The Square Petri Net

Jörg Desel[1,*], Julia Fleischer[1] and Moritz Sommer[2]

[1]FernUniversität in Hagen, Germany
[2]Luitpold-Gymnasium München, Germany

Abstract
A small high-level Petri net describing the behavior of a simple algorithm with one loop is presented and discussed. The net can run into a deadlock if and only if the algorithm terminates. The behavior of the net is surprisingly complex, so that we provide a tool to visualize possible runs. We offer the reader a puzzle, ask for a termination proof – and of course provide a solution.

Keywords
high-level Petri net, termination proof

1. Introduction

Very often, Petri nets and other formal modeling languages are taught independently of application domains such as software engineering. In addition, the methods used in formal techniques differ significantly from methods in other fields, although both are based on mathematics. Due to the usual study structure in separate modules, taught by different professors, students find it difficult to recognize connections and integrate their knowledge.

Since Petri nets have a particularly wide range of uses, it makes sense to use them as a comprehensive bridge in studies. This traditionally happens in some places, for example in the old textbook [1] of Rüdiger Valk and Eike Jessen, which is both a book on Petri nets and on applications in computing systems.

The primary modeling aspect of Petri nets is concurrency. However, many concepts developed for Petri nets are likewise applicable to sequential systems, which can be viewed as a special case of concurrent systems.

This short article presents an example – actually a puzzle or a challenging exercise for students – which integrates three areas of knowledge: programming practice, programming theory and Petri net modeling. The question to be answered is to find a descending measure for a loop, proving that a given algorithm terminates. The algorithm is presented as a Petri net, so that the termination proof is at the same time a termination proof for the Petri net behavior. Since the behavior of the algorithm under consideration turns out to be surprisingly complex, we provide as a web-based educational resource a Python program. This program can be modified by the user and contributes to understanding the behavior of the algorithm. This understanding is essential for finding the required termination argument.

We found the algorithm under consideration by chance. The question in this article meets the two most important beauty criteria: The algorithm itself is very simple and immediately understandable, whereas the question to the students is surprisingly difficult but solvable. Let us end this introduction by providing the algorithm in natural language:

> *Given two nonzero integers, repeat the following until one of the numbers is zero:*
> *If the numbers have different signs, add the second number to the first; otherwise, subtract the first from the second.*

PENGE 2024, Geneve, June 2024
*Corresponding author.
joerg.desel@fernuni-hagen.de (J. Desel); julia-anna.fleischer@fernuni-hagen.de (J. Fleischer)

So, starting e.g. with the pair $(5, 7)$, we obtain the sequence

$$(5, 7), (5, 2), (5, -3), (2, -3), (-1, -3), (-1, -2), (-1, -1), (-1, 0)$$

which terminates with the second number being zero, whereas the pair $(-8, 6)$ leads to the sequence

$$(-8, 6), (-2, 6), (4, 6), (4, 2), (4, -2), (2, -2), (0, -2).$$

2. The Algorithm

Instead of starting with a textual representation of the algorithm, we formalize it by means of the following high-level Petri net (for the ocurrence rule of high-level Petri nets see [2] or [3]). Notice that the initial values of the two tokens a and b in the two places represent the initial values of the two integers.

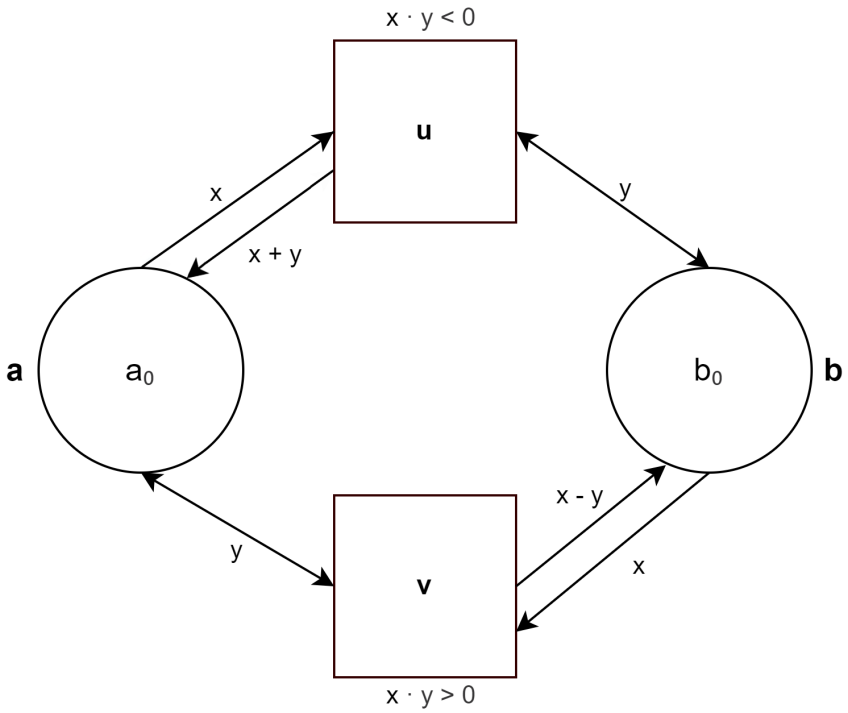

Figure 1: The Square Petri net

This Petri net corresponds to the following algorithm without INPUT and OUTPUT operations, when a_0 is the input value for the variable a and b_0 is the input value for the variable b.

INPUT $a, b \in \mathbb{Z}$
WHILE $a \cdot b \neq 0$
    IF $a \cdot b < 0$ THEN
        $a \leftarrow a + b$
    ELSE
        $b \leftarrow b - a$
OUTPUT a, b

Actually, repeated calculation of the product $a \cdot b$ is not the most efficient way to evaluate the conditions after WHILE and IF. Instead, a real implementation would rather check $a \neq 0$ **and** $b \neq 0$ instead of $a \cdot b \neq 0$ and $a > 0$ **xor** $b > 0$ instead of $a \cdot b < 0$.

This example departs from common applications of Petri nets. It deals with integers, a scenario somewhat atypical in the Petri net world, since token numbers on places (token counts) can never be negative. Consequently, these integers are not represented as markings of places but rather as high-level tokens.

Let us start with some observations:

- There is an obvious invariant stating that, for each reachable marking, both places carry one token representing an integer each. Therefore, for any reachable marking m, we can write $m(a) = z$ instead of $m(a) = \{z\}$, and the same applies to the place b.
- A transition is only enabled at a marking m satisfying $m(a) \neq 0$ and $m(b) \neq 0$. Since each transition occurrence changes only one of the two tokens, we have $m(a) \neq 0$ or $m(b) \neq 0$ after any transition occurrence. Consequently, the marking m satisfying $m(a) = m(b) = 0$ can only be reached if this is the initial marking.
- Taking the transition guards into account, each reachable marking enables at most one of the two transitions. It enables no transition if and only if one of the two tokens represents the value 0. Therefore, the behavior is deterministic for each initial marking.

We are interested in termination of the algorithm: Does it terminate for arbitrary input values a_0 and b_0, or only for some and, if so, for which ones? In terms of Petri nets, we address the question of whether the Petri net can reach a deadlock and, if so, from which initial markings a deadlock can be reached. Given that the behavior of the net is neither concurrent nor nondeterministic, such a deadlock would inevitably be eventually reached – if it is reachable at all.

This question is naturally a task for you, the reader, or for your students.

3. Finding a solution

How would you approach finding an answer? Naturally, you would first test the algorithm with different input values a_0 and b_0 and find that during the process the numbers sometimes become positive, sometimes negative, and also vary in their magnitude …and that, ultimately, regardless of the initial values of the variables, one of the two - and thus the product - inevitably turns to zero. The remaining number is then always the greatest common divisor of the initial values, or its negative value, if this concept is generalized from natural numbers to integers.

To prove termination of the algorithm, it is very helpful to find a descending measure, that is, a function $\mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ that allocates a natural number to each assignment of the variables a and b. This number should decrease by at least one with every loop iteration (see e.g. [4] for this and more general termination criteria). In Petri net language, the function, applied to $m(a)$ and $m(b)$, decreases with every transition occurrence. Once a suitable descending measure f is found, $f(a_0, b_0)$ indicates for initial values a and b how many times the transitions can fire at most. That, is, after a finite number of loop iterations there can eventually be no more iterations of the loop, or in Petri net terminology, a deadlock is reached. In our example, this happens precisely when $m(a) = 0$ or $m(b) = 0$.

Before we present a solution, a bit more about the genesis of this puzzle. It was planned as a regular exercise for our students and meant to be quite harmless, in the expectation that the algorithm either does not terminate or that a suitable decreasing measure is easy to find. But neither is the case! Since after many experiments it seemed unlikely that the algorithm would not terminate for any initial marking, we concentrated on the search for a descending measure, proving termination generally.

When looking for a descending measure, it helps to look at the algorithm in more detail. For this purpose, it has proven useful to note the respective values of a and b in a coordinate system, with the x-axis representing a and the y-axis representing b. Then arrows represent the changes caused by a loop execution, i.e., if the values a and b are changed to a' and b', then there is an arrow from the point (a, b) to the point (a', b'). That is exactly what we have done for several example values in Figure 2.

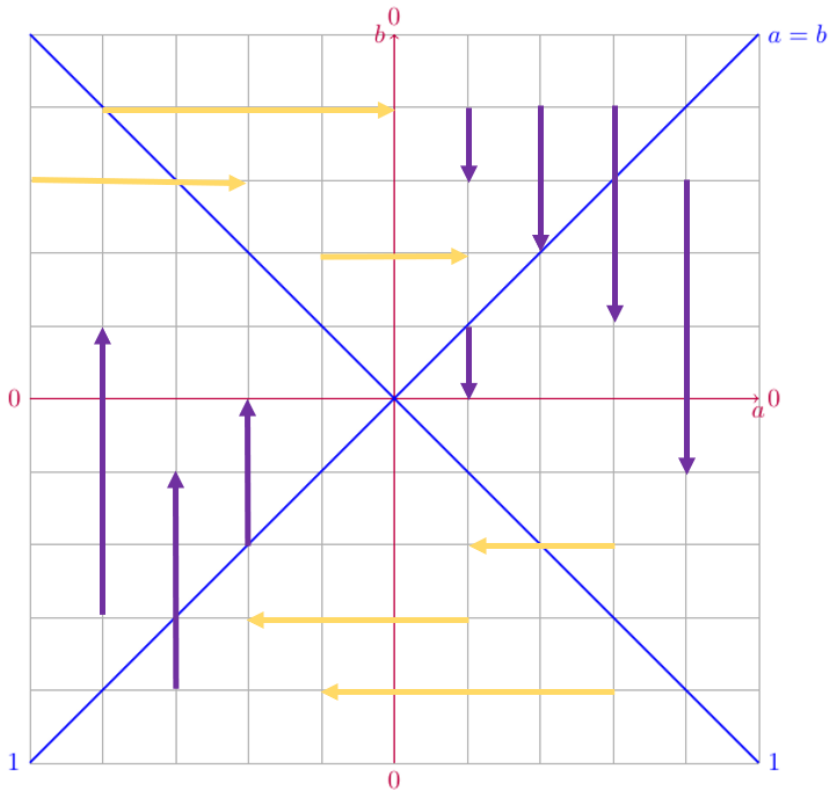

Figure 2: Visualization of the behavior of the algorithm

Yellow (horizontal) arrows represent occurrences of transition u and blue (vertical) arrows represent occurrences of transition v.

And because it's so easy to make a mistake, we implemented the algorithm and the visualization mentioned above. So you don't have to repeat this effort. We are happy to provide the tool, which is accessible via https://kalliope1907.github.io/jupyter/notebooks/index.html?path=visualisierung_algorithmus.ipynb. The tool allows to play with the algorithm. It visualizes the live of both variables for arbitrary initial values. Since it can be freely modified by the user – provided she has some practice in Python programming – not only the initial values but also the algorithm itself can be changed. And finally, the tool draws surpringly nice pictures, see the screenshot in Figure 3 for an example.

Before we developed this tool we gave the task to 600 students at the FernUniversität in Hagen in the "Software Engineering" module as an assignment and actually received three responses. The solution we're about to present is actually a combination of two proposals from students – our previous solution turned out to be less elegant.

4. Problem Statement

We are looking for a mapping $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ such that

- $f(a, b)$ decreases for any loop execution, i.e.,

$$a \cdot b < 0 \implies f(a + b, b) < f(a, b) \tag{1}$$

$$a \cdot b > 0 \implies f(a, b - a) < f(a, b) \tag{2}$$

- If $f(a, b) = 0$ then the loop terminates, i.e.,

$$f(a, b) = 0 \implies a \cdot b = 0 \tag{3}$$

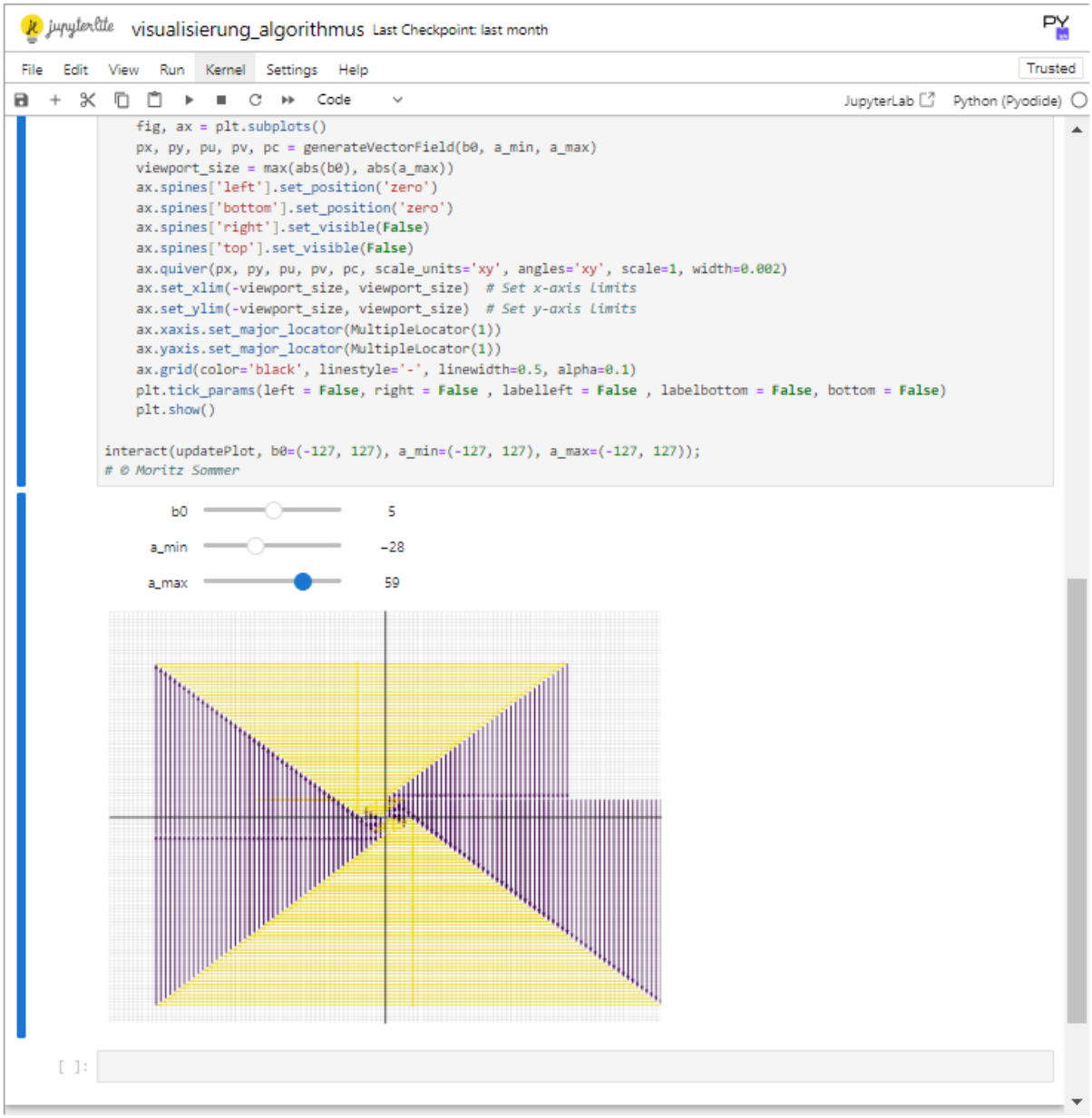

Figure 3: Screenshot of the tool

5. Our Solution

The formal definition of our descending measure turns out to be somewhat tedious. It is better understood when you consider the following eight areas of the coordinate system.

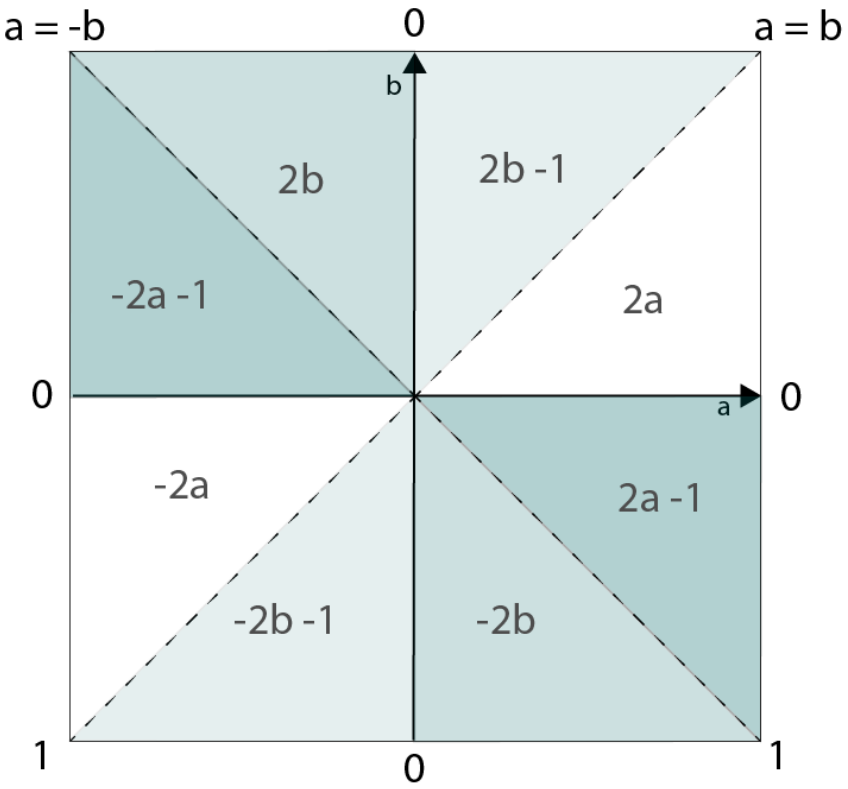

Figure 4: Illustrating the definition of the mapping f

We define a mapping $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ and distinguish ten (!) cases:

$$
\begin{array}{rll}
f(a,b) = & & \\
\quad 2b-1 & \text{for} & a>0, b>0, |a|<|b| \\
\quad 2a & \text{for} & a>0, b>0, |a|>|b| \\
\quad 2a-1 & \text{for} & a>0, b<0, |a|>|b| \\
-2b & \text{for} & a>0, b<0, |a|<|b| \\
-2b-1 & \text{for} & a<0, b<0, |a|<|b| \\
-2a & \text{for} & a<0, b<0, |a|>|b| \\
-2a-1 & \text{for} & a<0, b>0, |a|>|b| \\
\quad 2b & \text{for} & a<0, b>0, |a|<|b| \\
\quad 1 & \text{for} & a\neq 0, b\neq 0, |a|=|b| \\
\quad 0 & \text{else} & (a=0 \text{ or } b=0)
\end{array}
$$

A more compact version of the definition might be nicer, although less understandable:

$$
\begin{array}{lll}
f(a,b) = & & \\
2 \cdot \max(|a|,|b|) & \text{for} & |a| \cdot a \cdot b > |b| \cdot a \cdot b \\
2 \cdot \max(|a|,|b|) - 1 & \text{for} & |a| \cdot a \cdot b < |b| \cdot a \cdot b \\
1 & \text{for} & |a| \cdot a \cdot b = |b| \cdot a \cdot b \neq 0 \\
0 & \text{for} & |a| \cdot a \cdot b = |b| \cdot a \cdot b = 0
\end{array}
$$

As one of our students found out, $2 \cdot \max(|a|,|b|)$ can be equivalently replaced by $|a+b| + |a-b|$.

This definition clearly satisfies the second requirement: $f(a,b) = 0$ implies $a \cdot b = 0$. In fact, since a and b are integers, we have $f(a,b) \geq 1$ for all other cases.

Why is this function truly a descending measure, i.e., why does its value always decrease? We shall now prove that with each move from values a and b to values a' and b' we get $f(a,b) > f(a',b')$. For this proof, we restrict our considerations to the special case $a > 0$ and $b > 0$, i.e., to the first quadrant of the coordinate system. It is easy to see that by rotation symmetry the same arguments apply to the other three cases. Note that for $a = 0$ or $b = 0$ nothing has to be shown, because in both cases $a \cdot b = 0$ and therefore the loop stops.

Since we assume $a > 0$ and $b > 0$ we obtain $a \cdot b > 0$. Therefore $a' = a$ and $b' = b - a$. We thus have to prove $f(a, b-a) < f(a,b)$.

We distinguish five cases (see the five downward-pointing blue arrows in the coordination system of Figure 2):

Case 1: $b > 2 \cdot a$.
Then $b - a > a$. We get $f(a, b-a) = 2(b-a) - 1 < 2b - 1 = f(a,b)$ since $a > 0$.

Case 2: $b = 2 \cdot a$.
Then $b - a = a$. We get $f(a, b-a) = 1 < 2b - 1 = f(a,b)$ since $b = 2a \geq 2$.

Case 3: $2 \cdot a > b > a$.
Then $b - a < a$. We get $f(a, b-a) = 2a < 2b - 1 = f(a,b)$ since $a < b$.

Case 4: $b = a$.
Then $b - a = 0$. We get $f(a, b-a) = 0 < 1 = f(a,b)$ since $a, b > 0$.

Case 5: $b < a$.
Then $-a < b - a < 0$. We get $f(a, b-a) = 2a - 1 < 2a = f(a,b)$.

So in all of these five cases the inequality holds, and it holds in each case for a different reason. This completes the proof (we would in fact have 20 different cases if we would consider all four quadrants explicitly).

6. Conclusion

We presented an algorithm and its Petri net representation and proved that this algorithm terminates for all initial values, or, equivalently, that the Petri net terminates for every initial marking.

This exercise links program analysis and Petri net analysis and demonstrates how Petri nets can be used as a (programming) language independent means for the representation of algorithms. In addition, a connection to programming practice is made, as it can be helpful for students to create or use and modify an illustrative tool to understand the behavior of the algorithm.

As Petri nets are particularly useful for modeling the concurrent behavior of systems, they represent distributed algorithm in a natural way. Therefore, Petri net analysis techniques can be applied to analyse termination and other behavioral properties of distributed algorithm, as shown in many examples in [5], for another example see [6].

References

[1] E. Jessen, R. Valk, Rechensysteme - Grundlagen der Modellbildung, Studienreihe Informatik, Springer, 1987.

[2] W. Reisig, Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies, Springer, 2013. URL: https://doi.org/10.1007/978-3-642-33278-4. doi:10.1007/978-3-642-33278-4.

[3] K. Jensen, Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1, Second Edition, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 1996. URL: https://doi.org/10.1007/978-3-662-03241-1. doi:10.1007/978-3-662-03241-1.

[4] H. R. Jr., Theory of recursive functions and effective computability (Reprint from 1967), MIT Press, 1987. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3182.

[5] W. Reisig, Elements of distributed algorithms: modeling and analysis with Petri nets, Springer, 1998.

[6] J. Desel, E. Kindler, T. Vesper, R. Walter, A simplified proof for a self-stabilizing protocol: A game of cards, Inf. Process. Lett. 54 (1995) 327–328. URL: https://doi.org/10.1016/0020-0190(95)00065-K. doi:10.1016/0020-0190(95)00065-K.