Software and Systems Modeling
https://doi.org/10.1007/s10270-021-00917-7

REGULAR PAPER

Graphic modeling in Distributed Autonomous and Asynchronous Automata (DA^3)

Wiktor B. Daszczuk[1]

Received: 23 April 2020 / Revised: 11 July 2021 / Accepted: 20 July 2021

Abstract
Automated verification of distributed systems becomes very important in distributed computing. The graphical insight into the system in the early and late stages of the project is essential. In the design phase, the visual input helps to articulate the collaborative distributed components clearly. The formal verification gives evidence of correctness or malfunction, but in the latter case, graphical simulation of counterexample helps for better understanding design errors. For these purposes, we invented Distributed Autonomous and Asynchronous Automata (DA^3), which have the same semantics as the formal verification base—Integrated Model of Distributed Systems (IMDS). The IMDS model reflects the natural characteristics of distributed systems: unicasting, locality, autonomy, and asynchrony. Distributed automata have all of these features because they share the same semantics as IMDS. In formalism, the unified system definition has two views: the server view of the cooperating distributed nodes and the agent view of the migrating agents performing distributed computations. The automata have two formally equivalent forms that reflect two views: Server DA^3 for observing servers exchanging messages, and Agent DA^3 for tracking agents, which visit individual servers in their progress of distributed calculations. We present the DA^3 formulation based on the IMDS formalism and their application to design and verify distributed systems in the Dedan environment. DA^3 formalism is compared with other concepts of distributed automata known from the literature.

Keywords Distributed systems · Distributed system modeling · Distributed automata · Graphic 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 formalism reflects the behavior of components of distributed systems: servers in a distributed environment act asynchronously and make decisions autonomously. However, many modeling and verification formalisms provide mechanisms based on simultaneous actions of processes, like synchronous transitions on common symbols in Büchi automata [2] or Timed Automata [3], synchronization on send and receive operations in CSP [4] or Uppaal Timed Automata [5], synchronous operations on complementary input and output ports in CCS [4] or Occam [6]. Several automata-based formalisms (including Büchi automata and Timed Automata) use synchronous coordination. By contrast, IDMS models may be checked asynchronously.

The main feature of our IMDS is providing two views of a distributed system: cooperating servers or migrating agents. The system definition is uniform, but two projections allow for observation and verification of different properties: communication deadlock in the server view, and resource deadlock and distributed termination in the agent view. Mostly they are the same deadlocks but observed from different perspectives [1]. The vital feature of IMDS is finding partial deadlock/termination, in which a subset of processes is involved.

Students and engineers avoid formal methods, especially model checking, because determining the properties of a system in temporal formulas is not an easy task [7]. Avoiding formal methods is also a summary of our several years of observation. In order to persuade users to use formal methods, on the one hand, it is necessary to facilitate the

Communicated by Gordon Blair.

✉ Wiktor B. Daszczuk
wbd@ii.pw.edu.pl

[1] Institute of Computer Science, Warsaw University of Technology, Nowowiejska Str. 15/19, 00-665 Warsaw, Poland

Published online: 01 November 2021

specification of the system to be verified, and on the other hand, to simplify the verification process as much as possible. We achieved the latter by "push the button" verification, as we developed general temporal formulas, independent of the structure of the system under test. These formulas are built into the verifier and invisible to the user, who sees only the result of the verification in the form of meeting certain properties or not. A counterexample or a witness allows seeing the sequence of system behavior leading to the given result.

However, the verification itself must be preceded by a formal system specification, which is not an easy task. Ideally, we can describe the system components, and the verifier will automatically build a global model. These are the reasons for creating, for example, temporal verification systems for source languages, such as Bandera [8]. Likewise, in our Dedan environment, the imperative higher-level programming language Rybu [9] has been developed, however, it is beyond the scope of this article.

Also, the observation of the results and verification in the form of sequence diagrams is not an easy matter in the case of large systems. The development of an automatic model of system components, the semantics of which corresponds to a formal mathematical model, can facilitate both the specification of the system as a graphical input of the verifier and the observation of the verification result as a simulation of a counterexample over a graphical representation in the form of automata.

In this article, we describe just such an automata-based model, which is a graphical counterpart of the formal IMDS model. Together with automatic verification, our distributed automata make it easy for students and other users to enter easily and apply formal verification of distributed systems. Just as IMDS allows the system to be projected onto cooperating nodes or migrating agents, likewise distributed automata in graphical representation take the form of server automata or agent automata. The designer can choose between these two forms of specification and observation, and the verifier allows for easy switching between server and agent views, even with a partially completed model.

The contribution of this article is the introduction of DAAA—Distributed, Asynchronous and Autonomous Automata for modeling distributed systems (shortened to DA^3—D-triple-A or DA-cubed). Their goal is to define distributed systems graphically and to simulate modeled systems graphically. An essential function of the simulation is to show changes in system components according to the verification counterexample. There are two versions of DA^3 that reflect the communication duality: Server DA^3 (S-DA^3) and Agent DA^3 (A-DA^3). Both have the semantics equivalent to the IMDS specification, and therefore to each other. These two forms reflect two ways of describing distributed systems: as cooperating servers in the Client–Server-like specification or as traveling agents in the Remote Procedure Call (RPC) view [10]. They also reflect the fundamental characteristics of distribution: unicast communication, locality of execution, autonomy of decisions, and asynchrony of operation and communication.

First, we introduce the idea of DA^3. Figure 1 presents an informal view of a distributed system `buffer`. This figure enumerates the elements of the example system, which are introduced formally later in the article. The system consists of 1-element buffer `buf`, and two users—producer `Sprod` and consumer `Scons`, so there are three *servers* presented in red. Each server can be in one of the *states* shown as red ovals; the *initial states* have a double border. Servers offer *services* presented by green arrows. These services are invoked by *messages* sent in the context of distributed computations called *agents*. The agents `Aprod` and `Acons` are initiated on the `Sprod` and `Scons` servers, respectively, and they migrate in the system to the server `buf` and back using messages identified with servers' services. Here we do not distinguish between the concepts of service and messages, this distinction will be described in Sect. 5. Besides,

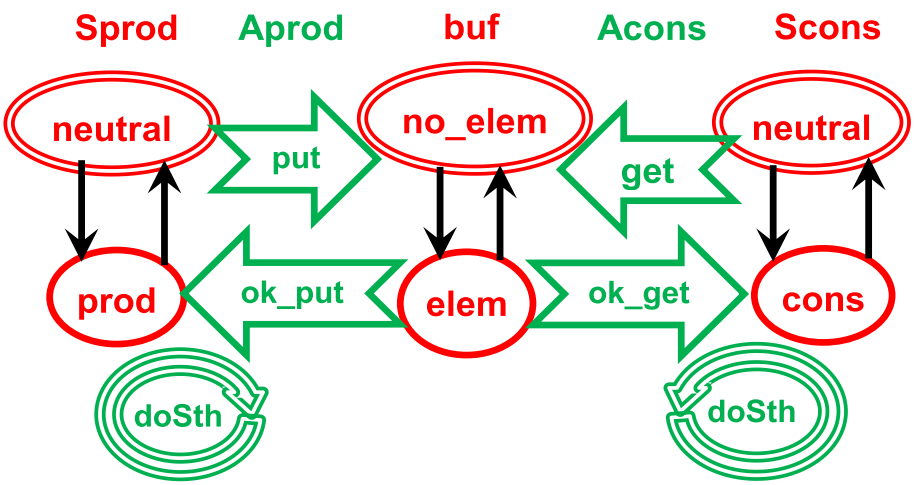

Fig. 1 The example 1-element buffer system

the agents call from their home servers `doSth` services on the same servers, which models performing of activities other than operations on `buf`. Those messages are shown as Looping arrows. Initial messages (just `doSth`) have a double border.

The system evolves by performing an action: each action takes a message in a given server state and creates a new server state and a new agent message that invokes the next action on some server.

The automata are graphical Mealy-style [11] projections of the system onto distributed components, i.e., servers or agents. Server automata S-DA^3 are presented in Fig. 2. The automata headings are the server names in rounded red boxes. The nodes of individual automata (we do not call them "states" to avoid ambiguity) are placed vertically under the automata headings. Automata *nodes* are server states. In the full notation provided in Sect. 4, they have the form (*server, value*), here shortened to the *value* only, because the server name is the automaton heading. Actions are modeled as the transitions from a node to another node, having agent messages being input and output symbols. Messages are triples (*agent, server, service*) made up of the service name, and the server and agent names that match the action context. Service names, being part of message triples, are underlined. They are introduced in the informal Fig. 1. The automata are equipped with input sets of pending messages that are in no way organized into data structures such as vectors, stacks, or queues. The sets are shown as bags under the automata, with their initial contents. The view of these S-DA^3 automata in the Dedan program is presented later in Fig. 6.

The same system projected onto agents is presented in Fig. 3. In this view, everything is dual: the automata nodes are the agents' messages, while the servers' states occur as the transition input/output symbols. The automata headings are the agent names in rounded green boxes. The nodes of individual automata are placed vertically under the automata headings. States (*server, value*) consist of the server name and the state value. Message triples in the full notation provided in Sect. 4 are (*agent, server, service*), here shortened to the pairs (*server, service*), because agent name is the automaton heading. States are pairs (*server, value*) made up of the server name and state value. State value names, being part of state pairs, are underlined. They are introduced in the informal Fig. 1. The states are input and output symbols on the automata transitions. There is also a global vector of current server states, shown below the automata. The view of the A-DA^3 automata in the Dedan program is presented later in Fig. 7.

The basic formalism for specifying and verifying distributed systems is IMDS, rather than the DA^3. Automata facilitate the specification of the graphics system 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 track the steps of the counterexample obtained from model checking. The source IMDS code of a distributed system has semantics in the form of labeled transition system (LTS). A fragment of the LTS of the example system is presented in Fig. 4. We will show that the LTS-es produced by both forms of DA^3 define the same semantics.

Our approach is not completely different from the other proposed automata models. The main difference is in the combination of the automata-based and the process-based model, the two formalisms with equivalent semantics but different

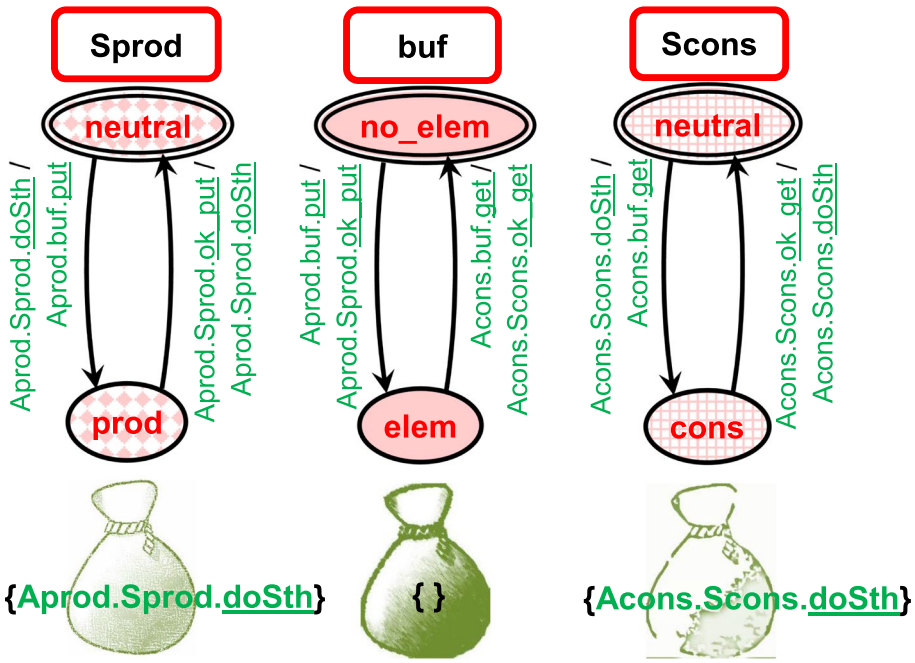

Fig. 2 Server automata for the *buffer* system

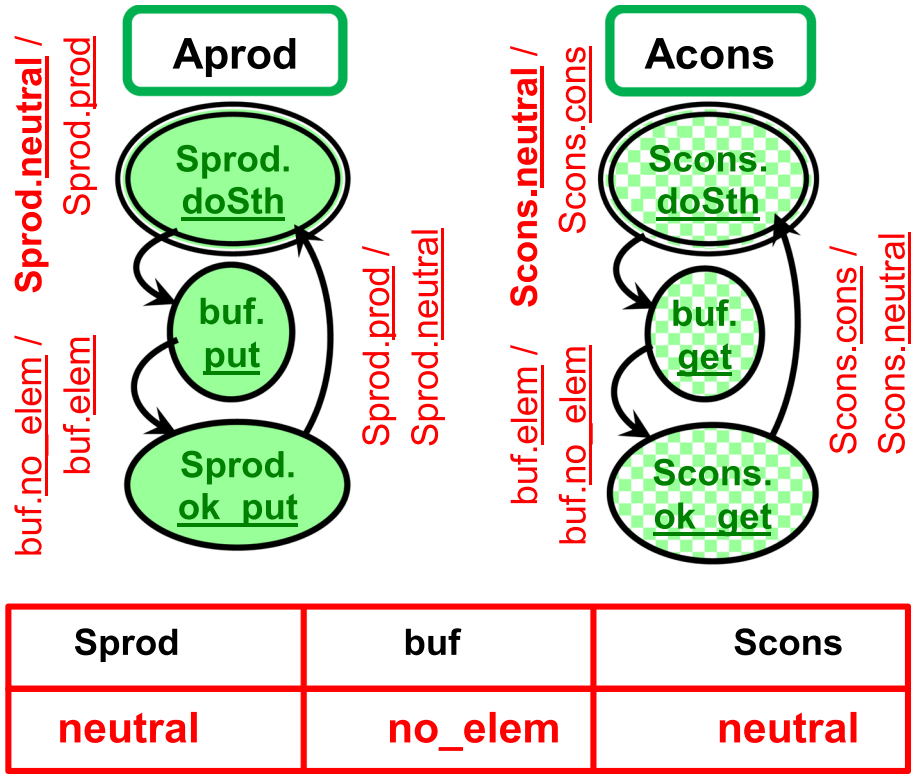

Fig. 3 Agent automata for the *buffer* system

in the way they express the behavior of servers and migrant agents. The textual process model is ideal for studying the characteristics of specific systems using temporal logic and other algebraic methods. On the other hand, the automata are closer to the designer's intuition and are perfectly suited to the specification of the system, its simulation or the observation of the course of counterexamples obtained from temporal verification. The duality of the automata model is important in this context, as it emphasizes the cooperation of distributed servers, or the migration of agents traveling between nodes. It is an extension of Lauer Needham's concurrency dualism postulate [12] to distributed environments.

IMDS as a broad theoretical base has been discussed in several articles, including [1]. The present article is the continuation of the work presented in [13], in which the informal background of DA^3 was given. The novelty of the approach presented in the present article is:

- More formal specification, as the construction of the LTS for a set of automata was not defined yet;
- On the base of the DA^3 LTS, and the IMDS LTS the specification, the equivalence between IMDS and DA^3 is described;
- The application of DA^3 in modeling and verification of distributed systems was previously presented in brief three paragraphs, now we present some procedures of verification and simulation in the Dedan environment in detail; these procedures evolved during several years of developing and using Dedan, mainly in education;
- Only a tiny model was provided as the subject to IMDS and DA^3 specification, now we discuss a reasonable size system;
- A new feature is added to the Dedan verification environment and described in this article: the simulation of DA^3 run following a counterexample; this is very useful in the verification procedure.

The article is organized as follows. The numerous formalisms known as distributed automata are discussed in Sect. 2. The definition of the basic IMDS formalism is given in Sect. 3. Section 4 provides an IMDS specification of the same *buffer* system as seen above. An informal view of the distributed automata DA^3 is given in Sect. 5, and the definition in two forms S-DA^3 and A-DA^3, in Sect. 6. A practical example of DA^3 application is described in Sect. 7. Section 8 presents an example of how Dedan works when simulating an example from Sect. 4. Also shown are examples of S-DA^3 and A-DA^3 views in the Dedan environment. Conclusions and further work are covered in Sect. 9. The equivalence of basic IMDS and both versions of DA^3 is presented in the Appendix.

2 Related work

Discrete event systems are those whose evolution is triggered by symbols produced and consumed or exchanged with the external world. They are especially useful for

modeling digital distributed systems. Among them, the most common are Petri nets, process algebras, and finite automata [15]. Generally, automata-based specification consists in connecting some *states* by means of *transitions* [16]. Formally, states are valuations of internal variables of a system. The transitions show changes between these states in response to occurring *symbols* (letters of the input alphabet). In practice, these symbols denote external events sent to the automaton, such as signals or messages. Distribution is often modeled as decomposing the global automaton into a set of concurrently behaving component automata, and splitting the set of symbols into subsets, which can overlap. Private symbols mean local actions, while common symbols are treated as communication [17].

An important element of the automaton is the current state, which illustrates the phase in the automaton evolvement. The automaton starts from the initial state. Additionally, some approaches require final states and some do not. In formal linguistics, final states accept input sequences, while in behavioral automata they are treated as termination.

Automata are used to express the behavior of state-based systems that are most commonly used in computer science and technology. The second area of application is formal linguistics, used to check whether a given sequence belongs to a language or to transform an input sequence to an output sequence. Sometimes these two functions are combined, for example in compilers. This article covers the first area that describes the behavior of a system as a state-transition diagram. Whereas actions typically happen on the transitions, in some approaches the states also encapsulate activity, consisting of sequences or parallel combinations of actions. Examples of activity states include the tasks of BPMN (Business Process Model and Notation) [18] and the states of UML (Unified Modeling Language) State Machine Diagrams [19]. An important feature of automata is their descriptive graphic form, friendly to human perception. Therefore, automata are widely used in education, digital circuit design, information systems specification, and other purposes.

Concurrency is a crucial issue in hardware and software design. For this purpose, sets of cooperating automata have been developed. In a system of concurrent automata, their states form a state vector describing the global state. Concurrent automata are used, for example, in hardware design, communication protocol specification, and parallel programs description. Two important considerations are: how the transitions in component automata are combined to global transitions and how the symbols are exchanged. Externals events may come from outside the system and events generated by automata, that are offered to other automata and to the external world. The system is open if events can come from the external world, and closed if they come only from automata.

The semantics of an automaton is defined such that if it remains in a given state and an input letter is proposed that matches a letter on the transition, the automaton changes its state to the target of the transition. If the same letter triggers multiple transitions, or if more than one letter is offered at the same time, the automaton is assumed to be nondeterministic. The semantics of concurrent automata concerns the way of combining the transitions of individual automata: in parallel, interleaved or mixed. Parallel semantics is the simultaneous execution of enabled transitions, interleaving means execution of one transition at a time. In the case of mixed semantics, for example in parallel Büchi automata [2], parallel transitions mean synchronization, while interleaved transitions refer to individual behavior. The communication structure can be designed in various ways, e.g., when signals are received by every automaton, it is assumed to be a kind of global ether through which all distributed components observe symbols. If the automata cooperate in pairs, synchronous or asynchronous communication channels are assumed. Some intermediate communication structures may exist, such as the previously mentioned stacks or queues, which further complicate the semantics and which may introduce additional asynchrony (aspects of asynchronous behavior are discussed below). Even more complicated structure of the multi-access Reo channels is proposed in [20]. Many types of concurrent or parallel automata were elaborated [21–23]. The timed versions are described in [24], and probabilistic versions in [25].

Many works describe formalisms called "distributed automata." There are fundamental differences between parallel but centralized, and distributed automata that model distributed computing. A significant difference between our automata and other presented models is the reflection of the actual features in distributed systems of communicating nodes. The features of these systems are unicasting, autonomy, locality, and asynchrony. It is imperative to fully explain our understanding of these concepts to avoid terminological confusion. Of course, other authors may use a different interpretation of the given terms, and they have every right to do so. However, we will focus on the interpretation that, in our opinion, reflects the specifics of the operation of actual components of existing distributed systems, because we focus on modeling and verification of such systems.

- *addressed communication*—there are different communication methods in distributed systems, and here we have no ambition to cover all possible ways. Systems that disseminate information about events or provision of services may naturally use a broadcasting model of communication (messages are sent to every component node). Systems that consist of replicated server clusters may naturally use a multicasting communication model (messages are targeted at relevant subsets of the component nodes, of which one or another node may choose to respond). Systems that model distributed computing

explicitly are best described using a unicasting model (messages target specific component nodes in the system, where migrating agents may execute sequences of operations on the target server). Examples of the kind of distributed processing environment suitable for unicasting include: object-oriented middleware, in which processes migrate between distributed objects that execute these; or systems of communicating physical agents, such as autonomous vehicles, or collections of communicating nodes in IoT networks, or similarly in web services computing. Each event should be routed precisely to the server automaton rather than available for all components. We deliberately exclude broadcast or multicast-based systems that are modeled for example in CSM (Concurrent State Machines [26]);

- *locality*—in a distributed environment, the components do not have access to the global state. The automaton's knowledge of other automata is limited to the last communication with them. The decision of each automaton is made on the basis of its local state and received events. In a centralized solution, an automaton can observe the global state, i.e., the values of all global items: states of other automata, globally pending symbols and/or global variables, if they are present in the formalism. The automata shown in Fig. 5 , which are components of automata systems (PDA—Pushdown Distributed Automaton, MPA—Message Passing Automaton and DA^3—our distributed automaton), can be assumed distributed when composed with other automata of the same communication scheme, because the automaton and its input structure (stack, queue or set, respectively) provide a local framework for making decisions which transition to execute. We could think of a set of locally pending messages as an extension of the server state, but we think that it is more convenient to extract them as carriers of distributed agents. Whether the server remains in a state or leaves—it depends entirely on its current state and pending messages, it is not affected by any non-local system elements: the states of other servers, the actions they take, or refraining from taking any action;
- *autonomy*—the automaton autonomously decides which event is accepted and causes the transition to be executed. No action or property of the environment can order or prevent a server from taking a certain action. It is modeled as collecting events on the automaton's input, that can trigger transitions outgoing from a given state, each triggered by an input symbol (or in some solutions by a combination of symbols). In nondeterministic models, the same symbol can trigger more than one transition. In some formalisms, the freedom of the automaton is limited, as depicted in Fig. 5 . It shows two concepts of automata that only have access to one of the pending symbols. They are: PDA that gets the symbol on the top of the stack, and MPA that gets the symbol waiting the longest in the input queue. The third type DA^3, the subject of this article, chooses freely from its input set. The random selection of the input message is realized comparably to the guarded select statement. Of course, what actions can be taken under what circumstances depends on the "program" contained on the server, which we model as a set of actions that have the server states and messages directed to this server on their input. It is up to the designer to accept the message and take action (or not accept a certain message in a particular state). Messages are treated the same way: there is no data structure that stores messages and makes them available in a particular way—like a queue or a stack. If the designer wants to accept messages in a specific regime, he or she can program this feature in the server action set. Significantly, delivering new messages to a server can never reduce the set of actions allowed in a given state, it can at most increase that set. Obviously, accepting the message leads the server to a new state in which specific actions may be forbidden. Autonomy is closely related to locality, but in our opinion, it is not conceptually identical with it;
- *asynchrony*—can be considered in three aspects: asynchrony of actions in processes, asynchronous delivery of messages, and whether the communication primitives are blocking or non-blocking.

i. The *asynchrony of actions* consists in the fact that two processes provide in a certain way the input elements necessary for its execution, in our case: the state of the server and the message. There may times when the process itself provides all the elements (in the DA^3 case, it happens when a message is sent to the same server—then the same process provides the state and the message). In this case, both are delivered synchronously, but the synchrony is hidden inside the process. If two processes need to deliver the items at the same time, or if the delivery of the item depends on what is happening in the other process, we will call the action synchronous. This is the idea behind Büchi automata and Timed Automata [27], where two (or more) automata perform a synchronous joint action (although the authors call them asynchronous because the remaining local actions are performed completely independently). The necessary condition for the performance of this joint action is that both automata reach the states preceding these synchronous transitions, which we can interpret precisely as providing the elements necessary for its implementation. If the processes (in our case two, but in other formalisms there may be more of them) provide the elements necessary to perform the action without looking at the other—then the action is asynchronous.

In our case, apart from the situation of sending the message to the same server, in the action λ of the server s, the output message m to is delivered to another server s'—the message m target server. The servers' can already be waiting in the state p' for the message m. In such a case, an action λ' will be initiated on the target server s' (see Sect. 3 for formal details of action as the relation between messages and states). If the message m is not an input pair with the current state p' of the server s', then the message m will wait on the server s' for such a state. The new state p'' of the server s, generated in the action λ, can match one or more messages waiting on the server s, which together with the state p'' will initiate another action λ' on the server s. If the state p'' does not match any of the messages pending at the server s—it will wait for the matching one to appear. On both servers s and s', the actions are initiated asynchronously: the state is waiting for a message, or the message is waiting for the matching state. Of course, this always happens in the interleaved model, which is ours. In some models with coincident semantics, the state and message may appear simultaneously but accidentally. However, this accidental synchrony is not required for the asynchronous execution of actions.

ii. *Communication asynchrony*—the moment of sending the message does not depend in any way on the recipient's readiness to accept it, the recipient's state or actions performed in it. Models such as the coincident, precisely synchronous message delivery in two Büchi automata or Timed Automata are not adequate for distributed processing in our opinion. In the Büchi and TA formalisms, automata communicate by performing a joint transit with the same symbols. This is usually interpreted as sending and receiving a message. In TA implementation of the Uppaal verifier, this common symbol even comes with attributes ! and ?, denoting sending and receiving of the message. This corresponds to actions (which in Büchi and TA are equivalent to communication, but not in our case, because the actions are local) when automata reach certain states at the same time or need to find out what the current node of the other automaton is (i.e., synchrony or non-locality). In fact, in our opinion, there is no other way to know if the other server is willing to accept the message other than sending the message and waiting for a reply. Moreover, receiving a reply informs only the state of the second server at the time of sending the reply, but at the moment of receiving the reply, it is not known what could have happened after sending it. In some models, asynchrony is modeled by a kind of "buffer" separating servers [28]. The process can send to the buffer at any time, and the reception will take place at the right moment. In our opinion, this significantly limits modeling, as such buffers clearly organize messages (typically FIFO), and there is also a problem with determining the buffers capacity (see below). A direct consequence of the asynchrony of communication is the indefinite time of message delivery, which is also a way of defining this asynchrony [29]. On the other hand, in [30] [31], the communication asynchrony is defined as the separation of the operations of sending and receiving messages. In summary, it can be said that in the synchronous model, the automata synchronize to communicate, while in the asynchronous model, they communicate in order to synchronize.

iii. The third aspect is *blocking*: synchronous communication consists in passively waiting for sending/receiving a message when the partner is not ready, i.e., non-blocking communication as opposed to blocking [32, 33]. While sending, the sender waits until the recipient is ready to receive the message, unless it is already ready. When receiving, the recipient waits for the message after initiating the operation, unless it is already ready to receive it. So we refer again to knowing the state of the other server, or the communication takes place at the same time. While from the recipient's point of view, it is possible to imagine blocking communication, from sender's point of view it is unrealistic, for the reasons already discussed. In our model, communication is non-blocking both on the part of the sender's side, i.e., sending the message is sent without knowing the recipient's state, as well as on the receiver's side, because the recipient is always sensitive to receiving new messages from any sender, even if it is waiting for a specific message.The third aspect is *blocking*: synchronous communication consists in passively waiting for sending/receiving a message when the partner is not ready, i.e., non-blocking communication as opposed to blocking [32, 33]. While sending, the sender waits until the recipient is ready to receive the message, unless it is already ready. When receiving, the recipient waits for the message after initiating the operation, unless it is already ready to receive it. So we refer again to knowing the state of the other server, or the communication takes place at the same time. While from the recipient's point of view, it is possible to imagine blocking communication, from sender's point of view it is unrealistic, for the reasons already discussed. In our model, communication is non-blocking both on the part of the sender's side, i.e., sending the message is sent without knowing the recipient's state, as well as on the receiver's side, because the recipient is always sensitive to receiving new messages from any sender, even if it is waiting for a specific message.

All the above features of asynchrony are met by the construction both in the IMDS formalism and in the DA^3 model that is unambiguous with it. Note that a synchronous system can be defined as one in which all distributed components execute in lock-step, using the same central clock, for example CSM [26]. Therefore, an asynchronous system is one in which all distributed components execute following their own local clocks, out of lock-step. In our approach, we do not introduce any notion of clocks, neither central nor local.

Another aspect of distributed computations is *communication duality*—this non-obvious feature applies to two aspects of distributed computations. The famous Lauer-Needham postulate [12] concerns the equivalence of communication through messages and resources (variables). This duality expresses itself in distributed systems as the alternative of Remote Procedure Call versus Client–Server architectures. Thus, the system described in the two paradigms should have two forms of automata: distributed servers communicating via messages and traveling agents communicating by servers' states. This leads to unobvious assumptions about the states of distributed automata and exchanged events: in the view of the servers following the Client–Server architecture, there are natural automata nodes reflecting the server servers, and the transitions are triggered by events—messages exchanged between servers. However, in the view of RPC architecture, in which the computations take the form of agents traveling between servers, the automata nodes should be assigned to the messages by which the agents run, and the events relate to the states of the servers. This turns the description of the distributed system upside down, but after reflection it is as natural as the wave-particle duality in Physic. However, everything is fine if both descriptions—distributed server automata versus traveling agent automata—have the same semantics, described as all possible system executions.

Note that in Fig. 1 , and in all subsequent figures, all elements related to the servers and their states are colored red, while all elements related to agents and their messages are colored green.

In the literature, several types of automata are called distributed (DA).

1. Automata on distributed alphabets, communicating by means of common symbols, based on Zielonka's automata [34]. The automata are called DA in many works concerning the behavior of concurrent systems (sometimes additionally equipped with real-time clocks for time analysis with real-time constraints): [35–37]. Those automata are called asynchronous in [38, 39], although they perform actions (execute the transitions) asynchronously only if the input symbols are different. They make synchronous moves on common input symbols (and it is the only common aspect of separate automata). From our point of view, those automata should be called synchronous. Alur's Timed Automata [3] (sometimes called DA [35]) are very close to Zielonka's automata; they are additionally equipped with time constraints and time invariants. CSP processes are similar, synchronizing on ! and ? operations instead of common alphabet symbols. The advantage of CSP is to specify the direction of communication (! sends, ? receives), which in the case Zielonka's automata should be provided informally. A similar concept is used in modeling distributed hardware systems in [40]. The hierarchical description of distributed symbols is undertaken using the nested version of such automata [41].
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 [2]). They are called DA in [42].
3. Message Passing Automata (MPA, called DA in [43, 44]) are really distributed and asynchronous. They contain ordered sets on symbols waiting for acceptance, called buffers or queues. Such automata using FIFO buffers are called asynchronous DA in [45].
4. Pushdown Distributed Automata (PDA) are equipped with local memories of input symbols (stacks) [46].
5. The two former cases (MPA and PDA) are combined in [47, 48] and called DA.
6. Cellular automata are sometimes called DA [49]. They are synchronous in nature, but their evolution does not depend on the global state. Only the previous state of the cell and its neighbors' states are taken into account. Cellular automata with reduced communication for the purpose of distributed implementation are presented in [50].

Above listed formalisms are referred to as "automata," but there are other models of distributed systems, some of them even closer to some of the requirements presented, for example, synchronous statecharts with sets of input events [51] and asynchronous statecharts with input dispatchers [14]. Communicating stream X-machines [52] with input buffers communicate asynchronously. We can treat any buffered communication as asynchronous because no knowledge about the target component state is required. However, they are formally intractable because of unbounded delays. Moreover, such buffers are unrealistic to implement, and if they are bounded, incoming messages must be blocked or lost in a case of a full buffer. Therefore, some knowledge about the receiver state comes back. In addition, the size of the input buffers is difficult to define, and too small buffers can cause deadlocks [53]. Note that in our formalism, the messages are the carriers of agents, therefore the total number of messages pending on input of servers is limited to the number of agents. Also, the agent is the context of the message, and the message is part of the action definition, so the

maximum number of messages in a given server is limited to the number of agents that can invoke the server's services. Message designates something like "agent state," and therefore there is exactly one system-wide message for each non-terminated agent, so the same agent cannot route multiple messages to a node, for example, messages coming from different servers.

In our opinion, none of the aforementioned formalisms fully meet the requirements set by DA described above: addressed communication, locality, autonomy, asynchrony, and communication duality. We developed Distributed Autonomous, Asynchronous Automata—DA^3 (D-triple-A or DA-cubed, to distinguish them from all the outlined formalisms, all called DA). Our automata reflect the behavior of distributed components. Servers independently make decisions (perform actions) without knowing the state and pending messages in other servers (autonomy), and messages are sent regardless of the states and messages in the target servers (asynchrony). Since there are two distributed system views in IMDS, two forms of DA^3 have been developed—Server-DA^3 and Agent-DA^3 (S-DA^3 and A-DA^3). We present them in Sect. 5 after describing the basic IMDS formalism. Both forms of DA^3 are equivalent to IMDS.

3 Integrated Model of Distributed Systems (IMDS)

We model closed distributed systems, i.e., those in which all activities depend on internal components and are not possible to interact with the outside world. All the signals that cause changes to the system come from its distributed components. If there are user processes that interact with the system, they must be modeled as components inside. No external signal can be observed, but the signals generated in the system can be observed from the outside, informing about system activity.

In IMDS [1], a distributed system is simply a set of *actions*, having pairs of state and message on input and on output. It is based on the observation that the server is activated by an incoming message in a distributed environment. If the message is accepted—which depends on the server's current state—it performs some action. The action executes the specified service and generates the next server state. The messages are the carriers of *agents*. The agent has a mission to fulfill in a distributed environment. This mission is accomplished through distributed computing performed on servers. Therefore, the action on its output generates a new agent message in addition to the new server state. Typically, a new message is sent to continue the agent computation on a different server or sometimes on the same server. Thus, the system actions are the relation Λ on the set P of *servers' states* and the set M of *agents' messages*. Precisely, the action $\lambda \in \Lambda$ connects two pairs: an input pair (*message, state*) and a similar output pair:

$$\Lambda \subset (M \times P) \times (M \times P) \tag{3.1}$$

There are sets of servers $S = \{s_1, s_2, \ldots, s_n\}$ and agents $A = \{a_1, a_2, \ldots, a_k\}$. The set P is split into disjoint subsets attributed to the servers: $P_{s1}, \ldots, P_{sn}$, while the set M is split into disjoint subsets attributed to the agents: $M_{a1}, \ldots, M_{ak}$. The mapping Ms is the same as assigning agent visits to servers; it means that the corresponding agent messages will be seen by the appropriate servers. The action $\lambda \in \Lambda$, $\lambda = ((m, p), (m', p'))$ involves server s_i and agent a_j. The input and output state belong to the same server: $p, p' \in P_i$, and the input and output message belong to the same agent: $m, m' \in M_j$. Each state is attributed to a server, and each message is directed to the specific server to invoke its service, which is modeled by functions Ps: $P \rightarrow S$, Ms: $M \rightarrow S$, Ma: $M \rightarrow A$. In each pair (m, p) which is the input of action λ, the server components must match: $Ms(m) = Ps(p)$.

Agents can be infinite or may terminate in special actions of the form $\lambda = ((m, p), (p'))$, where the output message is absent. Note that we exclude broadcast and multicast communication because the messages are agents carriers.

The behavior of the distributed system is determined by its labeled transition system—LTS [54]. The vertex in LTS (we do not use the name "state" to avoid ambiguity) is a configuration T of IMDS model: a set of current states of all servers and current messages of all agents (except terminated). The initial configuration T_0 contains initial states P_0 and initial messages M_0. The input configuration $T_{inp}(\lambda)$ of the action $\lambda = ((m, p), (m', p'))$ contains m and p belonging to its input pair (m, p) and the output configuration $T_{out}(\lambda)$ contains m' and p' of its output pair (m', p'). Obviously, $T_{inp}(\lambda)$ and $T_{out}(\lambda)$ are not functions because they include the elements of the pairs (m, p) and (m', p'), respectively, all states and messages about servers and agents other than those participating in λ are arbitrary.

$$\begin{aligned} &\text{LTS} = \langle Q, q_0, W \rangle \text{where}: \\ &\quad Q = \{T_0, T_1, \ldots\} \text{ (vertices)} \\ &\quad q_0 = T_0 \text{(initial vertex)} \\ &\quad W = \{ (T, \lambda, T'), \ldots \mid \lambda \in \Lambda, \\ &\qquad T = T_{\text{inp}}(\lambda), T' = T_{\text{out}}(\lambda) \} \text{ (transitions)} \end{aligned} \tag{3.2}$$

Actions are interleaved (one action executed at a time [55]). Note that each server performs its action locally and autonomously (only the state of the server and the messages *pending* on that server are taken into account). In addition, the communication is asynchronous: the server process sends a message to another server process (or the agent sets the server state for another agent) regardless of the current situation of the target process (and any other process). The pair (*message, state*) that triggers the action does not occur synchronously: the state is waiting for a matching message,

or the message is waiting for a matching state. The only exception is the message sent to the server itself: the new state and the new message appear in the server synchronously. This is natural because synchrony is allowed on a single server. As a result, we may call the processes *autonomous* and *asynchronous*, and the action is executed *locally*.

Note that in real distributed systems, there is no interleaving; unrelated activities may run in parallel but are not synchronized. Interleaving is one of the possible semantics of parallelism, another is the semantics of simultaneity, more akin to the operation of a distributed system. We use interleaved semantics because of its simplicity, and Manna and Pnueli have shown that it is equivalent to concurrent [56]. Also, please note that the servers are autonomous, thus no action can invalidate another action in other server. On the other hand, the actions enabled in the same server are always in conflict due to interleaving: one of them is executed at a time and typically it disables other actions prepared in the same server (if the output state is different from the input one).

Processes in the system are defined as sequences of actions. Suppose two consecutive actions in the process are connected by the server state. In that case, it is a server process that communicates with other server processes by means of messages (states are carriers of server processes). If two consecutive actions in the process are connected by an agent message—it is an agent process that communicates with other agent processes through the states of the servers (agents' messages are carriers of agent processes). As some actions may be inaccessible in a given system, they are not included in any process, as they are not part of any sequence. To avoid such "orphan" actions, we allocate them to sets instead of sequences. The process B_i of the server s_i is the set of actions with the sever s_i states on input. The process C_j of the agent a_j is the set of actions with the agent a_j message on input.

$$\begin{aligned} B_i &= \{\lambda_1, \lambda_2, \ldots \in \Lambda | \lambda = \big((m,p),\big(m',p'\big)\big) \\ &\quad \vee \lambda = \big((m,p),\big(p'\big)\big), p, p' \in P_i\}, i = 1, \ldots, n \\ C_j &= \{\lambda_1, \lambda_2, \ldots \in \Lambda | \lambda = \big((m,p),\big(m',p'\big)\big) \\ &\quad \vee \lambda = \big((m,p),\big(p'\big)\big), m, m' \in M_j\}, j = 1, \ldots, k \end{aligned} \tag{3.3}$$

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

$$\begin{aligned} B &= \{B_1, B_2, \ldots, B_n\} \\ C &= \{C_1, C_2, \ldots, C_k\} \end{aligned} \tag{3.4}$$

Examples of distributed systems modeled in IMDS can be found in [57]. In [58], the verification of the Karlsruhe Production Cell is covered, in which servers implement devices in the cell and agents implement metal plates that are processed. In automatic vehicle guidance system [59]—servers implement road segment controllers, and agents implement vehicles.

4 Simple IMDS example: buffer

The Dedan program was developed for the specification and verification of distributed systems. A distributed system in IMDS is simply a set of actions and initial items (states and messages), but for programming purposes, the input form of Dedan contains a definition of types and variables, formal and actual parameters, and an initialization part. In addition, actions are grouped for individual servers or agents, and repeaters can be applied to facilitate the definition of sets of similar actions. For the action $\lambda = ((m, p), (m', p'))$, treated as the execution of the service on the server, a more convenient notation is used, in which the server state p is denoted as pair (s, v), where s is the *server* and v is the *value* of the state, $s \in S$, $v \in V$. This allows the introduction of server types with similar sets of states, but differing in s. We have two server vectors `mE[2]` and `lotE[2]` in AVGS source code in Sect. 7. The message m is denoted as triple (a, s, r), in which *agent* a invokes the *service* r is on the *server* s, $a \in A$, $s \in S$, $r \in R$. Therefore, instances of the agent type can differ only in a or in a and s. We have the agent vector `AMP[N]`, $N = 2$ in AVGS code. The server type can offer a number of services, for example, *wait* and *signal* on a semaphore, *put* and *get* on a buffer, etc. The action $\lambda = (((a, s, r), (s, v)), ((a, s', r'), (s, v')))$ has the form `{a.s.r, s.v}->{a.s'.r', s.v'}` in Dedan source code. Note that the symbol $\rightarrow$ does not represent a function; it is introduced to show the progress of the action from its input to its output items and is similar to the graphical representation of an automaton transition as an arrow. Also note that a single action is deterministic as it is a tuple in the Λ relation. The nondeterminism lies in the fact that the same pair (m, p) can be an input pair of many actions, corresponding to nondeterministic divergences, for example $\lambda_1 = ((m, p), (m', p'))$, $\lambda_2 = ((m, p), (m'', p''))$. Nondeterminism can refer to the server: $m' = m''$, $p' \neq p''$, to the agent: $m' \neq m''$, $p' = p''$, or both: $m' \neq m''$, $p' \neq p''$. Yet another type of nondeterminism arises from interleaving, since when multiple actions are enabled, only one of them is executed. If the actions are in different servers, then the execution of the action cannot invalidate the action in the other server.

The more abstract imperative language Rybu has been developed [9], but the basic IMDS notation is more appropriate for the purposes of this article, so we do not refer to Rybu here.

A simple example of a *buffer* with producer and consumer agents (each one originating from its own server) is provided in the listings below, to illustrate the two views of a distributed system. First, the server view follows. The notation is intuitive: server types are defined (lines 2, 9, 16). The formal parameters specify the agents and other servers used. Each server type contains states (3, 10), services (4, 11), and

actions (6–7, 13–14). Then the server and agent variables are declared (17,18). Variables can have the same names as the types; they are distinguished by context. If a variable has the same identifier as its type, then the variable:type declaration may be suppressed to a single identifier, as in the example. Finally, servers (20–22) and agents (23,24) are initialized, and variable names are associated with formal parameters of servers.

Sample action (6) reads: when the agent `Aprod` invokes the service `put` on the server `buf`, and the server is in `no_elem` state, then the server changes its state to `elem` and the message `ok_put` is issued to the server `Sprod` in the context of the same agent. The system converted to the agent view (this is done automatically by the Dedan program) is as follows.

```
1.  system buffer_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. }.
```

```
1.   system buffer_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,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,Scons),
     ... // similar to Aprod
17.  agents Aprod,Acons;
18.  servers buf,Sprod,Scons;
19.  init -> {
20.     Aprod(buf,Sprod).Sprod.doSth,
21.     Acons(buf,Scons).Scons.doSth,
22.     buf.no_elem, Sprod.neutral, Scons.neutral,
23.  }.
```

Note that the set of system actions is uniform; the views differ only in a manner actions are grouped. In the server view, the actions are grouped into individual server types, while in the agent view, they are grouped into individual agent types.

A fragment of the LTS of the example system is presented in Fig. 4 . The vertices in the first line display messages of the agents `Aprod` and `Acons` are (without agent identifiers), and the states of all servers (`buf`, `Sprod`, `Scons`) are displayed in the second line (without server identifiers). Of course, the LTS'es generated from both the server view and the agent view are identical because they are projections on servers and on agents of a uniform system, and the views are its decompositions.

In the example, the servers and agents are deterministic, just to reduce the system and its LTS to a small size. Nondeterminism in the LTS comes from concurrency. Nondeterministic automata can be found in our other papers, where automata notation is used informally for the illustration of distributed systems designed in IMDS: A special kind of bounded buffer with agents switching between producer and consumer roles [60] or buffer with a capacity > 1, that can accept both putting and getting if it is neither full nor empty [1]. In student exercise examples shown later in Fig. 24, S-DA^3 views on the left are almost all nondeterministic: more than one transition is enabled (highlighted). The system components in the present example could be easily converted to nondeterministic, allowing the producer and consumer to do something different in many steps between their activities connected with the buffer. Actions introducing nondeterminism in producer would, according to the server view, be additional action 13a:

```
13a.  {Aprod.Sprod.doSth, Sprod.neutral}—>{Aprod.Sprod.doSth, Sprod.neutral}.
```

and according to the agent view, additional action 8a (in fact, it is the same action buf in different grouping):

```
8a.  {Aprod.Sprod.doSth, Sprod.
neutral}—>{Aprod.Sprod.doSth, Sprod.
neutral}.
```

Nondeterminism can result from multiple agents that run is a single server. For example, if we have several producers, any of them can call *put* on the *buffer*. The changes of the source code in the server view are (original line numbers are preserved, and line 0 is added for a constant definition, the differences are underlined):

```
0.   #DEFINE K 3
1.   server: buf (agents Aprod[K],Acons; servers Sprod,Scons),
6.   <i=1..K>{Aprod[i].buf.put,buf.no_elem}   ->
              {Aprod[i].Sprod.ok_put, buf.elem},
9.   server: Sprod (agents Aprod[K]; servers buf),
18.  agents Aprod[K],Acons;
19.  init -> {
20.     Sprod(Aprod[1..K],buf).neutral,
22.     buf(Aprod[1..K],Acons,Sprod,Scons).no_elem,
23.  <i=1..K>Aprod[i].Sprod.doSth,
```

Using Dedan, communication deadlocks in the server view and resource deadlocks in the agent view can be identified, and distributed termination can be checked. There is no deadlock in our example; deadlocked systems are presented with counterexamples in [57][58][59]. For a simple, practical example, see Sect. 7. A counterexample or other behavior can be tracked in Dedan using the simulator. The simulation is performed over the LTS of the verified system. Often, however, the simulation would be better performed over the components (servers and agents) of the verified system, shown separately and cooperating with each other. For this reason, we have introduced distributed automata—an alternative formulation of IMDS systems. Our distributed automata are equivalent to IMDS, but allow for graphical definition and graphical simulation of distributed systems in terms of its components. Of course, the graphic form should maintain the locality of actions, the autonomy of components and the asynchrony of their behavior. The duality of communication must also be preserved, so two forms of graphical specification have been developed: one for the server view and the other for the agent view.

5 Informal view of Distributed Automata DA^3

We are introducing DA^3 informally but systematically. Since everything is dual in IMDS, we build the automata so that in server automata, the transitions lead from state to state (of the same server), having messages as input and output symbols on the transitions. In the opposite view, agent automata lead from message to message (of the same agent), having servers' states as input and output symbols on the transitions. So for the action $((m, p), (m', p'))$, we have the transition $p \xrightarrow[p\ /\ p]{m\ /\ m} p'$ in server automaton $s = Ps(p)$ and the transition $m \xrightarrow{a} m'$ in agent automaton $a = Ma(m)$.

Let us look at the server view first. To avoid ambiguity, we call the automaton states *nodes*, as the term "state" is attributed to IMDS servers. Each server process is modeled using a server automaton equipped with its input message set. This models distributed servers communicating through message passing, similar to the Client–Server model of distributed systems [10]. There are three severs automata in the *buffer* example depicted in Fig. 2. In the automaton *Sprod*, its nodes are simply the states of the server *Sprod*: (*Sprod, neutral*) and (*Sprod, prod*). Since the name of the server is the same in both pairs, the figure shows only the state values in the ovals of the nodes. Each transition relates to an action in the server *Sprod*, triggered by a message addressed to the server, which matches its current state. The figure shows the initial content of the input sets. The initial nodes of the automata have double borders. The view of these server automata in the Dedan program is presented in Fig. 6 .

For example, the node (*Sprod, neutral*) accepts the message (*Aprod, Sprod, doSth*), which is the input symbol of the automaton transition

$(Sprod, neutral) \xrightarrow{(Aprod, Sprod, doSth) / (Aprod, buf, put)} (Sprod, prod)$. The message must be stored in the input set of the automaton to fire the transition. The action generates the new state of the server (*Sprod, prod*) and the new message (*Aprod, buf, put*), which are modeled as the destination node (*Sprod, prod*) of the transition and the output symbol of the transition (*Aprod, buf, put*): the message generated on the transition. Since the action took the only message from the input set, and the new message is addressed to the *buf* automaton, the *Sprod* input set becomes empty. Therefore, no action can be executed in the automaton *Sprod* at this time. However, if in the future the automaton can receive a message to its input set, and if this message matches the state (*Sprod, prod*), then the transition in the automaton will be possible.

The input set of the automaton *buf* can contain two messages simultaneously: (*Aprod, buf, put*) and (*Aprod, buf, get*). However, only one of them can be accepted in a given node, which triggers the appropriate transition: the first one in the node (*buf, no_elem*), the second in the node (*buf, elem*). Initially, the input set is empty.

Figure 4 shows a set of agent automata, which is equivalent to the previously described set of server automata. However, this form of automata highlights the resource sharing aspect of a distributed system, where traveling agents communicate by setting the server state values. This model conforms to the Remote Procedure Call paradigm [10]. In the figure, each agent travels between its home server and the shared *buf* server. Let us focus on producer *Aprod*. The agent name is omitted from message triples because it is common to all *Aprod* messages. The automaton starts from the initial node (message) of the *Aprod* automaton (*Aprod, Sprod, doSth*). The *Sprod* server initial state: (*Sprod, neutral*) is the input symbol of the transition from the node (*Aprod, Sprod, doSth*) so that the transition can be executed, leading to the next node (message) of the agent: (*Aprod, buf, put*) which transfers the agent to the *buf* server. In addition, the action generates a new state of the *Prod* server (*Sprod, prod*), which is the output symbol of the transition. If the current state of the *buf* server is (*buf, no_elem*), which means this symbol is present, the next *Aprod* transition can be fired. If not, the agent is waiting for such state of the *buf* server to occur. The next transition leads from (*Aprod, buf, put*) to (*Aprod, Sprod, ok_put*) and it changes the current state of *buf* from (*buf, no_elem*) to (*buf, elem*), which are the input and output symbol of the transition $(Aprod, buf, put) \xrightarrow{(buf, no\_elem) / (buf, elem)} (Aprod, Sprod, ok_{put})$. Thus, in the agent automaton, we see the messages as the nodes, and the automata states as input and output symbols of the transitions. This is counterintuitive to a user who is used to other kinds of automata and takes some getting used to.

We see that server states are common to some agents; in the example, the state of the *buf* server is important to both agents: *Aprod* and *Acons*. Thus, we have a vector of the current server states, common to all agents, which contains input symbols for transitions and receives their output symbols. This is safe because the transitions are interleaved, so there can be no conflict between the transitions in distinct agent automata that set new states of the same server.

Automata in their nondeterministic form, in which the producer can perform several steps of doing something else between producing subsequent elements, are presented in Fig. 8, as sever automaton and agent automaton (nondeterministic transitions are shown as bold blue arrows).

6 Distributed Autonomous and Asynchronous Automata (DA^3)

6.1 *Server automata S-DA^3*

The IMDS system in the server view can be represented as a set of communicating automata S-DA^3 (distributed server automata), similar to MPA (point 3 in the enumeration in Sect. 2). The server states are the nodes (we do not use "state" to avoid ambiguity) of the corresponding automaton.

The initial state of the server is the initial node of the automaton. The server process actions are automaton. The automaton is Mealy-style [11], the transition labels in the automaton have the form extracted from actions; IMDS action $\lambda = ((m, p), (m', p'))$ is converted to a transition from p to p' with the label m/m' (m is the input symbol firing the transition while m' is the output symbol generated on the transition). Transitions in the server s_i automaton are the relation in $P_i \times M \times M \times P_i$. Of course, m fulfils $Ps(m) = s_i$. 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 the set of actions may contain nondeterministic actions $\lambda_1 = ((m, p), (m', p'))$, $\lambda_2 = ((m, p), (m'', p''))$, $m' \neq m''$. The automaton is equipped with an input set—a set of input symbols, corresponding to a set of messages pending at the server. Firing the transition $(p, m/m', p')$ in the server s automaton retrieves the symbol m from the input set of this automaton and inserts the symbol m' to the input set of the server s' automaton, appointed by m', $Ms(m') = s'$. The initial input set consists of the initial agents' messages that are directed to this server. The special *agent-terminating action* $\lambda = ((m, p), (p'))$ is converted to a transition that generates no output symbol. To distinguish between the two types of actions, we call the former form a *progress action*.

Formally, having the definition of *S, A, V, R, P, M* in IMDS (respectively: servers, agents, values, services, states, messages), we have the set Ψ (originally we used reflected *S*, but there was a problem with typesetting of this symbol in the article) of n distributed sever automata,

$\Psi = \{\psi_i \mid i = 1, \ldots, n\}$, where n is the number of servers in set S. Intuitively, Ψ is the set of automata in which each automaton ψ_i corresponds to the server s_i in the IMDS system. Therefore, S is the set of servers while Ψ is the set of server automata. In the definition below, $exp(arg)$ is used for powerset 2^{arg}. The ith distributed server automaton is $\psi_i = (s_i, P_i, p_{0i}, F_i, X_i, X_{0i}) \in \Psi$, where:

- $s_i \in S$—the ith server,
- P_i—the set of nodes, which are the states of s_i,
- $p_{0i} \in P_i$—the 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 $\lambda = ((m, p_1), (p_2)) \mid p_1, p_2 \in P_i, \exists a_j \in A: m, m' \in M_j\}$—the set of transitions (for ordinary actions and agent-terminating actions, respectively),
- $X_i \in exp(\{m \mid Ms(m) = s_i\})$—the input set; X_i is the variable having a set value: each transition $\lambda = ((m_1, p_1),(m_2, p_2))$ removes its input message m_1 from the input set of the server s_1 appointed by the message m_1, $Ms(m_1) = s_1$, and inserts the message m_2 to the input set of the server s_2 appointed by m_2, $Ms(m_2) = s_2$ (except for the agent-terminating transition), accordingly to the rules for semantics described below,
- $X_{0i} \in exp(\{m \mid Ms(m) = s_i, m \in M_0\})$—the initial input set of the server s_i.

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

- $\forall m_1 \in X_i, m_2 \in X_j, m_1 \in M_k, m_2 \in M_l$: $m_1 \neq m_2 \Rightarrow k \neq l$: for each agent at most one message can exist in the global configuration,
- $\forall (p_1, m/m', p_2) \in F_i \; \exists F_j$: $m' \in F_j$: each output symbol (m') is an input symbol of an automaton belonging to Ψ.

The server automata of the example buffer system in the server view are illustrated in Fig. 3. The initial nodes of the server automata have double borders. Server names are omitted in the node labels, because they are identical for all nodes of a given server automaton. The input sets of the automata X_{buf}, X_{Sprod} and X_{Scons} are shown with their initial contents at the bottom of the picture; they change as the automata run.

The semantics of Ψ is defined as global vertex space $(\{T_\Psi\}, T_{\Psi_0}, \text{next}\Psi)$, where $\{T_\Psi\}$ is a set of global vertices, T_{Ψ_0} is the initial global vertex, and next Ψ is the transition relation, defined as follows:

- The global vertex of Ψ is $T_\Psi = ((p_1, X_1), (p_2, X_2), \ldots, (p_n, X_n))$ (current states and input sets of pending messages of all servers).
- If in T_Ψ there exists an ψ_i in which (p_{i1}, X_i), $(p_{i1}, m/m', p_{i2}) \in F_i$, $m \in X_i$, $Ms(m') = s_j$ then a possible next global vertex T'_Ψ is:

$$T'_\psi = \left((p'_k, X'_k) \mid \left\{ \begin{array}{ll} p'_k = p_k & \text{for } k \neq i \\ p'_k = p_{i_2} & \text{for } k = i \end{array} \right\}, \left\{ \begin{array}{ll} X'_k = X_k \backslash \{m\} \cup \{m'\} & \text{for } i = k = j \\ X'_k = X_k \backslash \{m\} & \text{for } k = i \neq j \\ X'_k = X_k \cup \{m'\} & \text{for } k = j \neq i \\ X'_k = X_k & \text{for } i \neq k \neq j \end{array} \right\}, k = 1, \ldots, n \right) \tag{6.1}$$

(the automaton ψ_i changes its node from p_{i1} to p_{i2}, all other automata preserve their nodes; the message m is extracted from the input set X_i of the automaton ψ_i, the continuation message m' is inserted into the input set X_j of the automaton ψ_j appointed by m', $Ms(m') = s_j$, all other input sets remain unchanged; the special case is for $(i = k = j$, first case), where the server s_i sends the message m' to itself).

- If in T_Ψ there exists an ψ_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 vertex is:

$$T'_\psi = \left(\left(p'_k, X'_k\right) \mid \left\{ \begin{array}{ll} p'_k = p_k & for\ k \neq i \\ p'_k = p_{i_2} & for\ k = i \end{array} \right\}, \left\{ \begin{array}{ll} X'_k = X_k \backslash \{m\} & for\ k = i \\ X'_k = X_k & for\ k \neq i \end{array} \right\}, k = 1, \ldots, n \right) \tag{6.2}$$

(the automaton ψ_i changes its node from p_{i1} to p_{i2}, all other automata preserve their nodes; the message m is extracted from the input set X_i of the automaton ψ_i, all other input sets remain unchanged).

- The initial global vertex is $T_{\Psi 0} = ((p_{01}, X_{01}), (p_{02}, X_{02}), \ldots, (p_{0n}, X_{0n}))$.
- For a given global vertex T_Ψ, the transition relation $nextT_\Psi(T_\Psi)$ is the set of pairs (T_Ψ, T'_Ψ). The transition relation $next_\Psi = \bigcup_{T_\Psi} nextT(T)$.

  Comment: in the execution, if for T_Ψ there exist multiple possible next global vertices, one of them is chosen in nondeterministic way; however, it is not important in the definition of the global vertex space.

The global vertex space of Ψ cooperation is a counterpart to the LTS of IMDS system: global vertices contain states of all servers, input symbol (message) of a transition should be attributed to a source global vertex, while output symbol (message) to a target global vertex. The fragment of a global vertex space for the buffer system is presented in Fig. 9.

The nodes of server automata $\Psi = \{\psi_{\text{buf}}, \psi_{\text{Sprod}}, \psi_{\text{Scons}}\}$ (server states) shown in Fig. 3 are (note that the server state has the form (server, value)):

$P_{buf} = \{$(buf, no_elem), (buf, elem)$\}$,
$P_{Sprod} = \{$(Sprod, neutral), (Sprod, prod)$\}$,
$P_{Scons} = \{$(Scons, neutral), (Scons, cons)$\}$.

The sets of transitions are (the message has the form (*agent*, *server*, *service*), and the transition has the form (*input node* = (*s*, *v*), input symbol = (*a*, *s*, *r*)/ output symbol = (*a*,*s*', *r*'), output node = (*s*, *v*')), or $(s, v) \xrightarrow{(a,s,r)/(a,s',r')} (s, v')$, where nodes are states and symbols are messages):

$F_{buf} = \{$((buf, no_elem), (Aprod, buf, put)/ (Aprod, Sprod, ok_put), (buf, elem)), ((buf, elem), (Acons, buf, get)/ (Acons, Scons, ok_get), (buf, no_elem))$\}$,
$F_{Sprod} = \{$((Sprod, neutral), (Aprod, Sprod, doSth)/ (Aprod, buf, put), (Sprod, prod)), ((Sprod, prod), (Aprod, Sprod, ok_put)/ (Aprod, Sprod, doSth), (Sprod, neutral))$\}$
$F_{Scons} = \{$((Scons, neutral), (Acons, Scons, doSth)/ (Acons, buf, get), (Scons, cons)), ((Scons, cons), (Acons, Scons, ok_get)/ (Acons, Scons, doSth), (Scons, neutral))$\}$

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

$X_{buf} \in exp(\{$(Aprod, buf, put), (Acons, buf, get)$\})$,
$X_{Sprod} \in exp(\{$(Aprod, Sprod, doSth), (Aprod, Sprod, ok_put)$\})$,
$X_{Scons} \in exp(\{$(Acons, Scons, doSth), (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 of ψ_{buf}, ψ_{Sprod} and ψ_{Scons} are:

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

S-DA^3 are similar to Message Passing Automata. The difference is in the ordering of messages at the input of the automaton: in MPA, pending messages are ordered in the input queue (or input buffer) [43][44], while in S-DA^3 any message form the input set may fire a transition (no ordering). If the input buffers in the MPA implementation are limited, a deadlock can occur due to all processes sending messages/ data to the full buffers. This situation can occur when the size of the buffers is set to a too small value [53]. IMDS helps overcome this problem by posing an accurate size limit to the input set: it is simply the number of agents, or precisely: the number of agents visiting the server (occurring in its actions). For example, the repertoire of agents putting their messages into the input set of the buf automaton is {Aprod, Sprod}, while in the case of the Sprod automaton, it is {Aprod}. Therefore, the sizes of the input sets in the implementation of the servers should be: 2 and 1, respectively.

6.2 Agent automata A-DA^3

The IMDS system in the agent view can be shown as a set of communicating automata A-DA^3 (agent distributed autonomous and asynchronous automata). We use the term node in these automata instead of state, because the states are attributed to servers in IMDS, and this could be misleading. The A-DA^3 automata are similar to timed automata with variables used in Uppaal [5] (but here we only consider timeless systems):

- The agent messages are the nodes of the automaton.
- The agent's initial message is the automaton initial node.
- The actions of the agent process are the automaton transitions.
- The automaton is Mealy-style [11]; the labels of the transitions in the automaton have the form extracted from actions; IMDS action $\lambda = ((m, p), (m', p'))$ is converted to a transition $(m, p/p', m')$ from the node m to the node m' with the label p/p' (p is the input symbol conditioning transition, while p' is the output symbol generated 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 $M_i \times P \times P \times M_i$ because of the possible nondeterminism in the set of actions: $\lambda_1 = ((m, p), (m', p'))$, $\lambda_2 = ((m, p), (m', p''))$, $p' \neq p''$.
- In the case of the agent-terminating action $\lambda = ((m,p), (p'))$, the special *terminating node* t_a in the automaton is added to be the target node of the transition, $Ma(m) = a$, and the transition is of the form $(m, p/p', t_a)$. For node t_a, there are no outgoing transitions. The action $((m, p), (p'))$ in the agent a automaton, $a = Ma(m)$, has the form $m \xrightarrow{p/p'} t_a$ (see below), while in the server automaton Ψ_s, $s = Ps(p)$, the transition is $p \xrightarrow{m/} p'$.

- The automata system is equipped with a *global input vector*—a vector of current input symbols for the transitions of the agent automata: servers' states. The vector is indexed with the servers. Firing the transition (*m, p/p', m'*) in the automaton replaces the symbol *p* in the vector with the new symbol *p'* in the position of the server $s = Ps(p)$. The initial global input vector consists of initial states of all automata.

Note that in server automata, the transitions are stretched between states, and the input/output symbols are agent messages. Everything is dual in the agent view of the system, i.e., the transitions lead from one message to another, while server states are input and output symbols. This is counterintuitive to a user who is used to other kinds of automata and takes some getting used to.

Formally, having the definition of *P*, *M*, *S*, *A*, *V*, *R* from IMDS (respectively: states, messages, servers, agents, values, services), we have the set $\Uparrow$ (*A* upside down, rounded to distinguish it from the general quantifier ∀) of *k* distributed agent automata, $\Uparrow = \{\mathfrak{v}_i \mid i = 1, \ldots, i\}$, 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 agent automaton definition below). Intuitively, $\Uparrow$ is the set of automata in which each automaton $\mathfrak{v}_i$ corresponds to the agent ai in the IMDS system. Therefore, A is the set of agents while $\Uparrow$ is the set of agent automata. The *i*th distributed agent automaton is $\mathfrak{v}_i = (a_i, M_i, m_0 i, t_{\mathfrak{v}i}, G_i, Y, Y_0) \in \Uparrow$, where:

- a_i—the *i*th agent,
- $M_i \cup \{t_{\mathfrak{v}i}\}$—the set of nodes, which are the messages of the agent a_i; $t_{\mathfrak{v}i}$ is the destination node of the agent-terminating transitions if the agent a_i terminates ($t_{\mathfrak{v}i}$ appears as the target node of the agent-terminating transition),
- $m_{0i} \in M_i$—the initial node,
- $G_i = \{ (m_1, p/p', m_2)$ for $\lambda = ((m_1, p), (m_2, p'))$ or $(m_1, p/p', t_{\mathfrak{v}i})$ for $\lambda = ((m_1, p), (p')) \mid m_1, m_2 \in M_i, \exists s_j \in S: p,p' \in P_j\}$—the set of transitions,
- $Y = [p_1, \ldots, p_n] \mid p_j \in P_j$—the global input vector (common for all automata $\mathfrak{v}_i$ in the system); *Y* is the vector of variables, *Y/j* is the *j*th position of *Y*, every variable *Y/j* has the range over the set of states of the server s_j: the action $\lambda_1 = ((m, p), (m', p'))$ changes the value of the variable *Y/j* at the position of its input and output state server, $Ps(p) = Ps(p') = s_j$; accordingly to rules for semantics below,
- $Y_0 = [p_{01}, \ldots, p_{0n}] \mid p_j \in P_j, p_j \in P_0$—the initial global input vector, consisting of initial states of all servers.

The semantics of $\Uparrow$ is defined as global vertex space ($\{T_\Uparrow\}$, $T_{\mathfrak{v}_0}$, *next*$\Uparrow$), where $\{T_\Uparrow\}$ is a set of global vertices, $T_{\mathfrak{v}_0}$ is initial global vertex, and *next*$\Uparrow$ is a transition relation, defined as follows:

- The global vertex of $\Uparrow$ is $T_\Uparrow = \{m_1, \ldots, m_k, Y\}$, $m_i \in M_i \cup \{t_{\mathfrak{v}i}\}$. If in $T_\Uparrow$ 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, $Ms(m_i) = s_x$, and a message m_j to a server s_y, $Ms(m_j) = s_y$ is issued) then a possible next global vertex is:

$$T'_\Uparrow : \forall m \in T_\Uparrow : \begin{Bmatrix} m = m_i \wedge m = m_j \Rightarrow m_i \in T'_\Uparrow \\ m = m_i \wedge m \neq m_j \Rightarrow m_j \in T'_\Uparrow \\ m \neq m_i \Rightarrow m_i \in T'_\Uparrow \end{Bmatrix};$$

$$Y' = [p'_1, \ldots, p'_n] \mid p'_k = \begin{Bmatrix} Y/k & for\ k \neq x \\ p' & for\ k = x \end{Bmatrix}, \ k = 1, \ldots, n \tag{6.3}$$

(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*, in the position appointed by *p* and *p'*, $s_x = Ps(p) = Ps(p')$, all other elements of the vector *Y* remain unchanged).

- If in $\{ T_\Uparrow\}$ 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 m_i is the last message in the run of agent a_i, then the agent terminates, the sever s_x appointed by the message m_i, $Ms(m_i) = s_x$, changes its state from *p* to *p'*) then a possible next global vertex is:

$$T'_\Uparrow : \forall m \in T_\Uparrow : \begin{Bmatrix} m = m_i \Rightarrow m_i \notin T'_\Uparrow \\ m \neq m_i \Rightarrow m_i \in T'_\Uparrow \end{Bmatrix};$$

$$Y' = [p'_1, \ldots, p'_n] \mid p'_k = \begin{Bmatrix} Y/k & for\ k \neq x \\ p' & for\ k = x \end{Bmatrix}, \ k = 1, \ldots, n \tag{6.4}$$

(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 $T_{\Uparrow 0} = \{m_{01}, \ldots, m_{0n}, Y_0\}$. For a given global vertex $T_\Uparrow$, transition relation $nextT_\Uparrow(T_\Uparrow)$ is a set of pairs $(T_\Uparrow, T'_\Uparrow)$.
- The transition relation $next\Uparrow = \mathrm{U}T_\Uparrow nextT_\Uparrow(T_\Uparrow)$. Comment: in the execution, if there are multiple next vertices possible, one of them is chosen in nondeterministic way; however, it is not important in the definition of the global vertex space.

The distributed agent automata for the *buffer* system $\Uparrow = \{\mathfrak{v}_{Aprod}, \mathfrak{v}_{Acons}\}$ are illustrated in Fig. 4. The initial nodes of the automata, being the initial messages of the agents, have double borders. Agent identifiers are omitted

in message labels (nodes of the automata), because they are identical for all messages in the given agent.

The nodes of sever automata (agent messages) shown in Fig. 4 are (note that the message has the form (agent, server, service)):

M_{Aprod} = {(Aprod, buf, put), (Aprod, Sprod, doSth), (Aprod, Sprod, ok_put)},
M_{Acons} = {(Acons, buf, get), (Acons, Scons, doSth), (Acons, Scons, ok_get)}.

As the agents do not terminate in the *buffer* system, there are not terminating nodes. Note that the server state has the form (*server*, *value*), and the transition has the form (*input node* = (*a*,*s*, *r*), *input symbol* = (s, v)/*output symbol* = (*s*, *v*'), *output node* = (*a*, *s*', *r*')), or $(a, s, r) \xrightarrow{(s,v)/(s,v')} (a, s', r')$, where nodes are messages and symbols are states. The same action (((*a*, *s*, *r*), (*s*, *v*)), ((*a*, *s*', *r*'), (*s*, *v*'))) as the transition in the server automaton ψ_s has the form $(s, v) \xrightarrow{(a,s,r)/(a,s',r')} (s, v')$. The sets of transitions ε_{Aprod} and ε_{Acons} are:

M_{Aprod} = {((Aprod, Sprod, doSth), (Sprod, neutral)/(Sprod, prod), (Aprod, buf, put)),
((Aprod, buf, put), (buf, no_elem)/(buf, elem), (Aprod, Sprod, ok_put)),
((Aprod, Sprod, ok_put), (Sprod, prod)/(Sprod, neutral), (Aprod, Sprod,doSth))}
M_{Acons} = {((Acons, Scons, doSth), (Scons, neutral)/(Scons, cons), (Acons, buf, get)),
((Acons, buf, get), (buf, elem)/(buf, no_elem), (Acons, Scons, ok_get)),
((Acons, Scons, ok_get), (Scons, cons)/(Scons, neutral), (Acons, Scons, doSth))}

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)].

The global vertex space of ⊎ collaboration is the IMDS LTS equivalent: global vertices contain the messages of all non-terminated agents, and the states of all servers in the global input vector. The fragment of the global vertex space for the *buffer* system is presented in Fig. 10.

7 Practical example of DA³: Automatic Vehicle Guidance System

The *buffer* example is tiny, just to present the main ideas. Now we introduce the automatic vehicle guidance system (AVGS) from [59]. The system consists of road markers and warehouse lots, presented in Fig. 11, communicating with each other to guide autonomous moving platforms (AMPs) from LotE1 to LotE2 or vice versa. The two AMPs might collide in section MarkerM, but there is the chance that one could wait temporarily in LotM, until the other has passed. There are six servers representing the Lots and Markers controllers, with a protocol for requesting and granting road segments managed by the controllers.

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. Only the MarkerM controller actions are shown, as other controllers are quite simple: request and grant if not occupied.

```
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 mE[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.takeE[j], mM.resE[j]}->{AMP[i].mM.switch[3-j], mM.occ},
28. <i=1..N><j=1..2> {AMP[i].mM.switch[j], mM.occ} -> {AMP[i].mE[j].tryM[j], mM.occ},
29. <i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.occ} -> {AMP[i].mE[j].takeM, mM.free},
30.
31. //on a way to ME1 or ME2 may go to LE if MEi occupied
32. <i=1..N><j=1..2> {AMP[i].mM.notE[j], mM.occ} -> {AMP[i].lotM.try[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.
35. //going from PL2 - goes to RM1(mE[1]) or RM3(mE[2])
36. <i=1..N><j=1..2> {AMP[i].mM.tryL[j], mM.free} -> {AMP[i].lotM.ok[j], mM.resL[j]},
37. <i=1..N><j=1..2> {AMP[i].mM.takeL[j], mM.resL[j]}-> {AMP[i].mE[j].tryM[j], mM.occ},
38. <i=1..N><j=1..2> {AMP[i].mM.okE[j], mM.occ} -> {AMP[i].mE[j].takeM, mM.free},
39. };
40. server: lotE(agents AMP[N];servers mE),
41. //Edge Warehouse Lot
42. //…
50. server: lotM(agents AMP[N];servers mM),
51. //Middle Warehouse Lot
52. //…
59. servers mE[2],mM,lotE[2],lotM;
60. agents AMP[N];
61. init -> {
62.        <j=1..2> mE[j](AMP[1..N],mM,lotE[j]).free,
63.                 mM(AMP[1..N],mE[1,2],lotM).free,
64.        <j=1..2> lotE[j](AMP[1..N],mE[j]).occ,
65.                 lotM(AMP[1..N],mM).free,
66.        <j=1..2> AMP[j].lotE[j].start,
67. }.
```

The designer defines server (or agent) types rather than individual instances, choosing a graphic or text form. Having the types defined, individual server/agent variables can be declared as type instances. In this example, some servers and agents are grouped into vectors (lines 59, 60). Moreover, some formal parameters are in the form of vectors (2, 19, …). Services and state values can also be vectors (4, 23). For a compact definition, repeaters precede the actions (10–17, …). The indices of agents, states and services indicate individual instances (26–29). The markers E and M names are shortened to `mE` and `mM`. The server type `lotE` is shown as S-DA^3 automaton in Fig. 12. Note the transition from `res` to `occ` (the label is surrounded by a blue ellipse)—it is the agent-terminating transition when `AMP` has reached its destination. There is no output message in this case.

The server view of the system is automatically converted to the agent view by Dedan. In the agent view, actions are grouped by agent. During conversion, the type `AMP` is split into two separate types and renamed to separate `AMP` and `AMP__1`, due to different action sequences on controllers. The agent view shows the system from the point of view of the `AMP` vehicles (listing below). Figure 14 presents a fragment of the `AMP` agent type automaton. The view of the agent automaton in Dedan, which is the visualization of the `AMP`[1] agent specification in IMDS, is presented in Fig. 15.

This image has been obtained automatically: from the graphical specification of server types in the graphical design tool, through the binding of specific server and agent instances in the IMDS source code, automatic conversion from server view to agent view, to display the automata in the A-DA^3 graphical simulator. The arrangement of the nodes in the plane results from an iterative algorithm that moves nodes in an invisible mesh if any transition intersects the node. We are still working on this algorithm. The user can rearrange the nodes to get a clearer view.

A fragment of the `AMP` automaton is visible; the other is named `AMP__1`. The program tries to find agents that have common action sets and combine them into vectors, in this case it would be `AMP`[2], but in this case, it fails because the agents pass through the servers in a different index order: the instance `AMP`[1] ascending and the instance `AMP`[2] descending.

```
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},
10.                    {AMP.lotE[2].take, lotE[2].res} -> {lotE[2].occ},
11. <j=1..2>   {AMP.lotM.try[j], lotM.free} -> {AMP.mM.okL[j], lotM.res[j]},
12. <j=1..2>   {AMP.lotM.ok[j], lotM.occ[j]} -> {AMP.mM.takeL[j], lotM.free},
13. <j=1..2>   {AMP.lotM.take[j], lotM.res[j]} -> {AMP.mM.tryL[j], lotM.occ[j]},
...
```

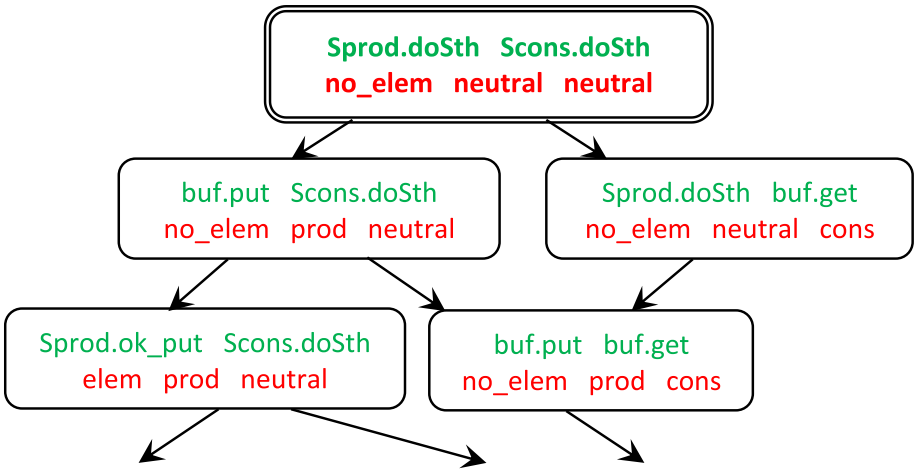

Fig. 4 A fragment of the LTS for the *buffer* system

8 Using DA^3 in the Dedan environment

Editing server automata or agent automata, and simulation over two versions of automata are four different tools in the Dedan integrated verification framework.

We describe a typical procedure during verification: the user defines the system graphically, usually in S-DA^3, then verifies in IMDS: detects deadlocks or missing termination, or confirms the inevitable termination of the system, then Dedan generates a counterexample (in case of confirmed termination called a witness). When the analysis of the counterexample is not easy, because it is either long or difficult to analyze due to a large number of servers/agents, the designer proceeds to simulate the counterexample. Simulation is possible in a configuration graph (LTS, see Fig. 16); however, the graph is usually huge, and the user sees a configuration consisting of the states of all servers and all agents' messages, which often obscures the matter. Then it is possible to proceed to the distributed automata simulation, where the designer can place panels of important automata next to each other and observe their progress step by step. The S-DA^3 view is frequently selected, for example, when verifying IoT sensor systems and data processing nodes, but the A-DA^3 view is also very useful, for example, when tracking the positions of moving parts in Karlsruhe Production Cell [58], vehicles running in the AVGS system [59] or special agents monitoring satellite equipment [61]. The latter move between ground services and individual satellite modules, checking the correctness of the sequences of actions.

The above design, verification and simulation scenarios are typical for students verifying their solutions to synchronization tasks, and implementing projects of autonomous vehicles cooperation or collaborating IoT controllers. The Dedan system was used by over 300 students, most often defining distributed systems textually, but in many cases also graphically in DA^3. Usually, at the design stage, the simulation function is used, in which the designer selects one automaton from among those containing the enabled actions, and makes a nondeterministic choice if more than one action is enabled. Typical student solutions consist of 8–16 servers, 4–6 agents, and the total number of states is typically less than 100 to about 400 actions. The number of states in the server type varies from 2 (semaphores) to 15, as does the number of server services. The number of agent messages (derived from the number of agents, the number of servers visited, and the number of their services) varies considerably from case to case, reaching up to hundreds. Thus, tracking agents is not easy, but much easier than analyzing an LTS or simulating a counterexample in the LTS over configurations. Compare the LTS fragment of AVGS in Fig. 16 with the server automata in Fig. 13 and agent automata in Fig. 15.

After designing the system, verification takes place, the important feature of which is automatism, because Dedan has built-in general formulas that check total or partial deadlock and termination regardless of the structure of the specific system. This is important because the use of model checking and temporal logics is not easy [62] (especially for 2nd year students, but also many engineers avoid formal methods [7]). In about 10% of cases, the systems have errors, and the counterexample has from a few to several dozen steps. In the first case, these are most often errors in the structure of the solution, while the longer ones concern faulty synchronization. Then, the graphical simulation of the counterexample is most often used, in which the subsequent steps are determined by its course until the error or the desired situation occurs.

The basic form of specification used in Dedan is IMDS, because it enables automatic conversion between the server view and the agent view of a system, and conversion to other formalisms: Petri net for static analysis and finding deadlocks using siphons [63], Uppaal input for timed verification [64], etc. However, the specification in the form of a relation between pairs $\lambda = ((message, state), (message', state'))$ is exotic for the users. Therefore, two alternative input forms

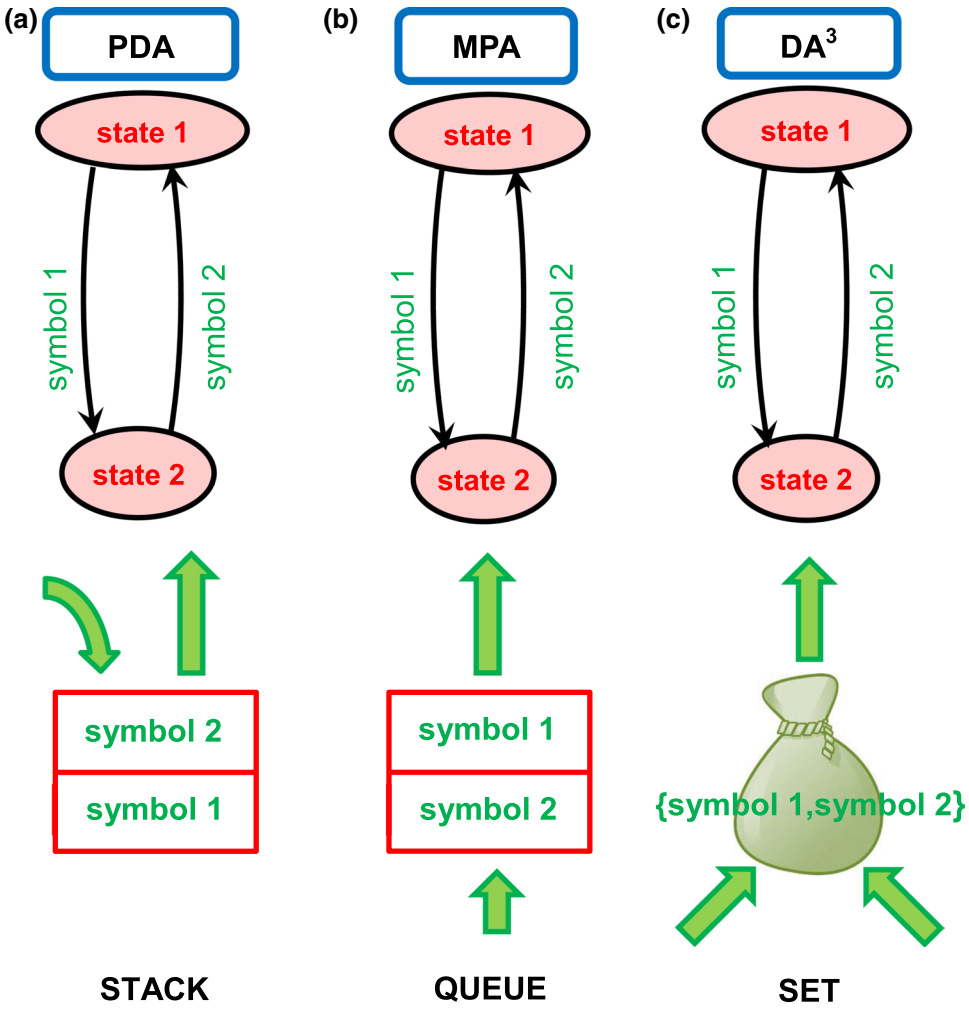

Fig. 5 Three types of automata, showing different scheduling semantics for events: **a** PDA—most recently received event presented to the automaton: transition *state 2*→*state 1* can be enabled if the current state is *state 2*, **b** MDA—least recently received event presented to the automaton: transition *state 1*→*state 2* can be enabled if the current state is *state 1*, and **c** DA^3 (also asynchronous statecharts with input dispatchers [14])—all events are presented simultaneously: some transition is enabled if any of the states is current

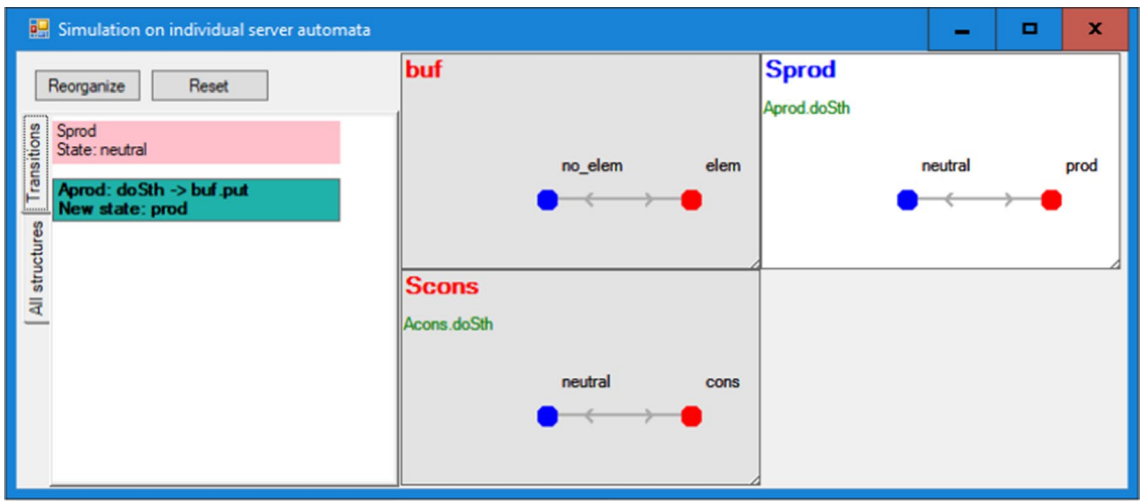

Fig. 6 Dedan S-DA^3 view of server automata shown in Fig. 2

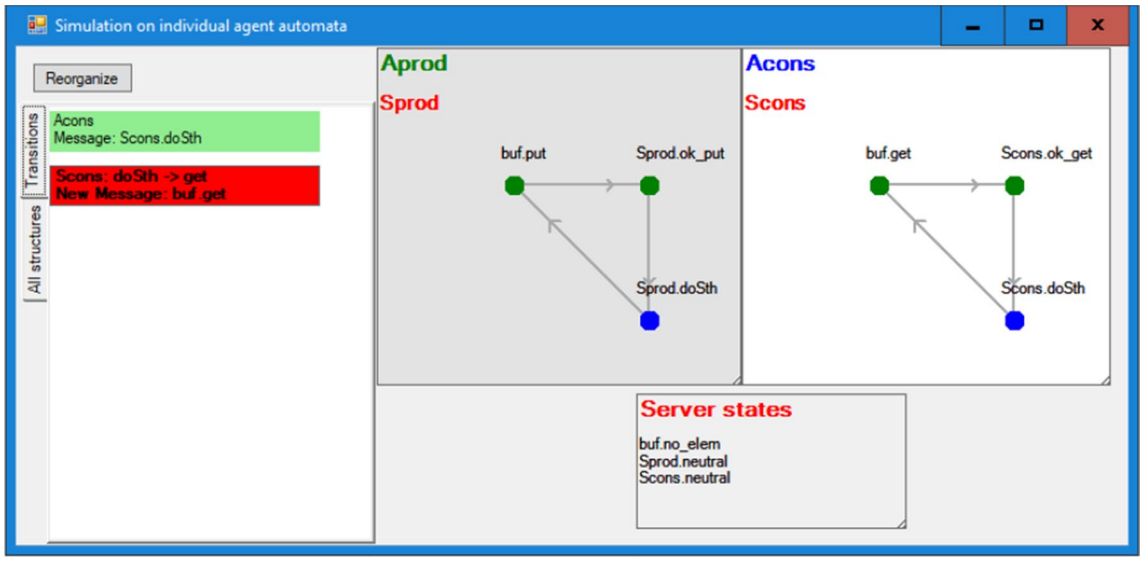

Fig. 7 Dedan A-DA^3 view of agent automata shown in Fig. 3

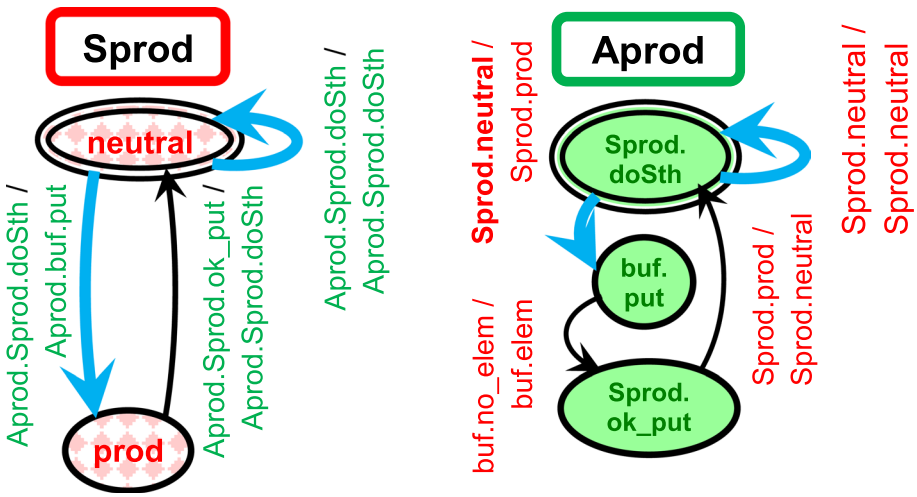

Fig. 8 Nondeterministic S-DA^3 and A-DA^3. Nondeterministic transitions are bold blue arrows. Note that these automata never coexist in the same view; they are parts of two alternative views of the system

are provided: imperative-style language Rybu (not covered in this article) and DA^3 automata.

In the graphic design, the user creates nodes and transitions by clicking on a window in desired coordinates. It should be emphasized that the automata type is being defined, and instances of this type can be declared in the system. New nodes are added by right-clicking on the surface (Fig. 17), a new node window appears as in Fig. 18 on the left (in this case, the state node, because it is S-DA^3). A transition is placed in the graph by clicking first on its source node, then on its target node, and then the label of this transition is defined in the right-hand window in Fig. 18. In this case, `AMP[1].markerE.tryM` is the transition input symbol and `AMP[1].lotE.try[2]` is the output symbol. The transition leads from `markerE.free` to `markerE.resM` state. The names of other servers/agents that appear in these windows are formal parameters of automata types. Typical editing operations are available, like moving nodes on the surface shown in Fig. 19.

The specified system is verified by searching for deadlocks or checking distributed termination (Fig. 20 on the left). The verification confirms desired features or presents a counterexample leading to an incorrect or desired configuration. In such a case, the designer analyzes the system under test, observing the counterexample as a sequence diagram in the server view (Fig. 20 center) or in agent view (Fig. 20 on the right) and inspecting the source code, graphical specification, and LTS fragments.

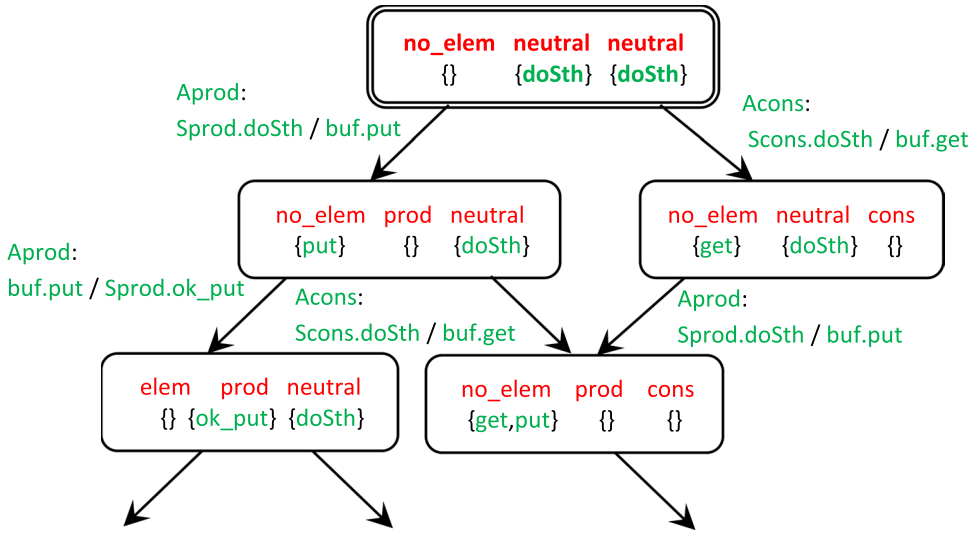

Fig. 9 The fragment of the global vertex space of server automata; the first row of a vertex contains the states of the servers, the second row the input sets

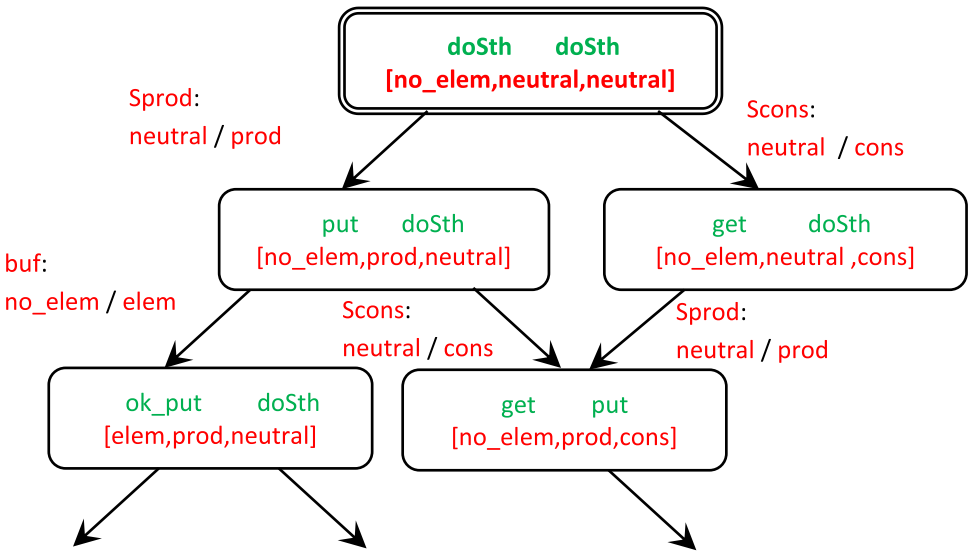

Fig. 10 The fragment of the global graph of agent automata; the first row of a vertex contains the agent messages, the second row the global input vector

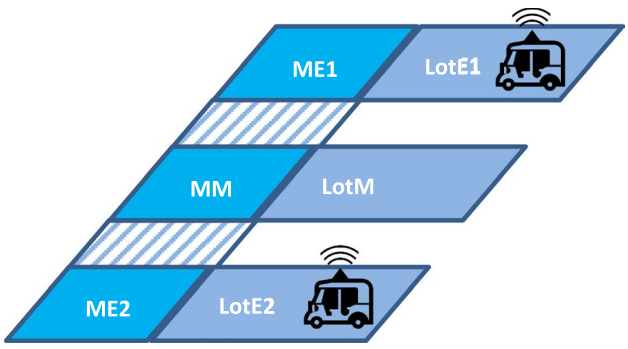

Fig. 11 The structure of road segment controllers. The AMPs are in their start positions

A distributed system can be simulated over the global space of configurations (LTS), but it can also be simulated in terms of DA[3], in the window illustrated in Fig. 6. All automata in the system are displayed, with input sets shown under the automata identifiers. The current states of the automata are blue, and all others are red.

The user can select the automaton (`Sprod` in the example, the selected automaton has a white background and blue name), then on the left side, a list of transitions outgoing from the current state of the selected automaton is displayed (enabled are highlighted; it is only one transition leading from `neutral` to `prod` in this case and it is enabled, with accepting of `doSth` message and issuing of the put message to `buf`). If the user clicks an enabled transition, it is "executed," and the destination automaton of the message becomes current (in this case: `buf`), as shown in Fig. 21.

Figure 22 shows such a situation after selecting `Scons` and executing a transition in it. This sends a message to

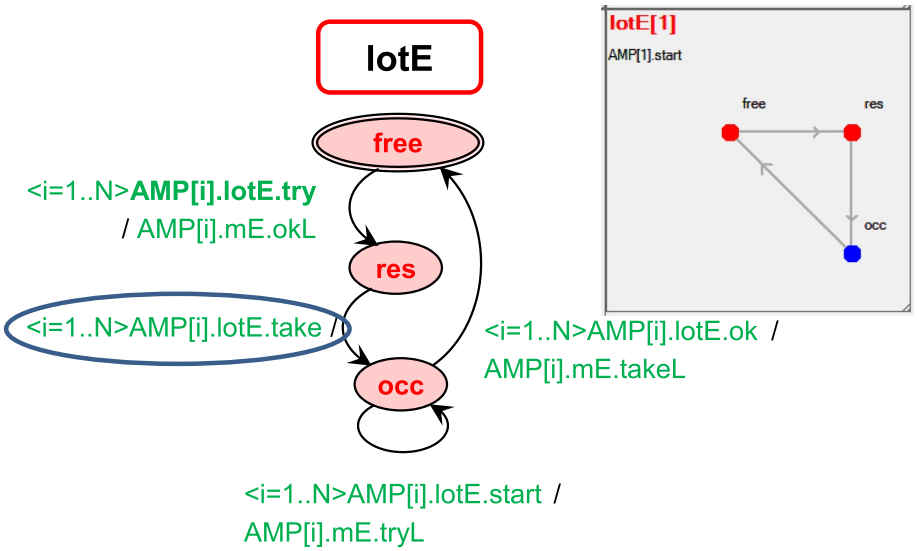

Fig. 12 The server automaton of `lotE` server type. There are two instances of this type, forming a 2-element vector `lotE[2]` (see the source code, line 59). A blue ellipse surrounds the agent-terminating transition label. The window of the Dedan program showing the automaton of the `lotE[1]` instance is shown on the right

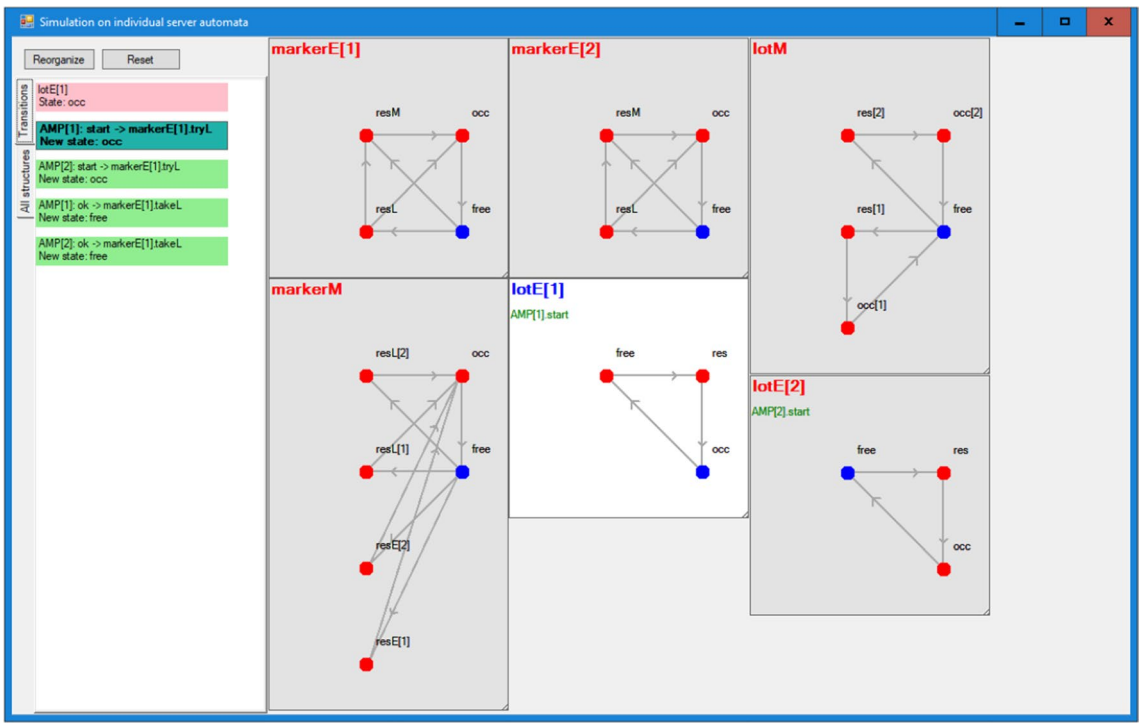

Fig. 13 The AVGS shown as S-DA^3. The chosen automaton is `LotE[1]` (light background and blue name), list of all transitions outgoing from the current state `occ` is on the left, with enabled transition highlighted (color figure online)

the `buf` automaton: now two messages are pending at this automaton. Of course, only one of them can be accepted, namely, the one invoking the action `{Aprod.buf.put, buf.no_elem}->{Aprod.Sprod.ok_put, buf.elem}`.

The deadlock situation is easy to identify. This is clearly shown on the sequence diagram (see Fig. 20 on the right). During the simulation, the deadlock is presented in Fig. 23: this is a version of the buffer system, in which users switch randomly between producer and consumer roles. For example, a deadlock occurs when all users choose to read from an empty buffer. This situation is shown in the figure: all messages are pending at the input of `buf`, but no action is enabled. Of course, it is a total deadlock. In the case of a partial deadlock, some automata work, so the deadlock should be observed in the comparison of the automata and

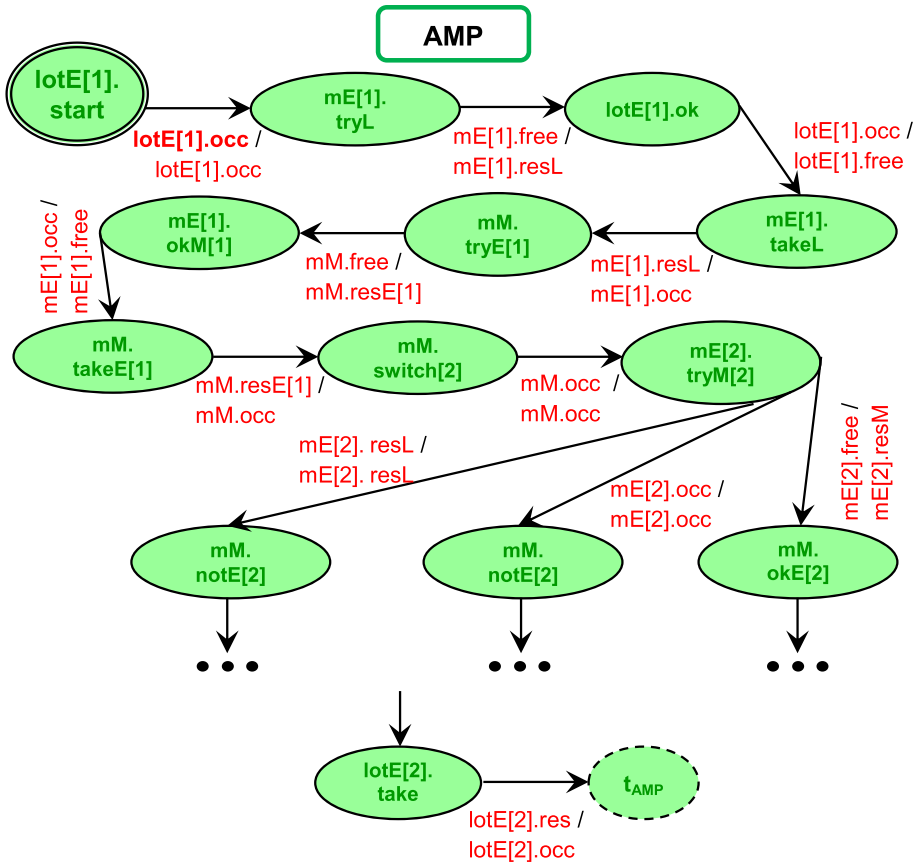

Fig. 14 The agent automaton of `AMP` agent type

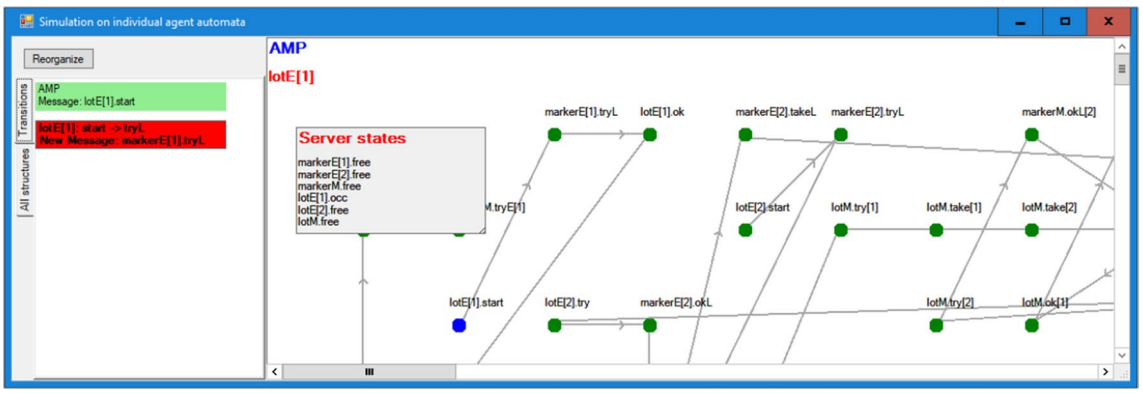

Fig. 15 The AVGS shown as A-DA^3. Server states—floating panel showing global input vector (of server states)

sequence diagram. In agent automata, the total deadlock is observed when every automaton has its own message, but neither automaton is enabled. Partial deadlock affects a subset of automata.

Figure 24 shows several examples of student exercises observed as S-DA^3 and A-DA^3.

9 Conclusions and further work

The Dedan program supports the engineer in the specification of distributed systems and their verification for deadlocks freeness and distributed termination. If a deadlock occurs, or a termination is checked, a sequence diagram of messages and states is generated, leading from the initial configuration to the deadlock/termination. If the deadlock/

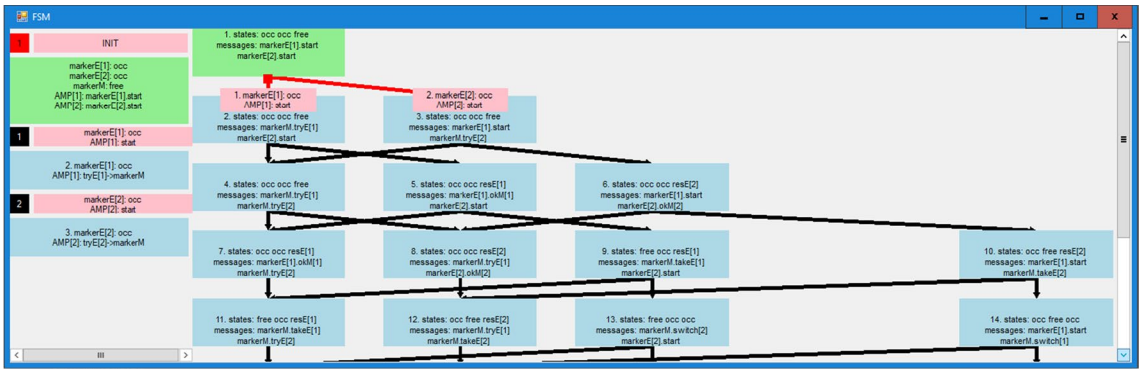

Fig. 16 Simulation in LTS. The current configuration is green, with states and messages shown without server and agent ids to save the space. The actions leading to its successors are red arrows, and input items of the actions are on the pink background. On the left, the output items of every action are displayed (color figure online)

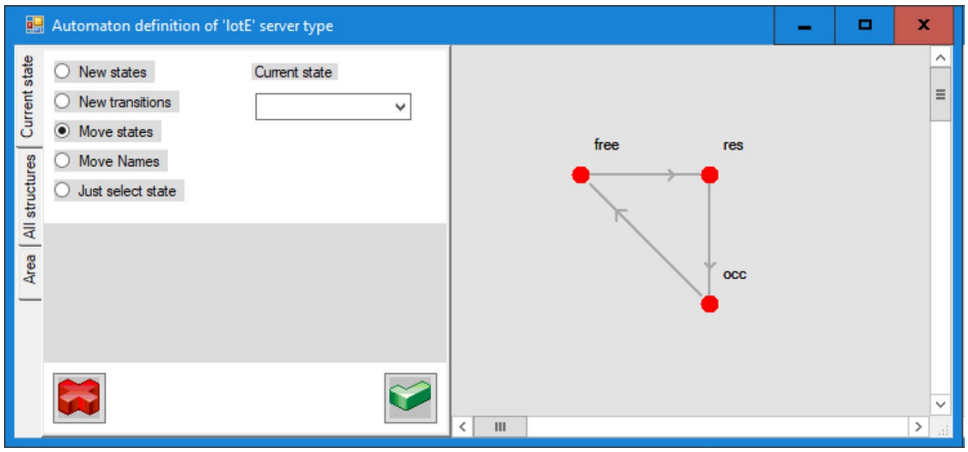

Fig. 17 The window for defining the position of nodes on the plane (S-DA^3 automaton type)

termination is not total, the involved servers/agents are shown. Distributed automata (in S-DA^3 or in A-DA^3 version) allow for the design of the system in graphical form, and to simulate system components and their interaction instead of simulating on the full configuration graph (LTS). Figures 6, 21, 22 show the simulation of the buffer system in Dedan. Also, a counterexample can be observed as a sequence of transitions in the cooperating DA^3 automata. Engineers are familiar with the concept of automata (S-DA^3 are similar to Message Passing Automata [43][44] and asynchronous statecharts [14], and A-DA^3 are like Timed Automata with global variables of Uppaal [5]), which can be naturally used in distributed systems design. For example, some models of transport cases were modeled. The server view is equivalent to the exchange of messages between road segment controllers that automatically lead the vehicles through the road segments [59]. In the agent view, it is the observation of vehicles moving on the road, with interactions to other vehicles occupying some segments of the road, via their controllers. Possible deadlocks in communication can be easily identified, and the verifier shows the behavior of vehicles leading to a deadlock as transitions of DA^3 automata. Table 1 compares the features of a distributed system, observed in equivalent formalisms: IMDS and DA^3.

Several years of Dedan development showed possible improvements in using DA^3 in both forms:

- Translation of graphic BPMN diagrams to IMDS allows for verification of them against partial deadlocks and checking partial termination (which is rare between verifiers). Especially, simulation of counterexamples in DA^3 with simultaneous observation of the current state in source BPMN is winning. ICS students carry out this project, but its scope exceeds this article. This successful project showed that specification in graphic domain specific languages (DSL) could be useful, when the designer can compare the counterexample in the form of sequence diagram with simulation in DA^3 and with the current state shown in source specification. Currently, DA^3 specifications for autonomous vehicles and web-service com-

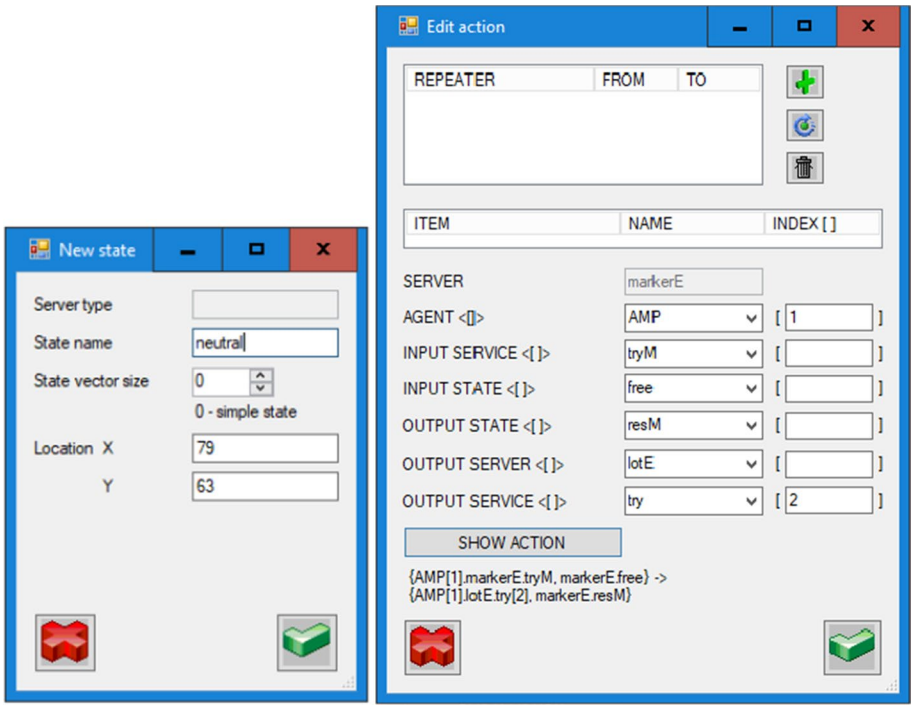

Fig. 18 Defining the state—the node in S-DA^3 and action, i.e., the transition in both S-DA^3 and A-DA^3

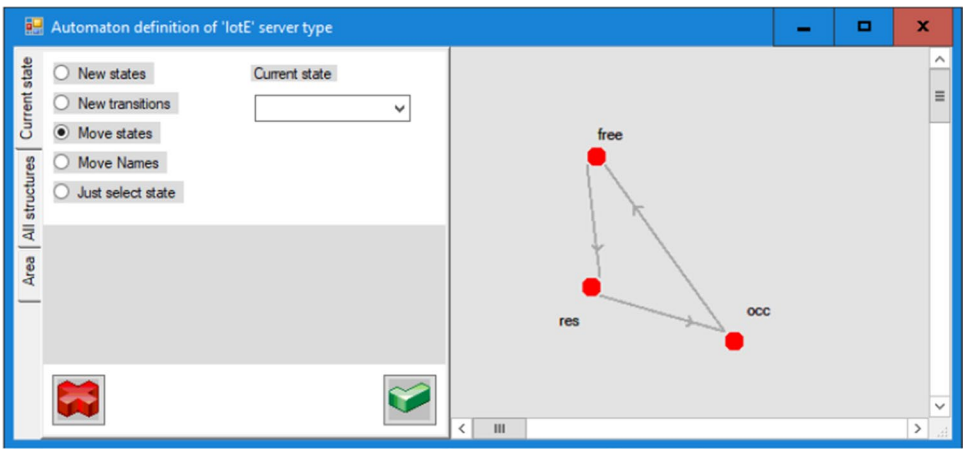

Fig. 19 Moving nodes on a plane: click a node and then click at its new location. It is the same S-DA^3 automaton type (`lotE`) as before, only with different node positions

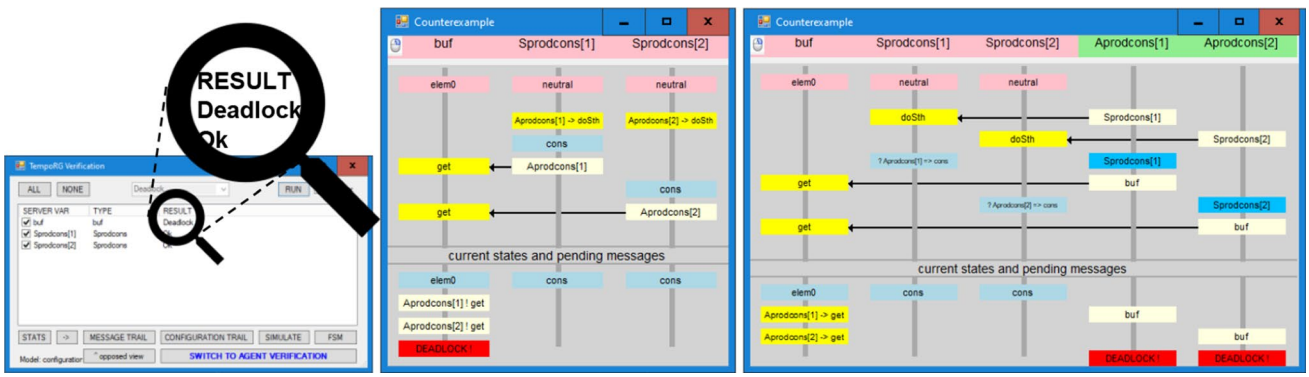

Fig. 20 The verification window (left), the sequence diagram of found communication deadlock in server `buf` (center) and the same situation seen in resource deadlock in the agent view of user agents (right)

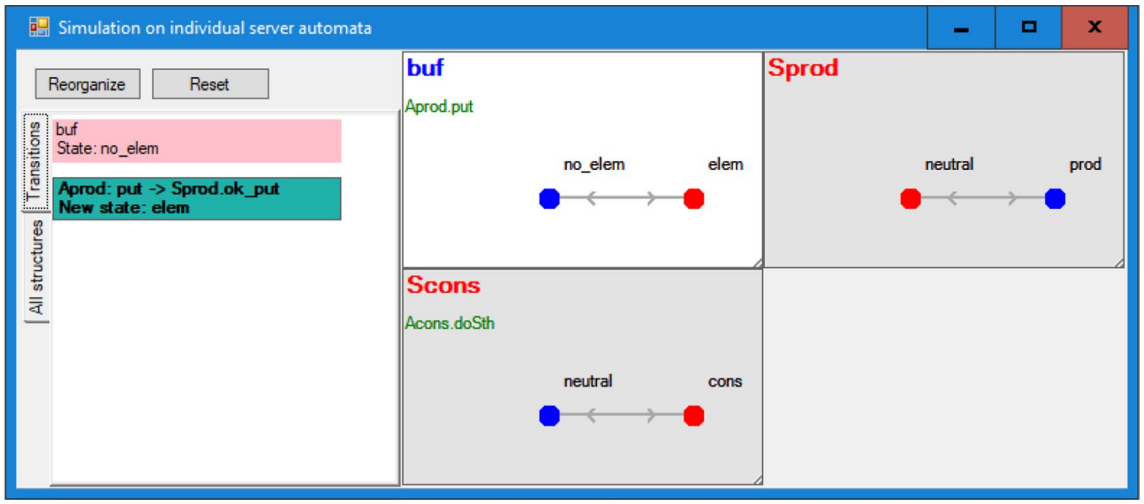

Fig. 21 Simulation over S-DA^3 after executing the transition in the automaton `Sprod`, highlighted in Fig. 3: `{Aprod.Sprod.doSth, Sprod.neutral}->{Aprod.buf.put, Sprod.prod}`. The target automaton of the action is buf, and it becomes the current one

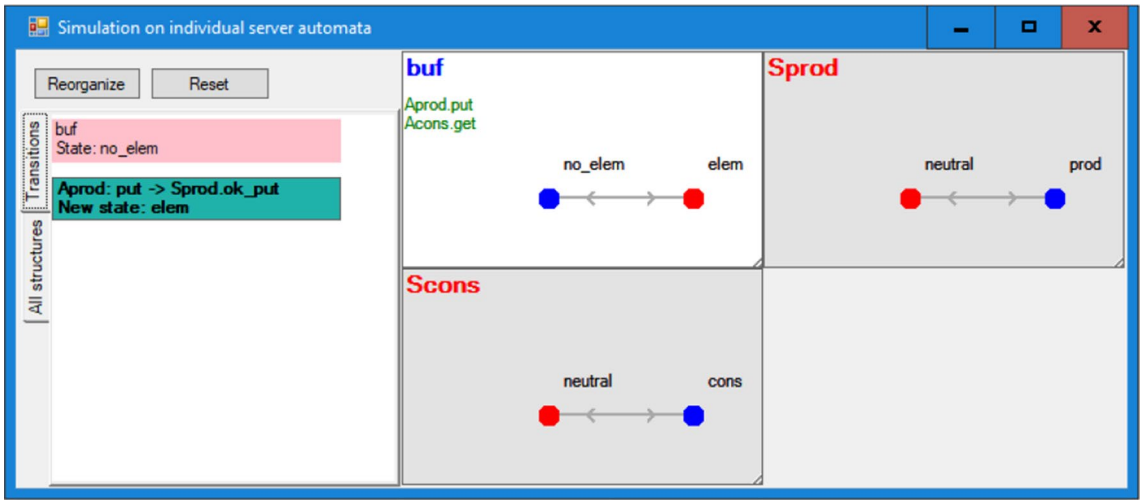

Fig. 22 Simulation over S-DA^3 after manually changing the current automaton to `Scons` and executing the transition in the automaton `Scons`. Now the automaton `buf` is current, and two messages pending at this automaton are displayed: `Aprod.put` and `Acons.get` (automaton name in the messages is omitted, because in both cases it is `buf`)

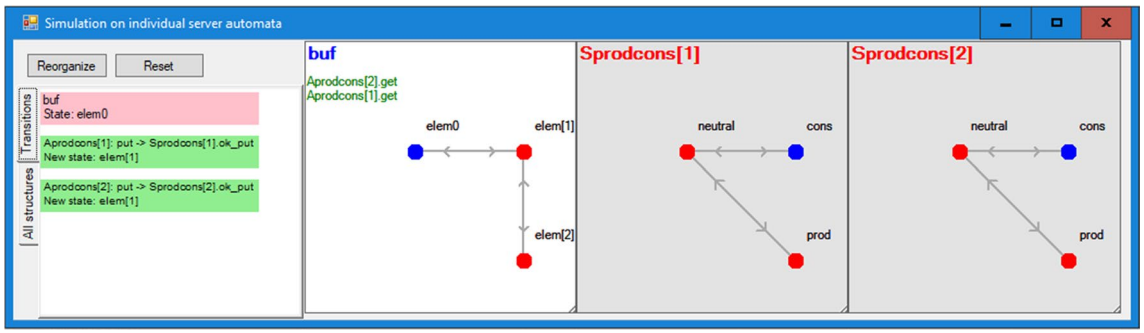

Fig. 23 There is no move: there are two messages in the `buf` server, both do not match the state of this node, and there is no other enabled agent in the system. If such an agent existed, it could potentially send a message to the `buf` server that would match the `elem0` state and cause an action to change that state. It is a graphical interpretation of the deadlock in the system

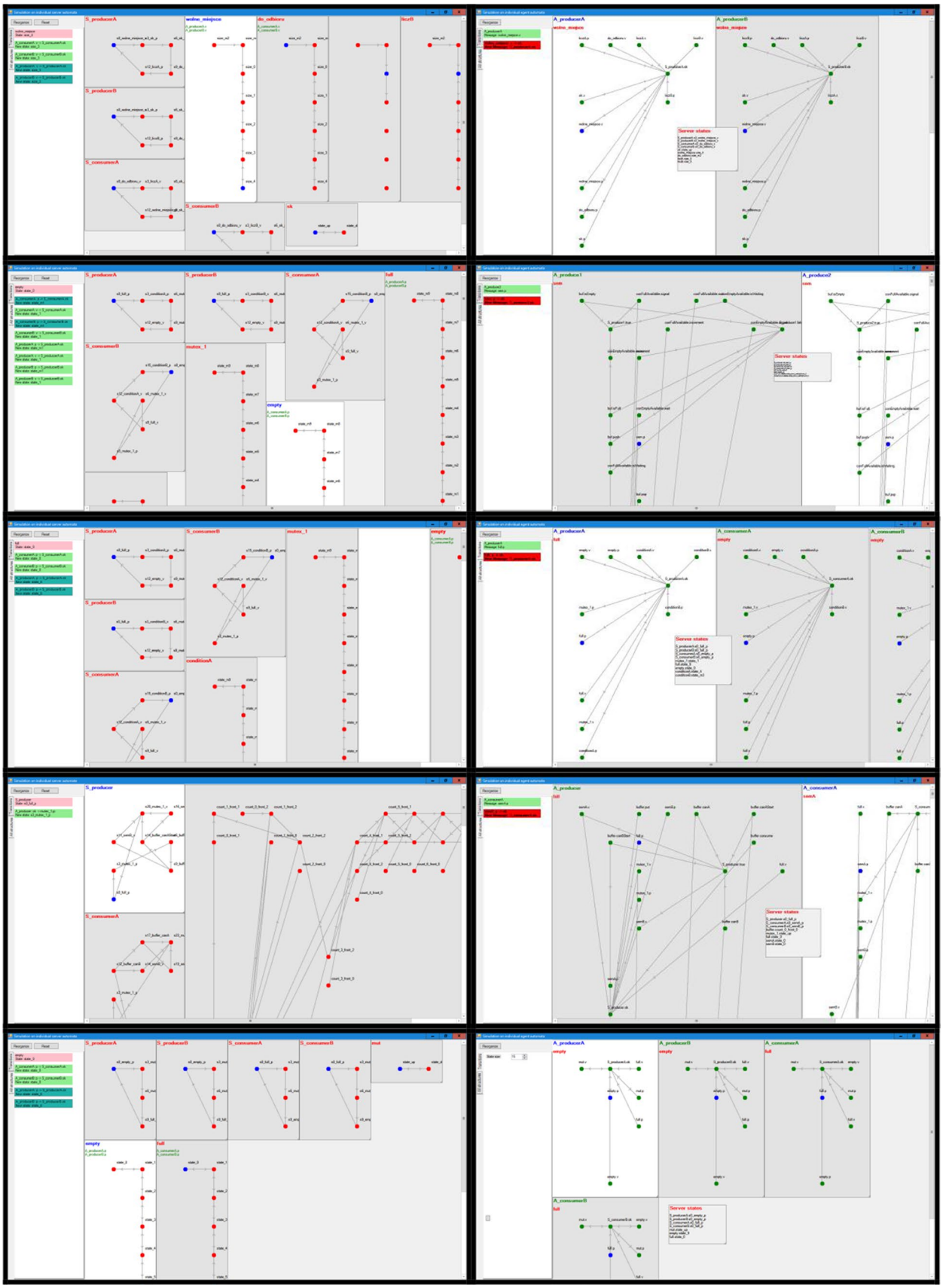

Fig. 24 Several examples of student exercises shown in S-DA^3 (left) and A-DA^3 (right) views

Table 1 Verification facilities in the three equivalent formalisms

Formalism:	IMDS	S-DA^3	A-DA^3
Main features	Specification, model checking, simulation	Graphic input, simulation	Graphic 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 vertex	Global vertex
	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 vertex space: all states and input sets in global vertices, input and output symbols on transitions	Global vertex space: all messages and global input vector in global vertices, 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	Graphic definition of a system (as servers) Simulation over individual server automata Counterexample projected onto individual server automata Counterexample-guided simulation	Graphic definition of a system (as agents) Simulation over individual agent automata Counterexample projected onto individual agent automata Counterexample-guided simulation

position, which are being developed by our students, are to be considered as kinds of DSL for these domains. In our laboratory, we are working on DSL for the digital twin specification for railway signaling. The important feature of these projects is a simultaneous observation of sequence diagram counterexample, DA^3 simulation, and the state of source specification.

- In the simulation of DA^3, some new features are planned, like starting the simulation from a configuration pointed in the sequence diagram, backtracking the simulation, and diverging from counterexample simulation to free choice of enabled transitions.
- During the simulation, we plan to show in separate windows a few kinds of counterexample tracking simultaneously: DA^3 concurrently in the two forms, sequence diagram and configuration graph (LTS)—Figs. 16, 20, 21, 23, respectively. A move (or backward move) in one of the windows would cause similar moves in the other windows. In the case of a DSL specification mentioned above, simultaneously, the current state in the source graphical specification can be shown.

In addition, some new features in the development of IMDS and the Dedan program are planned; some of them are subject of student projects:

- Non-exhaustive timeless and timed verification, using 2-vagabonds algorithm and various types of heuristics [65]. This could allow for verification of huge systems, whose LTS cannot be represented in the computer memory.
- Probabilistic DA^3 automata to identify deadlock probability 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 for the agent view (exploiting traveling of agents and resource sharing in a distributed environment); the initial version of the declarative language-based preprocessor Rybu for the server view of verified systems is developed by the students of ICS, WUT (Institute of Computer Science, Warsaw University of Technology) [9]. Rybu is not covered in this paper, as it is based on textual representation.
- Agents' own actions—equipping the agents with their own sets of actions, carried in their "backpacks," parameterizing their behavior; this will allow modeling 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 the student laboratory in ICS, WUT. Students verify their solutions to synchronization problems. The graphic definition of the component automata and simulation over distributed automata supports the verification procedure.

Appendix. 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. Equivalence is based on similar LTS structures, i.e., each vertex of LTS—including initial one—should contain the same items (states and messages) as the corresponding vertices in other LTS'es. This means that the corresponding vertices should contain exactly the same sets of elements, but in various forms: a set of items (messages and states) in IMDS, automata nodes and elements of input sets of S-DA^3, nodes and elements of the input vector in A-DA^3, except for terminating nodes which are absent in IMDS and in S-DA^3. Furthermore, transitions should connect the respective vertices in all three LTS'es. To show equivalence, the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$ for the action λ is needed. This rule comes directly from the IMDS definition:

$$\begin{aligned}&\forall_{\lambda\in\Lambda}\lambda=\left((m,p),\left(m^{'},p^{'}\right)\right)T_{inp}(\lambda)\supset\{m,p\}\Rightarrow T_{out}(\lambda)=T_{inp}(\lambda)\backslash\{m,p\}\cup\left\{m^{'},p^{'}\right\}\\&\forall_{\lambda\in\Lambda}\lambda=\left((m,p),\left(p'\right)\right)T_{inp}(\lambda)\supset\{m,p\}\Rightarrow T_{out}(\lambda)=T_{inp}(\lambda)\backslash\{m,p\}\cup\left\{p'\right\}\end{aligned}\tag{A.1}$$

1. LTS vertex
   - IMDS: $T=\{\ p_1, \ldots, p_n, m_1, \ldots, m_x \mid p_i\in P, m_j\in M, x\le k\}$, Every p from a different s, every m from a different a except for terminated a,
   - S-DA^3: $T_\Psi=((p_1, X_1), \ldots,(p_n, X_n))$, Each p from different s—from definition in T_Ψ each s participates (will be shown later). For each pair X_i, X_j, $i\neq j$ both cannot contain m appointing the same a (will be shown later). No X can contain m appointing the terminated a (will be shown later),
   - A-DA^3: $T_\Cup=\{m,m', m'', \ldots,Y\}$, Each m appoints different a—from definition in $T_\Cup$ each a participates, except for terminated ones (will be shown later). For each pair $m_i\neq m_j$, both cannot appoint the same a (will be shown later). No m can appoint the terminated a (will be shown later). Y contains the states of all servers—from definition.
2. Initial LTS vertex
   - IMDS: $T_0=\{p_{01}, \ldots, p_{0n}, m_{01}, \ldots,m_{0k} \mid p_{0i}\in P_0, m_{0j}\in M_0\}$,
   - S-DA^3: $T_{\Psi 0}=((p_{01}, X_{01}), \ldots, (p_{0n}, X_{0n}))$, Each p from different s—from definition. Each s participates—from definition. For each pair X_{0i}, X_{0j}, $i\neq j$, 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 $\forall a_i\in A$: $card(M_i\cap M_0)=1$ (the initial set of messages contains exactly one message for every agent). Each a has its m_0 in some X_{0i}—this from which the agents start (X_{0i} are indexed by servers and every m_0 belong to some M_i).
   - A-DA^3: $T_\Cup=\{m_{01}, \ldots,m_{0k}, Y_0\}$, In each pair $m_{0i}\neq m_{0j}$, both cannot appoint the same a (from definition, as $\forall a_i\in A$: $card(M_i\cap M_0)=1$ (the initial set of messages contains exactly one message for every agent). No m can be t_v—from definition. Every element of Y_0 appoints different server—from definition.
3. Transition in LTS (progress action)
   - IMDS: $(T_{inp}(\lambda), \lambda, T_{out}(\lambda)) \mid \lambda\in\Lambda, \lambda=((m, p), (m', p'))$
   - S-DA^3: From T_Ψ there exists a progress transition (p, m/m', p') to T'_Ψ corresponding to $\lambda=((m, p), (m', p'))$, in the automaton ψ_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_Ψ corresponds to $T_{inp}(\lambda)$, then according to (6.1) in T'_Ψ 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_Ψ and T'_Ψ, which fulfils the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$.

     For each progress action $\lambda=((m, p), (m', p'))$ having p on input, such a transition (p, m/m', p') exists in automaton ψ_i appointed by p, and no other transition exists in ψ_i with the same p, m, m', p', so it exactly corresponds to the set of progress actions having p on input.

     In each progress transition in ψ_i, the set of servers is preserved (as p and p' appoint the same server) and the set of agents—all except terminated ones—is preserved (as m and m' appoint the same agent).
   - A-DA^3: From $T_\Cup$, there exists a progress transition (m, p/p', m') to $T'_\Cup$ corresponding to $\lambda=(m, p), (m', p'))$, in the automaton v_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 $T_\Cup$ corresponds to $T_{inp}(\lambda)$, then according to (6.3) in $T'_\Cup$ m is replaced by m' and p in Y is replaced by p', and all other states and messages are equal in

$T_ŧ$ and $T'_ŧ$, which fulfils the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$.

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

In every progress transition in $ŧ_i$, the set of agents—all except terminated ones—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 action, or simply terminating action)

- IMDS: $(T_{inp}(\lambda), \lambda, T_{out}(\lambda)) \mid \lambda \in \Lambda, \lambda = ((m, p), (p'))$
- S-DA^3: From T_Ψ there exists a terminating transition $(p, m/, p')$ to T'_Ψ corresponding to $\lambda = ((m, p), (p'))$, in the automaton ψ_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_Ψ corresponds to $T_{inp}(\lambda)$, then according to (6.2) in T'_Ψ p is replaced by p' and m is extracted, and all other states and messages are equal in T_Ψ and T'_Ψ, 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'_Ψ is smaller by the agent appointed by m. Consequently, there is no way to reestablish a terminated agent.
- A-DA^3: From $T_ŧ$, there exists an agent a terminating transition $(m, p/p', t_ŧ)$ to $T'_ŧ$ corresponding to $\lambda = ((m, p), (p'))$, in the automaton $ŧ_i$ of the agent a_i appointed by m, to the message m', replacing the message m by $t_{ŧi}$.

  The state p belongs to a server s. If $T_ŧ$ corresponds to $T_{inp}(\lambda)$, then according to (6.4) in $T'_ŧ$ m is replaced by $t_{ŧi}$ and p is replaced in Y by p', all other states and messages are equal in $T_ŧ$ and $T'_ŧ$, which fulfils the rule of obtaining $T_{out}(\lambda)$ from $T_{inp}(\lambda)$ (terminating action).

  For each terminating action $\lambda = ((m, p), (p'))$ having m on input such a transition $(m, p/p', t_{ŧi})$ exists in automaton $ŧ_i$ 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 $ŧ_i$, the set of servers is preserved (as p and p' appoint the same server) and the set of agents in $T'_ŧ$ is smaller by the agent appointed by m, which is replaced by $t_{ŧi}$. Consequently, there is no way to reestablish a terminated agent.

5. Semantics of LTS execution,

- IMDS: interleaved, nondeterministic,
- S-DA^3: interleaved, nondeterministic,
- A-DA^3: interleaved, nondeterministic.

Author's contributions Development of IMDS and DA^3, development of the Dedan framework, supervision of students' works on the framework.

Funding This research received no external funding. The work is performed as an education project.

Availability of data and material Examples of the IMDS systems in the source code are available at http://staff.ii.pw.edu.pl/dedan/files/examples.zip.

Code availability The Dedan program is available online at http://staff.ii.pw.edu.pl/dedan/files/DedAn.zip.

Declarations

Conflict of interest The author declares no conflict of interest.

References

1. Daszczuk, W.B.: Specification and verification in integrated model of distributed systems (IMDS). MDPI Comput. **7**, 1–26 (2018). https://doi.org/10.3390/computers7040065
2. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**, 279–295 (1997). https://doi.org/10.1109/32.588521
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
4. Lanese, I., Montanari, U.: Hoare vs Milner: comparing synchronizations in a graphical framework with mobility. Electron. Notes Theor. Comput. Sci. **154**, 55–72 (2006). https://doi.org/10.1016/j.entcs.2005.03.032
5. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing UPPAAL over 15 years. Softw. Pract. Exp. **41**, 133–142 (2011). https://doi.org/10.1002/spe.1006
6. May, D.: OCCAM. ACM SIGPLAN Not. **18**, 69–79 (1983). https://doi.org/10.1145/948176.948183

7. Lutz, M.J.: Alloy, software engineering, and undergraduate education. In: ACM SIGSOFT First Alloy Workshop. Portland, Oregon, 6 Nov. pp. 1–2. ACM, New York, NY (2006)
8. Corbett, J.C., Dwyer, M.B., Hatcliff, J.: Roby: Bandera: extracting finite-state models from Java source code. In: 22nd International Conference on Software Engineering—ICSE '00, Limerick, Ireland, 9 June 2000. pp. 762–765. IEEE (2000). https://doi.org/10.1145/337180.337625.
9. Daszczuk, W.B., Bielecki, M., Michalski, J.: Rybu: imperative-style preprocessor for verification of distributed systems in the Dedan environment. In: KKIO'17—Software Engineering Conference, Rzeszów, Poland, 14–16 Sept. 2017. pp. 135–150. Polish Information Processing Society (2017).
10. Jia, W., Zhou, W.: Distributed network systems. from concepts to implementations. NETA vol. 15, Springer, New York (2005). https://doi.org/10.1007/b102545.
11. 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). https://doi.org/10.1109/CEC.2014.6900622.
12. Lauer, H.C., Needham, R.M.: On the duality of operating system structures. ACM SIGOPS Oper. Syst. Rev. **13**, 3–19 (1979). https://doi.org/10.1145/850657.850658
13. Daszczuk, W.B.: Distributed Autonomous and Asynchronous Automata (DA^3). In: Kacprzyk, J. (ed.) Integrated Model of Distributed Systems, pp. 125–137. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-12835-7_8
14. Ziadi, T., Helouet, L., Jezequel, J.-M.: Revisiting statechart synthesis with an algebraic approach. In: 26th International Conference on Software Engineering, Edinburgh, UK, 28 May 2004. pp. 242–251. IEEE Comput. Soc (2004). https://doi.org/10.1109/ICSE.2004.1317446.
15. Lodaya, K.: A regular viewpoint on processes and algebra. Acta Cybern. **17**, 751–763 (2006)
16. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, Cambridge (2009). https://doi.org/10.1017/CBO9781139195218
17. Phawade, R.: Kleene theorems for free choice automata over distributed alphabets. In: Koutny, M., Pomello, L., and Kristensen, L.M. (eds.) Transactions on Petri Nets and Other Models of Concurrency XIV, LNCS vol. 11790. pp. 146–171. Springer, Berlin (2019). https://doi.org/10.1007/978-3-662-60651-3_6
18. Morales, L.E.M.: Specifying BPMN diagrams with Timed Automata: Proposal of some mapping rules. In: 9th Iberian Conference on Information Systems and Technologies (CISTI), Barcelona, Spain, 18–21 June 2014. pp. 1–6. IEEE (2014). https://doi.org/10.1109/CISTI.2014.6876897.
19. Zhou, Y., Baresi, L., Rossi, M.: Towards a formal semantics for UML/MARTE state machines based on hierarchical timed automata. J. Comput. Sci. Technol. **28**, 188–202 (2013). https://doi.org/10.1007/s11390-013-1322-8
20. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**, 329–366 (2004). https://doi.org/10.1017/S0960129504004153
21. Martín-Vide, C., Mateescu, A., Mitrana, V.: Parallel finite automata systems communicating by states. Int. J. Found. Comput. Sci. **13**, 733–749 (2002). https://doi.org/10.1142/S0129054102001424
22. Stotts, P.D., Pugh, W.: Parallel finite automata for modeling concurrent software systems. J. Syst. Softw. **27**, 27–43 (1994). https://doi.org/10.1016/0164-1212(94)90112-0
23. Poizat, P., Choppy, C., Royer, J.-C.: From informal requirements to COOP: a concurrent automata approach. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99—Formal Methods, Toulouse, France, 20–24 Sept 1999, LNCS vol. 1709. pp. 939–962. Springer, Berlin Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_1
24. Grosu, R., Rumpe, B.: Concurrent Timed Port Automata. Technical Report TUM-19533, TU Munich (1995)
25. Martin, O.B., Williams, B.C., Ingham, M.D.: Diagnosis as approximate belief state enumeration for probabilistic concurrent constraint automata. In: Cohn, A. (ed.) AAAI'05: Proceedings of the 20th national conference on Artificial intelligence, Pittsburgh, PA, 9–13 July 2005, Vol. 1. pp. 321–326. AAAI Press, Palo Alto, CA (2005).
26. Mieścicki, J.: The use of model checking and the COSMA environment in the design of reactive systems. Ann. UMCS, Inform. Vol. AI. 4AI, 244–253 (2006). https://doi.org/10.17951/ai.2006.4.1.244-253.
27. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Automata, Languages and Programming. pp. 322–335. Springer, Berlin/Heidelberg (1990). https://doi.org/10.1007/BFb0032042
28. Lewerentz, C., Lindner, T. eds: Formal Development of Reactive Systems, LNCS 891. Springer, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-58867-1.
29. Babaoğlu, Ö., Bartoli, A., Dini, G.: Enriched view synchrony: a programming paradigm for partitionable asynchronous distributed systems. IEEE Trans. Comput. **46**, 642–658 (1997). https://doi.org/10.1109/12.600823
30. Quaglia, P., Walker, D.: On Synchronous and Asynchronous Mobile Processes. In: Tiuryn, J. (ed.) FoSSaCS 2000: Foundations of Software Science and Computation Structures, Berlin, Germany, March 25–April 2, 2000, LNCS vol. 1784. pp. 283–296. Springer, Berlin Heidelberg (2000). https://doi.org/10.1007/3-540-46432-8_19.
31. Gorla, D.: Comparing communication primitives via their relative expressive power. Inf. Comput. **206**, 931–952 (2008). https://doi.org/10.1016/j.ic.2008.05.001
32. Rowstron, A.: WCL: a co-ordination language for geographically distributed agents. World Wide Web. **1**, 167–179 (1998). https://doi.org/10.1023/A:1019263731139
33. van Schuppen, J.H., Boutin, O., Kempker, P.L., Komenda, J., Masopust, T., Pambakian, N., Ran, A.C.M.: Control of distributed systems: tutorial and overview. Eur. J. Control. **17**, 579–602 (2011). https://doi.org/10.3166/ejc.17.579-602
34. Zielonka, W.: Notes on finite asynchronous automata. RAIRO Theor. Informatics Appl. **21**, 99–135 (1987). https://doi.org/10.1051/ita/1987210200991
35. Krishnan, P.: Distributed timed automata. Electron. Notes Theor. Comput. Sci. **28**, 5–21 (2000). https://doi.org/10.1016/S1571-0661(05)80627-9
36. 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). https://doi.org/10.1007/978-3-662-47666-6_2
37. 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). https://doi.org/10.1007/978-3-642-32621-9_5
38. 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). https://doi.org/10.1142/9789814271059_0009.
39. Sandholm, A.B., Schwartzbach, M.I.: Distributed Safety Controllers for Web Services. BRICS Rep. Ser. 4, (1997). https://doi.org/10.7146/brics.v4i47.19268.
40. Baumann, C., Schwarz, O., Dam, M.: On the verification of system-level information flow properties for virtualized execution

platforms. J. Cryptogr. Eng. **9**, 243–261 (2019). https://doi.org/10.1007/s13389-019-00216-4
41. Bollig, B., Grindei, M.-L., Habermehl, P.: Realizability of concurrent recursive programs. Form. Methods Syst. Des. **53**, 339–362 (2018). https://doi.org/10.1007/s10703-017-0282-y
42. 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**, 3–18 (2006). https://doi.org/10.1016/j.entcs.2005.10.015
43. 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). https://doi.org/10.1007/978-3-540-28644-8_10.
44. 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). https://doi.org/10.1007/11562436_6
45. Reiter, F.: Asynchronous distributed automata: a characterization of the modal mu-fragment. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017), Warsaw, Poland, 10–14 July 2017. pp. 100:1–100:14. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.100
46. Balan, M.S.: Serializing the parallelism in parallel communicating pushdown automata systems. Electron. Proc. Theor. Comput. Sci. **3**, 59–68 (2009). https://doi.org/10.4204/EPTCS.3.5
47. 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). https://doi.org/10.4204/EPTCS.161.9
48. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: 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). https://doi.org/10.1145/1926385.1926419.
49. Liu, T.: Computation in the wild: reconsidering dynamic systems in light of irregularity. http://cs.williams.edu/~bailey/Li16.pdf (2016)
50. Kutrib, M., Malcher, A.: Iterative arrays with finite inter-cell communication. In: Castillo-Ramirez, A., de Oliveira, P.P.B. (eds.) AUTOMATA 2019: Cellular Automata and Discrete Complex Systems, Guadalajara, Mexico, 26–28 June 2019. pp. 35–47. Springer, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-030-20981-0_3.
51. Beeck, M.: A comparison of Statecharts variants. In: FTRTFT 1994: Formal Techniques in Real-Time and Fault-Tolerant Systems, Lübeck, Germany, 19–23 Sept. 1994, LNCS vol. 863. pp. 128–148. Springer, Berlin Heidelberg (1994). https://doi.org/10.1007/3-540-58468-4_163
52. Balanescu, T., Cowling, A.J., Georgescu, H., Gheorghe, M., Holcombe, M., Vertan, C.: Communicating stream X-machines systems are no more than X-machines. J. Univers. Comput. Sci. **5**, 494–507 (1999). https://doi.org/10.3217/jucs-005-09-0494
53. 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). https://doi.org/10.1109/ICASSP.2005.1416243.
54. 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, Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-18381-2_41
55. Penczek, W., Szreter, M., Gerth, R., Kuiper, R.: Improving partial order reductions for universal branching time properties. Fundam. Informaticae. **43**, 245–267 (2000). https://doi.org/10.3233/FI-2000-43123413
56. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York (1992). https://doi.org/10.1007/978-1-4612-0931-7
57. Daszczuk, W.B.: Deadlock detection examples: the Dedan environment at work. In: Integrated Model of Distributed Systems. pp. 53–85. Springer Nature, Cham, Switzerland (2020). https://doi.org/10.1007/978-3-030-12835-7_5
58. Daszczuk, W.B.: Asynchronous specification of production cell benchmark in integrated model of distributed systems. In: Bembenik, R., Skonieczny, L., Protaziuk, G., Kryszkiewicz, M., 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). https://doi.org/10.1007/978-3-319-77604-0_9
59. 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**, 1294–1301 (2016)
60. Daszczuk, W.B.: Fairness in temporal verification of distributed systems. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) 13th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, 2–6 July 2018, Brunów, Poland, AISC vol.761. pp. 135–150. Springer International Publishing, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-319-91446-6_14
61. Daszczuk, W.B.: Static and dynamic verification of space systems using asynchronous observer agents. Sensors. **21**, 1–24 (2021). https://doi.org/10.3390/s21134541
62. Lutz, M.J.: Modeling software the Alloy way. In: 2013 IEEE Frontiers in Education Conference (FIE), Oklahoma City, OK, 23–26 Oct. 2013. p. 3. IEEE (2013). https://doi.org/10.1109/FIE.2013.6684771
63. Abdul-Hussin, M.H.: Elementary siphons of petri nets and deadlock control in FMS. J. Comput. Commun. **3**, 1–12 (2015). https://doi.org/10.4236/jcc.2015.37001
64. Daszczuk, W.B.: Timed IMDS. In: Integrated Model of Distributed Systems. pp. 161–192. Springer Nature, Cham, Switzerland (2020). https://doi.org/10.1007/978-3-030-12835-7_10
65. Daszczuk, W.B.: 2-Vagabonds: non-exhaustive verification algorithm. In: Integrated Model of Distributed Systems. pp. 193–218. Springer, Cham, Switzerland (2020). https://doi.org/10.1007/978-3-030-12835-7_11

Wiktor B. Daszczuk was born in Olsztyn, Poland, in 1958. He received the joint Eng. and M.S. degree in computer science and technology from Warsaw University of Technology, Warsaw, Poland, in 1982 and the Ph.D. degree in computer science from the same university in 2003. He received the D.Sc. degree in 2020. From 1982 to 1990 he worked in the industry, from 1990 to 2003 he was a Research Assistant with the Institute of Computer Science, Warsaw University of Technology, and from 2003 he has been an Assistant Professor in the same institute. His research interest includes the specification and verification of distributed systems and modeling of urban transport systems. He is the author of more than 70 publications, including a monograph, articles, conference papers and research reports. He leaded the development of PowerWatch plant monitoring system and Feniks simulator of autonomous transport systems. Dr. Daszczuk received Siemens Prize for Applied Research in 1997 and Polish Prime Minister Award for Scientific Research in Eco-Mobility project in 2016.