On Representations of Waiting Queues for Semaphores in Logically Constrained Term Rewrite Systems*
(Short Abstract)

Misaki Kojima
Nagoya University
Nagoya, Japan
k-misaki@trs.css.i.nagoya-u.ac.jp

Naoki Nishida
Nagoya University
Nagoya Japan
nishida@i.nagoya-u.ac.jp

A transformation of concurrent programs with semaphore-based exclusive control and waiting queues into logically constrained term rewrite systems (LCTRSs, for short) has been proposed. To represent waiting queues, the transformation adopts a turn waiting system with number tickets, while the use of usual lists is a straightforward approach. In this abstract, we show a list-based approach to waiting queues and compare the two approaches by means of verification of race freedom of a simple reader-writer example.

1 Introduction

In the last decade, approaches to program verification by means of logically constrained term rewrite systems (LCTRSs, for short) [8] are well investigated [2, 13, 1, 11, 3, 4, 7, 5, 6]. LCTRSs are known to be useful as computation models of not only functional but also imperative programs. To apply techniques for LCTRSs to verification of practical programs such as automotive embedded systems, the transformation in [2] has been extended to concurrent programs with semaphore-based exclusive control and waiting queues [7].

The extended transformation does not use usual lists to represent waiting queues for semaphores, which are sequences of process identifiers, e.g., natural numbers. *Rewriting induction* [12, 2] used as a method for equivalence verification by means of LCTRSs makes a case analysis w.r.t. a *reduction-complete* position which ensures the exhaustiveness of the case analysis. Decidability of reduction-complete positions for constrained rewriting is not known yet. Even for decidable constraints, it is not so easy to decide whether a position is reduction-complete along with recursive data structures such as lists; it may be undecidable or not known yet. Note that we use stacks for function calls because such stacks do not prevent us from checking reduction-completeness of LCTRSs obtained from imperative programs.

To represent waiting queues for semaphores, the extended transformation adopts a so-called *turn waiting system* with number tickets. A semaphore $\mathtt{s}$ in the transformed LCTRS is represented as a term $\mathsf{sem}(v_{\mathtt{s}}, v_{\mathtt{d}}, v_{\mathtt{t}})$ such that $v_{\mathtt{s}}$ is a value of $\mathtt{s}$, $v_{\mathtt{d}}$ is a "display board" to permit processes to acquire the semaphore, $v_{\mathtt{t}}$ is a "ticket machine" to issue number tickets, and the display board and ticket machine are implemented as counters. A process P is represented as a term of the form $\mathsf{p}(u,n)$ such that u is a state of P, and n is either 0 or a positive integer: If $n = 0$, then P is active and does not wait for semaphore; otherwise, n (> 0) is the ticket for $\mathtt{s}$ and P is waiting for $\mathtt{s}$. If P wants to enter its critical section by acquiring $\mathtt{s}$ and the value of $\mathtt{s}$ is more than 0, then P can acquire $\mathtt{s}$ and enter its critical section; if P

*This work was partially supported by JSPS KAKENHI Grant Number 18K11160 and DENSO Corporation.

Submitted to:
WPTE 2023

Program 1: A simple reader-writer program with a counting semaphore.

```
1  semaphore s = 2;
2  int x = 0;
3
4  void reader(void)
5  {
6    while(true){
7      int y = 0;
8
9      down(&s);
10     y = x;
11     up(&s);
12   }
13 }
```

```
15 void writer(void)
16 {
17   while(true){
18     down(&s);
19     x = 1;
20     up(&s);
21   }
22 }
```

wants to enter its critical section but cannot acquire `s`, then P takes a ticket $v_{\mathtt{t}}$ to wait for `s` and the ticket machine prepares the next ticket by incrementing $v_{\mathtt{t}}$ by 2;[1] when P exits its critical section and releases `s`, if another process is waiting for `s` with the ticket $v_{\mathtt{d}}+2$, then the waiting process acquires `s` to enter its critical section, and otherwise, P increments `s` by 1.

Example 1 Let us consider Program 1 written in C language, together with a configuration consisting of three processes such that the first and second processes execute `reader`, and the third one executes `writer`: Functions are executed along the standard C semantics; processes are executed concurrently; `semaphore` is a type synonyms of `int`; the range of `int` is the integers; global variables are accessible from any process. For such a configuration, the LCTRS $\mathcal{R}_1$ in Figure 1 with the initial configuration $\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),\mathsf{p}(\mathsf{rdr}_7,0),\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(2,1,3),0,0)$ is generated by a *list-free* approach in [7], where rdr and wtr stand for `reader` and `writer`, respectively.

The list-free approach does not need any constraint in the theory of recursive data, and the representation of waiting queues does not need any rule for list operations. On the other hand, any practical advantage over the use of usual lists has not been shown yet.

In this paper, we show a list-based approach to waiting queues and compare the two approaches by means of verification of race freedom of Program 1.

2 A List-Based Approach

The list-based approach gives each process a positive integer as its identifier and uses usual integer lists consisting of list constructors $\mathsf{cons} : \mathit{int} \times \mathit{list} \to \mathit{list}$ and $\mathsf{nil} : \mathit{list}$.

As in the list-free approach, a process P is represented as a term of the form $\mathsf{p}(u,n)$ such that u is a state of P. Though, the use of the second argument n is different: If P is active, then the value is 0, and otherwise, it is the process identifier of P. For example, if P is the first process and is waiting for a semaphore, then the state of P is represented by a term of the form $\mathsf{p}(u,1)$.

[1] The reason of incrementing $v_{\mathtt{t}}$ by 2 is the reuse of numbers when the maximum number of processes is fixed and we use fixed-size bit vectors for the turn waiting system (see [7] for detail).

$$\left\{\begin{array}{lll}
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),p_2,p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(0),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0) & & \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{10}(y),0),p_2,p_3,\mathsf{sem}(s',d,t),x,0) & [s\neq 0\wedge s'=s-1] \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),t),p_2,p_3,\mathsf{sem}(s,d,t'),x,0) & [s=0\wedge t'=t+2] \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),n),p_2,p_3,\mathsf{sem}(s,d,t),x,1)\to\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{10}(y),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0) & [n=d\wedge n\neq 0] \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{10}(y),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{11}(x),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0) & \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{11}(y),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),p_2,p_3,\mathsf{sem}(s,d',t),x,1) & [t\neq d+2\wedge d'=d+2] \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{11}(y),0),p_2,p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),p_2,p_3,\mathsf{sem}(s',d,t),x,0) & [t=d+2\wedge s'=s+1] \\
\mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_7,0),p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_9(0),0),p_3,\mathsf{sem}(s,d,t),x,0) & & \\
\qquad\vdots & & \\
\mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_{11}(y),0),p_3,\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_7,0),p_3,\mathsf{sem}(s',d,t),x,0) & [t=d+2\wedge s'=s+1] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{19},0),\mathsf{sem}(s',d,t),x,0) & [s\neq 0\wedge s'=s-1] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},t),\mathsf{sem}(s,d,t'),x,0) & [s=0\wedge t'=t+2] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},n),\mathsf{sem}(s,d,t),x,1)\to\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{19},0),\mathsf{sem}(s,d,t),x,0) & [n=d\wedge n\neq 0] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{19},0),\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{20},0),\mathsf{sem}(s,d,t),1,0) & \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{20},0),\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s,d',t),x,1) & [t\neq d+2\wedge d'=d+2] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{20},0),\mathsf{sem}(s,d,t),x,0)\to\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s',d,t),x,0) & [t=d+2\wedge s'=s+1]
\end{array}\right\}$$

Figure 1: The transformed LCTRS $\mathcal{R}_1$ for Program 1 [7].

A semaphore $\mathtt{s}$ is represented by a term $\mathsf{sem}(v_{\mathtt{s}},w)$ such that $v_{\mathtt{s}}$ is the value of $\mathtt{s}$ and w is the waiting queues consisting of process identifiers. To add a process identifier to a waiting queue, the list operation $\mathsf{snoc} : \mathit{list}\times\mathit{int}\to\mathit{list}$ which appends an element to a given list is introduced:

$$\begin{aligned}\mathsf{snoc}(\mathsf{nil},y)&=\mathsf{cons}(y,\mathsf{nil})\\ \mathsf{snoc}(\mathsf{cons}(z,w),y)&=\mathsf{cons}(z,\mathsf{snoc}(w,y))\end{aligned}$$

To execute snoc atomically, we use the last argument of the configuration symbol cnfg as follows:

$$\begin{aligned}&\mathsf{cnfg}(\ldots,\ \mathsf{sem}(s,\mathsf{snoc}(\mathsf{nil},y)),\ldots,2)\to\mathsf{cnfg}(\ldots,\mathsf{sem}(s,\mathsf{cons}(y,\mathsf{nil})),\ldots,0)\\ &\mathsf{cnfg}(\ldots,\mathsf{sem}(s,\mathsf{snoc}(\mathsf{cons}(z,w),y)),\ldots,2)\to\mathsf{cnfg}(\ldots,\mathsf{sem}(s,\mathsf{cons}(z,\mathsf{snoc}(w,y))),\ldots,2)\end{aligned}$$

In adding a process identifier n to the waiting queue of $\mathtt{s}$, we call snoc in a right-hand side as follows:

$$\ell\to\mathsf{cnfg}(\ldots,\mathsf{sem}(v_{\mathtt{s}},\mathsf{snoc}(w,n)),\ldots,2)$$

Since snoc is defined by rules of cnfg, the reduction of the obtained LCTRS for configuration terms is topmost.

Example 2 We transform Program 1 into the LCTRS $\mathcal{R}_2$ in Figure 2. The initial configuration is $\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),\mathsf{p}(\mathsf{rdr}_7,0),\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(2,\mathsf{nil}),0,0)$. The last two rules represent snoc.

One may think that rules for snoc with cnfg are complicated and should be represented in the usual way, i.e, by defining it locally by rules without cnfg. For the atomicity of the execution of snoc, we have to prohibit the reduction of any process. To achieve it, a global definition of snoc—rules with cnfg—is necessary. As mentioned before, such as definition makes the reduction for configuration terms topmost. The topmost reduction reduces the search space for *reducts*, making verification tasks more efficient.

$$
\left\{\begin{array}{rll}
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),p_2,p_3,\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(0),0),p_2,p_3,\mathsf{sem}(s,w),x,0) & & \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),0),p_2,p_3,\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{10}(y),0),p_2,p_3,\mathsf{sem}(s',w),x,0) & & [s \neq 0 \wedge s' = s-1] \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),0),p_2,p_3,\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),1),p_2,p_3,\mathsf{sem}(s,\mathsf{snoc}(w,1)),x,2) & & [s=0] \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),n),p_2,p_3,\mathsf{sem}(s,\mathsf{cons}(k,w)),x,1) \to \mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{10}(y),0),p_2,p_3,\mathsf{sem}(s,w),x,0) & & [n=k \wedge n \neq 0] \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_9(y),n),p_2,p_3,\mathsf{sem}(s,\mathsf{nil}),x,1) \to \mathsf{error} & & \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{10}(y),0),p_2,p_3,\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{11}(x),0),p_2,p_3,\mathsf{sem}(s,w),x,0) & & \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{11}(y),0),p_2,p_3,\mathsf{sem}(s,\mathsf{cons}(k,w)),x,0) \to \mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),p_2,p_3,\mathsf{sem}(s,\mathsf{cons}(k,w)),x,1) & & \\
\mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_{11}(y),0),p_2,p_3,\mathsf{sem}(s,\mathsf{nil}),x,0) \to \mathsf{cnfg}(\mathsf{p}(\mathsf{rdr}_7,0),p_2,p_3,\mathsf{sem}(s',\mathsf{nil}),x,0) & & [s'=s+1] \\
\mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_7,0),p_3,\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_9(0),0),p_3,\mathsf{sem}(s,w),x,0) & & \\
\vdots & & \\
\mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_{11}(y),0),p_3,\mathsf{sem}(s,\mathsf{nil}),x,0) \to \mathsf{cnfg}(p_1,\mathsf{p}(\mathsf{rdr}_7,0),p_3,\mathsf{sem}(s',\mathsf{nil}),x,0) & & [s'=s+1] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{19},0),\mathsf{sem}(s',w),x,0) & & [s \neq 0 \wedge s' = s-1] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},3),\mathsf{sem}(s,\mathsf{snoc}(w,3)),x,2) & & [s=0] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},n),\mathsf{sem}(s,\mathsf{cons}(k,w)),x,1) \to \mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{19},0),\mathsf{sem}(s,w),x,0) & & [n=k \wedge n \neq 0] \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},n),\mathsf{sem}(s,\mathsf{nil}),x,1) \to \mathsf{error} & & \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{19},0),\mathsf{sem}(s,w),x,0) \to \mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{20},0),\mathsf{sem}(s,w),1,0) & & \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{20},0),\mathsf{sem}(s,\mathsf{cons}(k,w)),x,0) \to \mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s,\mathsf{cons}(k,w)),x,1) & & \\
\mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{20},0),\mathsf{sem}(s,\mathsf{nil}),x,0) \to \mathsf{cnfg}(p_1,p_2,\mathsf{p}(\mathsf{wtr}_{18},0),\mathsf{sem}(s',\mathsf{nil}),x,0) & & [s'=s+1] \\
\mathsf{cnfg}(p_1,p_2,p_3,\mathsf{sem}(s,\mathsf{snoc}(\mathsf{nil},y)),x,2) \to \mathsf{cnfg}(p_1,p_2,p_3,\mathsf{sem}(s,\mathsf{cons}(y,\mathsf{nil})),x,0) & & \\
\mathsf{cnfg}(p_1,p_2,p_3,\mathsf{sem}(s,\mathsf{snoc}(\mathsf{cons}(z,w),y)),x,2) \to \mathsf{cnfg}(p_1,p_2,p_3,\mathsf{sem}(s,\mathsf{cons}(z,\mathsf{snoc}(w,y))),x,2) & &
\end{array}\right.
$$

Figure 2: The LCTRS $\mathcal{R}_2$ with list-based waiting queues.

3 Comparison of List-Free and List-Based Approaches to Waiting Queues

To compare the list-free and list-based approaches, we solve the *all-path reachability problem* (APR problem, for short) for the verification of race freedom [5]. An *APR problem* of a rewrite system is a pair $P \Rightarrow Q$ of state sets P, Q and is *demonically valid* w.r.t. the system if every *finite* execution path—a reduction sequence starting with a state in P and ending with a terminating state (i.e., a normal form)—includes a state in Q. A proof system for APR problems of LCTRSs, DCC, and its simplified variant have been proposed in [1, 5, 6].

We have implemented the simplified proof system in a prototype of Crisys2,[2] a equivalence verification system based on *constrained RI* for LCTRSs [9, 2]. Roughly speaking, Crisys2 checks the reduction-completeness of a given constrained term $t\ [\phi]$ by a sufficient condition which is the validity of $\phi \Rightarrow \bigvee_{\ell \to r\ [\psi], t=\ell\theta, Y=\mathcal{V}ar(r,\psi)\setminus\mathcal{V}ar(\ell)} (\exists \overrightarrow{Y}.\ \psi)\theta$.

Using the implementation, we made experiments of the race-freedom verification of $\mathcal{R}_1$ and $\mathcal{R}_2$ by reducing to APR problems, which were conducted with in a 3,600s timeout on a machine running MacOS 13.4 on Apple M2 8 cores with 24GB memory; Z3 (ver. 4.12.1) [10] was used as an external SMT solver. The APR problem for $\mathcal{R}_1$ was solved in 287.52s, and that for $\mathcal{R}_2$ was in 351.17s, where the initial APR problems have been generalized in advance. Note that the APR problem without atomicity of snoc was solved in 2138.17s. Table 1 shows the results of experiments for m `readers` and n `writers` in Program 1, where the initial value of semaphore `s` is less than or equal to $m+n$. Note that for the case where $m+n \geq 4$, the implementation did not halt in 3,600s.

[2] https://www.trs.css.i.nagoya-u.ac.jp/~nishida/wpte2022/

Table 1: Experimental results of m `readers` and n `writers`.

`#reader`	`#writer`	Init. value of `s`	List-free approach	List-based approach
1	1	1	1.41s	2.36s
2	1	1	199.78s	274.42s
2	1	2	287.52s	351.17s
2	1	3	128.03s	148.18s

4 Conclusion

The computation to prove the demonical validity of APR problems for the non-occurrence of runtime errors often causes a state explosion because the search space is linear to multiplication of the numbers of rewrite rules for processes. For this reason, the fewer the number of rewrite rules, the better the efficiency of solving the APR problems. From the experiments in Section 3, the list-free approach in [7] has an advantage over the list-based approach in Example 2 regarding efficiency of proving APR problems for non-occurrence of a specified runtime error: The main reason of the efficiency must be the number of rewrite rules. Our future work is to make more experiments to compare the two approaches by means of e.g., runtime-error verification of practical larger programs.

References

[1] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In: *Proc. IJCAR 2018*, *LNCS* 10900, Springer, pp. 295–311, doi:10.1007/978-3-319-94205-6_20.

[2] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Trans. Comput. Log.* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[3] Yoshiaki Kanazawa & Naoki Nishida (2019): *On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems*. In: *Proc. WPTE 2018*, *EPTCS* 289, Open Publishing Association, pp. 34–52.

[4] Yoshiaki Kanazawa, Naoki Nishida & Masahiko Sakai (2019): *On Representation of Structures and Unions in Logically Constrained Rewriting*. IEICE Technical Report SS2018-38, IEICE. Vol. 118, No. 385, pp. 67–72, in Japanese.

[5] Misaki Kojima & Naoki Nishida (2022): *On Reducing Non-Occurrence of Specified Runtime Errors to All-Path Reachability Problems of Constrained Rewriting*. In: *Informal Proc. WPTE 2022*, pp. 1–16. Available at `https://easychair.org/publications/preprint/TM7q`.

[6] Misaki Kojima & Naoki Nishida (2023): *From Starvation Freedom to All-Path Reachability Problems in Constrained Rewriting*. In: *Proc. PADL 2023*, *LNCS* 13880, Springer Nature Switzerland, pp. 161–179, doi:10.1007/978-3-031-24841-2_11.

[7] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2020): *Transforming Concurrent Programs with Semaphores into Logically Constrained Term Rewrite Systems*. In: *Informal Proc. WPTE 2020*, pp. 1–12.

[8] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In: *Proc. FroCoS 2013*, *LNCS* 8152, Springer, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[9] Cynthia Kop & Naoki Nishida (2014): *Automatic Constrained Rewriting Induction towards Verifying Procedural Programs*. In Jacques Garrigue, editor: *Proceedings of the 12th Asian Symposium on Programming Languages and Systems*, *Lecture Notes in Computer Science* 8858, Springer, pp. 334–353, doi:10.1007/978-3-319-12736-1_18.

[10] Leonardo Mendonça de Moura & Nikolaj S. Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[11] Naoki Nishida & Sarah Winkler (2018): *Loop Detection by Logically Constrained Term Rewriting*. In: *Proc. VSTTE 2018*, *LNCS* 11294, Springer, pp. 309–321, doi:10.1007/978-3-030-03592-1_18.

[12] Uday S. Reddy (1990): *Term Rewriting Induction*. In Mark E. Stickel, editor: *Proceedings of the 10th International Conference on Automated Deduction*, *Lecture Notes in Computer Science* 449, Springer, pp. 162–177, doi:10.1007/3-540-52885-7_86.

[13] Sarah Winkler & Aart Middeldorp (2018): *Completion for Logically Constrained Rewriting*. In: *Proc. FSCD 2018*, *LIPIcs* 108, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 30:1–30:18, doi:10.4230/LIPIcs.FSCD.2018.30.