On the expressive power of communication primitives in parameterised systems*

Benjamin Aminof[1], Sasha Rubin[2], and Florian Zuleger[1]

[1] Technische Universität Wien, Austria
[2] Università degli Studi di Napoli “Federico II”, Italy

Abstract. We study foundational problems regarding the expressive power of parameterised systems. These (infinite-state) systems are composed of arbitrarily many finite-state processes that synchronise using a given communication primitive, i.e., broadcast, asynchronous rendezvous, broadcast with message loss, pairwise rendezvous, or disjunctive guards. With each communication primitive we associate the class of parameterised systems that use it. We study the relative expressive power of these classes (can systems in one class be simulated by systems in another?) and provide a complete picture with only a single question left open. Motivated by the question of separating these classes, we also study the absolute expressive power (e.g., is the set of traces of every parameterised system of a given class ω-regular?). Our work gives insight into the verification and synthesis of parameterised systems, including new decidability and undecidability results for model checking parameterised systems using broadcast with message loss and asynchronous rendezvous.

1 Introduction

Parameterised systems are composed of arbitrarily many copies of the same finite-state process. The processes in a system run independently, but are given a mechanism by which they can synchronise, e.g., in broadcast systems one process can send a message to all the other processes, while in a rendezvous system the message is received by a single process [12,17]. Examples of such systems abound in theoretical computer science (e.g., distributed algorithms [18]) and biology (e.g., cellular processes [15]).

Problem Statement. Different synchronisation mechanisms, or *communication primitives* as we call them here, yield systems with different capabilities. For instance, broadcast is at least as expressive as rendezvous since in two steps broadcast may simulate a rendezvous (I broadcast “I want to rendezvous”, and someone broadcasts the reply “I will rendezvous with you”, illustrated in Figure 5). On the other hand, intuitively, broadcast is more expressive than rendezvous (since to simulate a broadcast a process would have to rendezvous with

* Benjamin Aminof and Florian Zuleger are supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant ICT12-059. Sasha Rubin is a Marie Curie fellow of the Istituto Nazionale di Alta Matematica.

all other processes before anyone made a different move). The motivation of this paper is to formalise such reasoning and make such intuitions precise.

Communication Primitives. This paper focuses on representative primitives from the literature on formal methods for parameterised systems: Broadcast (BC), like CBP message passing can model ethernet-like broadcast, GSM's cell-broadcast, or the notifyAll method in Concurrent Java [5,12,20]; Asynchronous Rendezvous (AR) can model the notify method in Concurrent Java [5]; Broadcast with Message Loss (BCML) can model mobile ad hoc networks (MANETS) and systems that use selective broadcast with nodes that can be activated or deactivated at any time [6,7,8]; and Pairwise Rendezvous (PR), like CSP message passing, can model population protocols [4,17]. For comparison we also consider a primitive that admits cutoffs, i.e., disjunctive guards (DG) [10], a property not shared by the previous primitives.[3]

Executions of Parameterized Systems. We systematically compare communication primitives using the standard notion of executions from the point of view of single processes. Indeed, many papers (e.g. [5,7,6,11,12,13,9,17,2,3]) consider specifications from the point of view of single processes — important examples of such specifications are safety specifications like coverability and liveness specifications like repeated coverability and termination. Given a process P, and a communication primitive CP, let P^n_{CP} be the finite-state system composed of n copies of P that synchronise using CP (note that there is no special "controller" process). An *execution* is a (finite or infinite) sequence of labels[4] of states of a single process in P^n_{CP}. In many applications (e.g., in parameterised verification), one needs to consider systems of all sizes. Thus, we let P^∞_{CP} denote the infinite-state system consisting of the (disjoint) union of the systems P^n_{CP} for each $n \in \mathbb{N}$.

Relative Expressive Power. We define the natural comparison $\mathrm{CP} \leq_{\mathrm{IE}} \mathrm{CP}'$ as follows: for every process P that uses CP there is a process Q that uses CP$'$, such that P^∞_{CP} and $Q^\infty_{\mathrm{CP}'}$ have the same set of infinite executions. Similarly, we write $\leq_{\mathrm{FE}}$ if considering only finite executions. The informal meaning of these comparisons $\leq$ is that CP$'$ can simulate CP, with respect to linear-time specifications. All of our simulations (except of AR by PR) have the added properties that they also hold for systems of a fixed finite size, and that they are efficiently computable. This latter fact is useful for example for model checking (MC) classes of parameterised systems with respect to linear-time specifications (over a single process), i.e., if $\mathrm{CP} \leq \mathrm{CP}'$ and the translation from P to Q is efficient, then MC CP-systems is (immediately) reduced to MC CP$'$-systems. We remark that most decidability results for MC parameterised systems are for linear-time specifications, whereas for branching-time specifications it is typically undecidable [17,2,3,11].

Absolute Expressive Power. Motivated by the problem of comparing communication primitives, we also study their absolute expressive power. That is, a communication primitive CP determines a class of languages $\mathcal{L}_{\mathrm{CP}}$, i.e., the

[3] A cutoff is a maximal number of processes that needs to be model checked in order to guarantee correct behaviour of any number of processes. Our results show that, indeed, having a cutoff lowers the expressive power.

[4] Typically, each label is a set of atomic propositions.

sets of executions of such systems. How does the class $\mathcal{L}_{\text{CP}}$ relate to canonical classes of languages, such as regular, context-free, ω-regular, etc.? Answers to such questions allow one to deduce that certain communication primitives *cannot be simulated* by certain others, as well as directing one's choice of communication primitive for modeling and synthesis of distributed systems.

Our contributions.[5]
Relative Expressive Power. We provide a full picture of the relative expressive power of these communication primitives, see Figures 1 and 2 — an arrow from CP to CP$'$ means CP$' \leq$ CP,[6] and a mark across an arrow means that CP $<$ CP$'$.

Section 3 establishes all but three of the arrows in Figures 1 and 2: we get PR $\leq_{\text{IE}}$ BCML from Theorems 6 and 4, PR $\leq_{\text{FE}}$ DG from Theorems 2 and Proposition 5, and AR $\leq_{\text{FE}}$ PR from Proposition 5 and Theorem 3.

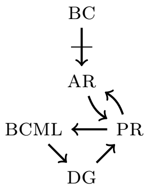


Fig. 1: $\leq_{\text{FE}}$

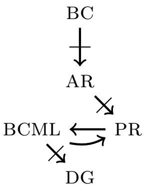


Fig. 2: $\leq_{\text{IE}}$

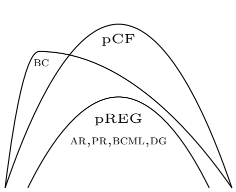


Fig. 3: Finite Executions of P^{∞}_{CP}

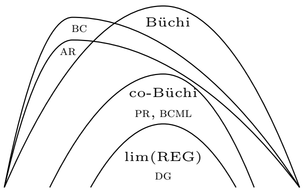


Fig. 4: Infinite Executions of P^{∞}_{CP}

Absolute Expressive Power. The classes of languages of finite executions generated by the different primitives are illustrated in Figure 3. BC can generate languages that are not context free, whereas DG, PR, BCML, AR generate exactly the prefix-closed regular languages (pREG). However, no communication primitive can generate all prefix-closed context-free languages (pCF). The case of infinite executions is illustrated in Figure 4. We show that DG can generate exactly limits of regular languages, BCML, PR can generate exactly co-Büchi languages, AR, BC can generate non ω-regular languages, whereas no communication primitive can generate all ω-regular languages. We present our results on absolute expressive power in Section 5. The strictness of the arrows in Figures 1 and 2 follow from our results on absolute expressive power. To get DG $<_{\text{IE}}$ PR (and thus also deduce DG $<_{\text{IE}}$ BCML) use Theorem 4 and Theorem 7 (and the fact that there are co-Büchi languages that are not the limit of any regular language, e.g., all words over $\{a, b\}$ with finitely many as). To get PR $<_{\text{IE}}$ AR use Proposition 9 and Theorem 6. To get AR $<_{\text{FE}}$ BC use Proposition 6 and Theorem 3. To get AR $<_{\text{IE}}$ BC use Proposition 8 and Theorem 5.

Model Checking Linear-Time Specifications. Our techniques yield new results about model checking (MC) AR and BCML parameterised systems for liveness

[5] For lack of space, some proofs are missing or only sketched and can be found in the full version of this paper.

[6] We note that the transitivity of the relations $\leq$ gives rise to additional simulations that, for clarity, are not drawn in the figures.

properties.[7] In particular, even the simplest liveness property (i.e., does there exist an infinite run) is undecidable for AR systems (Section 4). Also, liveness properties are decidable in PTIME for systems using BCML (a problem that was not even known to be decidable); this follows because BCML can be efficiently simulated by PR (Proposition 2), and the fact that MC of PR-systems can be done in PTIME (which itself follows from [17, Section 4]).

2 Definitions and Preliminaries

Tuples f over a set X may be written $(x_1, \cdots, x_k)$ or in functional notation $f \in X^{[k]}$, i.e., $f(i) = x_i$. Given a set Σ, we denote by Σ^*, Σ^+, Σ^ω the sets of all finite strings, all non-empty finite string, and all infinite strings, respectively, over Σ. Let u_i denote the ith letter of u. We write $\mathrm{pre}(L)$ for the set of *finite prefixes* of some language $L \subseteq \Sigma^*$ or $L \subseteq \Sigma^\omega$. A language $L \subseteq \Sigma^*$ is *prefix closed* if $L = \mathrm{pre}(L)$. The *limit* of a language $L \subseteq \Sigma^*$ is the language $\lim L \subseteq \Sigma^\omega$ such that $\alpha \in \lim L$ if and only if infinitely many prefixes of α are in L. A *labeled transition system (LTS)* is a tuple $\langle \Omega, A, Q, Q_0, \delta, \lambda \rangle$ where Ω is a set of *letters* (also called *observables*),[8] A is a set of *edge labels*, Q is a finite set of *states*, $Q_0 \subseteq Q$ are the *initial states*, $\delta \subseteq Q \times A \times Q$ is the *transition relation*, and $\lambda : Q \to \Omega$ is the *labeling function*. For $\tau = (q, a, q')$ we write $\mathsf{src}(\tau) = q$, $\mathsf{des}(\tau) = q'$ and $\mathsf{edglab}(\tau) = a$, and we also write $q \xrightarrow{a} q'$. A *path* of an LTS is a (finite or infinite) string of transitions $\pi := \pi_1\pi_2\ldots$ of δ such that $\mathsf{src}(\pi_{i+1}) = \mathsf{des}(\pi_i)$ for every i. We write $\mathsf{src}(\pi) := \mathsf{src}(\pi_1)$, and if π is finite we write $\mathsf{des}(\pi) := \mathsf{des}(\pi_{|\pi|})$. A *run* is a path π where $\mathsf{src}(\pi) \in Q_0$. We write $\mathsf{edglab}(\pi)$ for the sequence $\mathsf{edglab}(\pi_1)\mathsf{edglab}(\pi_2)\ldots$. Typically the edge-labels will carry information, i.e., an action (e.g., send message $\mathtt{m}$), and whether or not the edge is visible. See [22] for basic notions about automata. In particular, we use the following acronyms: NFW, NBW and NCW where N stands for "nondeterministic", F for "finite", B for "Büchi", C for "co-Büchi", and W for "word automata". Counter machines CM are standard variations of Minsky Machines, i.e., they have a fixed number of counters that can be incremented, decremented if not zero, and tested for zero. In the rest of this paper, the word "simulation" is used as in ordinary natural language, and not as part of the technical term "(bi)simulation relation".

A note about simulations and visibility. In order to reason about simulations we have to be able to hide some of the inner steps involved. Consider the following motivating example. All the x86 family of processes support the same basic instruction set, but they implement each instruction using their own sequences of microcode instructions. This is fine since to the running software these sequences of microcode are invisible and it can only see their effect on the observables, i.e., the values of the registers. In order to capture this basic trait of simulations, our definition of local process labels each transition with a Boolean flag indicating whether it is visible or not, with the added condition that invisible transitions

[7] As in [12,9] we formalise safety properties as regular sets (of finite words) and liveness properties as ω-regular sets (of infinite words).

[8] In applications one typically takes $\Omega := 2^{\mathsf{AP}}$ where AP is a set of atomic predicates.

do not change the observables[9]. To demonstrate, in the introduction we illustrated that BC (effectively) simulates PR by replacing every PR transition by two (successive) BC transitions. Thus, in order to preserve the set of executions, we have to hide one of these two transitions (see Figure 5).

System Model. For a set $\varSigma$, let $\varSigma_{\mathsf{sync}} = \{\mathtt{m}!, \mathtt{m}? \mid \mathtt{m} \in \varSigma\}$ be the *synchronisation-actions*. Let $\varPi$ be a set of *internal-actions*, disjoint from $\varSigma$. A *local process* is a finite LTS $P = \langle \varOmega, A, S, S_0, \delta, \lambda \rangle$ where $A := (\varSigma_{\mathsf{sync}} \cup \varPi) \times \mathbf{B}$ and for every $(q, (\sigma, b), q') \in \delta$, if $b = \mathbf{false}$ then we must have that $\lambda(q) = \lambda(q')$. A transition $\tau = (q, (\sigma, b), q')$ is called *visible* if $b = \mathbf{true}$ and *invisible* if $b = \mathbf{false}$. Thus, an invisible transition may change the state but not what is observed.

Define functions $\mathsf{act}, \mathsf{vis}$ such that $\mathsf{act}(\tau) = \sigma$ and $\mathsf{vis}(\tau) = b$. A state $s \in S$ is *able to receive* (resp. *able to send*) message $\mathtt{m} \in \varSigma$ if there is a transition $\tau \in \delta$ with $\mathsf{src}(\tau) = s$ and $\mathsf{act}(\tau) = \mathtt{m}?$ (resp. $\mathsf{act}(\tau) = \mathtt{m}!$). States and transitions of P are called *local states* and *local transitions*. Informally, local transitions with $\sigma \in \varPi$ are transitions that a single process must take alone, and are called *local internal transitions*, whereas local transitions with $\sigma \in \varSigma_{\mathsf{sync}}$ may involve synchronising with other processes, and are called *local synchronising transitions*.

For a local process P we now define the *global system*, i.e., the composition P^n_{CP} of n-many copies of P that communicate using CP. A *global state* of P^n_{CP} is an n-tuple of elements of S, collectively S^n. For $f = (s_1, \cdots, s_n), f' = (s'_1, \cdots, s'_n) \in S^n$ a *global transition* $\tau = (f, \nu, f') \in S^n \times (\varSigma \cup \varPi) \times S^n$ satisfies:

1. If $\nu \in \varPi$ then there exists i and $b \in \mathbf{B}$ such that $s_i \xrightarrow{\nu, b} s'_i$, and $s_\ell = s'_\ell$ for $\ell \neq i$ *(internal transition)*.
2. If $\nu = \mathtt{m} \in \varSigma$:
   - If CP = BC: there exist i and $b_i \in \mathbf{B}$ such that $s_i \xrightarrow{\mathtt{m}!, b_i} s'_i$ and letting R be the set of processes $j \neq i$ that are able to receive $\mathtt{m}$, we must have that R is non-empty and $s_j \xrightarrow{\mathtt{m}?, b_j} s'_j$ for all $j \in R$ (and some $b_j \in \mathbf{B}$), and $s_\ell = s'_\ell$ for $\ell \notin R \cup \{i\}$ *(broadcast transition)*.[10]
   - If CP = AR: there exist i and $b_i \in \mathbf{B}$ such that $s_i \xrightarrow{\mathtt{m}!, b_i} s'_i$ and either: there exists $j \neq i$ such that $s_j \xrightarrow{\mathtt{m}?, b_j} s'_j$ and $s_\ell = s'_\ell$ for $\ell \neq i, j$; or there is no $j \neq i$ such that j is able to receive $\mathtt{m}$, and $s_\ell = s'_\ell$ for $\ell \neq i$ *(asynchronous rendezvous transition)*.
   - If CP = PR: there exist $i \neq j$ and $b_i, b_j \in \mathbf{B}$ such that $s_i \xrightarrow{\mathtt{m}!, b_i} s'_i$ and $s_j \xrightarrow{\mathtt{m}?, b_j} s'_j$, and $s_\ell = s'_\ell$ for $\ell \neq i, j$ *(rendezvous transition)*.
   - If CP = BCML: there exist i and $b_i \in \mathbf{B}$ such that $s_i \xrightarrow{\mathtt{m}!, b_i} s'_i$ and there is some, possibly empty, set R of processes (not containing i) such that

[9] It is common to allow specifications (e.g., the LTL formula $\mathsf{G}\,p$) to be satisfied by computations that loop forever in the same state. Thus, we *don't* consider every transition in which the observables don't change to be invisible. In particular, we can have both visible and invisible self loops. Using the CPU analogy, the former corresponds to a NOP in the instruction set, and the latter to a NOP in microcode.

[10] A slightly different version of BC, in which R is also allowed to be empty, also appears in the literature [12]. Our results also hold for this version.

$s_j \xrightarrow{\mathtt{m}?,b_j} s'_j$ for all $j \in R$ (and some $b_j \in \mathbf{B}$), and $s_\ell = s'_\ell$ for $\ell \notin R \cup \{i\}$ *(broadcast with message loss transition).*

- If CP = DG: there exist $j \neq i$ and $b \in \mathbf{B}$ such that $s_i \xrightarrow{s_j?,b} s'_i$ and $s_j \xrightarrow{s_j!,\mathbf{false}} s_j$, and $s_\ell = s'_\ell$ for $\ell \neq i, j$ *(guarded transition).* [11]

A process k is said to be *involved* in a global transition τ if it takes a local transition γ from s_k to s'_k (e.g., in all cases above process i is involved in τ). Moreover, it is *visibly involved* if $\mathsf{vis}(\gamma) = \mathbf{true}$.

Finally, P^n_{CP} is the LTS $\langle \Omega^n, \Sigma \cup \Pi, S^n, S^n_0, \Delta, \Lambda \rangle$ where Δ consists of the global transitions (just defined), and $\Lambda(f)(i) := \lambda(f(i))$ for every $i \in [n]$. The infinite state LTS P^∞_{CP} is the disjoint union of P^n_{CP} for $n \in \mathbb{N}$, and it is called a *parameterised system*, or just a *system*.

Executions. Let π be a path of P^n_{CP}. We will relate π to paths in P corresponding to a single process. Fix a process index $k \in [n]$. Let $i_1 < i_2 < \ldots$ be the set of indices such that process k is visibly involved in the global transition π_{i_j}, and define $s_j := \Lambda(\mathsf{src}(\pi_{i_j}))(k) \in \Omega$ (for all j). If there are only finitely many indices $i_1 < i_2 < \cdots < i_l$, we let $\mathit{vislet}_k(\pi)$ be the concatenation of $s_1 s_2 \ldots s_l$ with the additional letter $\Lambda(\mathsf{des}(\pi_{i_l}))(k)$ at the end. Otherwise, we set $\mathit{vislet}_k(\pi) := s_1 s_2 \ldots$. We define the *set of* 1*-executions of* P^n_{CP} by

$$\textsc{Exec}(P^n_{\text{CP}}) := \{\mathit{vislet}_k(\pi) : k \in [n], \pi \text{ is a run of } P^n_{\text{CP}}\} \subseteq \Omega^\omega \cup \Omega^*$$

and the *set of* 1*-executions of* P^∞_{CP} as $\textsc{Exec}(P^\infty_{\text{CP}}) := \cup_{n \in \mathbb{N}^+} \textsc{Exec}(P^n_{\text{CP}})$. We denote the infinite (resp. finite) elements of $\textsc{Exec}(\cdot)$ by $\textsc{InfExec}(\cdot)$ (resp. $\textsc{FinExec}(\cdot)$). It is worth noting that if a run π is infinite, but $\mathit{vislet}_k(\pi)$ is finite, then process k was only doing finitely many meaningful moves in π (which is akin, in a system with only visible transitions, to it being scheduled only finitely many times) which is why we do not include such traces in InfExec.

3 Relative Expressive Power

For communication primitives CP, CP′, write CP $\leq_{\text{IE}}$ CP′ if for every local process P there is a local process Q (computable from P) such that $\textsc{InfExec}(P^\infty_{\text{CP}}) = \textsc{InfExec}(Q^\infty_{\text{CP}'})$. Similarly, define $\leq_{\text{FE}}$ with FinExec replacing InfExec. For $x \in \{\text{IE}, \text{FE}\}$, if CP $\leq_x$ CP′ $\leq_x$ CP then write CP $\equiv_x$ CP′. If CP $\leq_x$ CP′ and CP $\not\equiv_x$ CP′ then write CP $<_x$ CP′ (and define $<_x$ similarly). Informally, if CP $\leq_x$ CP′ we say that CP′ *simulates* CP. Note that, in the definition of $\leq_x$, if there is a PTime algorithm that given P produces the corresponding Q then we say that CP′ *efficiently simulates* CP. All the simulation results CP $\leq_x$ CP′ in this paper (except for AR $\leq_{\text{FE}}$ PR) are efficient simulations.

[11] If CP = DG then we also assume $\Sigma = S$ (i.e., the synchronization alphabet is the set of local states), and for every local state $s \in S$ there is a transition $s \xrightarrow{t!,a} r$ if and only if $s = r = t$ and $a = \mathbf{false}$ (i.e., the only transition τ with $\mathsf{act}(\tau) = s!$ is an invisible self-loop on state s).

Relationship with verification. Every regular language of finite words is called a *safety* property, and every ω-regular language of infinite words is called a *liveness* property, cf. [12]. The model checking problem for parameterised systems using CP for a given safety (resp. liveness) property L over Ω is the following: given P, decide whether or not $\text{FinExec}(P_{\text{CP}}^{\infty}) \subseteq L$ (resp. $\text{InfExec}(P_{\text{CP}}^{\infty}) \subseteq L$). This model checking problem is sometimes called the "parameterised model checking problem" or "parameterised verification", e.g., [11]. If CP$'$ effectively simulates CP then the parameterised verification problem for systems using CP is reducible to the parameterised verification problem for systems using CP$'$.

3.1 Simulations

The simulations DG $\leq_x$ PR $\leq_x$ BC, with $x \in \{\text{FE}, \text{IE}\}$, have already been discovered in the literature [9]; we illustrate PR $\leq_x$ BC in Figure 5. These results are the starting point for our fine-grained analysis. In this section we establish the simulations DG $\leq_x$ BCML $\leq_x$ PR $\leq_x$ AR $\leq_x$ BC for $x \in \{\text{IE}, \text{FE}\}$. All these simulations were not previously known. In all the proofs we efficiently construct, given a local process P, a local process Q such that $\text{Exec}(P_{\text{CP}}^{n}) = \text{Exec}(Q_{\text{CP}'}^{n})$.

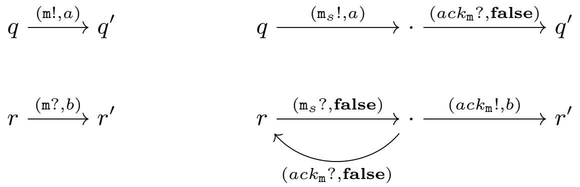
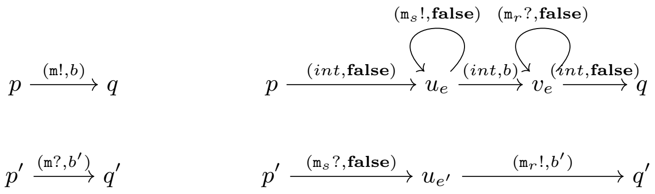


Fig. 5: Simulation of PR (left) by BC (right)

Proposition 1. AR $\leq_x$ BC*, for* $x \in \{\text{FE}, \text{IE}\}$*.*

Proof. Recall that the difference between PR and AR is only that in AR a process can send a message $\mathtt{m}$ even if there is no other process to receive it (but if there is, then one such process must receive $\mathtt{m}$). We divide the global transitions of an AR system into three types: internal transitions, synchronous transitions involving two processes, and those involving only one process. Given local process P, we build local process Q such that Q_{BC}^{n} simulates P_{PR}^{n} (for every $n > 2$), by using a sequence (called a *transaction*) of 1 or 2 global transitions. Simulating the internal transitions is done directly, the synchronous transitions involving two processes are simulated by a 2-step transaction as in the simulation of PR by BC, and the synchronous transitions in which there is only a sender are simulated by a single-step transaction as follows: let $e = (p, (\mathtt{m}!, b), q)$ be a local transition in P; in Q, a process can take the local transition $(p, (\mathtt{m}_{solo}!, b), q)$ broadcasting the message that it is simulating a send of $\mathtt{m}$ that should have no receivers, and every process that is in a state that is able to receive $\mathtt{m}$ in P, receives $\mathtt{m}_{solo}$ and invisibly moves to a new special "disabled" copy of its current state from which it can no longer do anything; all other processes simply receive $\mathtt{m}_{solo}$ and invisibly self-loop. The intuition is that by sending $\mathtt{m}_{solo}$ process i guessed that there is no process able to receive $\mathtt{m}$ in the simulated system, and thus we disable the processes that witness the fact that the guess is wrong — effectively making it right. Note that if we do not disable them then one of these processes will be in the wrong state (since in the AR system one of them must receive $\mathtt{m}$ and move, but in the simulating system none moved) and will be able to later allow moves in the simulating system that are not possible in the simulated one. □



$p \xrightarrow{(\mathtt{m}!,b)} q$
$p' \xrightarrow{(\mathtt{m}?,b')} q'$
$(\mathtt{m}_s!,\mathbf{false})$
$(\mathtt{m}_r?,\mathbf{false})$
$p \xrightarrow{(int,\mathbf{false})} u_e \xrightarrow{(int,b)} v_e \xrightarrow{(int,\mathbf{false})} q$
$p' \xrightarrow{(\mathtt{m}_s?,\mathbf{false})} u_{e'} \xrightarrow{(\mathtt{m}_r!,b')} q'$

Fig. 6: Simulation of BCML (left) by PR (right)

Proposition 2. BCML $\leq_x$ PR*, for* $x \in \{\mathrm{FE}, \mathrm{IE}\}$*.*

Proof. Given a local process P, we build a local process Q such that Q^n_{AR} simulates P^n_{PR} (see Figure 6). A global transition where i sends $\mathtt{m}$ by taking a local transition of the form $e := (p, (\mathtt{m}!, b), q)$, and a set R of processes receive, is simulated by a multi-step transaction. The transaction needs multiple steps because in PR only two processes move in every step. The main difficulty, and the reason the transaction is complicated, is that we must be careful not to introduce new executions that are not possible in the BCML system.

The simulation of sending the lossy broadcast message $\mathtt{m}$ is done in three stages: (a) process i internally and invisibly moves from state p to the new intermediate state u_e; state u_e has an invisible self-loop that sends message $\mathtt{m}_s$ (indicating it is trying to simulate sending $\mathtt{m}$); the self-loop enables the message to be sent to an arbitrary number of processes; (b) process i internally moves from state u_e to the new intermediate state v_e with visibility b; state v_e has an invisible self-loop that receives message $\mathtt{m}_r$; the self-loop allows to acknowledge that an arbitrary number of processes have received the message; (c) process i internally and invisibly moves from state v_e to state q.

The simulation of receiving message $\mathtt{m}$, by taking a local transition of the form $e' := (p', (\mathtt{m}?, b'), q')$, is in two stages: (a) process j invisibly moves from state p' to the new intermediate state $u_{e'}$ receiving message $\mathtt{m}_s$. (b) process j moves from state $u_{e'}$ to state q' with visibility b' sending message $\mathtt{m}_r$.

Unfortunately, we can not guarantee that this transaction is atomic, i.e., that no other global transitions intertwine with the simulation of a single lossy broadcast. We can not even guarantee that if a process j has received a message $\mathtt{m}_s$ from process k, then it is going to send $\mathtt{m}_r$ to process k, and not to another process. The solution is to consider the processes that performed the second stage with some process k as the ones which received the lossy broadcast message $\mathtt{m}$ from k. This works since, for each process j, the first stage of receiving a lossy broadcast message is invisible, and after that it can not do anything but participate in a second stage of receiving a message. $\square$

Proposition 3. $\textsc{pr} \leq_x \textsc{ar}$*, for* $x \in \{\mathrm{FE}, \mathrm{IE}\}$*.*

Proposition 4. $\textsc{dg} \leq_x \textsc{bcml}$*, for* $x \in \{\mathrm{FE}, \mathrm{IE}\}$*.*

Remark: There is a version of broadcast, lets call it ABC, where a process can broadcast a message even when no other process is able to receive it. All our results about BC hold also for ABC since $\textsc{bc} \equiv_x \textsc{abc}$, for $x \in \{\mathrm{FE}, \mathrm{IE}\}$.

4 Model Checking Asynchronous-Rendezvous Systems

The theorem below states that model checking even the most basic liveness properties of AR systems is undecidable. The proof of the theorem is an adaptation of the one used in ([12]) to prove a similar result for BC. Unfortunately, there is a serious complication: ([12]) makes central use of the fact that BC systems can elect a controller, but AR systems are not powerful enough to do that.

Fortunately, we can make do with a temporary controller, which AR can elect: from the initial state a process can send the message “I am now the controller” and enter the initial state of the “controller” component of the process template. If later on another process sends this message then it becomes the new controller, and the current controller, who receives this message, enters a special state D, from which it can do nothing. Thus, there are never two controllers at the same time, and at most n controller switches in a system with n processes.

The ability of AR to elect a temporary controller allows us not only to prove the theorem below, it also allows us to later show (see Figure 4) that AR systems have an expressive power that is in between PR (which cannot elect even a temporary controller) and BC (which can elect a permanent controller). However, interestingly enough, this is only true for infinite traces. For finite traces, having a temporary controller, in contrast to a permanent one, provides no extra expressive power (see Figure 3).

Theorem 1. *(i) Model checking liveness properties of parameterised systems communicating via* AR *is undecidable. (ii) In particular, the following problem is undecidable: given local process* P*, decide if* $\textsc{InfExec}(P^\infty_{\textsc{ar}})$ *is empty or not.*

Proof. For the first item, it is enough to reduce the halting problem for input-free deterministic counter machines CM (which is undecidable [19]) to the existence of a run in an AR system $P^\infty_{\textsc{ar}}$ that visits a halting state infinitely often. It is convenient to assume that when (and if) the halting location is reached then the CM resets itself, i.e., it decrements all its counters until they become zero and then loops back to the initial state. The basic encoding for the simulation is from [12]. It uses one process called the *controller* to orchestrate the simulation and store the line of the CM, and many *memory* processes. Each memory process stores one bit for each counter, and the value of a counter is the number of processes having a non-zero bit for it. Each process has a special dead state D, which once entered cannot be exited.

A process may, from the initial state, nondeterministically become either the temporary controller or a memory process. The transitions in a memory process are, for each counter $c \in C$: if the stored c-bit is 0, then it can send the message "inc(c)" and set the c-bit to 1; if the stored c-bit is 1, then it can send the message "notzero(c)" and leave c unchanged, or send the message "dec(c)" and set c to 0, or *receive* the message "iszero(c)" and go to state D. From every state of the controller there is a complementary send/receive transition as specified by the CM line that this state represents. Thus, for example, an "increment c" is simulated by the controller receiving an "inc(c)" and moving to the next line of the CM (or to the state D, if the current command to simulate is not "increment c"), and an "if $c = 0$ goto l_1 else goto l_2" command is simulated by the controller either receiving "notzero(c)" and moving to state l_2; or moving to state l_1 and sending an "iszero(c)" which, if counter c is zero, is not received, and otherwise is received by a memory process with a 1 c-bit which then enters state D.

It is not hard to show that P^n_{AR} can faithfully simulate the CM as long as the counters stay below $n - 1$. Thus, if the CM reaches the halting location h then there is an infinite run of P^n_{AR}, for a large enough n, in which the process playing the controller is in h infinitely often. For the reverse direction, the key point is that whenever the simulation makes an error (such as replacing the temporary controller in mid simulation, or having the controller guess a counter is zero when it is not, or when a memory process simulates a command that is not what the controller wants to simulate) one process dies (i.e., enters state D). Thus, since there are only finitely many processes participating in any execution of P^∞_{AR}, in every infinite run of P^∞_{AR}, from some point on, no more processes die, and thus from that point on the simulation is correct. It follows that if there is a run of P^∞_{AR} in which a process is in state h infinitely often then the run of the CM reaches the halting location. This completes the sketch of the proof of the first item. The second item uses a standard trick (see e.g. [12]) of adding an extra counter that increases in every step, and gets reset only when the halting state is reached. Thus, the system will run out of its finite number of memory processes and hang unless the CM reaches h. □

5 Absolute Expressive Power

Finite Executions. First note that, since every prefix of a finite run is a run, for every CP and P we have that $\textsc{FinExec}(P^\infty_{\text{CP}})$ is prefix-closed.[12]

Proposition 5. *For every* CP *and prefix-closed regular language L there exists P such that* $\textsc{FinExec}(P^\infty_{\text{CP}}) = L$.

[12] Although distributed systems are routinely studied this way, one may also introduce final states to the local process and restrict to runs that end in final states.[16]

Proof. Transform an automaton for L into P by pushing letters into states (i.e., by changing L so that it remembers the last read input-letter in its state, labeling each state by this letter, and adjusting the initial states), and making all local transitions of P internal (i.e., not synchronisation transitions) and visible. $\square$

Proposition 6 (cf. [16]). *(i) There is a* P *s.t.* $\textsc{FinExec}(P^{\infty}_{\text{BC}}) = \textit{pre}(\{a^n b^n \mid n \geq 1\})$. *Moreover,* BC *can generate non-context free languages; (ii) None of our communication primitives can generate all prefix-closed context-free languages.*

Theorem 2 ([17]). *For every* P*, the language* $\textsc{FinExec}(P^{\infty}_{\text{PR}})$ *is regular.*

Theorem 3. *For every* P*, the language* $\textsc{FinExec}(P^{\infty}_{\text{AR}})$ *is regular.*

Proof. Let $P = \langle \Omega, A, S, S_0, \delta, \lambda \rangle$ be some local process. We will construct a finite automaton (NFW) $\mathcal{A}$ that accepts exactly the traces in $\textsc{FinExec}(P^{\infty}_{\text{AR}})$. We call a local state $s \in S$ *unbounded* if for every $k \in \mathbb{N}$ there is an $n \in \mathbb{N}$, and a reachable global state f in P^{n}_{AR}, such that $|f^{-1}(s)| \geq k$. We denote by $U \subseteq S$ the set of *unbounded states* of P and by $B = S \setminus U$ the set of *bounded states*. Observe that $S_0 \cap B = \emptyset$, and that there is a $K \in \mathbb{N}$ such that $|f^{-1}(s)| \leq K$ for every $s \in B$, and every global state f in P^{∞}_{AR}.

We now define an automaton $\mathcal{A}$. States of $\mathcal{A}$ are pairs $\langle s, f \rangle$, where $s \in S$ is the state of the process whose execution we are observing, and $f \in S \to \{0, 1, \ldots, K\} \cup \{\infty\}$, is such that $f(u) = \infty$ for every $u \in U$. Intuitively, for each state in B, f keeps track of the number the other processes in that state. A state $\langle s, f \rangle$ of $\mathcal{A}$ is initial iff: $s \in S_0$ and $f(u) = 0$ for all $u \in B$. $\mathcal{A}$ has a transition from $\langle s, f \rangle$ to $\langle s', f' \rangle$ if there is a local transition $\tau \in \delta$ where the counter values of f change to f' according to τ (and any possible matching transition if τ is a synchronising transition), and if s is involved in τ then s changes to s'. Such a transition is labeled by $\lambda(s)$ if s was involved in the transition and $\mathsf{vis}(\tau) = \mathbf{true}$, and otherwise by ϵ. For example, if $\tau = (p, (\mathtt{m}?, \mathbf{true}), p')$ then, together with the any transition of the form $(q, (\mathtt{m}!, b), q')$ in δ, it induces the following transitions in $\mathcal{A}$: *(i)* a transition $\langle p, f \rangle \xrightarrow{\lambda(p)} \langle p', f' \rangle$ for every f, f' such that $f(q) \neq 0$, and f' is obtained from f by decrementing the value assigned to q and incrementing the value assigned to q'; *(ii)* a transition $\langle s, f \rangle \xrightarrow{\epsilon} \langle s, f' \rangle$ for every s and every f, f' such that $f(p) \neq 0, f(q) \neq 0$, and f' is obtained from f by decrementing the values assigned to p, q and incrementing the values assigned to p', q' (as usual, $\infty - 1 = \infty = \infty + 1$).

Clearly, $\mathcal{A}$ is a finite automaton. We show that $\mathcal{A}$ (with all states accepting) accepts exactly the traces in $\textsc{FinExec}(P^{\infty}_{\text{AR}})$. For every $n \in \mathbb{N}$, every execution obtained from some path of $\textsc{FinExec}(P^{n}_{\text{AR}})$ is accepted by a run of $\mathcal{A}$ that "simulates" this execution by correctly updating the components s, f of its states.

It remains to prove that every word accepted by $\mathcal{A}$ is in $\textsc{FinExec}(P^{\infty}_{\text{AR}})$. We claim (*): for every k there exists $n_k \in \mathbb{N}$ and a path π_k of $\textsc{FinExec}(P^{n_k}_{\text{AR}})$ reaching a global state such that there are at least k processes in every local state $s \in U$. To see that (*) yields $L(\mathcal{A}) \subseteq \textsc{FinExec}(P^{\infty}_{\text{AR}})$, let π be some run of $\mathcal{A}$, and take $k \geq 2|\pi|$. Observe that in such a run at most k processes are involved. We build a corresponding run in $P^{n_k}_{\text{AR}}$. First, (†): using (*) we take a path that results in at least k processes in every local state $s \in U$. Recall that $S_0 \cap B = \emptyset$ and thus, in particular, there is at least one process in each of the initial states. Then,(‡): the process we want to observe starts from the relevant initial state and we imitate the run π of $\mathcal{A}$ step by step. This is indeed possible since whenever a step of ‡ requires a process with a state in U then such a process is available, and the same for processes in B. The former is guaranteed by ‡, and the latter since (by induction on the step number) the number of processes with states in B is at least as specified by the function f of the mimicked point in π.

We now prove (*). Let $u_1, ..., u_m$ be the states in U. Inducting on $0 \leq i \leq m$, we construct paths π^i in systems $\textsc{FinExec}(P_{\textsc{ar}}^{n_i})$ such that load at least k processes in states $u_1, ..., u_i$. We start with the empty run π^0 in $\textsc{FinExec}(P_{\textsc{ar}}^{1})$. Clearly, π^0 satisfies the inductive claim. Given π^i, we construct π^{i+1} as follows: Let l_i be the length of π^i. By the definition of U, there is a path π in some system $\textsc{FinExec}(P_{\textsc{ar}}^{n})$ that ends with at least $l_i + k$ processes in state u_{i+1} and at least one process in each of the initial states (thus, executing π in a larger system does not force any of the additional processes out of the initial states). We set $n_{i+1} = n + n_i$ and define π^{i+1} to be the concatenation of π and π^i in the system $\textsc{FinExec}(P_{\textsc{ar}}^{n_{i+1}})$. Clearly, π^{i+1} loads at least k processes in states $u_1, ..., u_{i+1}$ (since π^i can remove at most l_i states from u_{i+1}). □

It is open if there is a constructive proof of Theorem 3.

Infinite Executions.

Theorem 4. *For every co-Büchi language L, and for* $\textsc{cp} \in \{\textsc{pr}, \textsc{bcml}, \textsc{ar}, \textsc{bc}\}$, *there is a local process P s.t.* $\textsc{InfExec}(P_{\textsc{cp}}^{\infty}) = L$.

Proof. Given an NCW $\mathcal{A}$ recognizing L we build a local process P, in which all transitions are visible, such that $\textsc{InfExec}(P_{\textsc{cp}}^{\infty}) = L$. The local process P has exactly the same structure, when viewed as a graph, as $\mathcal{A}$, with an added special sink state. In order to take a transition to a co-Büchi state (i.e., a state that an accepting run of $\mathcal{A}$ can only visit finitely many times) the process has to receive a message. A process that sends a message enters the sink state, and can not send again. Thus, in a system with n processes a process can visit a co-Büchi state up to $n-1$ times. Transitions to other states are internal transitions of the process, and can always be taken. □

We now show that not all ω-regular languages can be generated by our parameterised systems. In fact, the proof is general enough to apply to any reasonable notion of communication primitive (not only those defined in this paper), unless some additional fairness conditions are imposed. The proof employs a standard pumping argument to derive a contradiction by showing that if $ab^1ab^2a\ldots$ is in $\textsc{Exec}(P_{\textsc{cp}}^{n})$ then so is $ab^1ab^2\ldots ab^cab^\omega$, where c is the number of states in $P_{\textsc{cp}}^{n}$.

Proposition 7. *For every local process P, primitive* $\textsc{cp}$, *and $n \in \mathbb{N} \cup \{\infty\}$, the set* $\textsc{InfExec}(P_{\textsc{cp}}^{n})$ *is not equal to the ω-regular language $L \subseteq \{a, b\}^\omega$ consisting of all infinite sequences that contain infinitely many occurrences of a.*

The following is not hard to see:

Proposition 8. *There is a* BC*-system that can generate the non co-Büchi language* $\{a^l b^m c^\omega \mid l \geq m \geq 1\}$*)*.

We use a variation of the proof of Theorem 1 to show that AR-systems can generate languages that are not ω-regular:

Proposition 9. *There is an* AR*-system that can generate a language* $L \subset \{a,b\}^\omega$ *that has the property that*
1. every string $\alpha \in L$ *has a suffix* $(a^n b^n)^\omega$ *for some integer* $n \in \mathbb{N}$*, and*
2. every string $(a^n b^n)^\omega$ *is the suffix of some string in* L*.*
In particular, the language L *is not co-Büchi.*

Proof. Standard fooling arguments show that any L with the properties described is not Büchi (and thus not co-Büchi). We now describe an AR-system that can generate a language L with the properties stated in the lemma. The idea follows that in the proof of Theorem 1: a controller starts in mode a; in mode a it repeatedly increments a counter c; at some point it checks if all memory processes are 1 by issuing an "allone(c)" message (which can be implemented symmetric to the "iszero(c)" message), and moves to mode b; in mode b it repeatedly decrements the counter c; at some point it checks if all memory processes are 0 by issuing a "iszero(c)" message, and moves back to mode a to repeat the computation. Build the local process P based on M and note that a process that becomes a controller forever in P^l_{AR} does not err from some point on, and thus traces a path whose suffix is $(a^n b^n)^\omega$ with $n \leq l$. Note that an abdicating controller does not trace an infinite path (since the dead state is a dead-end). □

The following is proved in almost the same way as Theorem 3:

Theorem 5. *For every* P*, the language pre*$(\text{INFEXEC}(P^\infty_{\text{AR}}))$ *is regular.*

Model checking safety and liveness properties (given as automata) of parameterised systems communicating via PR is decidable in PTIME [17]. Actually:

Theorem 6 (implicit in [17]). *For every local process* P*, one can compute, in* PTIME*, a non-deterministic co-Büchi automaton for the set* $\text{INFEXEC}(P^\infty_{\text{PR}})$*.*

Theorem 7. *Every* $\text{INFEXEC}(P^\infty_{\text{DG}})$ *is the limit of a regular language.*

Proof. By [10] there exists $N \in \mathbb{N}$ such that $\text{INFEXEC}(P^\infty_{\text{DG}}) = \text{INFEXEC}(P^N_{\text{DG}})$ (the idea is to pick N large enough such that every reachable state can be reached and adding one extra process; this choice of N ensures that every reachable self-loop $(s, (s!, \textbf{false}), s)$ can always be fired; the extra process can therefore move unrestrained). The language $L := \text{FINEXEC}(P^N_{\text{DG}})$ is regular (because it is the projection of the finite-state machine P^N_{DG}). It is sufficient to prove that $\text{INFEXEC}(P^N_{\text{DG}}) = \lim L$. Clearly $\text{INFEXEC}(P^N_{\text{DG}}) \subseteq \lim L$. To see the converse let $\alpha \in \lim L$. So there exists $k \in [N]$ and an infinite set $I \subseteq \mathbb{N}$ such that for every $i \in I$ there exists a run ρ_i of P^N_{DG} such that the prefix of α of length i is equal to $\mathit{vislet}_k(\rho_i)$. The set $\text{pre}\{\rho_i : i \in I\}$ is an infinite tree (under the prefix-ordering) that is finitely-branching (this is where we use the fact that the ρ_is are in P^N_{DG} and not P^∞_{DG}), and thus by Kőnig's Lemma, it has an infinite branch ρ. Clearly ρ is an infinite run of P^N_{DG} and $\mathit{vislet}_k(\rho) = \alpha$. Thus $\alpha \in \text{INFEXEC}(P^N_{\text{DG}})$. □

6 Related Work and Conclusion

Related Work. The absolute and relative expressive power of Petri nets and their extensions were studied for finite and infinite executions, e.g., [14,16,1]. They show a strict hierarchy of relative expressive power: Petri nets (PN) are less expressive than Petri nets with non-blocking arcs (PN+NBA), which are less expressive than Petri nets with transfer arcs (PN+T). Translating these results into the language of parameterised systems, one finds that these extensions roughly correspond to a very powerful model of parameterised systems with a controller and in which processes can be created and destroyed at any time. By this translation, PNs correspond to communication by PR, PN+NBA to communication by AR, and PN+T to communication by BC. In contrast, we focus on the setting with no controller and with no process creation or destruction. Thus, neither their simulation nor separation results are directly applicable to our more restricted setting.

The paper [9] organises communication primitives by whether or not model checking (MC) is decidable. Although they do have a notion of simulation, that notion is based on reducing the MC problem of systems using one primitive to systems using another primitive. In particular, their reduction transforms, while ours preserves, the set of behaviours. For instance, despite their result that MC safety properties of DG- and PR-systems are inter-reducible, we prove that there is a set of traces of a PR system that can't be generated by any DG system.

It was previously known that MC safety properties for systems using each of the primitives in this paper is decidable, liveness for BC is undecidable, and liveness for PR and DG is decidable [12,17,10,6,9]. We complete the picture, and prove, in particular, that for AR systems liveness is undecidable. The result in [9] on the undecidability of liveness for AR systems makes the additional assumption that there exists a unique "leader" process. The presence of a leader usually dramatically increases the expressive power, cf. [11,17], and makes it easier to establish undecidability than in our fully symmetric case. A number of papers focus on supplying the exact complexity of MC various parameterised systems, e.g., [11,2,17,6,7,8,21].

Conclusion. Comparing the expressive power of various models of computation is a central theme in theoretical computer science. In our case, such comparisons can be used to transfer results from one model to another. For instance, we prove that AR can be effectively simulated by BC, and thus the fact that safety is decidable for BC (cf. [12]) implies that safety is decidable for AR ([9]). We also deduced the new result, using [17] and the fact that BCML can be efficiently simulated by PR, that liveness for BCML is decidable in PTime.

The results about absolute expressive power are useful not only to show, e.g., that PR can not simulate AR, but also to point to the inherent limitations of each communication primitive. Such results can be used in synthesis to show that certain specifications are not realisable. As a concrete example, a minor variation of our proof that no system can generate the language "infinitely many a's" (Proposition 7) yields that there is no parameterised system (and thus no point in trying to synthesise one without adding external fairness conditions) that satisfies the conjunction of the properties “every run has infinitely many grants” and “some run has arbitrarily large gaps between successive grants”.

References

1. Abdulla, P.A., Delzanno, G., Begin, L.V.: A classification of the expressive power of well-structured transition systems. Inf. Comput. 209(3), 248–279 (2011)
2. Aminof, B., Kotek, T., Rubin, S., Spegni, F., Veith, H.: Parameterized model checking of rendezvous systems. In: CONCUR, pp. 109–124. Springer (2014)
3. Aminof, B., Rubin, S., Zuleger, F., Spegni, F.: Liveness of parameterized timed networks. In: ICALP. pp. 375–387 (2015)
4. Aspnes, J., Ruppert, E.: An introduction to population protocols. In: Middleware for Network Eccentric and Mobile Applications, pp. 97–120. Springer-Verlag (2009)
5. Delzanno, G., Raskin, J.F., Begin, L.V.: Towards the automated verification of multithreaded java programs. In: TACAS. pp. 173–187 (2002)
6. Delzanno, G., Sangnier, A., Traverso, R., Zavattaro, G.: The cost of parameterized reachability in mobile ad hoc networks. CoRR abs/1202.5850 (2012)
7. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: CONCUR. LNCS, vol. 6269, pp. 313–327 (2010)
8. Delzanno, G., Sangnier, A., Zavattaro, G.: Verification of ad hoc networks with node and communication failures. In: FTDS. pp. 235–250 (2012)
9. Emerson, E., Kahlon, V.: Model checking guarded protocols. In: LICS. pp. 361–370. IEEE (2003)
10. Emerson, E., Kahlon, V.: Reducing model checking of the many to the few. In: CADE. pp. 236–254 (2000)
11. Esparza, J.: Keeping a crowd safe: On the complexity of parameterized verification. In: STACS (2014)
12. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. Symp. on Logic in Computer Science p. 352 (1999)
13. Esparza, J., Ganty, P., Majumdar, R.: Parameterized verification of asynchronous shared-memory systems. In: CAV. pp. 124–140 (2013)
14. Finkel, A., Geeraerts, G., Raskin, J., Begin, L.V.: On the *omega*-language expressive power of extended petri nets. Theor. Comput. Sci. 356(3), 374–386 (2006)
15. Fisher, J., Henzinger, T.A.: Executable cell biology. Nature biotechnology 25(11), 1239–1249 (2007)
16. Geeraerts, G., Raskin, J., Begin, L.V.: Well-structured languages. Acta Inf. 44(3-4), 249–288 (2007)
17. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
18. Lynch, N.: Distributed Algorithms. Morgan Kaufman Publishers, Inc., San Francisco, USA (1996)
19. Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
20. Prasad, K.V.S.: A calculus of broadcasting systems. Sci. Comput. Program. 25(2-3), 285–327 (1995)
21. Schmitz, S., Schnoebelen, Ph.: The power of well-structured systems. In: CONCUR. pp. 5–24 (2013)
22. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher Order Workshop. pp. 238–266 (1995)