false within `hc`, but true within a JVM. You may use that function in the model's source to tune that model to the way its properties will be computed by a model checker.

Let's modify `Microgrid.java` to adapt to `Model.isStatistical()`.

In the case of `Prism`, it can be more important to reduce the number of states. In the cases of JVM or of a simulator, reducing the number of operations may matter more. The original model aimed at the former – this is why `Microgrid.numJobs` was reset within the scheduler (Listing 7, line 16), so that households had to recompute exactly the same value. Good for `Prism`'s game engine, but not necessarily for a Monte–Carlo simulator.

Let us modify the method `Microgrid.newInterval()`, so that `numJobs` is not zeroed in the case of the statistical variant:

Listing 23: A conditional zeroing of `numJobs`.

```
1  public static void endInterval() {
2      numJobs = getNumJobs();
3      Model.stateAnd("measure");
4      if(!Model.isStatistical())
5          numJobs = 0;
6      // reduce job counters
7      for(int i = 0; i < Microgrid.NUM_HOUSEHOLDS; ++i)
8          households[i].tick();
9      ++time;
10 }
```

Now, a household, after the scheduler's first move, may read the initial value of `numJobs`, which is correctly zero, as no jobs had been generated so far. In the subsequent moves of households, the number of jobs is written to the field by the scheduler. The state space grows, but households may now just read the value in question, instead of computing it by itself. Let `getPrice()` be modified to enable that:

Listing 24: New version of `getPrice()`.

```
1 public static double getPrice() {
2     int n;
3     if(Model.isStatistical())
4         n = numJobs;
5     else
6         n = getNumJobs();
7     return n + 1.0;
8 }
```

And that's it – the automatons will generated as in the previous version of `Microgrid.java` if `hc` is not supplied with the option `-S`, that makes `Model.isStatistical()` true. On the JVM, in turn, a simulator–tuned version will be run, unless `Model.isStatistical()` is made false by setting a system property `hc.statistical=false`, e.g. by a JVM's `-D` option.

As simulators are often used specially to run large models, let the number of households in the grid be dependent on `Model.isStatistical()` as well:

Listing 25: Number of households made dependent on the model version.

```
1 public static int NUM_HOUSEHOLDS =
2     Model.isStatistical() ? 5 : 3;
```

Note that the field is not more final. This is because `hc` is more strict on final variables (also the blank ones) than a standard Java compiler – `hc` won't allow, that

two values of a final variable are ever seen. This is why the compiler analyses the code, that computes a final variable's initial value for possible reads of the same final variable. In order to do that, `hc` recurses through method calls amongst other. Yet, the compiler analyses exclusively these methods that belong to the class, to which also belongs a given final variable. Any methods of foreign classes are thus forbidden in declarations of finals. In our example, `NUM_HOUSEHOLDS` belongs to `Microgrid`, while `isStatistical()` belongs obviously to `Model`. In effect, we need to make the discussed field non–final, or use an intermediate helper variable, if `NUM_HOUSEHOLDS` needs to be final, e.g. to be present in the section of constants in the output file.

Note that a non–final `NUM_HOUSEHOLDS` won't be added to the state vector, as `hc` finds out, that the variable is never written to within the automaton threads, and thus can be seen by the automatons merely as a constant.

6.5. External constants

Sometimes it is desired to define a constant value not within the model sources, but at the command line, of either `hc` or a model checker. Let us modify `Microgrid.java` once again to use external constants.

Let there be two constants, whose values are missing from the model specification:

Listing 26: Number of households is an undefined value.

```
1 /**
2  * Number of households.
3  */
4 public static int NUM_HOUSEHOLDS = Model.intConst("HOUSEHOLDS");
```

and

Listing 27: Price limit is externally definable as well.

```
1 /**
2  * Price limit, above which this household may
3  * back-off.
4  */
5 protected final static double PRICE_LIMIT =
6     Model.doubleConst("PRICE_LIMIT");
```

The field `Microgrid.NUM_HOUSEHOLDS` must be set at `hc`'s command line, as the main thread makes use of that field's value:

```
$ hc Microgrid.java --const HOUSEHOLDS=3~9:2 -op -v0 -sh -i
```

See that there is a set of values that specifies the number of households: 3 to 9 with the step of 2. If there is at least a single constant with multiple values, `hc` changes the

structure of the output files into that of an *experiment mode*: a directory is created, and for every possible combination of constants a separate model is created. A file `key.txt` contains the key to the naming of the generated files.

The field `Household.PRICE_LIMIT` does not need to be specified during `hc`'s work, but it needs to be specified somewhere before the model can be computed, for example at `Prism`'s command line:

```
$ prism -const PRICE_LIMIT=1.5 Microgrid/3.nm
  Microgrid/3.pctl
```

If the model is run on the JVM, the constants should be defined as system properties whose names begin with `hc.const.`, e. g.

```
$ java -Dhc.const.HOUSEHOLDS=3 -Dhc.const.PRICE_LIMIT=1.5
  -cp $HC_HOME/lib/hc.jar:. example/Microgrid
```

7. Discussion

Model checkers are able to verify more and more advanced systems, not only in the sense of size, but also in the sense of automaton formalisms supported. Such systems may require more complex implementation, where a simple specification language is not enough. The current version of `hc` attempts to fulfil some of the new requirements – generation of additional classes of models is available, like `Prism`'s `pta` and `smg` ones, and ranges on variables and expressions help in both self–verification of large models, and in time– and memory–efficient analysis of them by model checkers.

Aknowledgement

This work has been supported by Polish Ministry of Science and Higher Education project N N516 407138, “Metody i narzędzia rozproszonego modelowania sieci bezprzewodowych”.

References

[1] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *Lecture Notes in Computer Science*, volume 3185, pages 200–236. Springer, 2004.

[2] H. Hildmann and F. Saffre. Influence of variable supply and load flexibility on demand-side management. In *Proc. 8th International Conference on the European Energy Market (EEM’11)*, pages 63–68, 2011.

[3] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[4] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. In Y. Lakhnech and S. Yovine, editors, *Proc. Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant Systems (FORMATS/FTRTFT'04)*, volume 3253 of *LNCS*, pages 293–308. Springer, 2004.

[5] D. Peled. Ten years of partial order reduction. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer Berlin Heidelberg, 1998.

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

[7] A. Rataj, B. Wozna, and A. Zbrzezny. A translator of Java programs to TADDs. In *Concurrency, Specification and Programming (CS&P 2008)*, pages 524–535, Gross Vaeter near Berlin, Germany, 2008.

Translacja gier probabilistycznych w J2TADD

Streszczenie

Artykuł prezentuje nową wersję translatora J2TADD. Dodane zostało tłumaczenie procesów markowowskich z niedeterministycznymi graczami, mogącymi formować koalicje mające różne cele. By ułatwić pisanie gier probabilistycznych dodane zostało kilka specyficznych dla gier konstrukcji, jak również specjalna biblioteka.

Aktualnej wersja posiada również kilka innych usprawnień:

- wybory, które są zwykłymi wyrażeniami języka Java, jednak `hc` tłumaczy je na specyficzne dla automatów rozgałęzienia probabilistyczne lub niedeterministyczne;
- można definiować dopuszczalne wartości zmiennych, co pomaga w sprawdzaniu wewnętrznej spójności modelu, a także może przyspieszyć jego rozwiązanie;
- różne metody specyfikacji niezmienników i warunków zegarowych.

Artykuł prezentuje jako przykład prostą grę probabilistyczną, modelującą rynek lokalnego dostawcy energii elektrycznej. W ramach przykładu omawiane są wersje automatów do rozwiązywania metodą analityczną i symulacyjną.