Article

Modelling Distributed Systems in Distributed Autonomous and Asynchronous Automata (DA^3)

Wiktor B. Daszczuk [1,*]

[1] Institute of Computer Science, Warsaw University of Technology, Nowowiejska Str. 15/19, 00-665 Warsaw, Poland; wbd@ii.pw.edu.pl

* Correspondence: wbd@ii.pw.edu.pl; Tel.: +48-22-234-7812

Abstract: Integrated Model of Distributed Systems is used for modeling and verification. In formalism, the distributed system is modeled as a collection of server states and agent messages. The evolution of the system takes the form of actions that transform the global system configuration (states and messages) into a new configuration. Formalism is used in the Dedan verification environment for finding different kinds of deadlocks: communication deadlocks in the server view and resource deadlocks in the agent view. For other purposes, a conversion has been developed to equivalent models: to Petri nets for structural analysis and do Distributed Autonomous and Asynchronous Automata (DA^3) for easy graphical modeling in terms of system components. In addition, it is possible to simulate a verified system on distributed components in DA^3. The automata have two forms: Server-DA^3 (S-DA^3) for the server view and Agent-DA^3 (A-DA^3) for the agent view. DA^3 formalism is compared to other concepts of distributed automata known from the literature.

Keywords: distributed systems; distributed system modeling; distributed automata; graphical modeling; formal methods

1. Introduction

IMDS (Integrated Model of Distributed Systems [1]) is a formalism used to identify and verify distributed systems, in particular to detect deadlocks and check distributed termination. The system is built on servers - cooperating distributed nodes - and the cooperation takes the form of agents who perform distributed computing through migration between servers. The main element of IMDS is an action that has the server's state and the agent's message on the input and a similar pair on the output. The modelled distributed system can be decomposed into server processes consisting of sequences of actions threaded by server states. Alternatively, the same specification can be spread across agent processes that travel between servers. Agent processes are sequences of actions threaded by means of messages. Server processes communicate using messages while agent processes communicate using server states. Therefore, two decompositions correspond to communication duality. The IMDS formalism has been used, along with the model checking technique [2], to develop the Dedan program that finds various types of deadlocks in the verified system: communication deadlock (in the server view), resource deadlock (in the agent view), partial deadlock (in which a subset of system processes participate) and total deadlock (when all processes are involved). Similarly, the termination can be partial or total. Although Dedan's main goal is deadlock and termination verification, the user may be interested in other properties of the distributed system, for example:

- automatic conversion between the server view and the agent view,
- observation of global transition graph,
- structural properties of the system: structural conflicts, dead code, whether the system is purely cyclic or not, etc.,

- temporal properties other than deadlock: if the system is safe from some erroneous situation, if given situations are inevitable, etc.,
- graphical definition of concurrent components of the system (servers or agents),
- graphical simulation in terms of concurrent components rather than in terms of a global graph.

In order to support these possibilities, additional facilities have been added to Dedan:

- export of the model to the Petri net (ANDL format [3]), for static structural analysis [4] under Charlie Petri net tool [5] ,
- export to external model checkers for temporal analysis: Promela format (LTL model checking in Spin [6]), SMV format (LTL and CTL model checking in NuSMV [7]), Timed Automata XML format (Timed CTL model checking in Uppaal [8]),
- alternative formulation of a system as Distributed Automata (precisely – Distributed Autonomous and Asynchronous Automata – DA^3), to facilitate system specification and simulation in terms of parallel components: automata represent server processes or agent processes.

This article focuses on a graphical automata-like model, equivalent to IMDS definition.

Servers in a distributed environment act asynchronously and make their decisions in autonomous manner. However, many modelling and verification formalisms use simultaneous activities of processes; like synchronous transitions on common symbols in Büchi automata [6] or Timed Automata [9], synchronization on send and receive operations in CSP [10], Occam [11] or Uppaal timed automata [8], synchronous operations on complementary input and output ports in CCS [10]. Several automata-based formalisms (including Büchi automata and Timed Automata) use synchronous coordination. There are several automata-based asynchronous models, they will be mentioned in Section 3. In IMDS, the autonomy of servers and agents is realized using asynchronous operations: sending the message to the server or setting the server's state is the only way of influencing their behavior. The process autonomously decides if and when the communication would be accepted and what activities it would cause.

The contribution of this paper is the introduction of DAAA – Distributed, Asynchronous and Autonomous Automata for modeling distributed systems (shortened to DA^3 – D-triple-A or DA-cubed). There are two versions of DA^3, following the communication duality, Server DA^3 (S-DA^3) and Agent DA^3 (A-DA^3). They are both equivalent to the IMDS specification and thus – to each other. Automata are not the basic formalism of specification and verification, that is the role of IMDS. The automata are an interface to the graphical system specification as a set of cooperating state machines. At the same time, in the Dedan environment it is possible to simulate system operation in terms of automata and to follow the steps of the counterexample obtained from model checking.

The definition of IMDS is given in Section 2. The distributed automata DA^3 are defined in Section 3. Also, the differences between DA^3 and other notions of Distributed Automata in the literature are discussed. The examples of Dedan operation on DA^3 are described in Section 4. The equivalence between IMDS and both versions of DA^3 is shown in Section 5. Conclusions and further work are covered in Section 6. Appendices present a larger example and operation of Dedan program.

2. Integrated Model of Distributed Systems (IMDS)

IMDS is defined in [1]. Here we use simplified version of IMDS, without dynamic process creation, which is suitable for conversion to finite automata, and for static model checking.

A distributed system is simply a set of actions, having pairs of state and message on input and on output. It is based on an observation, that in distributed environment a server is activated by an incoming message. If the message is accepted – which depends on a current state of the server – it performs some action, which is execution of a service, and changes its state. Also typically a new message is issued to continue the distributed computation on some other server. Thus system actions are the relation Λ on sets of *servers' states* P and *messages* M attributed to distributed computations, called *agents*:

$$\Lambda \subset (M{\times}P) \times (M{\times}P) \tag{2.1}$$

The set P is split into disjoint subsets attributed to servers in set S={s_1,s_2,...,s_n}, while the set M is split into disjoint subsets attributed to agents in set A={a_1,a$_2$,...,a_k}. In an action $\lambda\in\Lambda$, λ=((m,p),(m',p')), input and output state belong to the same server $m,m'\in M_i$, while input and output message belong to the same agent $p,p'\in P_j$. A message is sent to a specific server to invoke its service, which is modelled by a function $target : M \rightarrow S$. In every pair (m,p) which is the input of an action λ, the server component must match: $target(m)= s_i$, $p\in P_j$, $i = j$.

Agents may be infinite or they may terminate in special actions of the form λ=((m,p),(p')), where an output message is absent.

The behavior of a distributed system is determined by its Labelled Transition System – LTS [12]. A node in LTS (we do not use a name 'state' to avoid ambiguousness) is a configuration T of IMDS model: set of current states of all servers and current messages of all agents (except for terminated ones). An initial configuration T_0 contains initial states P_0 and initial messages M_0. An input configuration $T_{inp}(\lambda)$ of an action λ =((m,p),(m',p')) contains m and p of its input pair (m,p) and the output configuration $T_{out}(\lambda)$ contains m' and p' of its output pair (m',p').

$$\begin{aligned}
&\text{LTS} = \langle Q,q_0,W\rangle \text{ where:} \\
&\quad Q = \{T_0,T_1,\ldots\}\ (\mathit{nodes}) \\
&\quad q_0 = T_0\ (\mathit{initial\ node}) \\
&\quad W = \{\ (T,\ \lambda,\ T')\ \mid\ \lambda\in\Lambda,\ T{=}T_{inp}(\lambda),\ T'{=}T_{out}(\lambda)\ \}\ (\mathit{transitions})
\end{aligned} \tag{2.2}$$

Actions are executed in interleaving way (one action at a time [13]). Note that every server performs its action autonomously (only the server's state and the messages *pending* on this server are considered). Also, the communication is asynchronous: a server process sends a message to some other server process (or an agent sets the server's state for some other agent) regardless of the current situation of a process with which it communicates (and every other process). As a result, we may call the process *autonomous* and *asynchronous*.

The processes in the system are defined as sequences of actions. If two consecutive actions in a process are connected by a server state – it is a server process communicating with other server processes by means of messages (states are the carriers of server processes). If two consecutive actions in a process are connected by a message of an agent – it is an agent process communicating with other agent processes by means of servers' states (agent messages are the carriers of agent processes).

$$\begin{aligned}
&B_i = \{\ \lambda \in\Lambda\ \mid\ \lambda = ((m,p),(m',p'))\ \vee\ \lambda = ((m,p),(p')),\ p,p' \in P_i\ \} \\
&C_j = \{\ \lambda \in\Lambda\ \mid\ \lambda = ((m,p),(m',p'))\ \vee\ \lambda = ((m,p),(p')),\ m,m' \in M_j\ \}
\end{aligned} \tag{2.3}$$

The decomposition of a system into server processes is called a *server view*, the other one is an *agent view*.

$$\begin{aligned}
&B = \{\ B_i \mid i = 1\ldots n\ \} \\
&C = \{\ C_j \mid j = 1\ldots k\ \}
\end{aligned} \tag{2.4}$$

The examples of distributed systems modeled in IMDS may be found in [14]. In [15] the verification of Karlsruhe Production Cell is covered, where servers implement the devices in the cell, and agents implement metal plates that are processed. In Automatic Vehicle Guidance System [16] – servers implement road segment controllers and agents implement the vehicles.

3. Simple example – buffer

specification and verification of distributed systems the Dedan program was developed. For an action λ = ((m,p),(m',p')) a more convenient notation is used, in which a server state p is denoted as a pair (s,v), where s is a *server* and v is a *value* of a state, $s\in S$, $v\in V$. A message m is denoted as a triple (a,s,r), where a is an *agent* and r is a server's s *service* invoked by the message (a server may offer a number of services, for example *wait* and *signal* on a semaphore), $a\in A$, $s\in S$, $r\in R$. An action λ=((a,s,r),(s,v)),((a,s',r'),(s,v')) has the form `{a.s.r, s.v}->{a.s'.r', s.v'}` in Dedan source code.

To present the two views of a distributed system, a simple example of a buffer with producer and consumer agents (each one originating from its own server) is included in the listings below. First the server view follows. The notation is intuitional: server types are defined (lines 2, 9, 16). Formal parameters specify agents and other servers used. Every server includes states (l.3, 10), services (l.4, 11) and actions (l.6-7, 13-14). Then, server and agent variables are declared (l.17,18). The variables can have the same names as the types, they are distinguished by context. If a variable has the same identifier as its type, a declaration `variable:type` may be suppressed to a single identifier, as in the example. At the end, servers (l.20-22) and agents (l.23,24) are initialized, and variable names are bound with formal parameters of servers.

```
1.   system BUF_server_view;
2.   server: buf (agents Aprod,Acons; servers Sprod,Scons),
3.   services{put, get},
4.   states {no_elem,elem},
5.   actions {
6.      {Aprod.buf.put, buf.no_elem} -> {Aprod.Sprod.ok_put, buf.elem},
7.      {Acons.buf.get, buf.elem} -> {Acons.Scons.ok_get, buf.no_elem},
8.   }
9.   server: Sprod (agents Aprod; servers buf),
10.  services{doSth,ok_put}
11.  states {neutral,prod}
12.  actions {
13.     {Aprod.Sprod.doSth, Sprod.neutral} -> {Aprod.buf.put, Sprod.prod}
14.     {Aprod.Sprod.ok_put, Sprod.prod} -> {Aprod.Sprod.doSth, Sprod.neutral}
15.  }
16.  server: Scons (agents Acons; servers buf), // similar to Sprod
17.  servers buf,Sprod,Scons;
18.  agents Aprod,Acons;
19.  init -> {
20.     Sprod(Aprod,buf).neutral,
21.     Scons(Acons,buf).neutral,
22.     buf(Aprod,Acons,Sprod,Scons).no_elem,
23.     Aprod.Sprod.doSth,
24.     Acons.Scons.doSth,
25.  }.
```

The system converted to the agent view (this is done automatically by the Dedan program) is as follows.

```
1.   system BUF_agent_view;
2.   server: buf, services{put, get} states{no_elem, elem};
3.   server: Sprod, services{doSth, ok_put} states{neutral, prod};
4.   server: Scons, services{doSth, ok_get} states{neutral, cons};
5.   agent: Aprod (servers buf:buf,Sprod:Sprod),
6.   actions {
7.      {Aprod.buf.put, buf.no_elem} -> {Aprod.Sprod.ok_put, buf.elem},
8.      {Aprod.Sprod.doSth, Sprod.neutral} -> {Aprod.buf.put, Sprod.prod},
9.      {Aprod.Sprod.ok_put, Sprod.prod} -> {Aprod.Sprod.doSth, Sprod.neutral},
10.  };
11.  agent: Acons (servers buf:buf,Scons:Scons),// similar to Aprod
12.  agents Aprod:Aprod,Acons:Acons;
13.  servers buf:buf,Sprod:Sprod,Scons:Scons;
14.  init -> {
15.     Aprod(buf,Sprod).Sprod.doSth,
16.     Acons(buf,Scons).Scons.doSth,
17.     buf.no_elem, Sprod.neutral, Scons.neutral,
18.  }
```

The LTS of the example system is presented in Fig. 1. In the nodes, messages of the agents `Sprod` and `Scons` are displayed (without agent identifiers) in the first line and the states of all servers (`buf`, `Sprod`, `Scons`) are displayed in the second line (without server identifiers). In all figures, elements concerning servers are shown in red while elements concerning agents are shown in green.

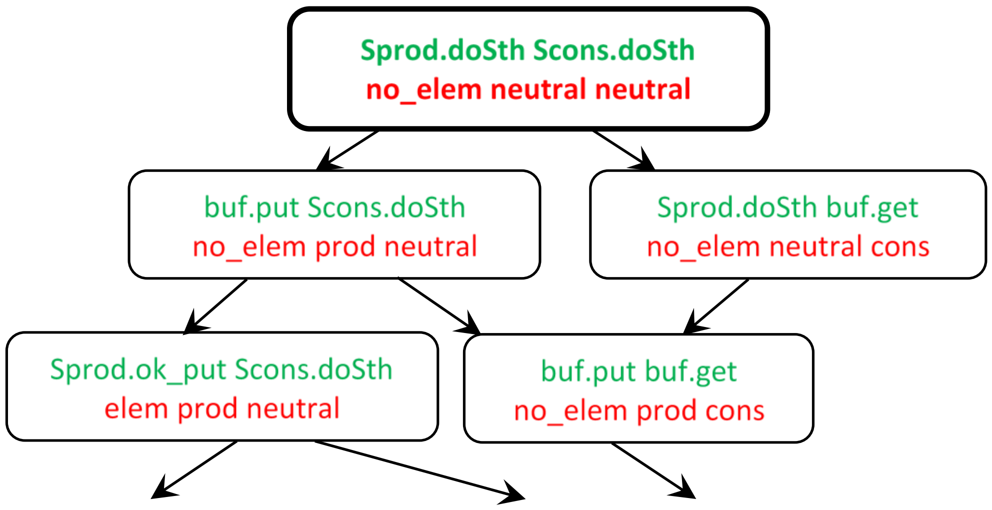


Figure 1. A fragment of LTS for the *buffer* system.

Of course, this LTS generated both from server view and from the agent view is identical, as they are projections onto servers and onto agents of a uniform system.

In Dedan program, communication and resource deadlocks may be identified and distributed termination may be checked. There is no deadlock in the example above, systems with deadlock are presented together with counterexamples in [14][15][16]. Also, a counterexample or other behaviour may be tracked in Deadn using a simulator. This simulation is performed over an LTS of a verified system. However, often a simulation should be performed over the components (servers and agents) of a verified system, shown separately and cooperating with each other. It is the reason of introducing Distributed Automata – an alternative formulation of IMDS system. Our distributed automata are equivalent to IMDS, but they allow for graphical definition and graphical simulation of distributed systems in terms of its components. Of course, the graphical form should preserve the autonomy of components and asynchrony of their behaviour. Also, communication dualism should be kept, therefore two forms of graphical specification are elaborated: one for the server view and the other for the agent view.

4.Distributed Autonomous and Asynchronous Automata (DA^3)

In computer engineering practice,various forms of automata are used to express the behavior of concurrent components. There are two reasons: graphical representation and individual modeling of distinct components. UML state diagrams are the good example [17]. For a graphical representation of distributed systems, and for a simulation in terms of parallel components of a system, distributed automata DA^3 were invented. We claim that our distributed automata are better to describe parallelism and cooperation in real distributed environment (with full asynchrony) than those enumerated below. Several different notions are called "distributed automata" in the literature.

1. Automata on distributed alphabets, communicating on common letters, based on Zielonka's automata [18]. The automata are called distributed automata in many papers concerning the behaviour of concurrent systems (in some of them additionally equipped with real time clocks for temporal analysis with real-time constraints): [19][20][21]. Those automata are called asynchronous in [22][23], although they perform actions (make the transitions) asynchronously only if the input letters are distinct. They make synchronous moves on common input letters (and it is the only common aspect of the automata). These automata should be called synchronous from our point of view. Alur's Timed Automata [9] (sometimes called

distributed [19]) are very close to Zielonka's automata, they are simply equipped with time constraints and time invariants. Similar are CSP processes, synchronizing on ! and ? operations rather than on letters of input alphabet. The advantage of CSP lays in specifying the direction of communication (! is sending, ? is receiving), which should be supplied informally in the case Zielonka's automata.

2. Close to Zielonka's automata are Büchi automata. They differ in distinguishing some states as accepting, used for LTL model checking (for instance in Spin [6]). They are called distributed automata in [24].
3. Message Passing Automata (MPA, called distributed automata in [25][26]) are really distributed and asynchronous. They have ordered sets on letters waiting for acceptance, called buffers or queues.
4. Pushdown Distributed Automata (PDA) are equipped with local memories of input symbols (stacks) [27].
5. The two former cases (MPA and PDA) are combined in [28][29] and called distributed automata.
6. The automata that are synchronous in fact, with central synchronizing server. In such an automaton, two independent actions are performed "simultaneously" by independent processors. The synchronizations between the processors are explicitly performed by a centralized processor: the synchronizer [30][31].
7. Grammar systems – languages for description of parallel systems, generated by automata with certain interleaving rules. They are called distributed automata in [32].
8. Single large automaton split into distributed parts called distributed automata [33].

We introduce a new version of automata, equivalent to IMDS formalism. We call them Distributed Autonomous, Asynchronous Automata - DA^3 (D-triple-A or DA-cubed) to distinguish them from all the listed formalisms, all called distributed automata. Our automata reflect the behavior of distributed components. The servers make decisions (perform actions) individually without any knowledge of other servers (autonomy) and messages are sent regardless of the states of target servers (asynchrony). As there are two views of a distributed system in IMDS, two forms of DA^3 were developed – Server-DA^3 and Agent-DA^3 (S-DA^3 and A-DA^3).

4.1. Server automata S-DA^3

An IMDS system in the server view may be shown as a set of communicating automata S-DA^3 (Distributed Server Automata), similar to MPA (point 3 in the enumeration above):

States of a server are nodes (we use node instead of state to avoid ambiguity) of corresponding automaton.

An initial state of the server is an initial node of the automaton. Actions of the server process are transitions of the automaton. The automaton is Mealy-style [34], labels of the transitions in the automaton have the form extracted from actions; an IMDS action $\lambda=((m,p),(m',p'))$ is converted to a transition from p to p' with a label m/m' (m is an input symbol conditioning the transition while m' is an output symbol produced on the transition); the transitions in the automaton of the server s_i are the relation in $P_i \times M \times M \times P_i$; of course m fulfil $target(m)=s_i$; note that "traditional" distinction between transition relation $P_i \times M \times P_i$ and output function $(P_i \times M \times P_i)\rightarrow M$ is not held, because in the set of actions can contain nondeterministic actions $\lambda_1=((m,p),(m_1,p'))$, $\lambda_2=((m,p),(m_2,p'))$, $m_2\neq m_1$. The automaton is equipped with an input set – a set of input symbols pending, corresponding to a set of pending messages at the server. Firing a transition $(p,m/m',p')$ in the automaton of server s retrieves the symbol m from the input set of this automaton and inserts the symbol m' to the input set of an automaton of the server s' appointed by m'. An initial input set consists of initial messages of agents directed to this server. The special agent-terminating action $\lambda=((m,p),(p'))$ is converted to a transition that does not produce any output symbol.

Formally, having the definition of S,A,V,R,P,M from IMDS (respectively: servers, agents, values, services, states, messages), we have the set $\mathcal{S}$ (reflected S) of n distributed sever automata

$\mathcal{Z}$={ z_i | i = 1...n }, where n is the number of servers in the set S. In the definition, $exp(arg)$ is used for powerset 2^{arg}. An ith distributed server automaton is $z_i=(s_i,P_i,p_{0i},F_i,X_i,X_{0i})$, where:

- $s_i \in S$ – ith server,
- P_i – set of nodes, states of s_i,
- $p_{0i} \in P_i$ – initial node,
- F_i={ $(p_1,m/m',p_2)$ for $\lambda=((m,p_1),(m',p_2))$ or $(p_1,m/,p_2)$ for $\lambda=((m,p_1),(p_2))$ | $p_1,p_2 \in P_i$, $\exists_{a_j \in A}$ $m,m' \in M_j$ } – set of transitions (for ordinary action and agent-terminating action, respectively),
- $X_i \in exp(\{m \mid target(m)=s_i\})$ – input set; X_i is a variable having a value of a set: a sender of a message (a transition delivering the message) inserts an element, execution of a transition removes an element, accordingly to rules for semantics below,
- $X_{0i} \in exp(\{m \mid target(m)=s_i,\ m \in M_0\})$ – initial input set.

The following conditions must hold, yet they are achieved by construction using the rules for semantics below:

- $\forall_{m1 \in Xi, m2 \in Xj, m1 \in Mk, m2 \in Ml}$ m1≠m2 ⇒ k≠l: for any agent at most one message may exists in the global configuration,
- $\forall_{(p1,m/m',p2) \in Fi}$ $\exists_{Fj}$ $m' \in F_j$: any output symbol is an input symbol of an automaton belonging to $\mathcal{Z}$.

The automata are simpler on intuitive level, illustrated in Fig. 2 for the server automata of buffer system. The input sets of the automata X_{buf}, X_{Sprod} and X_{Scons} are shown at the bottom of the picture, they change while the automata run.

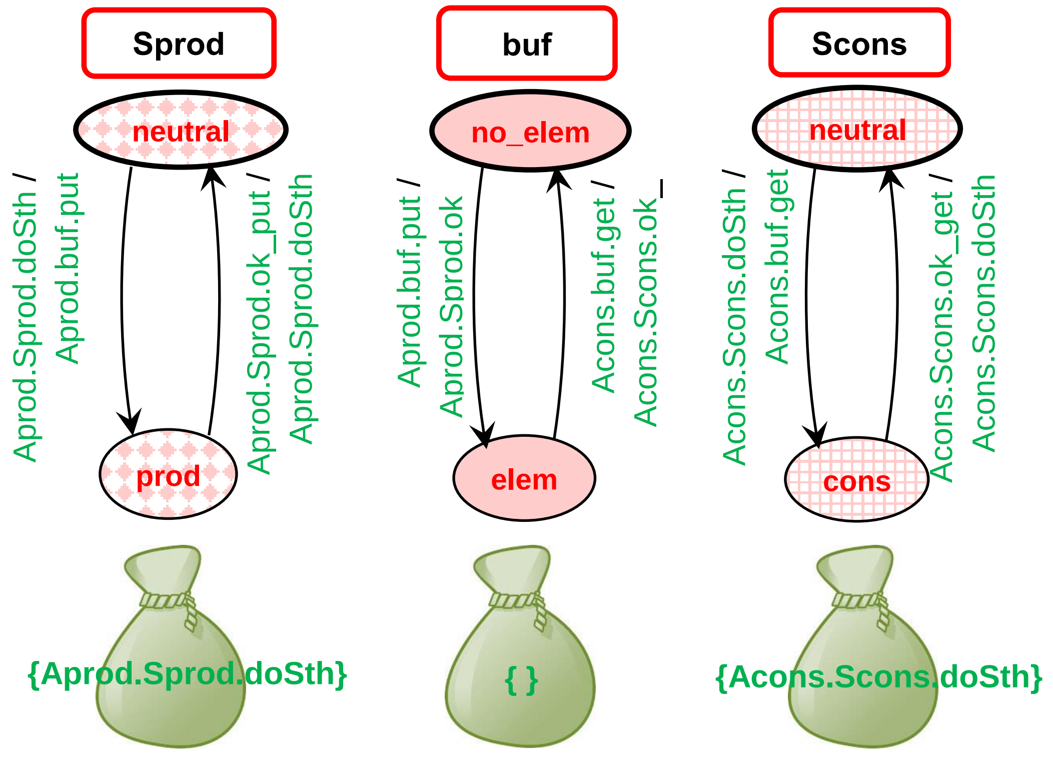


Figure 2. Server automata for *buffer* system. Under the automata there are *input sets* with their initial values.

The semantics of $\mathcal{Z}$ is defined as *global node space* ({ $T_{\mathcal{Z}}$ },$T_{\mathcal{Z}0}$,$next\mathcal{Z}$), where { $T_{\mathcal{Z}}$ } is a set of *global nodes*, $T_{\mathcal{Z}0}$ is an *initial global node* and $next\mathcal{Z}$ is a *transition relation*, defined as follows:

- The global node of $\mathcal{Z}$ is $T_{\mathcal{Z}}=((p_1,X_1),(p_2,X_2),\ldots,(p_n,X_n))$ (current states and input sets of pending messages of all servers).

- If in $T_{\mathcal{Z}}$ there exists an z_i in which (p_{i1},X_i), $(p_{i1},m/m',p_{i2})\in F_i$, $m\in X_i$, $target(m')=s_j$ then a possible next global node $T'_{\mathcal{Z}}$ is:

$$T'_{\mathcal{Z}} = \left((p_k,X'_k) \middle| \begin{cases} p'_k = p_k & for \quad k \neq i \\ p'_k = p_{i2} & for \quad k = i \end{cases} \begin{cases} X'_k = X_k \backslash \{m\} \cup \{m'\} & for \quad i = k = j \\ X'_k = X_k \backslash \{m\} & for \quad k = i \neq j \\ X'_k = X_k \cup \{m'\} & for \quad k = j \neq i \\ X'_k = X_k & for \quad i \neq k \neq j \end{cases}, k = 1 \ldots n\right) \tag{4.1}$$

(the automaton z_i changes its node to p_{i2}, all other automata preserve their nodes; the message m is extracted from the input set X_i of the automaton z_i, the continuation message m' is inserted into the input set X_j of the automaton z_j appointed by m', all other input sets remain unchanged; the special case is for ($i = k = j$, first case), where a server s_i sends a message to itself).

- If in $T_{\mathcal{Z}}$ there exists an z_i in which (p_1,X_i), $(p_{i1},m,p_{i2})\in F_i$, $m\in X_i$ (message m terminates the agent) then a possible next global node is:

$$T'_{\mathcal{Z}} = \left((p_k,X'_k) \middle| \begin{cases} p'_k = p_k & for \quad k \neq i \\ p'_k = p_{i2} & for \quad k = i \end{cases} \begin{cases} X'_k = X_k \backslash \{m\} & for \quad k = i \\ X'_k = X_k & for \quad k \neq i \end{cases}, k = 1 \ldots n\right) \tag{4.2}$$

(the automaton z_i changes its node to p_{i2}, all other automata preserve their nodes; the message m is extracted from the input set X_i of the automaton z_i, all other input sets remain unchanged).

- The initial global node is $T_{\mathcal{Z}0} = ((p_{01},X_{01}),(p_{02},X_{02}),\ldots,(p_{0n},X_{0n}))$.
- For given global node $T_{\mathcal{Z}}$, transition relation $nextT_{\mathcal{Z}}(T_{\mathcal{Z}})$ is a set of pairs $(T_{\mathcal{Z}},T'_{\mathcal{Z}})$. The transition relation $next\mathcal{Z} = \bigcup_{T_{\mathcal{Z}}} nextT_{\mathcal{Z}}(T_{\mathcal{Z}})$. If for $T_{\mathcal{Z}}$ there exist multiple possible next global nodes, one of them is chosen in nondeterministic way.

A global graph of $\mathcal{Z}$ cooperation may be elaborated in such a way that nodes are global nodes $T_{\mathcal{Z}}$, and edges are transitions in automata z_i. Of course, this graph is analogous to the LTS of IMDS system: global nodes contain states of all servers, input symbol (message) of a transition should be attributed to a source global node, while output symbol (message) to a target global node. The fragment of a global node space for the buffer system is presented in Fig. 3.

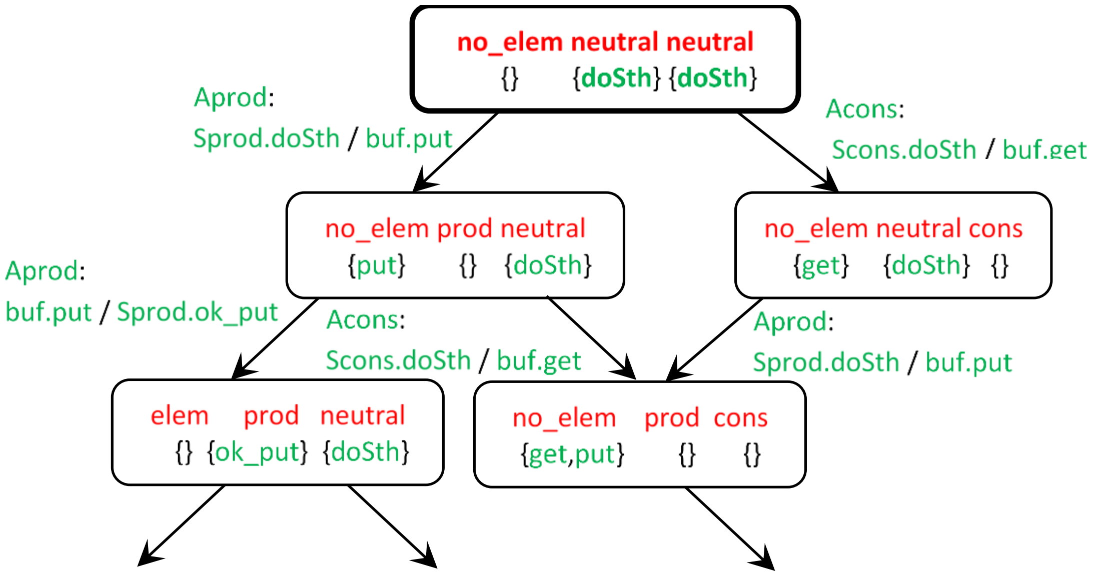


Figure 3. Fragment of global graph of server automata; first row contains states of the servers, second row input sets.

The initial sates of servers are in bold ovals. Server names are omitted in the state labels, because they are identical for all states in given server automaton. As the automata may be treated as patterns for creation of many instantiations of similar automata, agent and server formal parameters should be added for server types automata:

```
Sprod(Aprod,buf), buf(Aprod,Acons,Sprod,Scons), Scons(Acons,buf)
```

Every automaton is equipped with the input set of pending messages:

$X_{buf} \in exp(\{(\texttt{Aprod,buf,put}),(\texttt{Acons,buf,get})\})$.
$X_{Sprod} \in exp(\{(\texttt{Aprod,Sprod,doSth}),(\texttt{Aprod,Sprod,ok\_put})\})$.
$X_{Scons} \in exp(\{(\texttt{Acons,Scons,doSth}),(\texttt{Acons,Scons,ok\_get})\})$.

Note that both messages in the base set of X_{Sprod} cannot be included in X_{Sprod} at the same time, as they belong to the same agent, likewise in the case of X_{Scons}.

The initial input sets are:

$X_{0buf} = \emptyset$,
$X_{0Sprod} = \{(\texttt{Aprod,Sprod,doSth})\}$,
$X_{0Scons} = \{(\texttt{Acons,Scons,doSth})\}$.

The S-DA^3 are similar to Message Passing Automata. The difference is in the ordering of messages on the input of the automaton: in MPA pending messages are ordered in the input queue (or input buffer) [25][26], while in S-DA^3 any message form the input set may cause a transition (no ordering). If the input buffers are bounded, a deadlock may occur because of all processes sending to full buffers. Such a situation occurs when the size of buffers is taken too small [35]. IMDS helps to overcome this problem by posing an accurate limit for the input set maximum size (or the input buffer in the implementation): it is simply the number of agents. More precisely, it is the number of agents having their messages in the base set of the input set of the automaton.

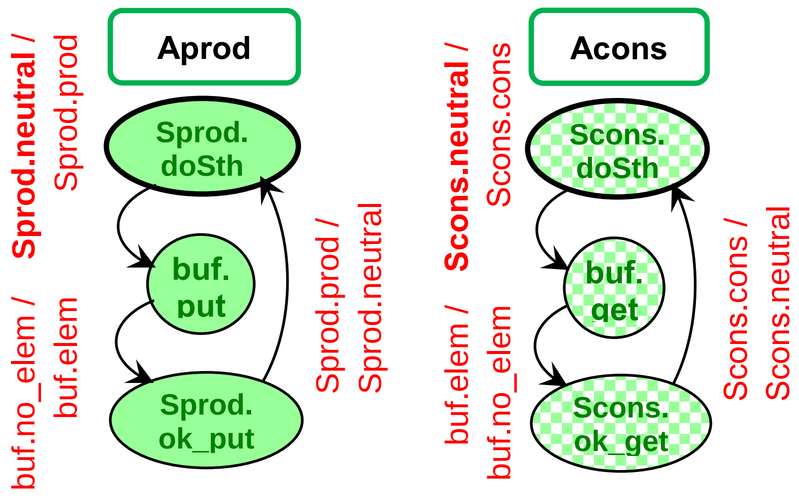
4.2. Agent automata A-DA^3

An IMDS system in the agent view may be shown as a set of communicating automata A-DA^3 (Agent Distributed Autonomous and Asynchronous Automata). We use term node in these automata instead of state, because states ate attributed to servers in IMDS and it may be misleading. The A-DA^3 automata are similar to Timed Automata with variables used in Uppaal [8] (but we consider only timeless systems here):

- Messages of an agent are nodes of a corresponding automaton.
- An initial message of the agent is an initial node of the automaton.
- Actions of the agent process are transitions of the automaton.
- The automaton is Mealy-style [34]; the labels of the transitions in the automaton have the form extracted from actions; an IMDS action λ=((m,p),(m',p')) is converted to a transition (m,p/p',m') from m to m' with a label p/p' (p is an input symbol conditioning the transition while p' is an output symbol produced on the transition; as before a transition relation in $M_i \times P \times M_i$ and output function $(M_i \times P \times M_i) \rightarrow P$ are replaced by a single relation in $M_i \times P \times P \times M_i$ due to possible nondeterminism in the set of actions: λ_1=((m,p),(m',p_1)), λ_2=((m,p),(m',p_2)), $p_2 \neq p_1$.
- For an agent-terminating action λ=((m,p),(p')), a special terminating node t in the automaton is added as transition destination node, and the transition is of the form (m,p/p',t). For t no outgoing transition is defined.
- The system is equipped with a global input vector – the vector of current input symbols – server's states. Firing a transition (m,p/p',m') in the automaton replaces the symbol p in the vector with the symbol p'. The initial global input vector consists of initial states of all automata.

Formally, having the definition of P,M,S,A,V,R from IMDS (respectively: states, messages, servers, agents, values, services), we have the set Ɐ (A upside down, rounded to distinguish it from the general quantifier) of k distributed agent automata $Ʉ = \{ ɐ_i \mid i = 1 \ldots k \}$, where k is the number of agents in the set A, and n is the number of servers in the set S (used in the definition of the agent automaton below). An ith distributed agent automaton is $ɐ_i = (a_i, M_i, m_{0i}, t_{ɐi}, G_i, Y, Y_0)$, where:

- a_i – ith agent,
- $M_i \cup \{t_{ɐi}\}$ – set of nodes, $t_{ɐi}$ is the destination node if the agent $ɐ_i$ terminates (it appears in terminating transition),
- $m_{0i} \in M_i$ – initial node,
- $G_i = \{ (m_1, p/p', m_2)$ for $\lambda = ((m_1,p),(m_2,p'))$ or $(m_1, p/p', t_{ɐi})$ for $\lambda = ((m_1,p),(p')) \mid m_1, m_2 \in M_i$, $\exists_{s_j \in S}\, p, p' \in P_j \}$ – set of transitions,
- $Y = [p_1, \ldots, p_n] \mid p_j \in P_j$ – global input vector (common for all a_i in the system); Y is a vector of variables, Y/i is the ith position of Y, every variable Y/i has a range over a set of states of server s_j: an action changes value of the variable at the position of its server: accordingly to rules for semantics below; ,
- $Y_0 = [p_{01}, \ldots, p_{0n}] \mid p_j \in P_j$, $p_j \in P_0$ – initial global input vector, consisting of initial states of all servers.



Sprod	buf	Scons
neutral	no_elem	neutral

Figure 4. Agent automata for *buffer* system. Under the automata there is the *global input vector* with its initial content.

The semantics of $Ʉ$ is defined as *global node space* $(\{ T_{Ʉ} \}, T_{Ʉ0}, nextɄ)$, where $\{ T_{Ʉ} \}$ is a set of *global nodes*, $T_{Ʉ0}$ is *initial global node* and $nextɄ$ is a *transition relation*, defined as follows:

- The global node of $Ʉ$ is $T_{Ʉ} = \{m_1, \ldots, m_k, Y\}$, $m_i \in M_i \cup \{t_{ɐi}\}$. If in $T_{Ʉ}$ there exists m_i, for which there exists $(m_i, p/p', m_j) \in G_i$, $p, p' \in P_x$ (message m_i causes a change of a state from p to p' in a server s_x, $target(m_i) = s_x$, and a message m_j to a server s_y, $target(m_j) = s_y$ is issued) then a possible next global node is:

$$T'_{Ʉ} : \forall_{m \epsilon T_{Ʉ}} \left\{ \begin{array}{r} m = m_i \wedge m = m_j \;\Rightarrow\; m_i \in T'_{Ʉ} \\ m = m_i \wedge m \neq m_j \;\Rightarrow\; m_j \in T'_{Ʉ} \\ m \neq m_i \;\Rightarrow\; m_i \in T'_{Ʉ} \end{array} \right\} ; \tag{4.3}$$

$$Y' = [p_1, \ldots, p_n] \mid p_k = \left\{ \begin{cases} Y/k & for\ \ k \neq x \\ p' & for\ \ k = x \end{cases} \right\} ,\ k = 1 \ldots n$$

(the automaton $\mathfrak{v}_i$ i changes its node to m_j, all other automata preserve their nodes; the state p in input vector Y is replaced by p' in the position of the server s_x, appointed by p and p', all other elements of the vector Y remain unchanged).

- If in { $T_{Ʉ}$ } there exists m_i, for which there exists $(m_i,p/p',t_{\mathfrak{v}i})\in G_i$, $p,p'\in P_x$ (message mi is a last message in the run of agent a_i, then the agent terminates, a sever s_x appointed by the message m_i, $target(m_i)= s_x$ changes its state from p to p') then a possible next global node is:

$$T'_{Ʉ} : \forall_{m\epsilon T_{Ʉ}} \begin{Bmatrix} m = m_i & \Rightarrow & m_i \notin T'_{Ʉ} \\ m \neq m_i & \Rightarrow & m_i \in T'_{Ʉ} \end{Bmatrix} \quad ;$$
$$Y' = [p_1, \dots, p_n] \ \mid \ p_k = \begin{Bmatrix} Y/k & for \ \ k \neq x \\ p' & for \ \ k = x \end{Bmatrix} \quad , \ k = 1 \dots n \tag{4.4}$$

(the automaton $\mathfrak{v}_i$ changes its node to $t_{\mathfrak{v}i}$ all other automata preserve their nodes; the state p in Y is replaced by p' as above). The initial global node $T_{Ʉ0}=\{m_{01},\dots,m_{0n},Y_0\}$. For given global node $T_{Ʉ}$, transition relation $nextT_{Ʉ}(T_{Ʉ})$ is a set of pairs $(T_{Ʉ},T'_{Ʉ})$. The transition relation $nextɄ = \bigcup_{T_{Ʉ}} nextT_{Ʉ}(T_{Ʉ})$. In given global node $T_{Ʉ}$, if there are multiple next nodes possible, one of them is chosen in nondeterministic way.

Distributed agent automata for the buffer system are illustrated in Fig. 4. The initial messages of the agents are in bold ovals. Agent identifies are omitted in message labels (nodes of the automata), because they are identical for all messages in given agent type automaton. Below is the global input vector of current states of servers:

Y = [(`Sprod`,*value* ∈{`neutral`,`prod`}), (`buf`,*value* ∈{`no_elem`,`elem`}),
(`Scons`,*value* ∈ {`neutral`,`cons`})].

The initial content of the global input vector is:

Y_0 =[(`Sprod`,`neutral`),(`buf`,`no_elem`),(`Scons`,`neutral`)].

A global graph of $Ʉ$ cooperation may be elaborated analogously to the graph of $\mathcal{Z}$: nodes of the global graph are global nodes $T_{Ʉ}$, and edges are transitions in automata $\mathfrak{v}_i$. This graph is analogous to the global graph of S-DA³ and to the LTS of IMDS system (global nodes contain messages of all agents, input symbol/state of a transition should be attributed to a source global node, while output symbol/state to a target global node).

5. Equivalence of the formalisms

In this section we show the equivalence of the IMDS model with both automata-based models: S-DA³ and A-DA³. the equivalence is based on similar LTS structures, i.e., every node of LTS – including initial ones – should contain the same elements (states and messages) as corresponding nodes in other LTS'es. That is, corresponding nodes should contain exactly the same sets of elements, but in various forms: set elements in IMDS, nodes and elements of input sets of S-DA³, nodes and elements of input vector in A-DA³, except terminating nodes. Furthermore, the transitions should join corresponding nodes in all LTS'es. To show the equivalence, the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$ for the action λ is needed. This rule comes directly ftom the definition of IMDS:

$$\forall_{\lambda\in\Lambda}\ \lambda=((m,p),(m',p'))\ T_{inp}(\lambda) \supset \{m,p\} \Rightarrow T_{out}(\lambda)=T_{inp}(\lambda)\setminus\{m,p\} \cup \{m',p'\} \tag{5.1}$$
$$\forall_{\lambda\in\Lambda}\ \lambda=((m,p),(p'))\ \ T_{inp}(\lambda) \supset \{m,p\} \Rightarrow T_{out}(\lambda)=T_{inp}(\lambda)\setminus\{m,p\} \cup \{p'\}$$

1. LTS node
   - IMDS: T={ p,m ∣ $p\in P$, $m\in M$ }, Every p from a different s every m from a different a except terminated a,
   - S-DA³: $T_{\mathcal{Z}}=((p_1,X_1),\dots,(p_n,X_n))$ Every p from a different s – from definition in $T_{\mathcal{Z}}$ every s participates (will be shown later). For every pair X_1,X_2 both cannot contain m appointing

the same a (will be shown later). No X can contain m appointing a terminated a (will be shown later),

- A-DA[3]: $T_Ʉ=\{m,m',m'',...,Y\}$ Every m appoints different a – from definition in $T_Ʉ$ every a participates, except terminated ones (will be shown later). For every pair m_1,m_2 both cannot appoint the same a (will be shown later). No m can appoint a terminated a (will be shown later). Y contains the states of all servers – from definition.

2. Initial LTS node
   - IMDS: $T_0=\{ p_0,m_0 \mid p_0\in P_{ini},\ m_0\in M_{ini} \}$,
   - S-DA[3]: $T_{Ƶ0}=((p_{01},X_{01}),...,(p_{0n},X_{0n}))$ Every p from a different s – from definition. Every s participates – from definition. For every pair X_{01},X_{02} both cannot contain m appointing the same a (from definition, as X_{0i} contains m_0 for a starting from the server s_i, and $\forall_{ai\in A}\ card(M_i \cap M_{ini})=1$ (the initial set of messages contains exactly one message for every agent). Every a has its m_0 in some X_{0i} – this from which the agents starts (X_{0i} are indexed by servers and every m_0 belongs to some M_i).
   - A-DA[3]: $T_Ʉ=\{m_{01},...,m_{0k},Y_0\}$ In every pair m_1,m_2 both cannot appoint the same a (from definition, as $\forall_{ai\in A}\ card(M_i \cap M_{ini})=1$ (the initial set of messages contains exactly one message for every agent). No m can be $t_ɐ$ – from definition. Every element of Y_0 appoints different server – from definition.

3. Transition in LTS (regular)
   - IMDS: $(T_{inp}(\lambda),\lambda,T_{out}(\lambda)) \mid \lambda\in\Lambda,\ \lambda=((m,p),(m',p'))$
   - S-DA[3]: From $T_Ƶ$ there exists a regular transition $(p,m/m',p')$ to $T'_Ƶ$ corresponding to $\lambda=((m,p),(m',p'))$, in the automaton $ƶ_i$ of the server s_i appointed by p to the state p', retrieving the message m from X_i and inserting a message m' to X_j appointed by m'.
     Both messages m,m' belong to the agent a. If $T_Ƶ$ corresponds to $T_{inp}(\lambda)$, then according to (4.1) in $T'_Ƶ$ p is replaced by p' and in the union of all X_i m is replaced by m', and all other states and messages are equal in $T_Ƶ$ and $T'_Ƶ$, which fulfils the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$.
     For every regular $\lambda=((m,p),(m',p'))$ having p on input such a transition $(p,m/m',p')$ exists in automaton $ƶ_i$ appointed by p, and no other than for such λ transition exists, so it exactly corresponds to the set of regular actions having p on input.
     In every regular transition in $ƶ_i$, the set of servers is preserved (as p and p' appoint the same server) and the set of agents (as m and m' appoint the same agent).
   - A-DA[3]: From $T_Ʉ$ there exists a regular transition $(m,p/p',m')$ to $T'_Ʉ$ corresponding to $\lambda=(m,p),(m',p'))$, in the automaton $ɐ_i$ of the agent a_i appointed by m to the message m', replacing the state p in Y by the state p' of the same server s appointed by states p,p' (in the position of the server s in Y).
     If $T_Ʉ$ corresponds to $T_{inp}(\lambda)$, then according to (4.3) in $T'_Ʉ$ m is replaced by m' and p in Y is replaced by p', and all other states and messages are equal in $T_Ʉ$ and $T'_Ʉ$, which fulfils the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$.
     For every regular $\lambda=((m,p),(m',p'))$ having m on input such a transition $(m,p/p',m')$ exists in automaton $ɐ_i$ appointed by m, and no other than for such λ transition exists, so it exactly corresponds the set of regular actions having m on input.
     In every regular transition in $ɐ_i$, the set of agents is preserved (as m and m' appoint the same agent) and the set of servers (as p and p' appoint the same server).

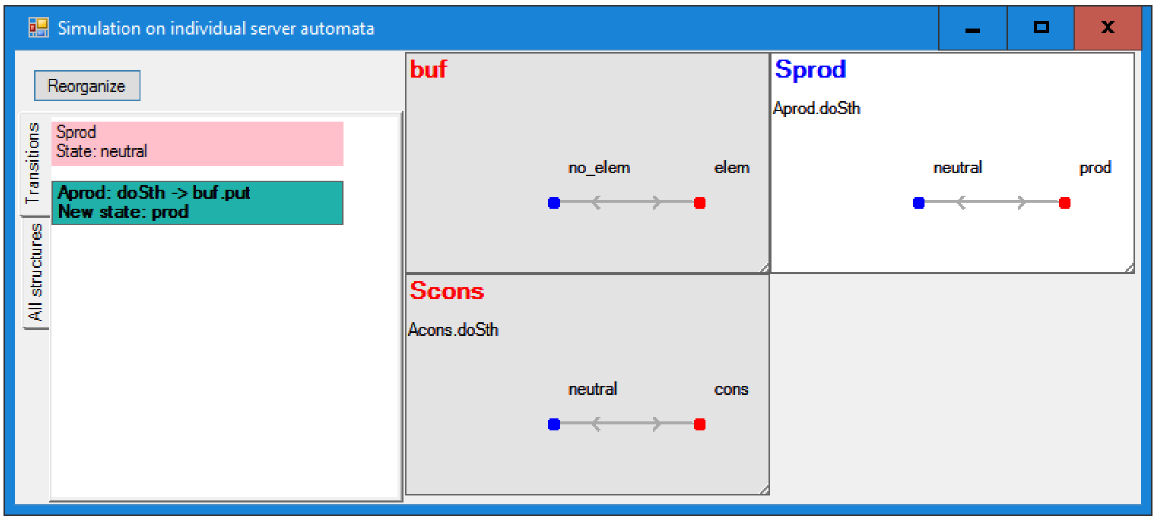
4. Transition in LTS (agent-terminating)
   - IMDS: $(T_{inp}(\lambda),\lambda,T_{out}(\lambda)) \mid \lambda\in\Lambda,\ \lambda=((m,p),(p'))$
   - S-DA[3]: From $T_Ƶ$ there exists a terminating transition $(p,m/,p')$ to $T'_Ƶ$ corresponding to $\lambda=((m,p),(p'))$, in the automaton $ƶ_i$ of the server s_i appointed by p, to the state p', retrieving the message m from X_i. The message m belongs to an agent a.
     If $T_Ƶ$ corresponds to $T_{inp}(\lambda)$, then according to (4.2) in $T'_Ƶ$ p is replaced by p' and m is extracted, and all other states and messages are equal in $T_Ƶ$ and $T'_Ƶ$, which fulfils the rule

of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$ (terminating action).
For every terminating $\lambda = ((m,p),(p'))$ having p on input such a transition $(p,m/,p')$ exists in automaton s_i appointed by p, and no other than for such λ transition exists, so it exactly corresponds the set of terminating actions having p on input.
In every terminating transition in s_i, the set of servers is preserved (as p and p' appoint the same server) and the set of agents in $T'_{\mathcal{Z}}$ is smaller by the agent appointed by m. Consequently, there is no way to reestablish a terminated agent.

- A-DA³: From $T_{\mathcal{U}}$ there exists an agent a terminating transition $(m,p/p',t_{\mathfrak{a}})$ to $T'_{\mathcal{U}}$ corresponding to $\lambda = ((m,p),(p'))$, in the automaton $\mathfrak{a}_i$ of the agent a_i appointed by m, to the message m', replacing the message m by $t_{\mathfrak{a}i}$.
The state p belongs to a server s. If $T_{\mathcal{U}}$ corresponds to $T_{inp}(\lambda)$, then according to (4.4) in $T'_{\mathcal{U}}$ m is replaced by $t_{\mathfrak{a}i}$ and p is replaced in Y by p', all other states and messages are equal in $T_{\mathcal{U}}$ and $T'_{\mathcal{U}}$, which fulfils the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$ (terminating action).
For every terminating $\lambda = ((m,p),(p'))$ having m on input such a transition $(m,p/p',t_{\mathfrak{a}})$ exists in automaton $\mathfrak{a}_i$ appointed by m, and no other than for such λ transition exists, so it exactly corresponds the set of terminating actions having m on input.
In every terminating transition in $\mathfrak{a}_i$, the set of servers is preserved (as p and p' appoint the same server) and the set of agents in $T'_{\mathcal{U}}$ is smaller by the agent appointed by m, which is replaced by $t_{\mathfrak{a}i}$. Consequently, there is no way to reestablish a terminated agent.

5. Semantics of LTS
- IMDS: Interleaving, nondeterministic,
- S-DA³: Interleaving, nondeterministic,
- A-DA³: Interleaving, nondeterministic.

6. Conclusions and further work



Sprod
State: neutral

Aprod: doSth -> buf.put
New state: prod

Figure 5. Simulation over SDA³ automata. Every automaton has its own sub-window, a chosen automaton has blue caption and light background. On the left, current state of the chosen automaton and a list of possible transitions are displayed.

The Dedan program supports an engineer in specification of distributed systems and their verification for deadlocks freeness and distributed termination. If a deadlock occurs, a sequence diagram of messages and states is generated, leading from the initial configuration to the deadlock. If the deadlock is not total, the servers/agents taking part in the deadlock is are shown. Distributed automata (in S-DA³ or in A-DA³ version) allow to design the system in graphical form, and to simulate the components of the system and their cooperation instead of a simulation over the full configuration graph (LTS). Fig. 5 shows the simulation of the buffer system in Dedan. Also, a counterexample may be observed as a sequence of transitions in cooperating DA3 automata. Engineers are familiar with the notion of automata (S-DA3 are similar to Message Passing Automata [25][26] and A-DA3 are like Timed Automata with global variables of Uppaal [8]) and they may be naturally used in distributed systems design. For example, some models of transport cases were modeled. Observation of the server view is equivalent to exchange of messages between road segment controllers that automatically lead the vehicles on the roads [16]. In the agent view, it is the observation of vehicles moving over the road, with interactions to other vehicles occupying some segments of the road. Possible deadlocks in communication may by easily identified, and the verifier shows the behavior of vehicles leading to a deadlock as transitions of DA3 automata. Table 1 compares the features of a distributed system, observed in equivalent formalisms: IMDS and DA3.

Table 1. Verification facilities in the three equivalent formalisms.

Formalism:	**IMDS**	**S-DA3**	**A-DA3**
Main features	Specification, model checking, simulation	Graphical input, simulation	Graphical input, simulation
Notions	state	node	element of global input vector, input/output symbol on transitions
	message	element of input set, input/output symbol on transitions	node
	configuration	global node	global node
	action	transition	transition
	initial state	initial node	initial element of global input vector
	initial message	initial element of input set	initial node
	initial configuration	initial nodes and initial input sets of all automata	initial nodes and initial global input vector
	Labeled Transition System	Global node space: all states and input sets in global nodes, input and output symbols on transitions	Global node space: all messages and global input vector in global nodes, input and output symbols on transitions
Features	• Resource deadlock • Communication deadlock • Partial deadlock • Total deadlock • Partial distributed termination • Total distributed termination • Counterexamples/ witnesses • Configuration space inspection • Simulation over configuration space	• Graphical definition of a system (as servers) • Simulation over individual server automata • Counterexample projected onto individual server automata • Counterexample–guided simulation	• Graphical definition of a system (as agents) • Simulation over individual agent automata • Counterexample projected onto individual agent automata • Counterexample–guided simulation

The next steps are:

- Timed DA[3] automata, in which time constraints will be added to actions and message passing. This will allow to check for deadlocks in real time-dependent systems.
- Probabilistic DA[3] automata allowing to identify a probability of a deadlock if the alternative actions in system processes are equipped with probabilities.
- Language-based input – elaboration of two languages for distributed systems specification: one for the server view (exploiting locality in servers and message passing) and the other one for the agent view (exploiting travelling of agents and resource sharing in distributed environment); a preliminary version of a declarative language-based preprocessor for a server view of verified systems is developed by the students of ICS, WUT (Institute of Computer Science, Warsaw University of Technology).
- Agent's own actions – equipping the agents with their own sets of actions, carried in their "backpacks", parametrizing their behavior; this will allow for modelling of mobile agents (agents carrying their own actions model code mobility) and to avoid many server types in specification, differing slightly.

The Dedan environment is successfully used in operating systems laboratory in ICS, WUT. The students verify their solutions of synchronization problems. Graphical definition of component automata and simulation over distributed automata supports the procedure of verification.

Supplementary Materials: The Dedan program is available online at http://staff.ii.pw.edu.pl/dedan/files/DedAn.zip, examples at http://staff.ii.pw.edu.pl/dedan/files/examples.zip.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

Appendix A - Practical example of DA[3]: Automatic Vehicle Guidance System

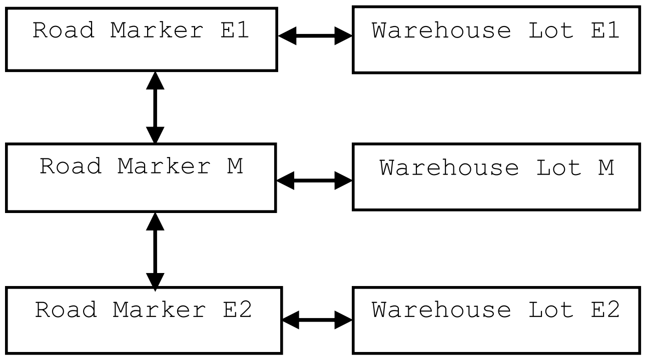


Figure A1. Structure of road segment controllers.

The buffer example is a tiny one, just to present the main ideas. Now we will introduce the Automatic Vehicle Guidance System (AVGS) from [16]. The system consists of road markers and warehouse lots, presented in Fig. A1, communicating with each other in order to guide autonomous moving platforms (AMPs) from `Lot_E1` to `Lot_E2` or reverse way. There is an obvious conflict in `MarkerM`, and it may be defeated using the `LotM` as a staggered arrangement. There are six servers representing the controllers of *Lots* and *Markers*, with a protocol of requesting and granting road segments governed by the controllers.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 December 2019 doi:10.20944/preprints201912.0400.v1

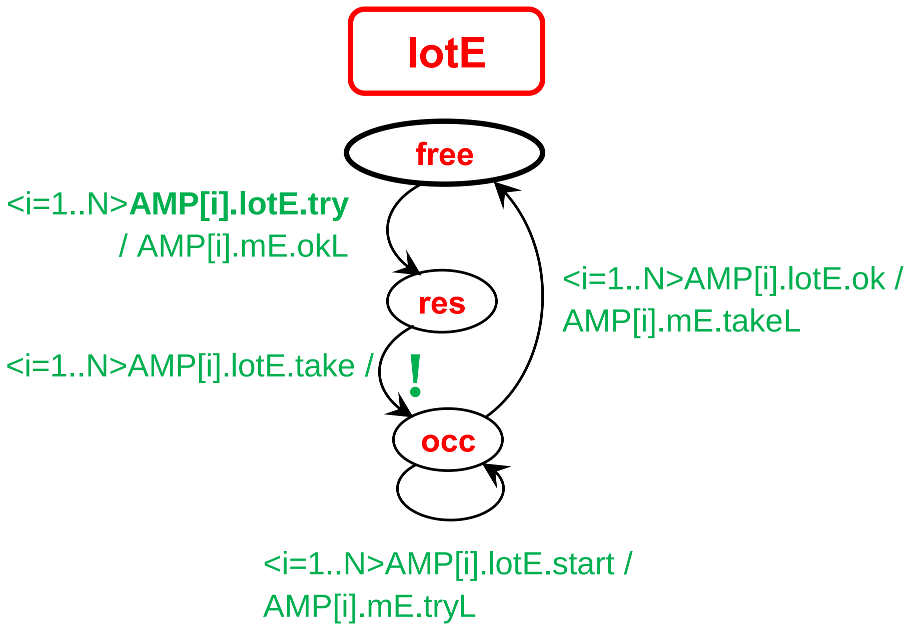

lotE
free
<i=1..N>AMP[i].lotE.try / AMP[i].mE.okL
res
<i=1..N>AMP[i].lotE.ok / AMP[i].mE.takeL
<i=1..N>AMP[i].lotE.take / !
occ
<i=1..N>AMP[i].lotE.start / AMP[i].mE.tryL

Figure A2. Server automaton of `lotE` server type.

The server view describes the system from the point of view of communicating controllers. The code of AVGS in IMDS source notation is given below.

```
#DEFINE N 2
server: mE (agents AMP[N]; servers mM,lotE),
//Edge Road Marker 4. ...
server: mM (agents AMP[N]; servers mE[2],lotM),
//Middle Road Marker
services {tryE[2],tryL[2],okE[2],notE[2],okL[2], takeE[2],takeL[2],switch[2]},
states {free,resE[2],resL[2],occ},
actions {
//going to ME1 or ME2
<i=1..N><j=1..2> {AMP[i].mM.tryE[j], mM.free} -> {AMP[i].mE[j].okM[j], mM.resE[j]},
<i=1..N><j=1..2> {AMP[i].mM.takeE[j], mM.resE[j]} -> {AMP[i].mM.switch[3-j], mM.occ},
<i=1..N><j=1..2> {AMP[i].mM.switch[j], mM.occ} -> {AMP[i].mE[j].tryM[j], mM.occ},
<i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.occ} -> {AMP[i].mE[j].takeM, mM.free},

//on a way to ME1 or ME2 may go to LE if MEi occupied
<i=1..N><j=1..2> {AMP[i].mM.notE[j], mM.occ} -> {AMP[i].lotM.try[j], mM.occ},
<i=1..N><j=1..2> {AMP[i].marker2.okL[j], mM.occ} -> {AMP[i].lotM.take[j], mM.free},

//going from PL2 - goes to RM1(mE[1]) or RM3(mE[2])
<i=1..N><j=1..2> {AMP[i].mM.tryL[j], mM.free} -> {AMP[i].lotM.ok[j], mM.resL[j]},
<i=1..N><j=1..2> {AMP[i].mM.takeL[j], mM.resL[j]} -> {AMP[i].mE[j].tryM[j], mM.occ},
<i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.occ} -> {AMP[i].mE[j].takeM, mM.free},
};
server: lotE(agents AMP[N];servers mE),
//Edge Warehouse Lot
...
server: lotM(agents AMP[N];servers mM),
//Middle Warehouse Lot
...
servers mE[2],mM,lotE[2],lotM;
agents AMP[N];
init -> {
    <j=1..2>     mE[j](AMP[1..N],mM,lotE[j]).free,
                 mM(AMP[1..N],mE[1,2],lotM).free,
    <j=1..2>     lotE[j](AMP[1..N],mE[j]).occ,
                 lotM(AMP[1..N],mM).free,
    <j=1..2>     AMP[j].lotE[j].start,
}.
```

In the example, some servers and agents are grouped into vectors (lines 59,60). Also, some formal parameters have the form of vectors (l.2, 19, ...). Services and states may also be vectors (l.4,23). For a compact definition, repeaters precede the actions in a server type (lines 10-17, ...). The indices of agents, states and services indicate individual instances (l.26-29). Markers E and M are shortened to `mE` and `mM`. The server type `lotE` is shown as S-DA^3 automaton in Fig. A2. Note the transition from res to `occ` (with exclamation mark) – it as an agent-terminating transition, as an `AMP` reached its destination. No output message is present. Multiple transitions are denoted by repeaters, as in the source code.

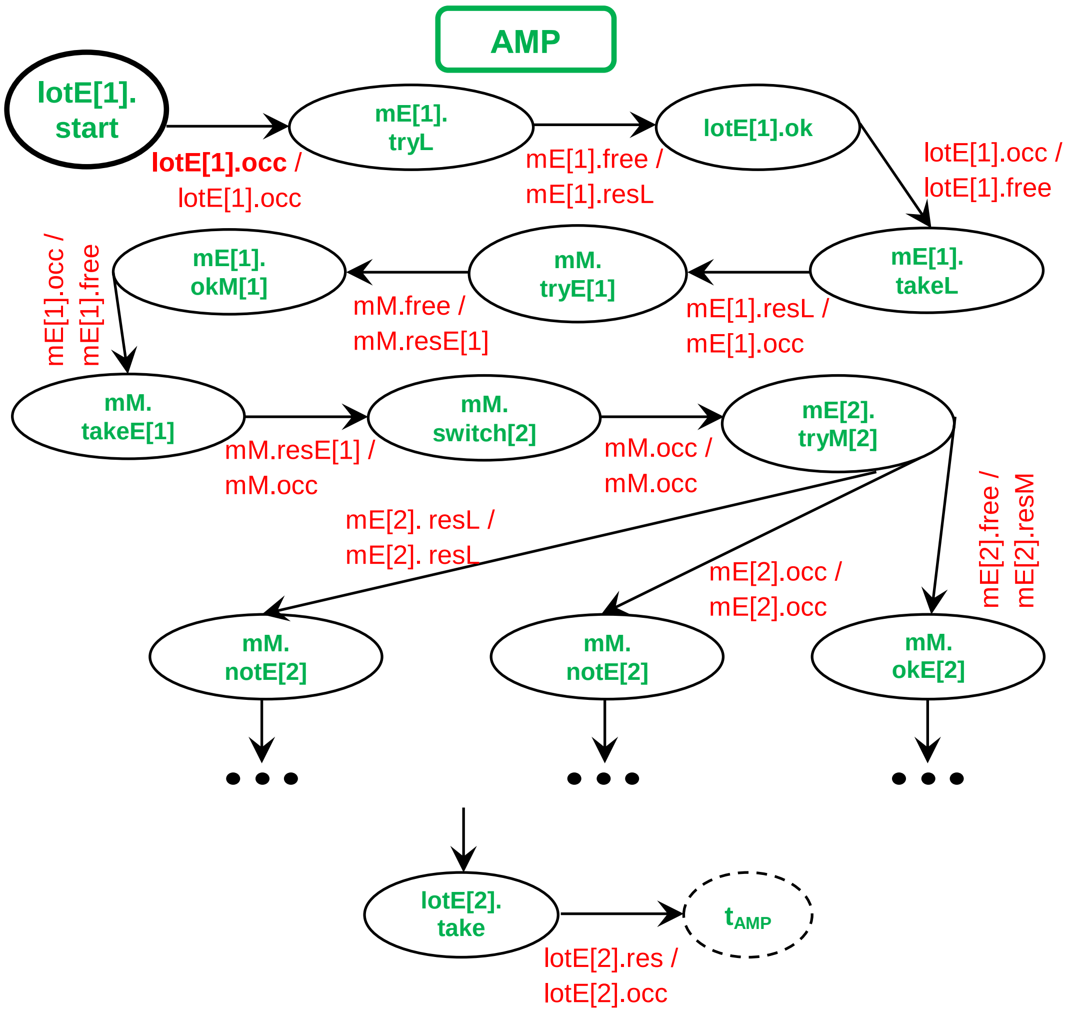


Figure A3. Agent automaton of `AMP` agent type.

The server view of the system is automatically converted to the agent view by the Dedan program. In the agent view, actions are grouped for individual agents. During the conversion, the type `AMP` is split into two separate types `AMP` and `AMP__1`, due to different initial messages. Consequently, vector elements `AMP[1]` and `AMP[2]` are renamed to separate agent variables `AMP` and `AMP__1`. The agent view shows the system from the point of view of the AMP vehicles (Listing below). Fig. A3 presents a fragment of the `AMP` agent type automaton.

```
1. agent: AMP (servers mE[2]:mE,mM:mM,lotE[2]:lotE,lotM:lotM),
2. actions {
3.          {AMP.lotE[1].try, lotE[1].free} -> {AMP.mE[1].okL, lotE[1].res},
4.          {AMP.lotE[1].ok, lotE[1].occ} -> {AMP.mE[1].takeL, lotE[1].free},
5.          {AMP.lotE[1].start, lotE[1].occ} -> {AMP.mE[1].tryL, lotE[1].occ},
6.          {AMP.lotE[1].take, lotE[1].res} -> {lotE[1].occ},
7.          {AMP.lotE[2].try, lotE[2].free} -> {AMP.mE[2].okL, lotE[2].res},
8.          {AMP.lotE[2].ok, lotE[2].occ} -> {AMP.mE[2].takeL, lotE[2].free},
9.          {AMP.lotE[2].start, lotE[2].occ} -> {AMP.mE[2].tryL, lotE[2].occ},
          {AMP.lotE[2].take, lotE[2].res} -> {lotE[2].occ},
<j=1..2> {AMP.lotM.try[j], lotM.free} -> {AMP.mM.okL[j], lotM.res[j]},
<j=1..2> {AMP.lotM.ok[j], lotM.occ[j]} -> {AMP.mM.takeL[j], lotM.free},
<j=1..2> {AMP.lotM.take[j], lotM.res[j]} -> {AMP.mM.tryL[j], lotM.occ[j]},
...
```

Appendix B - Using DA^3 in the Dedan program

The basic form used in Dedan program is IMDS, because it allows for automatic conversion between the server view and the agent view of a system. Yet, the specification in the form of a relation between pairs λ=((*state*,*message*),(*state'*,*message'*)) is exotic for the users. Therefore, an alternative input form of DA^3 automata is provided.

A system may be simulated over the global space of configurations (LTS), but it is also possible to simulate it in terms of S-DA^3, as illustrated in Fig. 5. All of the automata in the system are displayed, with input sets of pending messages under automata identifiers shown. The current states of the automata are blue.

A user can choose an automaton (Sprod in the example, the chosen automaton has white background and blue name), and then a list of transitions from the current state of the chosen automaton is displayed on the left (with enabled ones distinguished; it is only one transition in this case, and it is enabled). Next, the user may choose a transition from the enabled ones. In the example it is only one transition enabled, leading from neutral to prod (with acceptance of `doSth` message and issuing of put message to `buf`). If the user clicks an enabled transition, it is "executed" and a destination automaton of the message becomes current (`buf` in this case).

References

1. Daszczuk, W.B.: Specification and Verification in Integrated Model of Distributed Systems (IMDS). MDPI Comput. **7** (4), 1–26 (2018).
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge, MA (2008). ISBN 9780262026499
3. Schwarick, M.: MARCIE - Model Checking and Reachability Analysis Done EffiCIEntly. Eighth International Conference on Quantitative Evaluation of SysTems, Aachen, Germany, 5-8 Sept. 2011, pp. 91-100, IEEE (2013). doi: 10.1109/QEST.2011.19
4. Daszczuk, W.B.: Siphon-based deadlock detection in Integrated Model of Distributed Systems (IMDS). In: Federated Conference on Computer Science and Information Systems, 3rd Workshop on Constraint Programming and Operation Research Applications (CPORA'18), Poznań, Poland, 9-12 Sept. 2018. pp. 425–435. IEEE (2018). doi:10.15439/2018F114
5. Heiner, M., Schwarick, M., Wegener, J.-T.: Charlie – An Extensible Petri Net Analysis Tool. In: 36th International Conference, PETRI NETS 2015, Brussels, Belgium, 21-26 June 2015. pp. 200–211. Springer International Publishing, Cham, Switzerland (2015). doi:10.1007/978-3-319-19488-2_10
6. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23** (5), 279–295 (1997). doi:10.1109/32.588521
7. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. Int. J. Softw. Tools Technol. Transf. **2** (4), 410–425 (2000). doi:10.1007/s100090050046
8. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing UPPAAL over 15 years. Softw. Pract. Exp. **41** (2), 133–142 (2011). doi:10.1002/ spe.1006
9. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126** (2), 183–235 (1994). doi:10.1016/0304-3975(94)90010-8
10. Lanese, I., Montanari, U.: Hoare vs Milner: Comparing Synchronizations in a Graphical Framework With Mobility. Electron. Notes Theor. Comput. Sci. **154** (2), 55–72 (2006). doi:10.1016/j.entcs.2005.03.032
11. May, D.: OCCAM. ACM SIGPLAN Not. **18** (4), 69–79 (1983). doi:10.1145/ 948176.948183
12. Reniers, M.A., Willemse, T.A.C.: Folk Theorems on the Correspondence between State-Based and Event-Based Systems. In: 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, 22-28 Jan. 2011, LNCS vol. 6543. pp. 494–505. Springer-Verlag, Berlin Heidelberg (2011). doi:10.1007/978-3-642-18381-2_41
13. Penczek, W., Szreter, M., Gerth, R., Kuiper, R.: Improving Partial Order Reductions for Universal

Branching Time Properties. Fundam. Informaticae. **43** (1-4), 245–267 (2000). doi:10.3233/FI-2000-43123413

14. Daszczuk, W.B.: Deadlock Detection Examples: The Dedan Environment at Work. In: Integrated Model of Distributed Systems, SCI vol. **817**, pp. 53–85. Springer Nature, Cham, Switzerland (2020). doi: 10.1007/978-3-030-12835-7_5
15. Daszczuk, W.B.: Asynchronous Specification of Production Cell Benchmark in Integrated Model of Distributed Systems. In: Bembenik, R., Skonieczny, L., Protaziuk, G., Kryszkiewicz, M., and Rybinski, H. (eds.) 23rd International Symposium on Methodologies for Intelligent Systems, ISMIS 2017, Warsaw, Poland, 26-29 June 2017, Studies in Big Data, vol. **40**. pp. 115–129. Springer International Publishing, Cham, Switzerland (2019). doi: 10.1007/978-3-319-77604-0_9
16. Czejdo, B., Bhattacharya, S., Baszun, M., Daszczuk, W.B.: Improving Resilience of Autonomous Moving Platforms by real-time analysis of their Cooperation. Autobusy-TEST. **17** (6), 1294–1301 (2016). arXiv:1705.04263
17. UML, Available online: http://www.uml.org/ (accessed 6.12.2019)
18. Zielonka, W.: Notes on finite asynchronous automata. RAIRO - Theor. Informatics Appl. **21** (2), 99–135 (1987). doi: 10.1051/ita/1987210200991
19. Krishnan, P.: Distributed Timed Automata. Electron. Notes Theor. Comput. Sci. **28**, 5–21 (2000). doi:10.1016/S1571-0661(05)80627-9
20. Muscholl, A.: Automated Synthesis of Distributed Controllers. In: Automata, Languages, and Programming - 42nd International Colloquium, {ICALP} 2015, Kyoto, Japan, 6-10 July 2015, Part {II}. pp. 11–27 (2015). doi:10.1007/978-3-662-47666-6_ 2
21. Diekert, V., Muscholl, A.: On Distributed Monitoring of Asynchronous Systems. In: 19th International Workshop on Logic, Language, Information and Computation, WoLLIC 2012, Buenos Aires, Argentina, 3-6 Sept. 2012. pp. 70–84. Springer, Berlin Heidelberg (2012). doi:10.1007/978-3-642-32621-9_5
22. Mukund, M.: Automata on Distributed Alphabets. In: Modern Applications of Automata Theory. pp. 257–288. Co-Published with Indian Institute of Science (IISc), Bangalore, India (2012). doi:10.1142/9789814271059_0009
23. Sandholm, A.B., Schwartzbach, M.I.: Distributed Safety Controllers for Web Services. BRICS Rep. Ser. **4** (47), (1997). doi:10.7146/brics.v4i47.19268
24. Brim, L., Černá, I., Moravec, P., Šimša, J.: How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. Electron. Notes Theor. Comput. Sci. **135** (2), 3–18 (2006). doi:10.1016/j.entcs.2005.10.015
25. Bollig, B., Leucker, M.: Message-Passing Automata Are Expressively Equivalent to EMSO Logic. In: 15th International Conference CONCUR 2004 - Concurrency Theory, London, UK, 31 Aug. - 3 Sept. 2004. pp. 146–160. Springer, Berlin Heidelberg (2004). doi:10.1007/978-3-540-28644-8_10
26. Bollig, B., Leucker, M.: A Hierarchy of Implementable MSC Languages. In: Formal Techniques for Networked and Distributed Systems - FORTE 2005, Taipei, Taiwan, 2-5 Oct. 2005. pp. 53–67. Springer, Berlin Heidelberg (2005). doi:10.1007/11562436_6
27. Balan, M.S.: Serializing the Parallelism in Parallel Communicating Pushdown Automata Systems. Electron. Proc. Theor. Comput. Sci. **3**, 59–68 (2009). doi: 10.4204/EPTCS.3.5
28. Enea, C., Habermehl, P., Inverso, O., Parlato, G.: On the Path-Width of Integer Linear Programming. Electron. Proc. Theor. Comput. Sci. **161**, 74–87 (2014). doi:10.4204/EPTCS.161.9
29. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '11, Austin, TX, 26-28 Jan. 2011. pp. 283–294. ACM Press, New York, NY (2011). . doi:10.1016/j.entcs.2005.03.032.
30. Huguet, S., Petit, A.: Modular constructions of distributing automata. In: Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, 28 Aug. - 1 Sept. 1995. pp. 467–478. Springer, Berlin Heidelberg (1995). doi:10.1007/3-540-60246-1_152
31. Petit, A.: Recognizable trace languages, distributed automata and the distribution problem. Acta Inform. **30** (1), 89–101 (1993). doi:10.1007/BF01200264
32. Krithivasan, K., Ramanujan, A.: On The Power of Distributed Bottom-up Tree Automata. Int. J. Adv. Comput. Sci. **3** (04), 184–190 (2013). Available online: http://worldcomp-proceedings.com/proc/p2011/FCS2998.pdf (accessed 6.12.2019)
33. Gros-Desormeaux, H., Fouchal, H., Hunel, P.: A Comparison of Distributed Test Generation Techniques.

In: Ben Ayed, R. and Djemame, K. (eds.) Second international conference on Verification and Evaluation of Computer and Communication Systems VECoS'08, Leeds, UK , 2-3 July 2008. pp. 38–49. British Computer Society, Swinton, UK (2008). Available online: https://www.researchgate.net/publication/254361167_A_Comparison_of_Distributed_Test_Generation_Techniques (accessed 6.12.2019)

34. Dick, G., Yao, X.: Model representation and cooperative coevolution for finite-state machine evolution. In: 2014 IEEE Congress on Evolutionary Computation (CEC), Beijing, China, 6-11 July 2014. pp. 2700–2707. IEEE, New York, NY (2014). doi:10.1109/ CEC.2014.6900622

35. Olson, A.G., Evans, B.L.: Deadlock Detection For Distributed Process Networks. In: ICASSP '05. IEEE International Conference on Acoustics, Speech, and Signal Processing, Philadelphia, PA, 18-23 March 2005, Vol.V. pp. 73–76. IEEE, New York, NY (2005). doi:10.1109/ICASSP.2005.1416243