Challenges in simulating railway systems using Petri Nets

Joakim Bjørk, Anders M. Hagalissetto

Precise Modeling and Analysis,
Department of Informatics, University of Oslo

Abstract. In this paper we report our experiences by simulation of realistic railroad systems, based on Petri Nets. The railroad layout serves as a specification layer, from which we generate Petri Nets automatically. The Petri nets are then translated automatically to Maude code. The challenges in time and space complexity are hard, since railroad models are huge.

Keywords: railway systems, Petri Nets, simulation, case study

1 Introduction

In several papers [7], [4], [5], [3] we have explored the advantages of using Petri Nets as a modeling language for railway systems. We have focused on the structure of the railroad layout and how various behaviors like safety collision detection, sensitivity can be hard-coded into the Petri Net components and how these libraries of components relate through a syntactic refinement relation.

In this paper we report our experiences with simulating trains on a realistic model of Oslo Subway, that is isomorphic to the real railroad layout. At the Grand Challenge Workshop on the Railway Domain, Formal Methods, 19th of July 2005, we were asked from the audience how effective a simulation based on our approach would be. Frankly, we did not have any idea, and hoped the best. This paper addresses the question directly, by giving detailed measurements by metrics both in the size of the net and the number of trains on the line. Railway systems are large concurrent systems, and are by their nature hard to handle computationally. Feasible simulations require careful design of the railroad models and the underlying execution structure. One goal of our research is to conform to Dines Bjørner’s “Grand Challenge on the Railway Domain”\(^1\),

---
\(^1\) See FMRail http://www.railwaydomain.org/.
by giving generic components and methods for construction, analysis and standardization of notation in the railroad domain [1].

At the moment the standard Petri Net tools fail to support industrial applications of railroad models in several ways; editing of nets gets unfeasible because of the Petri Net hierarchies, the size of nets exceeds the allowed usage of memory, and the execution time for standard tools like Design/CPN\(^2\), Renew\(^3\) are not applicable even for small railroad nets. These tools work well for small Petri nets, but when even medium-sized railroads are represented, they quickly fail.

This paper is organized as follows: First in section 2, we give some technical background by introducing the appropriate class of Petri Nets, and give a short introduction to Maude. Then in section 3 the architecture of the simulator is presented. The translation from Petri Net to Maude is carried out by a small example, the turnout component in section 5. Implementation plays a crucial role for the computational complexity as discussed in section 6. Then finally in section 7, a case study involving three parts of Oslo Subway is considered; (i) the downtown piece of layout from Majorstuen to Grønland, (ii) the line 4 from Kolsås to Bergkrystallen and (iii) the whole system.

2 Technical preliminaries

In this section some technical preliminaries are given, including some functions, the definition of Petri Nets, and a brief explanation of Maude. If \( S \) is a set, then \(|S|\) measures the size of \( S \), and is defined by \(|\emptyset| = 0, |\{x\} \cup S'| = 1 + |S'|\). If \( n \) and \( m \) are numbers, then \( \max(n, m) = \begin{cases} n & \text{if } n > m \\ m & \text{else} \end{cases} \). A multiset \( Z \) is a pair \( Z = \langle A, f \rangle \), where \( A \) is a set and \( f \) is a function \( f : A \rightarrow \mathbb{N} \). The size of a multiset \( Z \), denoted \(|Z|\), is given by \( \sum_{(a,n) \in f} n \).

2.1 A short introduction to Petri nets

Petri net is a graphical programming language. A Petri Net is a triple, \( \langle P, T, A \rangle \), where \( P \) is a finite set of places, \( T \) is a finite set of transitions and \( A \) is a finite set of arcs, \( A \subseteq (P \times T \times E) \Delta (T \times P \times E) \). \( E \) denotes a set of expressions that can be boolean, assignments and time inscriptions. Composite expressions are permitted only as conjunctions. A transition is a 4-tuple \( \langle \text{TransId}, \text{Name}, \text{Sort}, \text{Expr} \rangle \), where TransId is

\(^2\) For details see http://www.daimi.au.dk/designCPN/.

\(^3\) For details see http://www.renew.de.
its unique identity, Name its name, Sort specifies the sort, and Expr is an expression, also called guard in [6]. The guard is a boolean expression. The guard may be empty and is then evaluated to true. A place is a triple \(\langle \text{PlaceId}, \text{Colour}, \text{Sort} \rangle\), that contains a unique identity, a color and a sort. Tokens might inhabit places. Suppose that \(N = \langle P, T, A \rangle\) is a Petri Net and that \(t \in T\) is a transition. The \textit{preset} of the transition \(t\), denoted \(\bullet t\), is the set of places with arcs pointing at \(t\): \(\bullet t = \{ p \in P \mid \exists e(\langle p, t, e \rangle \in A) \}\). The \textit{postset} of the transition \(t^*\) is the set of places with arcs from \(t\): \(t^* = \{ p \in P \mid \exists e(\langle t, p, e \rangle \in A) \}\). A \textit{marking} \(M\) of a Petri Net \(N = \langle P, T, A \rangle\) is a distribution of tokens over places. Let \(\mathcal{M}\) denote the set of possible multi-sets over the available tokens. Formally, a \textit{marking} of \(N\) is set of pairs of places and multi-sets of tokens, \(\langle \langle p, m \rangle \mid p \in P \land m \in \mathcal{M} \rangle\). The \textit{number of tokens} \(p(\text{tok})\) of a place \(p\) in a marking \(M\) is \(|m|\) if \(\langle p, m \rangle \in M\), else undefined. A net is \textit{1-safe} if \(|m| \leq 1\) for all \(m \in \mathcal{M}\).

Figure 1 shows a small Petri net representing a track segment of a railroad system. The circles in the figure denote places and the rectangles represents transitions. The arrows between places and transitions denote arcs, while tokens are drawn as filled circles inside the places.

A transition is enabled if all the requirements of the arc expressions can be met and the guard evaluates to true. In the Petri net of figure 1 the only enabled transition is \texttt{move+}. If this transition fires, tokens will be removed from the places according to the arc expressions of the arcs pointing at the transition. This means that a train in the positive direction is removed from the left place and a \texttt{noTrain} token is removed from the right. New tokens are added according to the expressions on the arcs pointing from the transition. So a no train is added to the left place and a train in the positive direction is added to the right place. This simulates a train moving from left to right in a track segment. The net in figure 1 are 1-safe, all the nets in this paper are. The no train token is a construction
to make sure that two train tokens can not be in the same place at the same time.

2.2 A short introduction to Maude

Maude is a functional language based on rewriting logic. Rewriting logic interpret inferences as transition rules in a possibly concurrent system. A typical Maude program contains functions, equations and rewriting rules. In Maude all functions are declared recursively by equations, for more details see [2]. Maude has no pointers or aliasing and thereby no side-effects. Hence both local and global properties can be proved by examination of each equation or rule independently.

The static parts of a system are described by equations. Equations in Maude are on the form:

\[ \text{eq } t_1 = t_2 . \]

This means that the term \( t_1 \) can be reduced to the term \( t_2 \). The rules describes the dynamic parts of a system. The syntax of the rules in Maude are:

\[ \text{rl [label] } t_1 \Rightarrow t_2 . \]

This means that the term \( t_1 \) can be rewritten to the term \( t_2 \) in one rewrite step. The label is the name of the rule. There are also conditional equations and rules.

There were several reasons why Maude was chosen as the execution language for the railroad models. First, since the language is based on term rewriting, the translation from Petri Nets to Maude is rather direct. Second, Maude executes the generated specifications directly, and efficient. Finally, Maude permits the user to search in the state space from a given initial state, thus it has a built-in model checker.

3 The Architecture of the simulator

In this section we outline the main design of our tool. The results for this paper only regards off-line simulation. A finite railroad simulation is a sequence of states from a given initial state to an end state. A finite railroad simulation is an off-line simulation if the user of the simulator only knows explicitly the initial state and the final state. A railroad simulation is called on-line if the simulation provides information about each state in the simulation. As figure 2 shows the system involves several modules. De-
sign/CPN is a program for designing and executing Petri nets. For more details on Design/CPN see [8]. RWSEditor lets the user design a specification and saturate it with atomic Petri nets. The module Xml2Maude is translates a Petri net (saved as a XML file) into a Maude module. The module Xml2Maude handles both list-markings and set-markings. The shaded part of the figure is under development.

The process of setting up a simulation is as following: The first phasis is manual. First, the atomic Petri net components are constructed, such as the turnout in figure 4, by using Design/CPN. Then RWSEditor is used to draw a specification such as the downtown layout in figure 6. The specification is thereafter saturated with the atomic Petri net components previously constructed[5]. This involves connecting the interfaces of the specification with places of the atomic Petri nets, whereby RWSEditor constructs a large Petri net of the entire specification automatic and saves the net to a file. Next the translator module Xml2Maude generates a Maude module of the entire Petri net. Finally Maude is used to execute the Maude module off-line.

4 Mapping railroad components to Petri nets

The modeling principles and the mapping of the railroad components is illustrated by considering the turnout component. The turnout component is equipped with movable points and makes it possible for trains to be routed to either of two tracks whenever they enter the component from the stem. This routing is determined by the positions of points which
are often operated automatically by a point machine [10]. The turnout component can be combined to construct other components. The turnout is an example of a component called a router, since there are more than two entrances into the component. A detailed list of components used in the modeling of Oslo Subway, as well as the Petri net mapping can be found in [5].

4.1 Modeling the turnout as a Petri net

Figure 4 shows a petri net model of a turnout component. This component has few elements and a simple behavior. The trains has a parameter that indicates the direction, and the model prevents two trains from occupying the same place. The movable points of the turnout is set, and it is not possible for a train to drive to the left in a turnout if the points will guide it to the right, and vice versa. The position of the points will be referred to as the state of the turnout. Trains in the model have no idea where they are going, and the model has no notion of time.

The turnout consists of two track segments, such as the net in figure 1, merged in the left places. If the point machine was omitted, it would be arbitrary which way a train took from the join place, if it was going in the + direction. The semaphores of the places R and L keeps track of the state of the turnout. The point machine can be operated by adding a semaphore token to the Change place. Then either the SetR transition or the SetL transition will become enabled. The firing of one of these transitions will always change the state of the turnout.

There is only one enabled transition in figure 4, the transition Ldir+. If this transition fires, the train token is consumed from the Join place, the noTrain token from the Left place, and the semaphore token from the L place. Furthermore, there will be added a noTrain token to the Join place, a train token to the Left place and a semaphore token to the L place. This simulates that a train has moved to the left in a turnout segment.
Fig. 4. A Petri net representation of a safe turnout.

Fig. 5. The safe turnout controlled by trains.
Figure 5 shows a refined version of the turnout in figure 4. In this component, all trains have a travel plan, and they will change the state of turnouts if they have to. All the blue arcs lead a noTrain token, the green arcs lead a semaphore token and the purple arcs lead an id token. The RId place contains an RouterId token which is holding the unique identity of this turnout. All the trains are equipped with a travel plan. Such a travel plan consists of pairs of router identities and a given direction, left or right. The expressions included in brackets close to some of the transitions are guards and must evaluate to true before the transition is enabled. The expression (RouterId, left) in Train.plan evaluates to true if the pair (RouterId, left) is in the travel plan of the train token.

If the train in the Join place of figure 5 is supposed to turn left in this turnout, then the Ldir+ transition will be enabled just as in figure 4. On the other hand if it is supposed to go to the right the transition TcJR is enabled. The firing of this will result in a change of the state of the turnouts. The TcJR transition is only enabled if there is no more than one train in the component. This is to prevent derailing if one train tries to change the state of the turnout while another is driving through it.

5 Translating Petri Nets to Maude

There are probably many ways to translate a petri net into a Maude module. We have focused on two approaches. They will be referred to as set-marking and list-marking. The names reflect the way markings are represented.

Set-marking In [9] Ölveczky suggests a translation where all the tokens with their position are stored in a set. The declaration of the initial state of the Petri Net in figure 4 will be:

\[
\text{eq initState} = \text{Join(train(+)) Left(noTrain) Right(noTrain) L(sem)} .
\]

The transitions will naturally be translated into rewrite rules. For instance will the transition Ldir+ become the rule:

\[
\text{rl [Ldir+] : Join(train(+)) Left(noTrain) L(sem) } \Rightarrow \text{Left(Train(+)) L(sem) Join(noTrain)} .
\]

\[
\text{4 TcJR is short for Train controlled point machine for a train in the Join place that is going to the Right place}
\]
The rule above is enabled if the current marking contains at least one occurrence of each of the three tokens in the left hand side of the rule. If the rule is executed, each of the tokens on the left hand side is replaced by the tokens on the right hand side of the rule.

**List-marking** In an attempt to avoid extensive pattern matching we have proposed a translation where all places are stored in a list. All the places then contains a set of tokens. This could be a list as well, but all our nets are 1-safe so it does not matter. The initial state of the net in figure 4 will then be:

\[
\text{eq initState} = (\text{Join(train(+)) Left(noTrain) Right(noTrain)} \quad L(\text{sem}) \quad R(\text{sem}) \quad \text{Change(null)})
\]

The rule for the transition $Ldir+$ will then be:

\[
\text{rl [Ldir+] : (Join(train(+)) Left(noTrain) Right(toks) \quad L(\text{sem}) \quad R(toks') \quad \text{Change(toks'')} \Rightarrow (Join(noTrain) Left(train(+)) Right(toks) \quad L(\text{sem}) \quad R(toks') \quad \text{Change(toks'''})}
\]

where $tok$, $tok'$ and $tok''$ are variables of token set. Note that all the places are represented in the rule, not only the ones influenced by the transition, as is the case for set-marking.

### 6 Computational complexity

In this section we discuss the computational cost both with respect to time an space, with respect to choosing either the set marking or list-marking approach.

The problem with the *set-marking* approach is that when the number of tokens get large there will be extensive pattern matching. Let $|tok|$ be the number of tokens in the system and let the transition $t$ have $|*i|$ input places and $|*o|$ output places. Since the Petri nets are 1-safe there is never more than one arc from a place to a transition and each arc always lead one token. Hence then the transition $t$ consumes exactly $|*i|$ tokens when it fires. The maximum number of pattern matches that have to be performed to verify whether a transition $t$ can fire is given by the expression:

\[
\frac{|tok|!}{(|tok| - |*i|)!}
\]

(1)
In a Petri Net $N = \langle P, T, A \rangle$, the number of times one have to pick an element from the set of tokens to be able to determine whether any transitions are enabled, will be given by the formula:

$$\sum_{t \in T} \frac{|tok^t|}{(|tok| - |\bullet t|)!}$$  \hspace{1cm} (2)

Expression (2) is the sum of expression (1) for all transitions $t \in T$.

If all names, and tokens have same size, the file size of a Petri net stored as a set-marking will be:

$$|P|*k_3 + |tok|*k_4 + k_5 + \sum_{t \in T} ((|t^*| + |t^*|)*k_1 + k_2).$$  \hspace{1cm} (3)

where all $k_i$ are constants. The part marked I measures the size declaration of places, which is a constant times the number of places. The part marked II measures the definition of the initial state. As shown at the beginning of this section all tokens are written in the definition of the initial state. The constant part of this definition is included in the miscellaneous part of expression (3), marked III. This is definitions and declarations not influenced by the size of the net. The part of expression (3) marked with IV gives the size of the generated rules. The size of each token in a rule is given by the constant $k_1$. To find the total size for a rule, just multiply the constant $k_1$ with the number of tokens in the rule ($|t^*| + |t^*|$) and add a small constant ($k_2$). To find the size of all the rules, simply add this number for all the rules.

Consider now the list-marking approach: The number of pattern matches that must be performed to decide whether a specific transition is enabled is given by the following expression:

$$\sum_{p \in P} \max(1, P_{tok})$$  \hspace{1cm} (4)

For each token in a place we must perform one pattern match. But since we must perform one match for places without tokens as well we have to add up $\max(1, P_{tok})$ for all places. Because our nets are one-safe expression (4) can be reduced to $|P|$. The worst case scenario for number of matches to say whether any transition can fire is:

$$|P| * |T|$$  \hspace{1cm} (5)

This means that to get the number of matches for all the transitions we have to multiply the number of matches for one transition by the number of transitions.
The main problem of this approach is that the modules tend to get enormous as the nets get larger. The file size of a one-safe Petri net translated to a Maude module with list-marking is given by:

\[
\frac{[T] \cdot (|P| \cdot k_6 + k_7) + |P| \cdot k_8 + k_9}{V} + \frac{[P] \cdot k_6 + k_7}{VI} + \frac{k_9}{VII},
\]

where all \( k_i \) are constants. The part of expression (6) marked with V gives the size of the rules. Each place in a rule takes up \( k_6 \) amount of space. Since all places is written in a rule the size of one rule is given by \( |P| \cdot k_6 + k_7 \). All the rules have the same size so we just multiply this number with the number of rules to get the total size of all the rules. The part marked VI gives the size of the initial state and some variable declarations, all dependent on the number of places. As in expression (3) we have a miscellaneous part, this is marked as VII. The list-marking makes the files much larger while we hope to decrease the execution time.

7 A critical part of the line - the downtown scenario

The figure 6 shows the specification of the subway of downtown Oslo, the track from Majorstuen to Grønland. This part of the line is very important and safety critical, since trains from every line drive on the same track under the ground. Traffic jams are likely to happen, since the interval between the trains can be as low as 90 seconds. If accidents occur, there will soon be a queue of trains influencing the throughput on the complete subway system. The net in figure 6 consists of 93 components, ordinary track segments, turnouts, end segments, crossings and other components (for more details see [4]).

![Fig. 6. The downtown fragment of Oslo Subway.](image)

The history of the subway shows that large accidents occur rarely, mostly thanks to a very strict interlocking system. But small accidents with the electric power supply and malfunctioning caused by mechanical problems happen quite often, since the trains are old. Another not so well
known problem regards the driver profile. Since the trains are operated manually, the profile of the driver has a great impact on the throughput of trains. For Oslo Subway it is therefore an important problem to optimize, monitor and test scenarios with several trains running concurrently. But computerized support does not come for free.

8 Simulation of Oslo Subway

From the point of view of simulating the railroad, the size of the data structures and the time it takes to run several trains put hard requirements on the underlying implementation and the hardware, as described in section 6. Therefore, the subway was initially broken into smaller pieces,

Fig. 7. The Oslo subway system.

in order to compare empirically the difference of memory-usage and execution time with respect to the size of the net.

Three parts of Oslo subway system have been saturated with Petri net components as for instance the turnout in figure 5. The smaller of the three is the downtown part from Majorstuen to Grenland. The downtown segment was extended from Kolsås to Bergkrystallen, to form a complete line, line 4 in figure 7. Finally the entire system was saturated. Table 1 shows the sizes of the petri nets both in components and file sizes.

Maude was used to execute the nets with a various number of trains. First one rewrite step was performed, which corresponds to the firing of one transition in the Petri net. In practice this means that one train moves one step ahead or that a train changes the state of a point machine. Secondly Maude executed the net until a final state was reached. This

\footnote{Petri nets are saved as XML files by the DTD of the Design/CPN group}
corresponds to a simulation where all the trains will move as long as they can. The travel plans in the simulated nets are defined as one journey from one end station to another end station. The trains moved until they reached an end station or their running track was blocked by another train. In such cases the net reaches a final marking.

Let us describe one simulation in detail: The complete railroad of Oslo Subway was executed with four trains. The trains started their journey in the following positions:

- at Frognerseteren on line 1 eastwards.
- at Bergkrystallen on line 1 westwards.
- at Sognsvann on line 3 eastwards.
- at Mortensrud on line 3 westwards.

When the simulation ended all the trains had reached their end station. The train starting its journey from Frognerseteren had arrived at Bergkrystallen and the train at Bergkrystallen was now at Frognerseteren. The train that started from Sognsvann had driven to Mortensrud, and the one that came from Mortensrud had arrived at Sognsvann. The simulation took about four seconds, which is far below real time where these trips last for 42 to 55 minutes.

The table 2 in the Appendix shows how long time Maude used to compute (i) the initial state, (ii) perform one rewrite, and (iii) execute the net as far as possible. It also shows how many rewrites Maude used to reach a final marking. The final row in table 2 shows that simulating 32 trains in the complete railroad net results in 35630 rewrites in order to reach the final marking. The simulation time is 3.620 seconds. There are no available data of simulation times in case of the list-marking of the whole system in figure 7, because the computer crashed when we tried to open it in Maude. The files were simply too large.

We were caught by some puzzles. The first surprise was the simulation times did not depend much on the number of trains, even though an
increase in the number of trains caused an increased number of rewrites. A possible explanation might be that the number of tokens in the system is not influenced by the number of trains. Each train token added to the net will replace a noTrain token. If the number of trains on the line is increased, more transitions are enabled during execution, which will decrease the time it takes to find an enabled transition. Another unexpected result occurred when we executed the whole net. When we increased the number of trains from 16 to 32, the number of rewrites decreased. The number of rewrites are connected to the number of firing of transitions. The decreased number of rewrites is due to deadlocks in scissor components. This deadlock occurs since the scissor synchronizes two turnouts pairwise, and therefore two trains running in opposite direction might both try to access the synchronizing controller for the turnouts simultaneously. More details about the scissor can be found in [5].

9 Conclusion

Overall, the initial simulation proves that the approach has been promising. We got some major surprises while working. In the beginning we expected that the matching algorithm of Maude would be too expensive, since the worst case scenario of possible matches for the complete railroad net would make execution unfeasible. However, a simulation of the entire Oslo subway system is possible, with a realistic number of trains, and also in simulation time much faster than real time. Maude used about four seconds to simulate approximately one hour real time traffic.

A natural next task would be to saturate the specification with more advanced atomic Petri nets, involving time, block sections, collision detection, driver profiles, signals etc. These components will result in even larger nets and it will be exiting to see whether the Maude engine can handle it.

Acknowledgments

Thanks to Olaf Owe, Pål Enger, Eskil Brun, and Dines Bjørner. The authors would also like to thank Trygve Kaasa, Willy Ulseth and Thor Georg Sælid from Oslo Subway Company, for in depth knowledge about railroad engineering and the railroad layouts for Oslo Subway.

References

### Appendix

<table>
<thead>
<tr>
<th>segment</th>
<th>type</th>
<th>number of trains</th>
<th>Simulation time</th>
<th>number of rewrites</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>number of trains</td>
<td>Simulation time</td>
<td>number of rewrites</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initial state</td>
<td>initial state</td>
<td>number of rewrites</td>
</tr>
<tr>
<td></td>
<td></td>
<td>one rewrite</td>
<td>all rewrites</td>
<td>number of rewrites</td>
</tr>
<tr>
<td>Majorstuen - Grønland</td>
<td>list</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kolsås - Bergkrystallen</td>
<td>list</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1090</td>
<td>3680</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>650</td>
<td>750</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>490</td>
<td>680</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>700</td>
<td>610</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>260</td>
<td>790</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>400</td>
<td>460</td>
</tr>
<tr>
<td>The whole system</td>
<td>list</td>
<td>1</td>
<td>0</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2. Simulation data