Article

Modelling Distributed Systems in Distributed Autonomous and Asynchronous Automata (DA³)

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 (DA³) for easy graphical modeling in terms of system components. In addition, it is possible to simulate a verified system on distributed components in DA³. The automata have two forms: Server-DA³ (S-DA³) for the server view and Agent-DA³ (A-DA³) for the agent view. DA³ 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 – DA³), 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 DA³ – D-triple-A or DA-cubed). There are two versions of DA³, following the communication duality, Server DA³ (S-DA³) and Agent DA³ (A-DA³). 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 DA³ are defined in Section 3. Also, the differences between DA³ and other notions of Distributed Automata in the literature are discussed. The examples of Dedan operation on DA³ are described in Section 4. The equivalence between IMDS and both versions of DA³ 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:
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 $A=\{a_1,a_2,...,a_m\}$. In an action $\lambda \in \Lambda$, $\lambda=\{(m,p),(m',p')\}$, input and output state belong to the same server $m,m' \in M_i$, while input and output message belong to the same agent $p,p' \in P_i$. A message is sent to a specific server to invoke its service, which is modelled by a function $\text{target} : M \rightarrow S$. In every pair $(m,p)$ which is the input of an action $\lambda$, the server component must match: $\text{target}(m) = s$, $p \in P_i$, $i = j$.

Agents may be infinite or they may terminate in special actions of the form $\lambda = \{(m,p),(p')\}$, 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 $\text{input}(\lambda)$ of an action $\lambda = \{(m,p),(m',p')\}$ contains $m$ and $p$ of its input pair $(m,p)$ and the output configuration $\text{output}(\lambda)$ contains $m'$ and $p'$ of its output pair $(m',p')$.

$$\text{LTS} = (Q, q_0, W)$$

where:

- $Q = \{T_0, T_1, \ldots\}$ (nodes)
- $q_0 = T_0$ (initial node)
- $W = \{ (T, \lambda, T') \mid \lambda \in \Lambda, T=\text{input}(\lambda), T'=\text{output}(\lambda) \}$ (transitions)

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).

$$B_i = \{ \lambda \in \Lambda \mid \lambda = \{(m,p),(m',p')\} \lor \lambda = \{(m,p),(p')\}, p,p' \in P_i \} \quad (2.3)$$

$$C_i = \{ \lambda \in \Lambda \mid \lambda = \{(m,p),(m',p')\} \lor \lambda = \{(m,p),(p')\}, m,m' \in M_i \} \quad (2.3)$$

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

$$B = \{ B_i \mid i = 1...n \} \quad (2.4)$$

$$C = \{ C_i \mid j = 1...k \} \quad (2.4)$$

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 $\lambda = \{(m,p),(m',p')\}$ 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 service invoked by the message (a server may offer a number of services, for example $\text{wait}$ and $\text{signal}$ on a semaphore), $a \in A, s \in S, r \in R$. An action $\lambda = \{(a,s,r),(s,v),(a,s',r'),(s,v')\}$ has the form $\{a.s.r, s.v\} \rightarrow \{a.s'.r', s.v'\}$ 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.
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 (DA³)

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 DA³ 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

![Figure 1. A fragment of LTS for the buffer system.](image-url)
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 - DA³ (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 DA³ were developed – Server-DA³ and Agent-DA³ (S-DA³ and A-DA³).

4.1. Server automata S-DA³

An IMDS system in the server view may be shown as a set of communicating automata S-DA³ (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 \( s \) are the relation in \( P_1 \times M \times P_1 \); of course \( m \) fulfil target\((m) = s \); note that "traditional" distinction between transition relation \( P_1 \times M \times P_1 \) and output function \( (P_1 \times M \times P_1) \to M \) is not held, because in the set of actions can contain nondeterministic actions \( \lambda = ((m,p),(m',p')) \), \( \lambda = ((m,p),(m,p')) \), \( m\neq m' \). 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 \( Z \) (reflected \( S \)) of \( n \) distributed sever automata
\[ \mathcal{Z} = \{ z_i \mid i = 1 \ldots n \} \], where \( n \) is the number of servers in the set \( S \). In the definition, \( \exp (arg) \) is used for powerset \( \mathcal{P} \). An \( i \)th distributed server automaton is \( \mathcal{A}_i = (s_i, P_i, p_{0i}, F_i, X_i, X_{0i}) \), where:

- \( s_i \in S \) – \( i \)th server,
- \( P_i \) – set of nodes, states of \( s_i \),
- \( p_{0i} \in P_i \) – initial node,
- \( F_i = \{ (p_i, m/m', p_2) \mid \lambda = (m, p_1), (m', p_2) \mid p_i, p_2 \in P, \exists a_j \in A, m, m' \in M_j \} \) – set of transitions (for ordinary action and agent-terminating action, respectively),
- \( X_i \in \exp (\{ m \mid \text{target}(m) = s_i \}) \) – 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,
- \( X_{0i} \in \exp (\{ m \mid \text{target}(m) = s_i, \ m \in M_0 \}) \) – initial input set.

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

- \( \forall m_1 \in X_i, m_2 \in X_j, m_1 \neq m_2 \Rightarrow k \neq l \): for any agent at most one message may exists in the global configuration,
- \( \forall (p_i, m, m' \in P_i) \exists m' \in F_i \): 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](image_url)

**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** \( \{ \mathcal{T}_\mathcal{Z}, \mathcal{T}_\mathcal{Z}, \text{next}_\mathcal{Z} \} \), where \( \{ \mathcal{T}_\mathcal{Z} \} \) is a set of **global nodes**, \( \mathcal{T}_\mathcal{Z} \) is an **initial global node** and \( \text{next}_\mathcal{Z} \) is a **transition relation**, defined as follows:

- The **global node of** \( \mathcal{Z} \) is \( \mathcal{T}_\mathcal{Z} = (p_1, X_1), (p_2, X_2), \ldots, (p_n, X_n) \) (current states and input sets of pending messages of all servers).
If in $T_2$ there exists an $x_i$ in which $(p_0, X_i), (p_3, m/m', p_2) \in E_i$, $m \in X_i$, $target(m') = s_i$ then a possible next global node $T'_2$ is:

$$T'_2 = \left\{ \left. \left( p_k, X_k' \right) \right| \begin{array}{l}
p'_k = p_k \quad \text{for} \quad k \neq i \\
p'_k = p_{i2} \quad \text{for} \quad k = i
\end{array} \right\} \left\{ \begin{array}{l}
X'_k = X_k \setminus \{m\} \quad \text{for} \quad i = k = j \\
X'_k = X_k \cup \{m\} \quad \text{for} \quad k = i \neq j \\
X'_k = X_k \quad \text{for} \quad k = j \neq i \\
X'_k = X_k \quad \text{for} \quad i \neq k \neq j
\right\}, \quad k = 1 \ldots n$$

(4.1)

(the automaton $x_i$ 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 $x_i$ the continuation message $m'$ is inserted into the input set $X_i$ of the automaton $x_i$ 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 $T_2$ there exists an $x_i$ in which $(p_1, X_i), (p_3, m, p_2) \in E_i$, $m \in X_i$ (message $m$ terminates the agent) then a possible next global node is:

$$T'_2 = \left\{ \left. \left( p_k, X_k' \right) \right| \begin{array}{l}
p'_k = p_k \quad \text{for} \quad k \neq i \\
p'_k = p_{i2} \quad \text{for} \quad k = i
\end{array} \right\} \left\{ \begin{array}{l}
X'_k = X_k \setminus \{m\} \quad \text{for} \quad k = i \neq i \\
X'_k = X_k \quad \text{for} \quad k = j \neq i
\right\}, \quad k = 1 \ldots n$$

(4.2)

(the automaton $x_i$ 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 $x_i$ all other input sets remain unchanged).

- The initial global node is $T_{0_\emptyset} = ((p_0, X_0), (p_3, X_0), \ldots, (p_9, X_0))$.
- For given global node $T_{z_\emptyset}$ transition relation $nextT_{z_\emptyset}(T_{z_\emptyset})$ is a set of pairs $(T_{z_\emptyset} T'_{z_\emptyset})$. The transition relation $nextT_2 = \cup_{z_\emptyset} nextT_{z_\emptyset}(T_{z_\emptyset})$. If for $T_2$ there exist multiple possible next global nodes, one of them is chosen in nondeterministic way.

A global graph of $z$ cooperation may be elaborated in such a way that nodes are global nodes $T_2$, and edges are transitions in automata $x_i$. 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](image-url)  
**Figure 3.** Fragment of global graph of server automata; first row contains states of the servers, second row input sets.
The initial states 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:

\[
\text{Sprod}(A\text{prod},buf), \text{buf}(A\text{prod},A\text{cons},S\text{prod},S\text{cons}), \text{Scons}(A\text{cons},buf)
\]

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

\[
\begin{align*}
X_{\text{buf}} &\in \exp(\{(\text{Aprod, buf, put})(\text{Acons, buf, get})\}). \\
X_{\text{Sprod}} &\in \exp(\{(\text{Aprod, Sprod, doSth})(\text{Aprod, Sprod, ok}_\text{put})\}). \\
X_{\text{Scons}} &\in \exp(\{(\text{Acons, Scons, doSth})(\text{Acons, Scons, ok}_\text{get})\}).
\end{align*}
\]

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

The initial input sets are:

\[
\begin{align*}
X_{\text{buf}} &= \emptyset, \\
X_{\text{Sprod}} &= \{(\text{Aprod, Sprod, doSth})\}, \\
X_{\text{Scons}} &= \{(\text{Acons, Scons, doSth})\}.
\end{align*}
\]

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 are 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 \(\lambda=((m,p),(m',p'))\) 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: \(\lambda=((m,p),(m',p'))\), \(\lambda=((m,p),(m',p'),p\exp)\).
- For an agent-terminating action \(\lambda=((m,p),(p'))\), 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 \(\mathcal{U}\) (A upside down, rounded to distinguish it from the
general quantifier) of k distributed agent automata $\mathcal{U} = \{ u_i \mid i = 1...k \}$, 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 $i^{th}$ distributed agent automaton is $u = (a, M, m_0, t_0, G_i, Y, Y_0)$, where:

- $a_i$ - $i^{th}$ agent,
- $M \cup \{ t_0 \}$ - set of nodes, $t_0$ is the destination node if the agent $u_i$ terminates (it appears in terminating transition),
- $m_0 \in M$ - initial node,
- $G_i = (\{ m_i, p_1, p' \} \text{ for } \lambda = ((m_1, p), (m_2, p')) \text{ or } (m_1, p/p', t_0) \text{ for } \lambda = ((m_1, p), (p')) \mid m, m_0 \in M, \exists \in p, p' \in P \}$ - set of transitions,
- $Y = [p_1, ..., p_n] \mid p \in P$ - global input vector (common for all $a_i$ in the system); $Y$ is a vector of variables, $Y/i$ is the $i^{th}$ position of $Y$, every variable $Y/i$ has a range over a set of states of server $s_0$; an action changes value of the variable at the position of its server: accordingly to rules for semantics below,
- $Y_0 = [p_0, ..., p_n] \mid p \in P_0$ - initial global input vector, consisting of initial states of all servers.

![Agent automata for buffer system](image)

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

The semantics of $\mathcal{U}$ is defined as **global node space** ($\{ T_u \}, T_{i0}, \text{next} \mathcal{U}$), where $\{ T_u \}$ is a set of global nodes, $T_{i0}$ is initial global node and next$\mathcal{U}$ is a transition relation, defined as follows:

- The global node of $\mathcal{U}$ is $T_u = [m_i, ..., m_i, Y], m_i \in M \cup \{ t_0 \}$. If in $T_u$ there exists $m_i$ for which there exists $(m_i, p/p', m) \in G_i, p, p' \in P_1$ (message $m_i$ causes a change of a state from $p$ to $p'$ in a server $s_0$, target$(m) = s_0$, and a message $m_i$ to a server $s_0$, target$(m) = s_0$ is issued) then a possible next global node is:

$$
T_u' : \forall m \in T_u \{ \begin{align*}
  m = m_i \land m = m_j & \Rightarrow m_i \in T_u' \\
  m = m_i \land m \neq m_j & \Rightarrow m_j \in T_u' \\
  m \neq m_i & \Rightarrow m_j \in T_u'
\end{align*} \} \text{ } \text{ } (4.3)
$$

- $Y' = [p_1, ..., p_n] \mid p_k = \begin{align*}
  Y/k & \text{ for } k \neq x \\
  p' & \text{ for } k = x
\end{align*} \} , \text{ } k = 1...n$
(the automaton $\varepsilon_i$ changes its node to $m_i$, all other automata preserve their nodes; the state $p$ in input vector $Y$ is replaced by $p'$ in the position of the server $s_i$, appointed by $p$ and $p'$, all other elements of the vector $Y$ remain unchanged).

- If in $\{ T_U \}$ there exists $m_i$, for which there exists $(m,p/p',t_a)\in G_i$, $p,p'\in P_s$ (message $m_i$ is a last message in the run of agent $s_i$, then the agent terminates, a sever $s_i$ appointed by the message $m_i$, target($m_i$) = $s_i$ changes its state from $p$ to $p'$) then a possible next global node is:

$$T_U' : \forall m\in T_U \begin{cases} m = m_i \Rightarrow m_i \notin T_U' \\ m \neq m_i \Rightarrow m_i \in T_U' \end{cases} ;$$

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

(the automaton $\varepsilon_i$ changes its node to $t_a$, all other automata preserve their nodes; the state $p$ in $Y$ is replaced by $p'$ as above). The initial global node $T_{i,0} = (m_{i0}, ..., m_{in}, Y_0)$. For given global node $T_U$, transition relation $nextT_U(T_U)$ is a set of pairs ($T_U, T_U' \varepsilon$). The transition relation $nextT_U = \cup_{T_U} nextT_U(T_U)$. In given global node $T_U$, 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:

$$Y = [(\text{Sprod}, \text{value} \in \{\text{neutral}, \text{prod}\}), (\text{buf}, \text{value} \in \{\text{no}\_\text{elem}, \text{elem}\}),$$

$$(\text{Scons}, \text{value} \in \{\text{neutral}, \text{cons}\})].$$

The initial content of the global input vector is:

$$Y_0 = [(\text{Sprod}, \text{neutral}), (\text{buf}, \text{no}\_\text{elem}), (\text{Scons}, \text{neutral})].$$

A global graph of $\Upsilon$ cooperation may be elaborated analogously to the graph of $\Sigma$: nodes of the global graph are global nodes $T_U$, and edges are transitions in automata $\varepsilon$. This graph is analogous to the global graph of S-DA$^3$ 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-DA$^3$ and A-DA$^3$. 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-DA$^3$, nodes and elements of input vector in A-DA$^3$, except terminating nodes. Furthermore, the transitions should join corresponding nodes in all LTS’es. To show the equivalence, the rule of obtaining $T_{\text{in}}(\lambda)$ from $T_{\text{out}}(\lambda)$ for the action $\lambda$ is needed. This rule comes directly from the definition of IMDS:

$$\forall \lambda \in \Lambda \exists (m,p),(m',p') \in T_{\text{out}}(\lambda) \cup \{m,p\} \Rightarrow T_{\text{out}}(\lambda) = T_{\text{out}}(\lambda) \setminus \{m,p\} \cup \{m',p'\}$$

$$\forall \lambda \in \Lambda \exists (m,p),(p') \in T_{\text{in}}(\lambda) \cup \{m,p\} \Rightarrow T_{\text{in}}(\lambda) = T_{\text{in}}(\lambda) \setminus \{m,p\} \cup \{p'\}$$

1. LTS node

- IMDS: $T = \{p,m \mid p \in P, m \in M\}$. Every $p$ from a different $s$ every $m$ from a different a except terminated $a$.

- S-DA$^3$: $T_2 = ((p_1,X_1), ..., (p_n,X_n))$. Every $p$ from a different $s$ – from definition in $T_2$ every $s$ participates (will be shown later). For every pair $X_i, X_j$ 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**: \( T_\ell = \{ m, m', m'', \ldots, Y \} \) Every \( m \) appoints different \( a \) – from definition in \( T_\ell \) every \( a \) participates, except terminated ones (will be shown later). For every pair \( m_i, m_2 \) 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**: \( T_{\sigma} = \{ p_0, m_0 | p_0 \in P_{\vec{m}}, m_0 \in M_{\vec{m}} \} \),
- **S-DA**: \( T_{\sigma} = \{ (p_0, X_0), \ldots, (p_{n_0}, X_n) \} \) Every \( p \) from a different \( s \) – from definition. Every \( s \) participates – from definition. For every pair \( X_0, X_n \) both cannot contain \( m \) appointing the same \( a \) (from definition, as \( X_0 \) contains \( m_0 \) for \( a \) starting from the server \( s \), and \( \forall a \in A \) \( card(M_s \cap M_m) = 1 \) (the initial set of messages contains exactly one message for every agent). Every \( a \) has its \( m_0 \) in some \( X_0 \) – this from which the agents starts (\( X_0 \) are indexed by servers and every \( m_0 \) belongs to some \( M_i \)).
- **A-DA**: \( T_\ell = \{ m_0, \ldots, m_8, Y \} \) In every pair \( m_i, m_\ell \) both cannot appoint the same \( a \) (from definition, as \( \forall a \in A \) \( card(M_s \cap M_m) = 1 \) (the initial set of messages contains exactly one message for every agent). No \( m \) can be \( t_\ell \) – from definition. Every element of \( Y \) appoints different server – from definition.

3. Transition in LTS (regular)

- **IMDS**: \( (T_{in}(A), A, T_{out}(A)) | A \in A, A = (m, p), (m', p') \) If the message \( S \) is replaced by \( T_m \) (in the automaton \( \mathcal{A} \) of the server \( s \) appointed by \( p \) to the state \( p' \), retrieving the message \( m \) from \( X \) and inserting a message \( m' \) to \( X \) appointed by \( m' \). Both messages \( m, m' \) belong to the agent \( a \). If \( T_\ell \) corresponds to \( T_{in}(A) \), then according to (4.1) in \( T_\ell \) \( p \) is replaced by \( p' \) and in the union of all \( X \) \( m \) is replaced by \( m' \), and all other states and messages are equal in \( T_\ell \) and \( T_\ell ' \), which fulfills the rule of obtaining \( T_{out}(A) \) from \( T_{in}(A) \).

For every regular \( A = (m, p), (m', p') \) having \( p \) on input such a transition \( (p, m/m', p') \) exists in automaton \( s \) appointed by \( p \), and no other than for such a transition exists, so it exactly corresponds to the set of regular actions having \( p \) on input.

In every regular transition in \( v \), 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**: From \( T_\ell \) there exists a regular transition \( (m, p/p', m') \) to \( T_\ell ' \) corresponding to \( A = (m, p), (m', p') \), in the automaton \( s \) of the agent \( a \) 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 \( T_\ell \) corresponds to \( T_{in}(A) \), then according to (4.3) in \( T_\ell ' \) \( m \) is replaced by \( m' \) and \( p \) in \( Y \) is replaced by \( p' \), and all other states and messages are equal in \( T_\ell \) and \( T_\ell ' \), which fulfills the rule of obtaining \( T_{out}(A) \) from \( T_{in}(A) \).

For every regular \( A = (m, p), (m', p') \) having \( m \) on input such a transition \( (m, p/p', m') \) exists in automaton \( v \) appointed by \( m \), and no other than for such \( A \) transition exists, so it exactly corresponds the set of regular actions having \( m \) on input.

In every regular transition in \( v \), 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**: \( (T_{in}(A), A, T_{out}(A)) | A \in A, A = (m, p), (p', p) \) From \( T_\ell \) there exists a terminating transition \( (p, m/p', m') \) to \( T_\ell ' \) corresponding to \( A = (m, p), (p', p') \), in the automaton \( \mathcal{A} \) of the server \( s \) appointed by \( p \) to the state \( p' \), retrieving the message \( m \) from \( X \). The message \( m \) belongs to an agent \( a \).

If \( T_\ell \) corresponds to \( T_{in}(A) \), then according to (4.2) in \( T_\ell ' \) \( p \) is replaced by \( p' \) and \( m \) is extracted, and all other states and messages are equal in \( T_\ell \) and \( T_\ell ' \), which fulfills the rule
of obtaining $T_{\text{out}}(\lambda)$ from $T_{\text{inp}}(\lambda)$ (terminating action).
For every terminating $\lambda = ((m,p),(p'))$ having $p$ on input such a transition $(p,m,p')$ exists in automaton $s$ appointed by $p$, and no other than for such $\lambda$ transition exists, so it exactly corresponds the set of terminating actions having $p$ on input.
In every terminating transition in $s$, the set of servers is preserved (as $p$ and $p'$ appoint the same server) and the set of agents in $T_{\sigma}$ is smaller by the agent appointed by $m$. Consequently, there is no way to reestablish a terminated agent.

A-DA$^3$: From $T_{\text{U}}$ there exists an agent $a$ terminating transition $(m,p/p',t_\alpha)$ to $T'_{\text{U}}$ corresponding to $\lambda = ((m,p),(p'))$, in the automaton $w$ of the agent $a_i$ appointed by $m$, to the message $m'$, replacing the message $m$ by $t_\alpha$.

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

5. Semantics of LTS
- IMDS: Interleaving, nondeterministic,
- S-DA$^3$: Interleaving, nondeterministic,

6. Conclusions and further work

Figure 5. Simulation over SDA$^3$ 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-DA$^3$ or in A-DA$^3$ 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.

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

The next steps are:
- Timed DA³ 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 DA³ 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 DA³: Automatic Vehicle Guidance System**

![Diagram of road segment controllers](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.
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.

```plaintext
1. #DEFINE N 2
2. server: mE (agents AMP[N]; servers mM,lotE),
3. //Edge Road Marker 4. ...
19. server: mM (agents AMP[N]; servers mM[2],lotM),
20. //Middle Road Marker
21. services {tryE[2],tryL[2],okE[2],notE[2],okL[2], takeE[2],takeL[2],switch[2]},
23. states {free,resE[2],resL[2],occ},
24. actions {
25. //going to ME1 or ME2
26. <i=1..N><j=1..2> {AMP[i].mM.tryE[j], mM.free} -> {AMP[i].mE[j].okM[j], mM.resE[j]},
27. <i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.resE[j]} -> {AMP[i].mM.switch[3-j], mM.occ},
28. <i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.occ} -> {AMP[i].mE[j].tryM[j], mM.resE[j]},
29. <i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.occ} -> {AMP[i].mE[j].takeM, mM.free},
30. //on a way to ME1 or ME2 may go to LE if MEi occupied
31. <i=1..N><j=1..2> {AMP[i].mM.notE[j], mM.occ} -> {AMP[i].lotM.try[j], mM.occ},
32. <i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.occ} -> {AMP[i].lotM.ok[j], mM.occ},
33. <i=1..N><j=1..2> {AMP[i].marker2.okL[j], mM.occ} -> {AMP[i].lotM.take[j], mM.free},
34. //going from PL2 - goes to RM1(mE[1]) or RM3(mE[2])
35. <i=1..N><j=1..2> {AMP[i].mM.tryL[j], mM.free} -> {AMP[i].lotM.ok[j], mM.resL[j]},
36. <i=1..N><j=1..2> {AMP[i].mM.okL[j], mM.resL[j]} -> {AMP[i].mE[j].tryM[j], mM.occ},
37. <i=1..N><j=1..2> {AMP[i].mM.okL[j], mM.occ} -> {AMP[i].mE[j].takeM, mM.free},
38. ...
39. }
40. server: lotE(agents AMP[N];servers mM),
41. //Edge Warehouse Lot
42. ...
50. server: lotM(agents AMP[N];servers mM),
51. //Middle Warehouse Lot
52. ...
59. servers mM[2],mL,lotE[2],lotM;
60. agents AMP[N];
61. init -> {
62. <j=1..2> mE[j](AMP[1..N],mM,lotE[j]).free,
63. mE[1..N],mL[1,2],lotM).free,
64. <j=1..2> lotE[j](AMP[1..N],mE[j]).occ,
65. ...
66. <j=1..2> AMP[j].lotE[j].start,
67. }.
```
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 \texttt{lotE} is shown as S-DA³ automaton in Fig. A2. Note the transition from \texttt{res} to \texttt{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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figureA3}
\caption{Agent automaton of AMP agent type.}
\end{figure}

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\} \rightarrow \{AMP.mE[1].okL, lotE[1].res\},
4. \{AMP.lotE[1].ok, lotE[1].occ\} \rightarrow \{AMP.mE[1].takeL, lotE[1].free\},
5. \{AMP.lotE[1].start, lotE[1].occ\} \rightarrow \{AMP.mE[1].tryL, lotE[1].free\},
6. \{AMP.lotE[1].take, lotE[1].res\} \rightarrow \{lotE[1].occ\},
Appendix B - Using DA³ 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 \( \lambda = \{(\text{state}, \text{message}), (\text{state}', \text{message}')\} \) is exotic for the users. Therefore, an alternative input form of DA³ 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-DA³, 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


