Information Societies Technology (IST) Programme

Future and Emerging Technologies - Open Domain

Project acronym: Advance

Project full title:

Advanced Validation Techniques
for Telecommunication Protocols

Contract no: IST-1999-29082

Deliverable D12.2

Parameterized Systems with Resource Sharing

Year 2 : October 2001 - September 2002

Date of preparation: September 2002

Contributors: Paris, Ericsson

1 Introduction

The work presented here (see [11] for more details) has been stimulated by one of the problems considered important in the design of the Ericsson's AXD 301 switch [9]. The topic, moreover, appears to be of a broader interest both for Ericsson as well as in general. It is the problem of verifying systems with concurrent processes accessing shared resources under the supervision of a certain resource management module. Clearly, such systems are quite frequent in practice. At the same time, verification of systems of this kind offers many interesting challenges, such as dealing with parameterization and/or infinite data structures.

The Problem. Ericsson's AXD 301 [9] is a high capacity ATM switch that is used, for example, in backbone networks to connect city telephone exchanges [5]. It embeds several workstations (at least two) together with some specialized hardware. The complete software equipment of AXD 301 is too complex to be fully verified using the current technology. Nevertheless, it is still possible to concentrate on some critical parts of the software which are especially worth of verification. One such part is related to the resource management problem: An arbitrary number of *client* processes competes for an *exclusive* or *non-exclusive* (shared) access to an arbitrary number of resources under the supervision of a single *locker* process. More specifically, in AXD 301, processes lock applications they are using to prevent them from being moved from one workstation to another (or, on the other hand, to allow them to be moved). The library designed for this reason is, however, written in such a way that it could be used in other contexts as well (possibly with different locker strategies).

In [4, 5], the authors tried to apply some finite-state verification methods to check several properties (like mutual exclusion or absence of starvation) of the resource management system of AXD 301. They verified the properties in several settings given by a certain number of participating clients and a certain number of accessible resources. Their results are, of course, quite significant, but it would further be interesting to be able to verify that the system satisfies the desired properties with *any* number of clients and/or with *any* number of resources. The verification method should remain as automatic as possible. Moreover, with respect to the importance of the resource management problem in different contexts, it is sensible to try to cover as many different types of client and locker behaviours as possible for generic property classes such as those expressible in (fragments of) suitable temporal logics.

The Approach. The above described parametric verification problem in its full generality is obviously very complex—it involves dealing with up to two parameters and several (possibly different) infinite data structures (such as queues, stacks, or lists) used in the lockers. Here, we suppose dealing with queue-based locker strategies, a fixed number of resources, and a parametric number of identical clients. (An extension to additionally having a fixed number of specific clients seems to be relatively easy to introduce.)

Our approach to the problem is to split it into two parts: (1) to verify that systems of arbitrary many clients behave well provided they are controlled by a locker that implements a certain locker strategy and (2) to check that for an arbitrary number of clients with an arbitrary behaviour, the given locker really implements the strategy we suppose it to implement. In this work, we concentrate on the first issue and we take into account two locker strategies (namely FIFO and FIFO with priorities) that are very common in practice. (Besides, we also already have some preliminary results on a FIFO locker with overtaking of blocked processes and on a service-in-random-order locker as it is mentioned in the end of the report.)

The Results. In the work, we propose a simple model based on extended automata with queues that is suitable for an abstract description of systems of clients competing for shared resources under the supervision of some of the considered queue-based lockers. Over models of this kind, we consider the parametric verification problem for various fragments and/or modifications of CTL* enriched with local and global process quantification. (We denote the quantified processes as *visible* in the following). The most relevant properties for the given setting, such as mutual exclusion, absence of starvation, or process deadlockability, are covered in this way.

We obtain three main kinds of results here. First, we prove that many verification problems for the parameterized systems we consider can be reduced to finite-state model-checking problems, i.e. verifying systems with up to a certain small number of clients is enough to verify systems with an arbitrary number of clients. These so-called *cut-off* sizes of systems depend on the number of resources used and the number of visible (quantified) processes in the formula of interest. Second, we show that some verification problems cannot be solved in the just mentioned way, but they may still be decidable because there may exist cut-offs additionally depending on the control structure of the clients and/or the structure of the formula. For a certain interesting class of liveness properties and for families with priorities, we then really derive such a complex cut-off. Finally, we prove that some verification problems are undecidable in the given setting.

Related Work. Several different approaches have been proposed for verifying parameterized networks of processes and/or systems dealing with infinite data structures. Let us mention, for example, the use of regular model checking [19, 22, 1, 12, 3], (automated) abstraction [6, 8, 7], or network invariants [23, 18, 20, 21]. The idea of finding instances of parameterized systems whose verification is sufficient to verify the whole parameterized system has already been used in several different contexts [17, 14, 15, 13] as well.

The paper [13] is probably the closest to this work. It also targets verification of systems with shared resources and employs cut-offs too. However, between [13] and this work there are some differences in the way of modelling clients and in the considered classes of properties to be verified. Moreover and most importantly, [13] allows for heterogeneous systems of clients, but service in random order is the only considered locker strategy here.

As we have already mentioned, some finite state instances of the problems considered here were examined in [4, 5] where a special emphasis was laid on working directly on the source code level.

2 Modelling and Specification

We now describe a little more the model we use for describing parameterized systems of clients and the way we specify properties to be verified.

The Model. For an abstract description of parameterized systems of clients competing for shared resources under the control of a certain queue-based locker, we propose a simple model in the form of the so-called *RTR families*. (RTR stands for request-take-release.) The model is based on extended automata with queues. More precisely, an RTR family is defined by a finite set of resources, a finite control automaton of clients, and a locker function.

Apart from the *idle* action, control automata of clients may contain actions of *requesting*, *taking*, and *releasing* subsets of the set of resources. We suppose the request operation to be a non-blocking operation which only informs the locker about the future interests of clients. A request operation may then be later followed by a take operation on (a subset of) the requested resources, which is a blocking operation that may be finished only when the appropriate resources can be granted to the client asking for them. Apart from the basic request and take operations, we also have their versions for *high-priority* or *non-exclusive* access to resources. Finally, we introduce combined *request-take* actions with no interleaving inbetween and we denote by (RT)R the families that allow such actions only.

Lockers. In our considerations, we work with two important locker strategies. The *FIFO* locker handles all request and take operations according to the strict first-in first-out strategy where the "first-in" relates to request operations and the "first-out" to take operations. The *PRIO* locker is a modification of the FIFO locker that allows for two priority levels of the operations where the high-priority requests may overtake the low-priority ones. We use RTR_{NFIFO} to refer to RTR families equipped with the FIFO locker and RTR_{FIFO} to refer to families with the FIFO locker and without the non-exclusive access option. A similar notation applies for the use of the PRIO locker as well.

Properties. We study different kinds of properties to be verified, but it is always possible to view them as expressible in some fragment and/or modification of CTL* enriched with process quantification. We cover the most important properties in the given setting including mutual exclusion, absence of starvation, and process deadlockability.

Our concept of process quantification is inspired by ICTL* [16]. *Global process quantification* selects some processes in some global state, and then we monitor their behaviour along state space paths leading from this state. Absence of starvation can, e.g., be expressed in this way: $\forall_p\ AG\ (in\_state(p, q_{request}) \Rightarrow AF\ in\_state(p, q_{grant}))$. *Local process quantification* selects some processes within the current global state, and then we check their current local states, which can be used, e.g., to express simultaneous local reachability: $EF\ \forall_p\ in\_state(p, q)$.

In our setting, we then interpret the *parametric verification problem* for a given property as checking whether there is a system (characterized by the number of its clients) in the RTR family at hand for which the property holds or, dually, whether the property holds for all systems of the family.

3 The Results

Safety. We consider a class of next-time-free linear-time *safety properties* definable on finite prefixes of behaviours of the considered systems, having a single global existential or universal process quantification on k processes, and containing atoms referring to the control states of the visible processes only. Mutual exclusion is an example of such a property: $\forall_{p_1 \neq p_2}\ A_{fin}\ G\ \neg(in\_state(p_1, q_{cs}) \wedge in\_state(p_2, q_{cs}))$.

For any RTR_{FIFO} family, the parametric verification problem for a safety property of the above kind can be efficiently decided by checking the property for the systems of the given family with $1, 2, ..., k$ processes (k is the number of the visible processes). This means that, for instance, to verify mutual exclusion in an RTR_{FIFO} family, it suffices to verify it for the system of this family with two processes.

In the domain of PRIO lockers, dealing with a number of processes depending only on the number of visible processes and/or the number of resources is in general insufficient. However, for $\mathrm{(RT)R}_{PRIO}$ families, the cut-off down to the number of visible processes remains applicable for the quite useful subclass of reachability and invariance properties ($F\psi$ or $G\psi$ where ψ is non-temporal). A full coverage of the problem that would provide us with at least a positive decidability result (e.g., via a cut-off depending on the structure of the control automaton and the formula being checked), remains open.

Although the above cut-offs allow us to get rid of the parameter, due to the unbounded queues of requests, we might still have to face dealing with infinite state spaces (in general, a process may have an arbitrary number of requests in a queue). Fortunately, if in the control automaton of a family with only exclusive access to resources there is no loop with more request operations involving a resource r than take operations on r, *the queue content is guaranteed to be bounded.* Moreover, it can be shown that if this requirement does not hold, there is either a possibility of a process deadlock, or the content of the queues may grow over every bound, which both is usually considered bad. So, we have a large, simply recognizable class of families (including all (RT)R families) where the above (and also the following) cut-offs directly take us to finite-state verification.

Liveness. We further examine next-time-free linear-time *liveness properties* which only differ from the above safety properties in that they are defined on unconditionally, weakly, or strongly fair behaviours. Unconditional fairness requires all processes to keep doing some steps; weak fairness and strong fairness are as usual.

Under *unconditional fairness*, we can transform the parametric verification problem of the described liveness properties with k visible processes to checking the properties over systems with $1, 2, ..., m + k$ processes where m is the number of resources in the given (RT)R_{FIFO} family. (For purely existential/universal[1] properties, k processes suffice and even in the RTR case.) The same result can also be used to handle at least some properties within (RT)R_{PRIO} families. It covers, e.g., 1-index properties (i.e. properties with just one visible process), such as absence of starvation. A better coverage of liveness with unconditional fairness in (RT)R_{PRIO} remains an open problem very close to the one of safety for these families.

For *weak/strong fairness*[2] and (RT)R_{FIFO} families, we can use the same cut-off down to $m + k$ processes as above. Unfortunately, for (RT)R_{PRIO}, any cut-off based just on the number of resources and visible processes is insufficient even in the case of 1-index properties. Nevertheless, the problem is decidable for 1-index properties and (RT)R_{PRIO} because we can show that it is sufficient to consider systems with up to a certain number of processes that depends on m and the structure of the control automaton. The general cut-off is quite big, but in some cases, we can use some more specialized and tighter results. For instance, for (RT)R_{PRIO} families in whose control automaton there is only one component in which the clients do not own any resources, dealing with systems with up to $2m + 2$ processes is sufficient for 1-index properties.

Deadlockability. The above subclasses of safety and liveness properties do not include the important notion of *process deadlockability* (i.e. the possibility that some process may deadlock within some system of a given family). We treat it in a separate way. So far, we know that to check process deadlockability in an RTR_{FIFO} family, it suffices to check it in the system of the given family with the number of processes equal to the number m of resources in the family.

Undecidability. Let us now proceed to some *undecidability results.* First, by simulating push-down automata with two stacks, we can show that general reachability referring both

[1]We ask whether the exists a system in a given family, there exist certain processes in this system, and there exist a behaviour such that something holds (or the dual).

[2]Due to the "eager" behaviour of the lockers we consider in our work, weak and strong fairness coincide [11].

to the current control locations of processes and to the content of queues is undecidable for RTR_{FIFO}. Moreover, the CTL (and even EF) fragment of CTL^* with *local process quantification* is undecidable in RTR_{FIFO} as well. Finally, the LTL\X fragment of CTL^* with local process quantification is undecidable even for $\mathrm{(RT)R}_{FIFO}$. (In all the cases, dealing with safety properties is enough to show the undecidability.) The undecidability results seem to extend to RTR_{PRIO} too.

4 Future Work

In the context of RTR_{FIFO} and RTR_{PRIO} families, which our current results concentrate on, there are still some configurations of the considered verification problems that are not known to be decidable nor undecidable. Moreover, some of the here presented cut-offs are not known to be optimal, and so it is still possible that they can be further minimized. For the future, these two issues seem to be especially interesting in relation to some of the configurations of checking liveness and deadlockability over families using the PRIO locker (as mentioned already in the text). Next, in the case of decidable problems for which finding small cut-offs will turn out to be hard or impossible, it is then possible to try to apply some other verification methods such as some kind of symbolic verification. (Besides, symbolic verification could be used for approximate verification even for undecidable problems.)

Nevertheless, the two main topics which remain open for the future research are the following.

First, RTR families with non-exclusive access should be examined in the same way as families without this operation. So far, we know that some of the here presented results (undecidability, most of safety and deadlockability, a part of liveness) can relatively easily be generalized for the new setting. However, the rest seems to be quite intricate. Additionally, it could also be interesting to examine some more locker strategies relevant for the practice. In fact, we have already started some studies on families with a FIFO-based locker allowing for overtaking of blocked processes and on families with a (lazy) service-in-random-order locker. While the former appears to be very close to the PRIO locker, in the case of the latter, some results related to the FIFO locker (safety, liveness under unconditional fairness) seem to be applicable, but otherwise some special treatment will probably be necessary.

Second, a big challenge is the problem of checking whether a given locker implements the strategy we suppose it to implement regardless of the number and behaviour of clients. Here, it is necessary to propose a suitable model and in the case the problem is undecidable in general (which is quite probable), try to define meaningful subclasses of the model for which the situation is better. We can again consider using cut-offs in this domain, but it is also possible to try out some other methods, e.g., the already mentioned (precise or approximate) symbolic model checking.

References

[1] P. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parametrized System Verification. In *Proc. 11th Intern. Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 134–145. Springer-Verlag, 1999.

[2] P.A. Abdulla, L. Boasson, and A. Bouajjani. Effective Lossy Queue Languages. In *Proc. 28th Intern. Coll. on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *LNCS*. Springer-Verlag, 2001.

[3] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 555–568. Springer-Verlag, 2002.

[4] T. Arts and C.B. Earle. Development of a Verified Erlang Program for Resource Locking. In *Proceedings of FMICS'01*, 2001.

[5] T. Arts, C.B. Earle, and J. Derrick. Verifying Erlang Code: A Resource Locker Case-Study. In *Proc. Int. Symposium on Formal Methods Europe 2002*, volume 2391 of *LNCS*, pages 183–202. Springer-Verlag, 2002.

[6] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Prameterized Networks. In *Proceedings of TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, 2000.

[7] K. Baukus, Y. Lakhnech, S. Bensalem, and K. Stahl. Networks of Processes with Parameterized State Space. In *Proc. ICALP 2001 Satellite Workshop VEPAS 2001*, volume 50 of *ENTCS*, pages 388–402. Elsevier, 2001.

[8] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. In *Proceedings of FTRTFT 2000*, 2000.

[9] S. Blau and J. Rooth. AXD 301—A New Generation ATM Switching System. *Ericsson Review*, No. 1, 1998.

[10] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The Power of QDDs. In *Proc. 4th Static Analysis Symposium*, volume 1302 of *LNCS*, pages 172–186. Springer-Verlag, 1997.

[11] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parameterized Concurrent Systems with Resource Sharing. A research report of the ADVANCE project (IST FET project No. IST-1999-29082). LIAFA, University Paris 7, France, 2002.

[12] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 539–554. Springer-Verlag, 2002.

[13] E.A. Emerson and V. Kahlon. Model Checking Large-Scale and Parameterized Resource Allocation Systems. In *Proceedings of TACAS 2002*, volume 2280 of *LNCS*, pages 251–265. Springer-Verlag, 2002.

[14] E.A. Emerson and K.S. Namjoshi. Reasoning about Rings. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 85–94, 1995.

[15] E.A. Emerson and K.S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proc. 8th Int. Conf. on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 87–98. Springer-Verlag, 1996.

[16] E.A. Emerson and A.P. Sistla. Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997.

[17] S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39(3):675–735, 1992.

[18] N. Halbwachs, F. Lagnier, and C. Ratel. An Experience in Proving Regular Networks of Processes by Modular Model Checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[19] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 424–435. Springer-Verlag, 1997.

[20] R.P. Kurshan and K.L. McMillan. A Structural Induction Theorem for Processes. *Information and Computation*, 117(1), 1995.

[21] A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 82–97. Springer-Verlag, 2001.

[22] P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *LNCS*, pages 88–97, Vancouver, 1998. Springer-Verlag.

[23] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407 of *LNCS*, pages 68–80, Grenoble, 1989. Springer-Verlag.

Verification of Parameterized Concurrent Systems with Resource Sharing

Ahmed Bouajjani, Peter Habermehl, Tomáš Vojnar
LIAFA, University Paris 7
{bouajjani,habermehl,vojnar}@liafa.jussieu.fr

September 2002

1 Introduction

This work has been stimulated by one of the problems considered important in the design of the Ericsson's AXD 301 switch [9]. The topic, moreover, appears to be of a broader interest both for Ericsson as well as in general. It is the problem of verifying systems with concurrent processes accessing shared resources under the supervision of a certain resource management module. Clearly, such systems are quite frequent in practice. At the same time, verification of systems of this kind offers many interesting challenges, such as dealing with parameterization and/or infinite data structures.

The Problem. Ericsson's AXD 301 [9] is a high capacity ATM switch that is used, for example, in backbone networks to connect city telephone exchanges [5]. It embeds several workstations (at least two) together with some specialized hardware. The complete software equipment of AXD 301 is too complex to be fully verified using the current technology. Nevertheless, it is still possible to concentrate on some critical parts of the software which are especially worth of verification. One such part is related to the resource management problem: An arbitrary number of *client* processes competes for an *exclusive* or *non-exclusive* (shared) access to an arbitrary number of resources under the supervision of a single *locker* process. More specifically, in AXD 301, processes lock applications they are using to prevent them from being moved from one workstation to another (or, on the other hand, to allow them to be moved). The library designed for this reason is, however, written in such a way that it could be used in other contexts as well (possibly with different locker strategies).

In [4, 5], the authors tried to apply some finite-state verification methods to check several properties (like mutual exclusion or absence of starvation) of the resource management system of AXD 301. They verified the properties in several settings given by a certain number of participating clients and a certain number of accessible resources. Their results are, of course, quite significant, but it would further be interesting to be able to verify that the system satisfies the desired properties with *any* number of clients and/or with *any* number of resources. The verification method should remain as automatic as possible. Moreover, with respect to the importance of the resource management problem in different contexts, it is sensible to try to cover as many different types of client and locker behaviours as possible for generic property classes such as those expressible in (fragments of) suitable temporal logics.

The Approach. The above described parametric verification problem in its full generality is obviously very complex—it involves dealing with up to two parameters and several (possibly different) infinite data structures (such as queues, stacks, or lists) used in the lockers. Here, we suppose dealing with queue-based locker strategies, a fixed number of resources, and a parametric number of identical clients. (An extension to additionally having a fixed number of specific clients seems to be relatively easy to introduce.)

Our approach to the problem is to split it into two parts: (1) to verify that systems of arbitrary many clients behave well provided they are controlled by a locker that implements a certain locker strategy and (2) to check that for an arbitrary number of clients with an arbitrary behaviour, the given locker really implements the strategy we suppose it to implement. In this work, we concentrate on the first issue and we take into account two locker strategies—namely FIFO and FIFO with priorities—that are very common in practice. (Besides, we also already have some preliminary results on a FIFO locker with overtaking of blocked processes and on a service-in-random-order locker as it is mentioned in the end of the report.)

The Results. In the work, we propose a simple model based on extended automata with queues that is suitable for an abstract description of systems of clients competing for shared resources under the supervision of some of the considered queue-based lockers. Over models of this kind, we consider the parametric verification problem for various fragments and/or modifications of CTL* enriched with local and global process quantification. (We denote the quantified processes as *visible* in the following.) The most relevant properties for the given setting, such as mutual exclusion, absence of starvation, or process deadlockability, are covered in this way.

We obtain three main kinds of results here. First, we prove that many verification problems for the parameterized systems we consider can be reduced to finite-state model-checking problems, i.e. verifying systems with up to a certain small number of clients is enough to verify systems with an arbitrary number of clients. These so-called *cut-off* sizes of systems depend on the number of resources used and the number of visible (quantified) processes in the formula of interest. Second, we show that some verification problems cannot be solved in the just mentioned way, but they may still be decidable because there may exist cut-offs additionally depending on the control structure of the clients and/or the structure of the formula. For a certain interesting class of liveness properties and for families with priorities, we then really derive such a complex cut-off. Finally, we prove that some verification problems are undecidable in the given setting.

Related Work. Several different approaches have been proposed for verifying parameterized networks of processes and/or systems dealing with infinite data structures. Let us mention, for example, the use of regular model checking [19, 22, 1, 12, 3], (automated) abstraction [6, 8, 7], or network invariants [23, 18, 20, 21]. The idea of finding instances of parameterized systems whose verification is sufficient to verify the whole parameterized system has already been used in several different contexts [17, 14, 15, 13] as well.

The paper [13] is probably the closest to this work. It also targets verification of systems with shared resources and employs cut-offs too. However, between [13] and this work there are some differences in the way of modelling clients and in the considered classes of properties to be verified. Moreover and most importantly, [13] allows for heterogeneous systems of clients, but service in random order is the only considered locker strategy here.

As we have already mentioned, some finite state instances of the problems considered here were examined in [4, 5] where a special emphasis was laid on working directly on the source code level.

Outline. The rest of the report is organized as follows. We first informally characterize the model we use for describing parameterized systems of clients and then we formalize it. We describe the way we specify properties to be verified. Subsequently, we present our cut-off results for the areas of safety, deadlockability, and liveness. We go on to the undecidability results we have. We finish by a short summary and an outline of the future work. Let us note that we only sketch proofs of the theorems here—the full proofs will be included in an extended version of this report.

2 Systems of Clients Competing for Resources

We now describe a little more the model we use for describing parameterized systems of clients accessing some shared resources. We also explain the two common locker strategies that we cover in our considerations.

2.1 The notion of RTR Families

For an abstract description of parameterized systems of clients competing for shared resources under the control of a certain queue-based locker, we propose a simple model in the form of the so-called *RTR families*. (RTR stands for request-take-release.) The model is based on extended automata with queues. More precisely, an RTR family is a set of systems of processes that are controlled by some *RTR control automaton* defined on a certain finite set of resources and a certain locker function. An RTR control automaton is a common finite automaton with transitions labeled by actions allowed for clients.

Apart from the *idle* action (denoted by τ), RTR automata may contain actions of *requesting*, *taking*, and *releasing* subsets of the set of resources. We suppose the request operation `req`(R) to be a non-blocking operation which only informs the locker that the appropriate client will (sooner or later) need the resources in the set R. A request operation may then be followed by a take operation `take`(R) on (a subset of) the requested resources. This operation is blocking and may be finished only when the appropriate resources can be granted to the client asking for them. The release operation `rel`(R) informs the locker that the current owner of R does not need these resources any more and they can be assigned to some other interested clients.

Apart from the basic request and take operations, we also have their versions for *high-priority* or *non-exclusive* access to resources (denoted by `preq`, `ptake` or `nreq`, `ntake`, respectively). Finally, we introduce combined *request-take* actions (denoted by `rqt`, `prqt`, `nrqt`) that correspond to a combination of the request and take actions with no interleaving inbetween (i.e. a client requests and immediately tries to take some set of resources). We denote by (RT)R the families that allow `rqt`-like actions only.

2.2 Lockers

The first locker we consider here is the *FIFO* locker. It handles all request and take operations according to the strict first-in first-out strategy where the "first-in" relates to request and the "first-out" to take operations. We define the locker to be eager in the sense that as soon as some resources may be granted to some client trying to take them, they will be granted to this client.

We illustrate the behaviour of systems with the FIFO locker in Fig. 1. Suppose we have an RTR family with two resources (`A` and `B`), the control automaton of clients shown in Fig. 1, and equipped with the FIFO locker. Further suppose we have the system of this family with three clients in its initial state (all clients are in the initial control state,

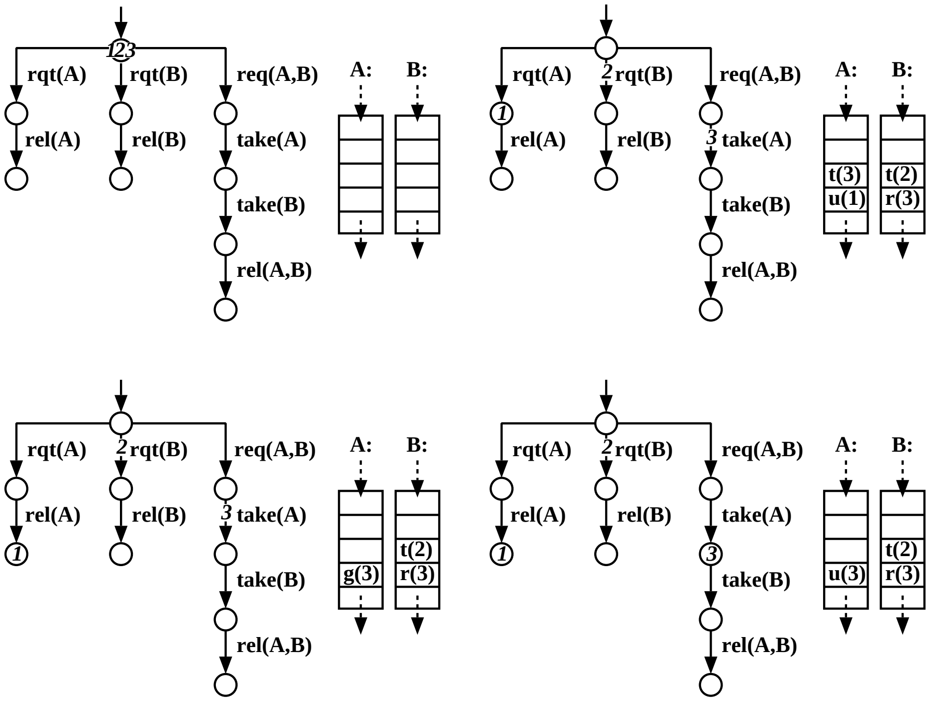

Figure 1: A sample system equipped with the FIFO locker

all queues that we associate with the resources are empty)—this situation is depicted in the upper left part of Fig. 1.

Now, assume process `1` requests and tries to take `A`. As `A` is free, it is granted to `1`, which may start (and suppose it starts) using it. The new state of `1` and the `u(1)` item in the queue associated with `A` record what has just happened. Then, if `3` requests `A` and `B`, this is recorded via `r(3)` in both the queues, and `3` may go on and try to take `A`. This is, however, impossible because `A` is already in use, and so `3` remains blocked in the middle of the take operation, and the fact that it wants to take `A` is recorded via `t(3)` in the queue of `A`. Subsequently, if `2` tries to take `B`, we arrive to the state depicted in the upper right part of Fig. 1. In the next step, assume that `1` releases `A`. Then, due to the eager nature of the locker, `A` is immediately granted to `3`, and we obtain the situation in the bottom left part of Fig. 1. (Here, such a behaviour is not so important, but in the case of FIFO with priorities, it rules out overtaking by processes that arrive only after the resources which their lower-priority predecessors are trying to take became free.) Finally, `3` is now unblocked and may start using `A`, which leads to the bottom right situation in Fig. 1.

Let us proceed to the *PRIO* locker. It is a modification of the FIFO locker that allows for two priority levels of the operations. A high-priority request is now allowed to overtake unserved low-priority requests provided it is not blocked in any other way (i.e. it is not blocked by any of the requested resources being in use or being requested by some high-priority requests that have not been served yet and that precede the given request). For example, let us have look at Fig. 2. In its left scenario, process `3` is blocked by `1`, process `2` whose take operation is of a high priority will overtake, and the free resource `B` will be granted to it. The middle scenario of Fig. 2 shows that the overtaking can also happen when some resources are released, and in such a case, a high-priority request can overtake a low-priority one even if the latter request is not blocked by any preceding requests. On the other hand, due to the eager behaviour of PRIO, when `3` tries to take `A` and `B` when

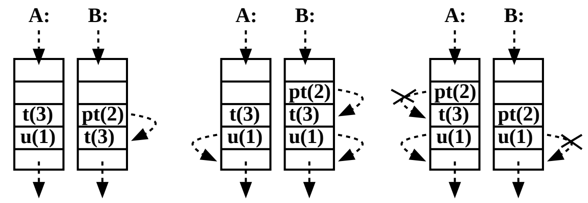

Figure 2: Typical overtaking under the PRIO discipline

they are both free, they will be granted to it, and a subsequent high-priority attempt of `2` to take `B` will be blocked. Moreover, as shown in the right scenario of Fig. 2, if a high-priority request cannot be granted, it does not block the low-priority requests preceding it—in the given scenario, after `1` releases `A` (but not `B`), `A` will be granted to `3`.

We use RTR_{NFIFO} to refer to RTR families equipped with the FIFO locker and RTR_{FIFO} to refer to families with the FIFO locker and without the non-exclusive access option. A similar notation applies for the use of PRIO as well.

3 The Formal Definition of RTR

In this section, we formalize the notion of RTR families which we use for describing parameterized systems of clients competing for shared resources. We first define the general notion of RTR families and then provide formal definitions of the FIFO and PRIO lockers.

3.1 The General Framework of RTR

We first define a notion of *RTR control automata* over a finite set of resources and a locker.

Definition 3.1 *An* RTR control automaton $\mathcal{A}$ *is a 5-tuple* (R, f, Q, q_0, T) *where* R *is a set of resources,* f *is a locker function described more precisely below,* Q *is a set of process control locations,* $q_0 \in Q$ *is an initial process control location, and* $T \subseteq Q \times A \times Q$ *is a process transition relation with* A *including the following (for all* $R' \subseteq R$, $R' \neq \emptyset$*):*

- τ
- $\texttt{req}(R')$, $\texttt{preq}(R')$, $\texttt{nreq}(R')$
- $\texttt{take}(R')$, $\texttt{ptake}(R')$, $\texttt{ntake}(R')$
- $\texttt{rqt}(R')$, $\texttt{prqt}(R')$, $\texttt{nrqt}(R')$
- $\texttt{rel}(R')$

The sets R, Q, T, *and* A *are finite, non-empty, pairwise disjoint, and disjoint with* $\mathbb{N}$.

An *RTR family* $\mathcal{F}(\mathcal{A})$ over an RTR automaton $\mathcal{A}$ is a set of *systems* S_n of concurrent processes having $n \geq 1$ identical processes with their finite control given by $\mathcal{A}$, working over the resources of $\mathcal{A}$, and identified by $p \in P_n = \{1, ..., n\}$. (In the following, if no confusion is possible, we often drop the reference to $\mathcal{A}$.)

As we have also already mentioned, we sometimes restrict ourselves to RTR families in which only the `rqt`, `prqt`, `nrqt` (together with τ and `rel`) operations can be used. We denote such families as *(RT)R families*.

Let $\Sigma = \{\mathtt{r}, \mathtt{pr}, \mathtt{nr}, \mathtt{t}, \mathtt{pt}, \mathtt{nt}, \mathtt{g}, \mathtt{ng}, \mathtt{u}, \mathtt{nu}\}$ be the *resource queue alphabet*. For a system S_n, we define its indexed queue alphabet $\Sigma_n = \{s(p) \mid s \in \Sigma \wedge p \in P_n\}$. The meaning is that a process is requesting a certain resource or wants to take it in the common, high-priority, or non-exclusive way, is granted the resource, or is using the resource in the exclusive or non-exclusive way.

Now, a *configuration* c of a system S_n from a family $\mathcal{F}(\mathcal{A})$ with $\mathcal{A} = (R, f, Q, q_0, T)$ can be formally defined as a function $c : (P_n \to Q \cup T) \cup (R \to \Sigma_n^*)$. The first component specifies control states of clients (note that they can blocked in the middle of some actions) and the second specifies the contents of queues. We will denote the set of all such configurations by C_n and use also $C = \bigcup_{n\in\mathbb{N}} C_n$.

Subsequently, we can restrict the definition of *locker functions* to functions $f : C \to 2^C$ such that $c \in C_n \Rightarrow f(c) \subseteq C_n$ for all $n \in \mathbb{N}$. The intuition behind this is that a locker takes a configuration (changed by adding/removing some request to/from some queues) and decides which resources can be granted to which processes. (Note that several different decisions can be possible for a single configuration.)

Let us now have a family $\mathcal{F}(\mathcal{A})$ with $\mathcal{A} = (R, f, Q, q_0, T)$ and a system $S_n \in \mathcal{F}$. We define *enabling* and *firing of transitions* in S_n via the predicate $en \subseteq C_n \times T \times P_n$ and the function $to : C_n \times T \times P_n \to 2^{C_n}$.

First, for all transitions $t = (q_1, \tau, q_2) \in T$ and $t = (q_1, a(R'), q_2) \in T$, $a \in \{\mathtt{req}, \mathtt{preq}, \mathtt{nreq}, \mathtt{rel}\}$, we define $en(c, t, p) \Leftrightarrow c(p) = q_1$. For the remaining kinds of actions $a(R') \in A$ and $t = (q_1, a(R'), q_2) \in T$, we define $en(c, t, p)$ to hold iff one of the following is satisfied:

1. $c(p) = q_1$,
2. $c(p) = t \wedge \forall r \in R'\ \exists w_1, w_2 \in \Sigma_n^*\ \exists s \in \{\mathtt{g}, \mathtt{ng}\} : c(r) = w_1.s(p).w_2$.

We will sometimes denote transitions enabled according to the first item above as $\mathtt{take}_\uparrow$, $\mathtt{rqt}_\uparrow$, etc. (they correspond to a process asking the locker to allow it to take some resources) and transitions enabled according to the other rule as $\mathtt{take}_\downarrow$, $\mathtt{rqt}_\downarrow$, etc. (they correspond to a process starting to use some resources after having been allowed to do so).

Let us next define what we can obtain by *firing an enabled transition*. The firing rules are in general nondeterministic meaning that any of the mentioned targets can be reached by firing a certain transition from a certain configuration. In the definitions, we will use the function $cfst$ which changes the status of the first record related to a process $p \in P_n$ within the queue content string $w \in \Sigma_n^*$ from $s \in \Sigma$ to $s' \in \Sigma$, i.e., $cfst(p, w, s, s')$ equals $w_1.s'(p).w_2$ if $\exists w_1 \in \Sigma_{n\setminus p}^*\ \exists w_2 \in \Sigma_n^* : w = w_1.s(p).w_2$, and w otherwise. (In $\Sigma_{n\setminus p}$, we exclude indexing by p.)

Now, for an enabled transition $t = (q_1, \tau, q_2) \in T$, we define $to(c, t, p) = f((c \setminus \{(p, q_1)\}) \cup \{(p, q_2)\})$. (We remove the mapping saying that p is in q_1 and add a new mapping saying that p is in q_2.)

For an enabled transition $t = (q_1, \mathtt{req}(R'), q_2) \in T$, we define $to(c, t, p) = f((c \setminus c^-) \cup c^+)$ where $c^- = \{(p, q_1)\} \cup \{(r, c(r)) \mid r \in R'\}$ and $c^+ = \{(p, q_2)\} \cup \{(r, c(r).\mathtt{r}(p)) \mid r \in R'\}$. The effect of firing transitions based on $\mathtt{preq}$ and $\mathtt{nreq}$ is the same up to $\mathtt{r}$ replaced by $\mathtt{pr}$ or $\mathtt{nr}$, respectively.

For an enabled transition of the form $t = (q_1, \mathtt{take}_\uparrow(R'), q_2) \in T$, we define $to(c, t, p)$ and c^- as above, but $c^+ = \{(p, t)\} \cup \{(r, cfst(p, c(r), \mathtt{r}, \mathtt{t})) \mid r \in R'\}$. Firing transitions based on $\mathtt{ptake}_\uparrow$ and $\mathtt{ntake}_\uparrow$ is the same up to $\mathtt{r}/\mathtt{t}$ replaced by $\mathtt{pr}/\mathtt{pt}$ or $\mathtt{nr}/\mathtt{nt}$, respectively.

In the case of $a \in \{\mathtt{take}, \mathtt{ptake}, \mathtt{ntake}\}$ and an enabled transition of the form $t = (q_1, a_\downarrow(R'), q_2) \in T$, we fix $to(c, t, p)$ to $f((c \setminus c^-) \cup c^+)$ with $c^- = \{(p, t)\} \cup \{(r, c(r)) \mid r \in R'\}$ and $c^+ = \{(p, q_2)\} \cup \{(r, cfst(p, c(r), s, s')) \mid r \in R'\}$ where $s = \mathtt{g}, s' = \mathtt{u}$ if $a \in \{\mathtt{take}, \mathtt{ptake}\}$, and $s = \mathtt{ng}, s' = \mathtt{nu}$ if $a = \mathtt{ntake}$.

The effect of firing $\texttt{rqt}_\uparrow$ can be defined as the obvious one-step combination of firing $\texttt{req}$ and $\texttt{take}_\uparrow$; firing $\texttt{rqt}_\downarrow$ is then to be the same as the one of $\texttt{take}_\downarrow$. Similar definitions can be introduced for $\texttt{prqt}$ and $\texttt{nrqt}$.

Finally, for each transition $t = (q_1, \texttt{rel}(R'), q_2) \in T$ that is enabled, we define $to(c,t,p) = f((c \setminus c^-) \cup c^+)$ with $c^- = \{(p,q_1)\} \cup \{(r, c(r)) \mid r \in R' \wedge \exists w_1 \in \Sigma^*_{n\setminus p}\ \exists w_2 \in \Sigma^*_n\ \exists s \in \{\texttt{u},\texttt{nu}\} : c(r) = w_1.s(p).w_2\}$ and $c^+ = \{(p,q_2)\} \cup \{(r, w_1.w_2) \mid r \in R' \wedge w_1 \in \Sigma^*_{n\setminus p} \wedge w_2 \in \Sigma^*_n \wedge \exists s \in \{\texttt{u},\texttt{nu}\} : c(r) = w_1.s(p).w_2\}$.

To conclude this subsection, we define the *initial configuration* c_0 of a system S_n such that $\forall p \in P_n : c_0(p) = q_0$ and $\forall r \in R : c_0(r) = \epsilon$. By a *finite behaviour* of S_n, we then understand a sequence $c_0, (p_1,t_1), c_1, ..., (p_l,t_l), c_l$ such that for each $i \in \{1,...,l\}$, $en(c_{i-1}, t_i, p_i)$ holds and $c_i \in en(c_{i-1}, t_i, p_i)$. An *infinite behaviour* of S_n can be defined in the obvious way.

3.2 The FIFO Locker

We will now define how the locker function $f : C \to 2^C$ over C of some family $\mathcal{F}$ should look like so that we can call it the *FIFO locker function*. For the rest of this subsection, let us suppose working with an arbitrary fixed $c \in C_n$ of a fixed family $\mathcal{F}$.

Let us first define P_1 and P_2 to be the sets of processes that will be enabled to fire some $\texttt{take}_\downarrow$, $\texttt{rqt}_\downarrow$, or $\texttt{ntake}_\downarrow$, $\texttt{nrqt}_\downarrow$ transitions, respectively. To be in P_1, a process must be in the middle of a $\texttt{take}$ or $\texttt{rqt}$ operation and it must be in the head of the queues of all the resources it is trying to take. In the case of P_2, the process must be in the middle of an $\texttt{ntake}$ or $\texttt{nrqt}$ operation and in the queues, it may be preceded by requests of other processes, but only when these requests are non-exclusive. High-priority actions are ignored here:

- $P_1 = \{p \in P_n \mid \exists q_1, q_2 \in Q\ \exists R' \subseteq R\ \exists a \in \{\texttt{take},\texttt{rqt}\} : c(p) = (q_1, a(R'), q_2) \wedge \forall r \in R'\ \exists w \in \Sigma^*_n : c(r) = \texttt{t}(p).w\}$.
- $P_2 = \{p \in P_n \mid \exists q_1, q_2 \in Q\ \exists R' \subseteq R\ \exists a \in \{\texttt{ntake},\texttt{nrqt}\} : c(p) = (q_1, a(R'), q_2) \wedge \forall r \in R'\ \exists w_1, w_2 \in \Sigma^*_n : c(r) = w_1.\texttt{nt}(p).w_2 \wedge w_1 \in \{s'(p') \mid s' \in \{\texttt{nr},\texttt{nt},\texttt{ng},\texttt{nu}\} \wedge p' \in P_n\}^*\}$.

Let further R_1 and R_2 be the sets of resources affected by P_1 and P_2:

- $R_1 = \{r \in R \mid \exists p \in P_1\ \exists w \in \Sigma^*_n : c(r) = \texttt{t}(p).w\}$.
- $R_2 = \{r \in R \mid \exists p \in P_2\ \exists w_1, w_2 \in \Sigma^*_n : c(r) = w_1.\texttt{nt}(p).w_2\}$.

We define $c^- = \{(r, c(r)) \mid r \in R_1 \cup R_2\}$ as the set of items that change due to granting some resources. Moreover, we introduce $c^+_1 = \{(r, \texttt{g}(p).w) \mid r \in R_1 \wedge p \in P_1 \wedge w \in \Sigma^*_n \wedge c(r) = \texttt{t}(p).w\}$ as the set of items recording new exclusive grants. Then, the set of items recording non-exclusive grants is $c^+_2 = \{(r, s'_1(p_1)...s'_l(p_l)) \mid r \in R_2 \wedge l \in \mathbb{N} \wedge s'_1, ..., s'_l \in \Sigma \wedge p_1, ..., p_l \in P_n \wedge \exists s_1, ..., s_l \in \Sigma : c(r) = s_1(p_1)...s_l(p_l) \wedge \forall i \in \{1,...,l\} : (p_i \in P_2 \wedge s_i = \texttt{nt} \Rightarrow s'_i = \texttt{ng}) \wedge (\neg(p_i \in P_2 \wedge s_i = \texttt{nt}) \Rightarrow s'_i = s_i)\}$. The latter expression is more complex because the same resource can be granted in the non-exclusive way to several processes at the same time, and so we have to carefully update the contents of the whole queue.

Now, we say that f is the *FIFO locker* iff $f(c) = \{(c \setminus c^-) \cup c^+_1 \cup c^+_2\}$ (for any $c \in C$). Finally, we establish the notion of RTR_{NFIFO} families as the RTR families based on the FIFO locker and excluding any use of the $\texttt{preq}$, $\texttt{ptake}$, and $\texttt{prqt}$ actions. In the context of RTR_{FIFO} families not allowing for the non-exclusive access, we then exclude any use of $\texttt{nreq}$, $\texttt{ntake}$, and $\texttt{nrqt}$ actions too.

3.3 The PRIO Locker

For the rest of this subsection, let us again suppose working with an arbitrary fixed $c \in C_n$ of a fixed family $\mathcal{F}$. We start by introducing the effect of processing high-priority requests and then use a reference to the FIFO locker.

First, we specify the set of high-priority processes that will be granted some resources as follows $P' = \{p \in P \mid \exists q_1, q_2 \in Q\ \exists R' \subseteq R\ \exists a \in \{\mathtt{ptake}, \mathtt{prqt}\} : c(p) = (q_1, a(R'), q_2) \wedge \forall r \in R'\ \exists w_1, w_2 \in \Sigma_n^* : c(r) = w_1.\mathtt{pt}(p).w_2 \wedge \neg(\exists p' \in P\ \exists s' \in \{\mathtt{pr}, \mathtt{pt}, \mathtt{g}, \mathtt{ng}, \mathtt{u}, \mathtt{nu}\}\ \exists w_1', w_2' \in \Sigma_n^* : w_1 = w_1'.s'(p').w_2')\}$. Let further R' be the set of resources affected by P': $R' = \{r \in R \mid \exists p \in P' \wedge \exists w_1, w_2 \in \Sigma_n^* : c(r) = w_1.\mathtt{pt}(p).w_2\}$.

We can now define the effect of granting some resources to some high-priority processes via $c^- = \{(r, c(r)) \mid r \in R'\}$ and $c^+ = \{(r, \mathtt{g}(p).w_1.w_2) \mid r \in R' \wedge p \in P' \wedge w_1, w_2 \in \Sigma_n^* \wedge c(r) = w_1.\mathtt{pt}(p).w_2\}$.

Then, provided f' is the FIFO locker, we say that f is the *PRIO locker* iff $f(c) = f'((c' \setminus c^-) \cup c^+)$ (for any $c \in C$ and the corresponding c'). Finally, we establish the notion of RTR_{NPRIO} families as the RTR families based on the PRIO locker. The more restricted notion of RTR_{PRIO} families excludes any use of `nreq`, `ntake`, and `nrqt` actions.

4 Specifying Properties To Be Verified

In this work, we study various generic classes of properties to be verified, but it is always possible to view them as expressible in some fragment and/or modification of CTL* enriched with process quantification. We take care to cover the most important properties in the given setting including mutual exclusion, absence of starvation, and process deadlockability.

Our concept of process quantification is inspired by ICTL* (see, e.g., [16]). We work with the following kinds of process quantification here:

- *Global process quantification.* In some global state of the system being investigated, we select some processes and then monitor their behaviour throughout some state space paths leading from this state. As an example of a property expressible in this way, we can mention absence of starvation: $\forall_p\ AG\ (.p = q_{request} \Rightarrow AF\ .p = q_{grant})$.[1]

- *Local process quantification.* Within the current global state, we select some processes and check their local states. This choice is independent of the choices made in other states. For example, we can express simultaneous local reachability in this way: $EF\ \forall_p\ .p = q$.

  Note that $\forall_p\ EF\ .p = q$ has a different meaning than the above formula. Here, the different processes may reach the location q within different global states.

We can add that some properties can be expressed using both global as well as local process quantification. Mutual exclusion is an example of such a property. In this case, we can use both $\forall_{p_1 \neq p_2}\ AG\ \neg(.p_1 = q_{cs} \wedge .p_2 = q_{cs})$ and $AG\ \forall_{p_1 \neq p_2}\ \neg(.p_1 = q_{cs} \wedge .p_2 = q_{cs})$.

For an RTR family $\mathcal{F}$ and a given formula Φ, we consider the *parametric verification problem* in the form $\forall S \in \mathcal{F} : S \models \Phi$ or the dual $\exists S \in \mathcal{F} : S \models \Phi$. So, we have three kinds of quantification here: (1) on systems in families, (2) on processes, and (3) on paths.

[1]We will use the notation $.p = q$ to express the state predicate "the control of process p is in location q".

5 Safety Properties

5.1 The Class of Safety Properties Considered

In this section, we will consider properties expressible in the linear-time fragment of CTL* enriched by process quantification only in the top-most level of the formulae and only by the global one (i.e. we will be interested in the linear-time behaviour of some selected processes from the initial state on). Moreover, as we are interested in safety now, we will interpret path quantifiers over finite paths in the considered state spaces (formally extended to infinite ones by repeating the last state).[2] We will not allow the contents of queues to be accessed.

More precisely, we will consider properties of the form

$$\Phi^k_{fin} \equiv [\exists|\forall]_{p_1,...,p_k|\iota}\ [E|A]_{fin}\ \varphi(p_1,...,p_k)$$

where

- ι is a conjunction of atoms $p_i \neq p_j$ (for $i \neq j$), which requires some process indices to refer to different processes, and
- $\varphi(p_1,...,p_k)$ is an LTL\X formula over atoms of the kind $.p_i = qt$ ($qt \in Q \cup T$) and/or $.p_i = .p_j$, which allows us to refer to control locations of the particular processes.

When we require $p_i \neq p_j$ for all $i \neq j$, we will write $\neq$ instead of ι (or, in the case of dealing with two processes, we will directly write $p_1 \neq p_2$). Let us add that the processes referred to via $p_1,...,p_k$ are the ones that we call visible through the given formula.

As an example of a safety property expressible in the above described way, we may use mutual exclusion: $\forall_{p_1 \neq p_2}\ A_{fin}\ G\ \neg(.p_1 = q_{cs} \wedge .p_2 = q_{cs})$.

Examining $[\exists|\forall] S \in \mathcal{F} : S \models \Phi^k_{fin}$ then corresponds to checking whether for some system (all systems) $S \in \mathcal{F}$, for some selection (all selections) of processes $p_1,...,p_k$ within S that fulfill ι, and for some finite behaviour (all finite behaviours) π of S, $\pi \models \varphi(p_1,...,p_k)$. In this context, with respect to the common definition of $\pi \models \varphi$ for LTL formulae, we suppose $\models$ in $\pi \models \varphi$ to additionally embed the restriction of π to a sequence of configurations (we leave out the transitions) and its subsequent extension by infinitely many times repeating the last configuration.

5.2 Safety in RTR$_{FIFO}$

We will now show that it is sufficient to consider systems comprising up to k processes (i.e. $S_1,...,S_k$) when verifying $\forall S \in \mathcal{F} : S \models \Phi^k_{fin}$ (or dually $\exists S \in \mathcal{F} : S \models \Phi^k_{fin}$) for an RTR$_{FIFO}$ family $\mathcal{F}$. Moreover, we will show that the basic verification task is to verify $S_n \models E_{fin}\ \varphi(1,...,n)$ for some suitably chosen values of n, $n \leq k$. In this way, we transform the parametric verification problem to a non-parametric one with a bound on the parameter equal to the number of visible processes in the formula of interest. This means that, for instance, to verify mutual exclusion in an RTR$_{FIFO}$ family, it suffices to verify it for the system of this family with two processes.

Below, we first state a basic cut-off lemma for a certain subclass of safety properties and subsequently we generalize it to the above result. In the lemma, we consider properties of the form "there exist k different processes and a finite behaviour for which a given LTL\X formula holds".

[2] Such an approach has been used in a similar setting in [13] as well.

Lemma 5.1 *For an RTR_{FIFO} family $\mathcal{F}$ and an LTL\X formula $\varphi(p_1, ..., p_k)$ with atoms based on the control locations of $p_1, ..., p_k$, the following holds for systems of $\mathcal{F}$:*

$$\forall n \geq k : S_n \models \exists_{p_1,...,p_k|\neq}\ E_{fin}\ \varphi(p_1, ..., p_k) \Leftrightarrow S_k \models E_{fin}\ \varphi(1, ..., k)$$

Proof. The case of $\Leftarrow$ is obvious—we let the additional processes of S_n idle. Let us concentrate on the $\Rightarrow$ case.

We have a witness behaviour π of S_n. Let $p_1, ..., p_k$ be such that $\pi \models \varphi(p_1, ..., p_k)$ and $p_i \neq p_j$ for $i \neq j$. We take the transition sequence π_t of π and we transform it to π'_t by removing all transitions done by processes different from $p_1, ..., p_k$ and by a subsequent one-to-one replacement of all appearances of $p_1, ..., p_k$ in the preserved part of π_t by $1, ..., k$. It remains to show that there is a behaviour π' of S_k corresponding to π'_t and that $\pi' \models \varphi(1, ..., k)$.

Under FIFO, transitions based on τ and `take` do not influence enabledness of transitions in other processes, and (as there is no overtaking here) transitions based on `req` and `rqt` can only block some processes, but they cannot unblock any process. The only transitions whose removal from π_t could disable some of the originally enabled transitions are the transitions based on `rel`. However, as we always start with empty queues and a process cannot influence queue items related to other processes, `rel` in processes different from $p_1, ..., p_k$ will not have any effect after transitions based on the other actions of these processes have been removed. So, by removing all transitions done by processes not among $p_1, ..., p_k$ from π_t, we obtain a transition sequence firable in S_n. Moreover, as the names of processes do not influence transition enabledness nor firing in RTR, π'_t yields a behaviour π' of S_k. Finally, since the states of $1, ..., k$ are being changed in the same way as in the case of $p_1, ..., p_k$ (and from the same initial values), the states of processes not among $p_1, ..., p_k$ are invisible for φ, φ does not depend on the concrete names of processes, and LTL\X is stuttering insensitive, we have $\pi' \models \varphi(1, ..., k)$. □

Theorem 5.1 *For each RTR_{FIFO} family $\mathcal{F}$ and each safety property expressed as Φ^k_{fin}, the validity of $[\exists|\forall]S \in \mathcal{F} : S \models \Phi^k_{fin}$ (with a possible additional restriction to systems with at least l processes) can be deduced from the validity of $S_n \models E_{fin}\ \varphi(1, ..., n)$ for (possibly several) $n \in \{1, ..., k\}$.*

Proof. With respect to the three levels of quantification, we can split the proof to 8 cases by considering $\exists$ or $\forall$ $S \in \mathcal{F}$ combined with $\exists$ or $\forall$ $p_1, ..., p_k$ and with E_{fin} or A_{fin}. Here, we cover only one of the cases—the others are similar (cf. the extended version of this report) or can be treated by negation.

In particular, we consider the (practically most important) case of $\exists S \in \mathcal{F} : S \models \exists_{p_1,...,p_k|\iota}\ E_{fin}\ \varphi(p_1, ..., p_k)$. The query $S \models \exists_{p_1,...,p_k|\iota}\ E_{fin}\ \varphi(p_1, ..., p_k)$ where ι does not necessarily relate all indexed processes can easily be converted to a disjunction of queries with fully specified relations among the visible processes. This can be done by considering all possible combinations of having $=$ or $\neq$ between the uncovered process indices. The use of $p_i = p_j$ can then be emulated by replacing all occurrences of p_j in φ and ι by p_i and considering $false$ all the queries with $p_i \neq p_i$ in their modified ι. In this way, we obtain an existentially quantified disjunction of queries of the form $S \models \exists_{p'_1,...,p'_{k'}|\neq}\ E_{fin}\ \varphi'(p'_1, ..., p'_{k'})$. Subsequently, as there is no choice of k' different indices in S_n with $1 \leq n < k'$ and (due to Lemma 5.1) $S_n \models \exists_{p'_1,...,p'_{k'}|\neq}\ E_{fin}\ \varphi'(p'_1, ..., p'_{k'})$ is equal to $S_{k'} \models E_{fin}\ \varphi'(1, ..., k')$ for all S_n with $n \geq k'$, we can transform the original query to $(S_{k'} \models E_{fin}\ \varphi'(1,...,k')) \vee (S_{k''} \models E_{fin}\ \varphi''(1,...,k'')) \vee ....$ The possible additional restriction to systems with at least l processes does not influence this result. $\square$

So, in Theorem 5.1, we have shown that the parametric verification problem of checking $[\exists|\forall] S \in \mathcal{F} : S \models \Phi^k_{fin}$ can be reduced to checking $S_n \models E_{fin}\ \varphi(1,...,n)$ for some suitably chosen values of $n \in \{1,...,k\}$. The reduction and the way of obtaining the values of n to be considered has been described in the proof of the theorem. They depend on the combination of quantifiers used and on the number of process quantifications with fully specified relations among the visible processes that are hidden in the original process quantification. Note also that Lemma 5.1 can be used not only within Theorem 5.1, but if need be, it can as well be applied to simplify the finite-state verification problem of verifying a system S_n with a certain given number of processes and belonging to some RTR_{FIFO} family.

5.3 Dealing With Queues of Requests

Even when we get rid of the parametric verification problem of $[\exists|\forall] S \in \mathcal{F} : S \models \Phi^k_{fin}$ and replace it by checking $S_n \models E_{fin}\ \varphi(1,...,n)$ for some values of $n \in \{1,...,k\}$, we may still have to deal with systems with infinite state spaces. The cause of this is working with queues whose contents may grow to infinity because, in general, a process may have an arbitrary number of requests in a queue.

Fortunately, as shown in Prop. 5.1, we can easily identify a significant class of RTR families in which the contents of queues is statically guaranteed to be bounded. This result can then be used to assure finiteness of the state spaces being dealt with not only in combination with the RTR_{FIFO} safety cut-off discussed above, but also in combination with the other results presented in this work.

Proposition 5.1 *Let $\mathcal{F}(\mathcal{A})$ be an RTR_{FIFO} or RTR_{PRIO} family over an RTR automaton $\mathcal{A} = (R, f, Q, q_0, T)$. If $\mathcal{A}$ does not contain any loop with more* `(p)req` *operations involving a certain resource $r \in R$ than* `(p)take` *operations involving the same resource,* the queue content is guaranteed to be bounded *for each $S_n \in \mathcal{F}$.*

Proof. As we have only exclusive ownership in the considered families (even for two requests of the same process) and `(p)take`$_\downarrow$ and `(p)rqt`$_\downarrow$ are blocking, the only way how a process may generate more than two items into a queue of a certain resource is via `(p)req`. (The two items are due to a process may take a resource and then request and try to take it for the second time, thus blocking itself forever.) Then, if the assumption of the proposition holds, when looping in any way (and as we have finite control structures, looping is necessary for being able to generate unboundedly many requests), all requested items will be taken and then released not to block the looping process. $\square$

Let us note that the class of families covered by the above is indeed quite meaningful because if the described requirement does not hold, there is either a possibility of a process deadlock in some of the systems of the family at hand, or the content of the queues may grow over every bound in them. Both of these possibilities are then usually considered bad. This phenomenon in RTR is due to only the process that generated some item into some queue may remove it. Consequently, there is no way how two processes could run in loops breaking the above condition and still mutually block themselves such that the queue content could not unboundedly grow. (Such a situation can happen, e.g., in systems of processes communicating via FIFO channels [10, 2]—consider one process looping through

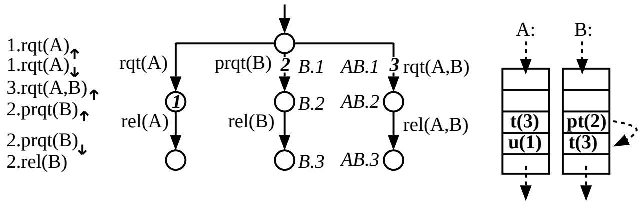

Figure 3: A scenario in (RT)R_{PRIO} problematic for the application of cut-offs

$c_1!m_1; c_2?m_2$ and another looping via $c_1?m_1; c_2!m_2$.) Finally, the class of (RT)R families that seems to be very common in practice is fully covered by the above result.

5.4 Safety in RTR$_{PRIO}$

Unfortunately, in the context of RTR$_{PRIO}$ (and even (RT)R$_{PRIO}$), we cannot—in general—apply the strongly simplifying result obtained for RTR$_{FIFO}$. The reason is that when we remove actions of some processes from a transition sequence firable in a system of an (RT)R$_{PRIO}$ family, we need not obtain a transition sequence that is still firable.

An example of a scenario illustrating the above phenomenon is shown in Fig. 3. In the system with three processes of the (RT)R$_{PRIO}$ family having resources `A` and `B` and the process control shown in Fig. 3, we have the behaviour shown on the left—note that process `2` overtakes process `3`. However, when we suppose that only processes `2` and `3` are visible for some property and we try to cancel all actions of process `1` as in Lemma 5.1, we will not obtain a behaviour. This is because both of the resources will immediately be granted to process `3` and no overtaking will be possible.

Using an extension of the example from Fig. 3, we can even show that the following theorem holds.

Theorem 5.2 *For safety properties expressed as Φ^k_{fin} and for (RT)R$_{PRIO}$ families $\mathcal{F}$ with m resources, the validity of $[\exists|\forall] S \in \mathcal{F} : S \models \Phi^k_{fin}$ is not, in general, decidable by using just the systems $S_1, S_2, ..., S_n \in \mathcal{F}$ with n being a bound derived from k and m.*

Proof. Let us consider the (RT)R$_{PRIO}$ family $\mathcal{F}$ with resources `A` and `B` and the process control shown in Fig. 3. We can ask whether a process can overtake another process within some system of $\mathcal{F}$ in the following way: $\exists S \in \mathcal{F} : S \models \exists_{p_1 \neq p_2} E_{fin}\ ((.p_2 \neq \mathtt{B.1})\ U\ (.p_1 = \mathtt{AB.1}))\ \wedge\ ((.p_1 \neq \mathtt{AB.2})\ U\ (.p_2 = \mathtt{B.2}))$. In the query, we refer to the control locations defined in Fig. 3 and we ask whether some process p_1 can request `A` and `B` (location `AB.1`) before some other process p_2 requests `B` (location `B.1`), but still the wish of p_2 is granted (`B.2`) before the wish of p_1 (`AB.2`). Such a situation can really happen in $S_n \in \mathcal{F}$ with $n \geq 3$ (cf. the run shown on the left in Fig. 3 and the overtaking between processes `2` and `3`). Note, however, that without an auxiliary process, the overtaking is not possible (when process `1` is left out from the presented witness behaviour, `A` and `B` are immediately granted to process `3`).

We can now extend the `B` branch by another pair of subsequent actions `prqt(B)` and `rel(B)` and the `AB` branch by another pair `rqt(A,B)` and `rel(A,B)`. Moreover, we can exclude the possibility of having more than one active process in each of these branches by adding `rqt(C)`/`rel(C)` (`rqt(D)`/`rel(D)`) at their beginnings and ends, respectively. Over the new family, we ask whether after the first overtaking, the processes can meet in `B.3` and `AB.3` and then exhibit another overtaking. As we do not extend the left-most branch, and no auxiliary processes can run in the middle or right branch, we need at least 4 processes to show satisfaction of the new property in the new setting. Obviously, we can keep extending the system and the formula in the described way (of course, without adding further operations on `C` and `D`), which will lead to a necessity of considering bigger and bigger systems although k as well as m remain the same. $\square$

Theorem 5.2 shows that the parametric verification problem for safety properties of the considered kind cannot be reduced to dealing only with systems with the number of processes bounded just by the number of resources and the number of visible processes. However, there is still some hope that the problem can be transformed into a finite-state verification problem with the upper bound on the number of processes to be considered depending also on the structure of the process control and on the structure of the formula under investigation. We leave the problem in its general form open for the future research. Nevertheless, we show below how to solve it for certain interesting subclasses of safety properties and the quite frequent (RT)R$_{PRIO}$ subclass of RTR$_{PRIO}$ families. In fact, we show that in their case, we can modify our proof constructions such that the results obtained for RTR$_{FIFO}$ families remain applicable.

The results we will present are related to (RT)R$_{PRIO}$ because in its case, we can replace overtaking in a witness behaviour by postponing the actions being overtaken. We can afford for this because the processes being overtaken are waiting inside some `rqt` actions and they do not do any actions between requesting the appropriate resources and starting to wait. Therefore, we can simply make them wait before the critical transitions rather than inside them. Note that this is impossible in the general case because there can be some additional steps between `req` and the corresponding `take`. The properties we will be able to handle in the just mentioned way must, of course, be insensitive to the described modification of the witness behaviours.

The first class of safety properties we will consider is the class of simple *reachability/invariance* properties

$$\Psi^k_{fin} \equiv [\exists|\forall]_{p_1,...,p_k|\iota}\ [E_{fin}\ F|A_{fin}\ G]\ \psi(p_1,...,p_k)$$

in which ι plays the same role as in Φ^k_{fin} and $\psi(p_1,...,p_k)$ is a non-temporal formula based on the same kind of atoms as the ones used in Φ^k_{fin}. Mutual exclusion is an example of a property that falls into this class. We will now prove a variant of Lemma 5.1 for Ψ^k_{fin}.

Lemma 5.2 *For each (RT)R$_{PRIO}$ family $\mathcal{F}$ and each non-temporal formula $\psi(p_1,...,p_k)$ with atoms based on the control locations of $p_1,...,p_k$, the following holds for systems of $\mathcal{F}$:*

$$\forall n \geq k : S_n \models \exists_{p_1,...,p_k|\neq}\ E_{fin}\ F\ \psi(p_1,...,p_k) \Leftrightarrow S_k \models E_{fin}\ F\ \psi(1,...,k)$$

Proof. The case of $\Leftarrow$ is obvious—we let the additional processes of S_n idle. Let us concentrate on the $\Rightarrow$ case.

We have a witness behaviour π of S_n. Let $p_1,...,p_k$ be such that $\pi \models F\ \psi(p_1,...,p_k)$ and $p_i \neq p_j$ for $i \neq j$. We transform the transition sequence π_t of π to π'_t by removing all transitions done by processes different from $p_1,...,p_k$ and by postponing all $\texttt{rqt}_\uparrow$ and $\texttt{prqt}_\uparrow$ transitions such that they get immediately before the corresponding $\texttt{(p)rqt}_\downarrow$ transitions if they exist. We move the preserved $\texttt{(p)rqt}_\uparrow$ transitions that do not have any corresponding $\texttt{(p)rqt}_\downarrow$ transition in π_t to the very end of the new transition sequence and we keep their original mutual ordering. Now, we prove that π'_t corresponds to some behaviour π' of S_n. For this, we have to show that all the involved $\texttt{(p)rqt}_\downarrow$ transitions are firable in the appropriate processes at all the points in π'_t where they appear. This is ensured because all (visible) `rel` transitions preceding a given (p)$\mathtt{rqt}_\downarrow$ transition still precede it, all the appropriate resources are thus released in time, and requests of all the processes that could compete with the given one, but should not win according to π_t, are moved behind the concerned (p)$\mathtt{rqt}_\downarrow$ transition.

Finally, we transform π'_t to π''_t by a one-to-one replacement of all appearances of $p_1, ..., p_k$ by $1, ..., k$. As the names of processes do not influence transition enabledness nor firing in RTR, π''_t yields a behaviour π'' of S_k. The transitions done by processes $1, ..., k$ in π'' are the same as the ones done by $p_1, ..., p_k$ in π which means that at the end of π'', the processes $1, ..., k$ will be in exactly the same control locations as the processes $p_1, ..., p_k$ at the end of π. As ψ does not depend on the concrete names of processes and it refers only to control locations of the involved processes, $\pi'' \models F\ \psi(1, ..., k)$ and $S_k \models E_{fin}\ F\ \psi(1, ..., k)$ hold, which concludes the proof. $\square$

Another class of safety properties which can be handled in the way discussed here is the class of properties that can be expressed as Φ^k_{fin} (i.e. using full LTL\X) but that do not distinguish whether a process is in a location from which it can fire (p)$\mathtt{rqt}_\uparrow$ or whether it is already "inside" (p)`rqt`. In other words, these are the properties of the Φ^k_{fin} form in which whenever we have a transition $t = (q, a(R'), q') \in T$ with $a \in \{\mathtt{rqt}, \mathtt{prqt}\}$, we never use $.p = q$ or $.p = t$, but at most $(.p = q \vee .p = t)$. Let us denote such properties by Υ^k_{fin}.

Lemma 5.3 *For each (RT)R$_{PRIO}$ family $\mathcal{F}$ and each property Υ^k_{fin} of the form $\exists_{p_1,...,p_k|\neq}\ E_{fin}\ \upsilon(p_1, ..., p_k)$, the following holds for systems of $\mathcal{F}$:*

$$\forall n \geq k : S_n \models \exists_{p_1,...,p_k|\neq}\ E_{fin}\ \upsilon(p_1, ..., p_k) \Leftrightarrow S_k \models E_{fin}\ \upsilon(1, ..., k)$$

Proof. We can use the same argument as in the proof of Lemma 5.2 to show that from a witness behaviour of S_n, we can obtain a behaviour of S_k. The fact that it is a witness behaviour follows from υ not being able to detect the removal of transitions fired in invisible processes, the change of names of processes, nor the postponed firing of some transitions. The last claim holds because the postponed firing manifests itself just by some processes staying longer in some locations q before going to t for some transitions $t = (q, (\mathtt{p})\mathtt{rqt}(R'), q') \in T$, which cannot be distinguished by $(.p = q \vee .p = t)$. $\square$

Now we are ready to state an analogy of Theorem 5.1 for the context of (RT)R$_{PRIO}$ families and the two above mentioned kinds of safety properties.

Theorem 5.3 *For each (RT)R$_{PRIO}$ family $\mathcal{F}$ and each safety property expressed as Ψ^k_{fin} or Υ^k_{fin}, the validity of $[\exists|\forall] S \in \mathcal{F} : S \models [\Psi|\Upsilon]^k_{fin}$ (with a possible additional restriction to systems with at least l processes) can be deduced from the validity of $S_n \models E_{fin}\ F\ \psi(1, ..., n)$ or $S_n \models E_{fin}\ \upsilon(1, ..., n)$, respectively, for (possibly several) $n \in \{1, ..., k\}$.*

Proof. We can use the same proof construction as in the case of Theorem 5.2 with references to Lemma 5.1 replaced by Lemmas 5.2 and 5.3. $\square$

For the sake of completeness and comparison with the further presented results, let us mention yet another class of properties for which the above result is applicable. It is the class of *single-index properties* with just one visible process. Clearly, when we monitor the behaviour of a single process, we cannot notice that some reordering has been used. This does not make much sense in the case of single-index safety properties where we suffice with one process, and so no reordering is in fact necessary. However, it is useful in the case of single-index liveness properties (as, e.g., starvation) where we might have to monitor a single process, but still accompanied with some number of invisible ones. Unfortunately, as we will show, in the case of liveness in (RT)R$_{PRIO}$, we have to (in general) deal with a number of processes depending on the structure of the control automata of the families under investigation even in the case of single-index properties.

6 Process Deadlockability

Given an RTR family $\mathcal{F}$ and a system $S_n \in \mathcal{F}$, we say that a *process p is deadlocked* in some configuration $c \in C_n$ if there is no configuration reachable from c from which we could fire some transition in p. Process deadlockability does not fit the different subclasses of safety and liveness properties that we examine in other parts of this work, and so we treat it separately.

6.1 Process Deadlockability in RTR$_{FIFO}$

We now show that in order to check whether some process can deadlock in some system of an RTR$_{FIFO}$ family $\mathcal{F}$, it is enough to check whether some process may deadlock within $S_{max(m,2)} \in \mathcal{F}$ where m is the number of resources in $\mathcal{F}$. In this way, we transform the parametric process-deadlockability verification problem to a non-parametric one with the bound on the parameter depending on the number of resources in the given family. As we have already indicated, this result can be combined with the result on dealing with queues of requests formalized in Prop. 5.1.

Theorem 6.1 *Let us have an RTR$_{FIFO}$ family $\mathcal{F}(\mathcal{A})$ over an RTR automaton $\mathcal{A} = (R, f, Q, q_0, T)$ with $|R| = m$. Then, if some process may deadlock in some $S_n \in \mathcal{F}$, some process may deadlock in $S_{max(m,2)} \in \mathcal{F}$ as well.*

Proof. Clearly, if some process may deadlock in $S_n \in \mathcal{F}$ with $n \leq max(m, 2)$, it can also deadlock in $S_{max(m,2)}$. We will now show that the theorem holds for S_n with $n > max(m, 2)$ too.

We neglect the trivial case of having a control automaton with terminal states. Then, if some process p may deadlock in some S_n, $n > max(m, 2)$, it is due to the fact that we can reach a configuration c in which p is in the middle of some $\mathtt{take}(R')$ or $\mathtt{rqt}(R')$ transition, but it can never be granted some $r \in R'$ in any of the future behaviours leading from c. A trivial way how p can deadlock in some take operation is that it does not issue a request for it. We exclude this situation in the following—it can be detected using a single process.

It is easy to see that from c, we can reach a configuration c' in which all the processes to which the head items in the queues of R' belong are deadlocked or (due to the properties of the control automaton) cannot release these resources any more. (Otherwise, p would eventually be granted its desired resources.) We can repeat this argument and due to dealing with finite R, we see that a configuration c'' can be reached from c' in which all the "head" processes P'', $P'' \subseteq P_n$, of some set of resources R'', $R' \subseteq R'' \subseteq R$, are deadlocked or cannot release these resources any more, and, moreover, when they are deadlocked, the resources they want to take are in R''.

If some $p'' \in P''$ is not deadlocked but cannot release some $r'' \in R''$ any more, the situation is again trivial. Then, to show process deadlockability, we suffice with S_2 (but we can also use $S_{max(m,2)}$)—one process first goes to the control location in which it blocks

r'' and from which it cannot release r'' any more, and then another process tries to repeat its behaviour.

We are left with the situation in which all processes in P'' are deadlocked. Note that as $|R''| \leq m$, $|P''| \leq m$ (we have at most as many different head processes as resources). As the processes in P'' are the heads of the queues of all the R'' resources and they are trying to take resources from among of R'' only, they are mutually blocked, and this situation cannot change when we remove requests of the other processes. It is then straightforward to show that this deadlock situation can be reached via a behaviour obtained from the original one by only preserving actions of the processes in P''. So, we can rename the processes in P'' and we have the desired witness in $S_{max(m,2)}$. □

6.2 Process Deadlockability in RTR$_{PRIO}$

The problem of checking process deadlockability in RTR$_{PRIO}$ remains an issue for the future work. Let us just note that compared to RTR$_{FIFO}$, it brings (at least) two additional difficulties: (1) When we leave out actions from some firable transition sequence in the RTR$_{PRIO}$ setting, we need not obtain a firable transition sequence—fortunately, we have already shown how to deal with such a problem at least in (RT)R$_{PRIO}$. (2) It may happen that a process is deadlocked while none of the resources it wants to take is blocked forever. This can be caused by processes with high-priority requests always overtaking the blocked process. Such a scenario is very close to starvation, but we check whether a process *may* starve (i.e. whether there is a behaviour in which it is always being overtaken from some point on), whereas for deadlockability, we need to check whether in *all* behaviours from a certain point on, the process is always being overtaken. It will be interesting to see whether the just described similarity requires a similar solution to the one of starvation (discussed in the following), or whether a simpler approach is possible.

7 Liveness Properties

In this section, we give some results concerning liveness properties.

7.1 The class of Liveness Properties Considered

We consider properties of the form

$$\Phi^k_{xf} \equiv [\exists|\forall]_{p_1,\ldots,p_k|\iota}\ [E|A]_{xf}\ \varphi(p_1, \ldots, p_k)$$

where ι and φ are as in Φ^k_{fin} and xf represents one of the following: uf *unconditional fairness*, sf *strong fairness* and wf *weak fairness*.

Unconditional fairness means that only runs of the systems where each process makes infinitely often some steps are considered. Weak fairness means that only behaviours where a process which is always enabled will eventually move are considered and strong fairness means that only behaviours where a process which is always eventually enabled will eventually move are considered. Weak and strong fairness coincide in our model because the locker is operating in such a way, that a resource granted to a process remains granted as long as the process does not use it.

7.2 Liveness Properties under Unconditional Fairness

In this subsection, we will show that for an (RT)R$_{FIFO}$ family $\mathcal{F}$ using m resources it is sufficient to consider systems comprising up to $m + k$ processes (i.e. S_1,..., S_{m+k}) when verifying $\forall S \in \mathcal{F} : S \models \Phi^k_{uf}$ (or dually $\exists S \in \mathcal{F} : S \models \Phi^k_{uf}$). Moreover, we will show that the basic verification task is to verify $S_n \models E_{fin}\ \varphi(1, ..., n)$ for some suitably chosen values of n ($n \leq m + k$). Fur purely existential/universal k processes suffice (even in the RTR case).

Below, we first state a basic cut-off lemma for a certain subclass of liveness properties and subsequently we generalize it to the above result.

Lemma 7.1 *For an (RT)R_{FIFO} family $\mathcal{F}$ with $m = |R|$ and an LTL$\backslash$X formula $\varphi(p_1, ..., p_k)$ with atoms based on the control locations of $p_1, ..., p_k$, the following holds for systems of $\mathcal{F}$:*

$$\forall l \geq m + k : S_l \models \exists_{p_1,...,p_k|\neq}\ E_{uf}\ \varphi(p_1, ..., p_k) \Leftrightarrow S_{m+k} \models E_{uf}\ \varphi(1, ..., k)$$

Proof. The case of $\Rightarrow$ is the same as in the proof for the corresponding Lemma 5.1 considering safety properties : We just remove some non visible processes from a witness behaviour π for $\varphi(p_1, ..., p_k)$ with more than $m + k$ processes and obtain a new witness with only $m + k$ processes. Notice, that this works also for k processes only.

The case of $\Leftarrow$ is different from the safety case, because the unconditional fairness condition means that the additional processes should be able to run and not be blocked. It is clear that the witness behavior consists of an initial part and a repeating part (loop). We consider two cases: No resource is ever released in the loop or at least one resource is always eventually released.

If no resource is ever released in the loop of the witness behaviour, then at most m processes can use some resource in the loop. Because there are $m + k$ processes, at least k processes never own any resource in the loop. The behaviour of these processes can be mimicked by additional processes.

If at least one resource is always eventually released in the loop of the witness, then there is a state s in the loop where at least 1 resource is unused. This means that at most $m - 1$ processes may use some resources in s. Therefore at least $k + 1$ processes do not use any resource in s. Hence, there is an invisible process p not using anything in s. The behaviour of p can be mimicked by the additional processes (let us suppose that there are $a = l - (m + k)$) in a new witness the following way: In the new witness, each of the additional processes goes to s at the beginning. Then, the loop of the original witness is repeated $a + 1$ times, first with p doing his actions, when the a additional process do one after the other all the actions of p. □

Theorem 7.1 *For each (RT)R_{FIFO} family $\mathcal{F}$ with $m = |R|$ resources and each liveness property under unconditional fairness expressed as Φ^k_{uf}, the validity of $[\exists|\forall] S \in \mathcal{F} : S \models \Phi^k_{uf}$ can be deduced from the validity of $S_n \models E_{uf}\ \varphi(1, ..., n)$ for the appropriate values of $n \in \{1, ..., k + m\}$.*

Proof. Using the basic cut-off property of Lemma 7.1 the proof is the same as for Theorem 5.1 concerning safety properties with the following difference: When considering a formula with k' variables required to refer to different processes (which has been obtained within the procedure of converting Φ^k_{uf} to formulae with fully specified relations among the visible processes), we have to consider *all* systems between k' and $m + k'$. The reason is that in $S_{k'}$, we can already choose the appropriate number of processes to be monitored, but Lemma 7.1 allows us to generalize the obtained results starting from $S_{m+k'}$ only. □

For purely existential formulae, we obtain a better result. It suffices to consider only systems with up to k processes instead of $k+m$ ones (even for RTR$_{FIFO}$ families).

Theorem 7.2 *For each RTR$_{FIFO}$ family $\mathcal{F}$ the validity of $\exists S \in \mathcal{F} : S \models \exists_{p_1,...,p_k|\iota} E_{uf}\varphi(p_1,...,p_k)$ can be deduced from the validity of $S_n \models E_{uf}\ \varphi(1,...,n)$ for (possibly several) $n \in \{1,...,k\}$.*

Proof. To prove this, we only need the $\Rightarrow$ direction of Lemma 5.1 which is already true for $l \geq k$ processes (like in the safety case). $\square$

7.3 Liveness under Weak or Strong Fairness

In this subsection, we give some results concerning liveness properties under weak and strong fairness. First, we show a basic cut-off and than we generalize. Notice, that results are different from the unconditional fairness case.

Lemma 7.2 *For an (RT)R$_{FIFO}$ family $\mathcal{F}$ with $m = |R|$ and an LTL\X formula $\varphi(p_1,...,p_k)$ with atoms based on the control locations of $p_1,...,p_k$, the following holds for systems of $\mathcal{F}$:*

$$\forall l \geq m+k : S_l \models \exists_{p_1,...,p_k|\neq} E_{wf/sf}\ \varphi(p_1,...,p_k) \Leftrightarrow S_{m+k} \models E_{wf/sf}\ \varphi(1,...,k)$$

Proof. ($\Rightarrow$): As in the safety case we remove some non visible processes from a witness behaviour π for $\varphi(p_1,...,p_k)$ and obtain a new witness behaviour with at most $m+k$ processes. We can not just remove *all* non visible processes from π because it can happen that a process in this behaviour is eventually always enabled but not taking any action: In the original witness he requests a resource which is owned by a non visible process and is therefore blocked (this behaviour is weakly/strongly fair) but by removing this non visible process, the process will be enabled. Therefore, in this case, we keep the invisible process which blocks forever the resource in the witness. Because there are m resources keeping additional invisible process are needed at most m times. ($\Leftarrow$): For adding additional process to a witness we consider three cases: 1. All processes are running. Then we proceed as for unconditional fairness (Lemma 7.1). 2. No process is running. Then we can add invisible process which will be blocked. 3. Otherwise: Let b be the number of blocked processes. Then if at least 1 process p out of the b processes does not use any resource (it is just asking for some), p can be easily mimicked. If all of the b processes use some resources then $b \leq m$. At least b resources cannot be used by the looping processes. This means, that at most $m - b = m'$ resources can be used by these processes. There are $m + k - b = m' + k$ looping processes. With m' and $m' + k$, we can use similar arguments as for unconditional fairness. $\square$

Theorem 7.3 *For each (RT)R$_{FIFO}$ family $\mathcal{F}$ with $m = |R|$ resources and each liveness property under weak/strong fairness expressed as $\Phi^k_{wf/sf}$, the validity of $[\exists|\forall] S \in \mathcal{F} : S \models \Phi^k_{wf/sf}$ can be deduced from the validity of $S_n \models E_{wf/sf}\ \varphi(1,...,n)$ for the appropriate values of $n \in \{1,...,k+m\}$.*

Proof. Using the basic cut-off property of Lemma 7.2 the proof is the same as for Theorem 7.1. □

Theorem 7.3 is not true for *(RT)R$_{PRIO}$* . Even for 1-index queries, there is not a cut-off based just on m and k here. For example, we need $l+2$ invisible process to show starvation in the (RT)R$_{PRIO}$ family whose control automaton is shown in Fig. 4.

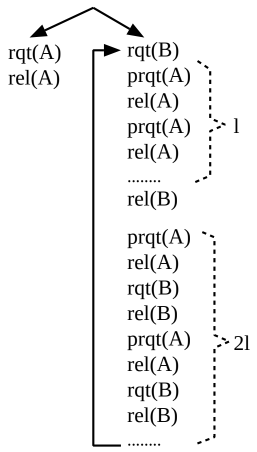

Figure 4: An RTR$_{PRIO}$ family problematic even wrt. 1-index liveness queries

The visible process goes to the left branch and request A and only if there are $l+2$ invisible process running in the right branch, the visible process can be always disabled. The underlying reason that there is no simple cut-off is that a lot of invisible process which request resources with a high priority can be needed to block a visible process.

Within (RT)R$_{PRIO}$, we can show a simple cut-off for 1-index liveness properties under weak/strong fairness if the control automaton of S contains at most one *free area* through which processes may loop. A free area is the set of states of a strongly connected sub-component of the control automaton of S such that in all these states the process is not owning any resource (see Fig. 5).

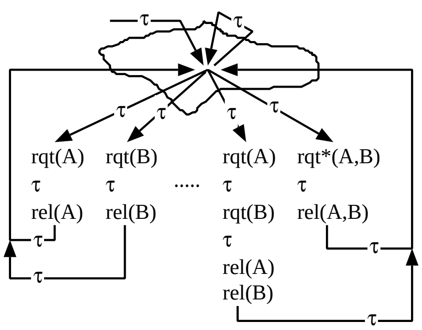

Figure 5: A control automaton with one free area

Theorem 7.4 *For each (RT)R$_{PRIO}$ family $\mathcal{F}$ with one free area and with $m = |R|$ resources and for each liveness property under weak/strong fairness expressed as $\Phi^1_{wf/sf}$, the validity of $\exists S \in \mathcal{F} : S \models \Phi^1_{wf/sf}$ can be deduced from the validity of $S_n \models E_{wf/sf}\ \varphi(1, ..., n)$ for the appropriate values of $n \in \{1, ..., 2m + 2\}$*

Proof. Ensuring that the *blocked visible process will remain blocked* when removing some processes from the witness: We have $2m+2$ processes—1 visible blocked, up to m invisible blocked, at least $m+1$ running forever. When a process releases l resources, at most $m-l$ processes can be using some resources. We have $(m+1)-(m-l)-1=l$ processes ready in the free area to start blocking the released resources. □

Furthermore, we have also shown that the parametric verification problem of $\Phi^1_{wf/sf}$ over general (RT)R$_{PRIO}$ families is decidable due to a cut-off result depending on the number of resources and the number of control locations and transitions of the control automaton of the family at hand. For an exact formulation of this result (and for a more detailed description of the other results mentioned in this section—including their full proofs), we refer an interested reader to the extended version of this report.

8 Some Undecidability Results

Let us now present several undecidability results related to verifying systems of clients competing for the use of shared resources under the supervision of a certain locker. They form some kind of bounds on what can be verified in an exact way in this domain.

8.1 General Reachability in RTR$_{FIFO/PRIO}$

We first show that in general, to exactly represent the reachability set of any entire RTR$_{FIFO}$ family, we need a Turing-powerful formalism. We do this by simulating push-down automata (PDAs) with two stacks.

To be able to formalize the above claim, we need a few auxiliary notions. For any given family $\mathcal{F}$ and any system $S_n \in \mathcal{F}$, we define the *reachability set* of S_n, $reach(S_n)$, as the set of all the configurations $c \in C_n$ for which there is a finite behaviour of S_n starting in $c_0 \in C_n$ and leading to c. Furthermore, we define a function $\lambda_n : C_n \times R \to Q^*$ which for a given configuration, projects the content of the queue associated with a certain resource to the string of the current control states of the processes present in the queue: $\forall c \in C_n\ \forall r \in R\ \forall s_1(p_1)...s_l(p_l) \in \Sigma_n^* : c(r) = s_1(p_1)...s_l(p_l) \Rightarrow \lambda_n(c,r) = c(p_1)...c(p_l)$. We will use λ_n to decode words that are recorded in queues and whose particular symbols are represented by the control states of the enqueued processes.

Theorem 8.1 *For each PDA M with two stacks and an input alphabet Σ_M, there exist a regular language CS of checksums and an RTR$_{FIFO}$ family $\mathcal{F}$ utilizing among all some resources* `output` *and* `checksum` *such that* $\forall w \in \Sigma_M^* : w \in L(M) \Leftrightarrow (\exists S_n \in \mathcal{F}\ \exists c \in reach(S_n)\ \exists cs \in CS : \lambda_n(c, \texttt{output}) = w \wedge \lambda_n(c, \texttt{checksum}) = cs)$.

Proof. We only sketch here the basic principles of how to simulate PDAs using RTR$_{FIFO}$ in a way that can easily be generalized to using any number of stacks. (For more details, an interested reader is referred to the extended version of this report.)

A given PDA M can be simulated in a non-deterministic way. States, input symbols as well as stack symbols of M can be represented by state, input-symbol, and stack-symbol processes, respectively. All these processes (together with an initialization process) must have an identical control, but their roles may be distinguished w.r.t. the control branches they are running in. For example, a process that non-deterministically chooses to simulate a stack symbol waits in a special input stack, then it goes to the PDA stack, which simulates the push operation, then it leaves the PDA stack, which corresponds to the pop operation, and, subsequently, it terminates. The most complex role is played by

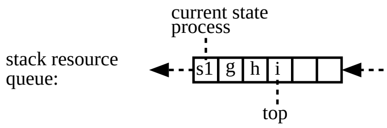

Figure 6: The stack in a simulation of a PDA

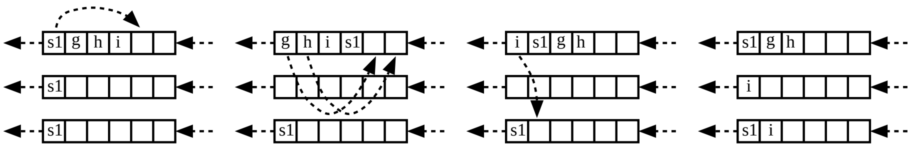

Figure 7: Popping the top of the stack

state processes that also simulate transitions from one state to another. To distinguish different states, stack symbols, and input symbols, we introduce multiple copies of the control branches implementing the appropriate general behaviour. A multiple use of the same state, input or stack symbol in a run corresponds to using multiple processes of the appropriate type.

We interpret the content of the queues by projecting their items to the control states of the appropriate processes. The simulation is controlled by state processes; if the current-state process p deadlocks, the whole system deadlocks. The stack is simulated by a resource that is normally owned by p and stack-symbol processes wait in its queue—the top of the stack corresponds to the tail of the queue (see Fig. 6)

A step of a PDA starts from a source state, processes some input symbol, checks and removes the top of the stack, pushes certain new symbols to the stack, and finishes in a certain target state. The two key operations we need to simulate such a step are the following: (1) To be able to pop exactly the top of the stack. (Principles of the implementation of this operation can also be used when introducing a certain number of new symbols from the input stack, etc.) (2) To be able to check that processes that are unblocked to do what is characteristic for the role they have chosen to play (entering the stack, leaving the stack, coming from the intput, etc.) are of the type the state process controlling the simulation expects.

Reading of the top stack symbol (i.e. tail queue symbol here) can be implemented by p releasing the stack, taking it again, and ensuring that after some symbol process releases the stack and does not take it back (i.e. it leaves as the read symbol), all further symbol processes will block before releasing the stack. In this way, the contents of the queue representing the stack is rotated, and only the tail symbol can leave and at the same time unblock the controlling state-symbol process. We use two additional resources to implement this trick—see Fig. 7.

To test whether a process plays the role p expects it to play, p may let it take and release a resource characteristic for its control branch and release only a certain resource before such a check (cf. Fig. 8).

In the above described way, a state process may check and remove the top of the stack, introduce the right new symbols from a pool of fresh processes, and pass over the control to a new state process representing the appropriate target state. The construction ensures that at most the required number of processes leaves/enters a certain queue. To ensure that it is exactly the required number (and also that thy system was correctly initialized),

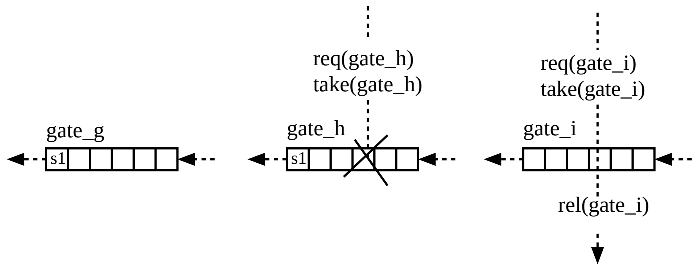

Figure 8: Checking whether a process represents the right symbol

the processes record their steps by sending requests to a special checksum queue. It is then easy to detect that some process failed to do something, which manifests itself by breaking a certain regular pattern of the contents of the checksum queue. □

Let us note that the above construction is in the fragment of RTR$_{FIFO}$ where the contents of queues is guaranteed to be bounded—there are no loops with more request than take operations on any resources here. The construction trivially covers RTR$_{PRIO}$ (high-priority actions need not be used), but it is outside (RT)R$_{FIFO}$. As a corollary, we can state the following theorem.

Theorem 8.2 *General* reachability queries *referring arbitrarily both to the current control locations of processes and to the content of queues are* undecidable *in RTR$_{FIFO}$ and RTR$_{PRIO}$ even when the contents of the involved queues of requests is guaranteed to be bounded.*

Proof. With respect to the proof construction of Theorem 8.1, it is undecidable to check whether there exists a system in the given family within which we can reach a state characterized by a boolean combination of atomic formulae of the kind $\exists p : .p = q$ and (non-temporal) formulae over the contents of queues powerful enough to test whether a queue contains two consecutive requests from the same process. The test on queues stems from dealing with the checksum queue in the proof of Theorem 8.1. □

The question of decidability of general reachability queries for (RT)R$_{FIFO}$ and (RT)R$_{PRIO}$ remains open for the future research.

8.2 Undecidability of Temporal Logics in RTR$_{FIFO/PRIO}$

The proof construction of Theorem 8.1 can easily be modified to prove undecidability of certain temporal logics enriched with *local process quantification* in the context of RTR$_{FIFO}$ and RTR$_{PRIO}$. Namely, undecidability of the parametric verification problem for CTL (and even its EF fragment) with local process quantification and for LTL\X with local process quantification can be proved. In the latter case, the undecidability result can even be shown in the context of (RT)R$_{FIFO}$/(RT)R$_{PRIO}$. The problem of verifying CTL (or EF) with local process quantification over (RT)R$_{FIFO}$/(RT)R$_{PRIO}$ remains open.

Theorem 8.3 *For RTR$_{FIFO}$ and RTR$_{PRIO}$, the problem of checking whether there exists $S \in \mathcal{F}$ which fulfills a given formula of the EF logic enriched with local process quantification is undecidable.*

Proof. We exploit the proof construction of Theorem 8.1. We leave out dealing with concrete input symbols and we formulate the problem of checking non-emptiness of the language of PDAs with two stacks as checking certain EF formulae based on non-temporal subformulae of the kind $\exists p : .p = q$. □

Theorem 8.4 *For (RT)R$_{FIFO}$/(RT)R$_{PRIO}$, the problem of checking whether there exists $S \in \mathcal{F}$ which fulfills a given formula of LTL\X enriched with local process quantification is undecidable.*

Proof. We proceed in a similar way as in the case of Theorem 8.3. We replace the use of separate `req` operations allowing us to book some resources in advance by suitable subformulae of the formulae to be checked. □

Let us note that when we use the local quantification operator $\exists_{p1 \neq p2}$, we can formulate undecidable queries in LTL\X even without using the until operator.

9 Summary and Future Work

Summary. In this work, we have concerned the problem of verifying parameterized systems of client processes competing for access to shared resources under the supervision of some locker module. We have proposed a partition of this very complex problem to two subproblems, which are interesting and significant by itself, and we have concentrated on one of them. Namely, we have examined the problem of verifying systems with arbitrary many clients provided that they are controlled by a locker implementing a certain locker strategy. In particular, we have covered two important resource management strategies: FIFO and FIFO with priorities. We have dealt with verification of various generic property classes expressible in (fragments of) temporal logics and including certain forms of safety, deadlockability, as well as liveness.

We have obtained three kinds of results here. (1) We have proved some parametric verification problems arising in the given context to be efficiently reducible to finite-state verification. This was done by arguing that it is often sufficient to consider at most a certain (cut-off) number of client processes to prove a given property for any number of them. Such cut-offs were shown to depend on the number of visible processes in the formula of interest and/or the number of resources in the family being verified. (2) For some problems in the domain of families with the locker based on FIFO with priorities, we have proved that using a simple cut-off as in the previous case is in general insufficient. However, such problems may still be decidable and there may exist cut-offs taking additionally into account the structure of the control automaton of the given family and/or the formula to be verified. In particular, we have derived such a cut-off for 1-index liveness properties of (RT)R$_{PRIO}$. (3) Finally, we have come up with some undecidability results related to the given area.

Future Work. In the context of RTR$_{FIFO}$ and RTR$_{PRIO}$ families, which our current results concentrate on, there are still some configurations of the considered verification problems that are not known to be decidable nor undecidable. Moreover, some of the here presented cut-offs are not known to be optimal, and so it is still possible that they can be further minimized. For the future, these two issues seem to be especially interesting in relation to some of the configurations of checking liveness and deadlockability over families using the PRIO locker (as mentioned already in the text). Next, in the case of decidable problems for which finding small cut-offs will turn out to be hard or impossible, it is then possible to try to apply some other verification methods such as some kind of symbolic verification. (Besides, symbolic verification could be used for approximate verification even for undecidable problems.)

Nevertheless, the two main topics which remain open for the future research are the following.

First, RTR families with non-exclusive access should be examined in the same way as families without this operation. So far, we know that some of the here presented results (undecidability, most of safety and deadlockability, a part of liveness) can relatively easily be generalized for the new setting. However, the rest seems to be quite intricate. Additionally, it could also be interesting to examine some more locker strategies relevant for the practice. In fact, we have already started some studies on families with a FIFO-based locker allowing for overtaking of blocked processes and on families with a (lazy) service-in-random-order locker. While the former appears to be very close to the PRIO locker, in the case of the latter, some results related to the FIFO locker (safety, liveness under unconditional fairness) seem to be applicable, but otherwise some special treatment will probably be necessary.

Second, a big challenge is the problem of checking whether a given locker implements the strategy we suppose it to implement regardless of the number and behaviour of clients. Here, it is necessary to propose a suitable model and in the case the problem is undecidable in general (which is quite probable), try to define meaningful subclasses of the model for which the situation is better. We can again consider using cut-offs in this domain, but it is also possible to try out some other methods, e.g., the already mentioned (precise or approximate) symbolic model checking.

References

[1] P. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parametrized System Verification. In *Proc. 11th Intern. Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 134–145. Springer-Verlag, 1999.

[2] P.A. Abdulla, L. Boasson, and A. Bouajjani. Effective Lossy Queue Languages. In *Proc. 28th Intern. Coll. on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *LNCS*. Springer-Verlag, 2001.

[3] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 555–568. Springer-Verlag, 2002.

[4] T. Arts and C.B. Earle. Development of a Verified Erlang Program for Resource Locking. In *Proceedings of FMICS'01*, 2001.

[5] T. Arts, C.B. Earle, and J. Derrick. Verifying Erlang Code: A Resource Locker Case-Study. In *Proc. Int. Symposium on Formal Methods Europe 2002*, volume 2391 of *LNCS*, pages 183–202. Springer-Verlag, 2002.

[6] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Prameterized Networks. In *Proceedings of TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, 2000.

[7] K. Baukus, Y. Lakhnech, S. Bensalem, and K. Stahl. Networks of Processes with Parameterized State Space. In *Proc. ICALP 2001 Satellite Workshop VEPAS 2001*, volume 50 of *ENTCS*, pages 388–402. Elsevier, 2001.

[8] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. In *Proceedings of FTRTFT 2000*, 2000.

[9] S. Blau and J. Rooth. AXD 301—A New Generation ATM Switching System. *Ericsson Review*, No. 1, 1998.

[10] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The Power of QDDs. In *Proc. 4th Static Analysis Symposium*, volume 1302 of *LNCS*, pages 172–186. Springer-Verlag, 1997.

[11] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parameterized Concurrent Systems with Resource Sharing. A research report of the ADVANCE project (IST FET project No. IST-1999-29082). LIAFA, University Paris 7, France, 2002.

[12] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. 14th Intern. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 539–554. Springer-Verlag, 2002.

[13] E.A. Emerson and V. Kahlon. Model Checking Large-Scale and Parameterized Resource Allocation Systems. In *Proceedings of TACAS 2002*, volume 2280 of *LNCS*, pages 251–265. Springer-Verlag, 2002.

[14] E.A. Emerson and K.S. Namjoshi. Reasoning about Rings. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 85–94, 1995.

[15] E.A. Emerson and K.S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proc. 8th Int. Conf. on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 87–98. Springer-Verlag, 1996.

[16] E.A. Emerson and A.P. Sistla. Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997.

[17] S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39(3):675–735, 1992.

[18] N. Halbwachs, F. Lagnier, and C. Ratel. An Experience in Proving Regular Networks of Processes by Modular Model Checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[19] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 424–435. Springer-Verlag, 1997.

[20] R.P. Kurshan and K.L. McMillan. A Structural Induction Theorem for Processes. *Information and Computation*, 117(1), 1995.

[21] A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 82–97. Springer-Verlag, 2001.

[22] P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *LNCS*, pages 88–97, Vancouver, 1998. Springer-Verlag.

[23] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407 of *LNCS*, pages 68–80, Grenoble, 1989. Springer-Verlag.