"The Computer Journal", vol. 60 (2017), No. 5, pp. 729-750, DOI: 10.1093/comjnl/bxw099

Communication and Resource Deadlock Analysis using IMDS Formalism and Model Checking

Wiktor B. Daszczuk

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

Modern static deadlock detection techniques deal with the global properties of the verified systems, using methods that explore the state space. Local features, like partial deadlocks or individual process terminations, are not easily expressed or checked by such methods. Also the distinction between communication deadlocks and resource deadlocks, common in dynamic waits-for methods, cannot be addressed or verified by static methods. An Integrated Model of Distributed Systems (IMDS) is proposed which specifies distributed systems as sets of servers' states, sets of messages and sets of actions. The message passing/resource sharing dualism of distributed systems is provided by projections: on servers (server view) and on agents (agent view), yet the uniform specification of a verified system is preserved. A progress of computation is defined in terms of actions which change (local) states and generate messages.

Distributed actions do not depend on global states and are independent from one another. Therefore local features of subsystems can be easily described in IMDS. Communication and resource deadlocks can be handled separately and total and partial deadlocks and terminations can be distinguished from each other. Integration of IMDS with model checking is outlined and temporal formulas for deadlock and termination checking are discussed.

Keywords: Deadlock Detection; Distributed Termination; Distributed Systems; Communication Dualism; Model Checking

Received 00 March 2016; revised 00 Month 2016

1. INTRODUCTION

A distributed system is typically described in terms of servers exchanging messages. A process in such a system is defined as a sequence of changes of a server state. The states of servers are internal to the processes, which communicate by message passing. This is a classical description of a distributed system, understood by many authors as natural: servers cooperating by messages (a client-server model) [1].

Yet, another model is possible: if a process is associated with a travelling agent, it migrates between servers and performs steps of its calculations in distinct servers. An agent communicates with other travelling agents by servers' states. Messages are internal to a process. In such a way, a system is described in terms of resource sharing instead of message passing. This is similar to a Remote Procedure Calling model (RPC [1], yet, in general it is not necessary for a process to return to the calling server).

The crucial fact is that it is *the same* system, shown in one of the two views, depending on the manner of connecting the actions. If a sequence of actions is connected by server states – it is a server view. Server states are the carrier of the process, while messages are the means of communication. If a sequence is connected by messages – it is an agent view. Messages are the carrier of the process, while servers' states serve to communicate processes. The two views are decompositions showing message passing and resource sharing aspects of a distributed system. Fig. 1 presents a metaphor for the decompositions: it is a Polish spiral cake called *strucla*. If the cake is cut crosswise – an observer sees a spiral. But if the cake is cut along – it presents itself as a layered cake. But still it is the same cake. The two kinds of strucla cutting may be named strucla decompositions.

We propose the original Integrated Model of Distributed Systems formalism (IMDS [2]) for a specification of cooperation in distributed environment. Informally, IMDS is based on states, messages and actions. A message invokes a service at a server. The

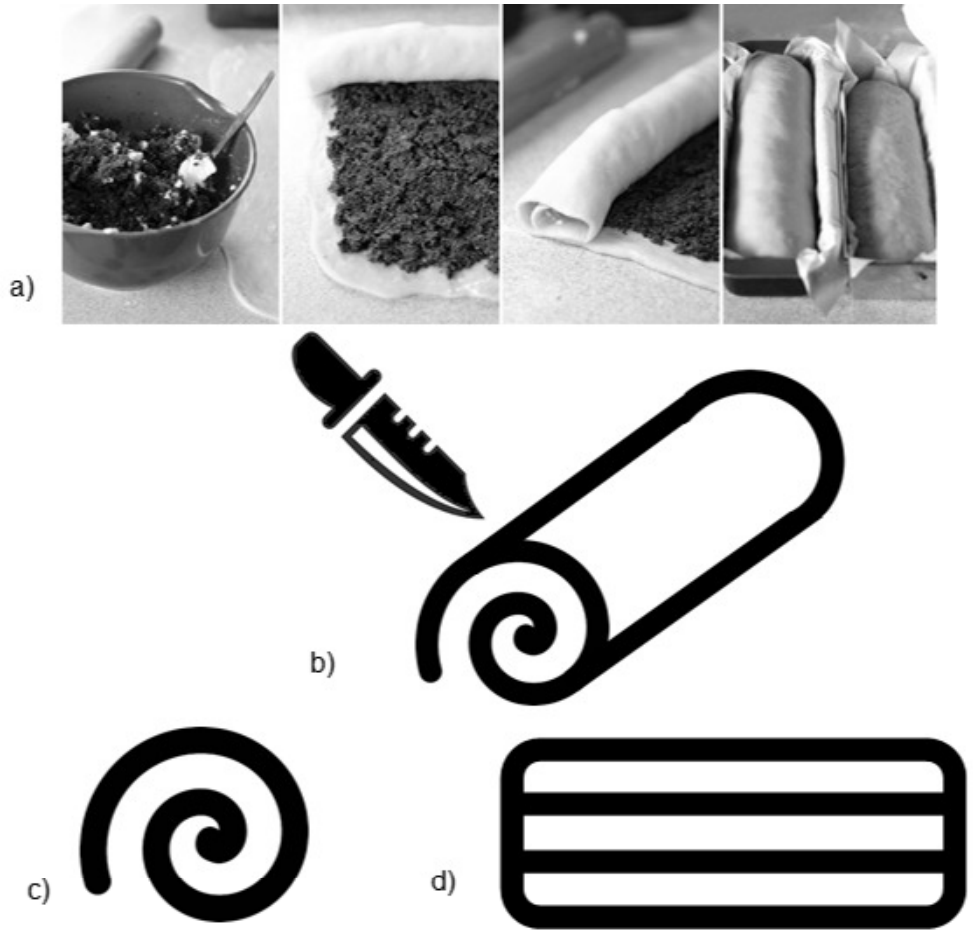

FIGURE 1. Strucla decompositions (cuts): a) preparation of strucla, b) cutting the cake, c) crosswise cut - spiral cake, d) along cut - layered cake.

execution of a service is called an action. If the server state allows for the execution of the service (we say that the state and the message *match*), the server may perform an action, i.e., execute a service. The action changes the server's state to another one and issues a next message, invoking a service at another server. Formally, a server's state is a pair *p=(server, value)* and a message is a triple *m=(agent, server, service)*. An agent is an identifier distinguishing a distributed computation from other computations. An action is a relation λ between an input pair *(m,p)* and an output pair *(m',p')*: $(m,p)\lambda(m',p')$. A process is a sequence of actions: in the same server (server process) or in the same agent (agent process).

Processes of a concurrent system may fall into a deadlock. An example of deadlock in a server view is when two servers wait for messages from each other. It is a communication deadlock, as the manner of process cooperation is message passing. A communication deadlock is a situation in which there are pending messages (at least one) on a server, but they cannot be served. A deadlock may concern a single server or more than one server – not all servers need to be involved.

In an agent view, a deadlock is a situation in which an agent has a message pending on some server, but the agent cannot make any progress (the message cannot be served, but messages of other agents possibly can). Again, a deadlock may concern a subset of system agents.

A termination in IMDS is simply a disappearing of an agent: a special termination action has the form $(m,p)\lambda(p')$. There is a new state of the server but there is no new message.

We propose a connection of our IMDS formalism with model checking: this coupling allows the designer to specify a distributed system, to observe its server and agent views of it and to easily find deadlocks and distributed termination of processes. The system may not work correctly: a deadlock is possible or a desired termination may not occur. A model checker can automatically detect those situations. The model checker creates a counterexample for each program defect, which supports the designer. A counterexample is a sequence of states and messages leading from an initial situation to an erroneous situation.

The paper is organized as follows: in Section 2 some

deadlock detection techniques are presented and their disadvantages are discussed. Section 3 introduces the IMDS formalism: the static description of distributed system's state by states and messages and the behaviour of distributed system in terms of actions. The semantics of distributed system behaviour is introduced as a *labelled transition system* (LTS [3]) over the actions. Deadlock and termination are defined in terms of LTS. This LTS is a basis of temporal model checking with our Dedan environment, described in Section 4. An example of a distributed system and its verification under Dedan is presented in Section 5. The future development of Dedan is described in Section 6.

2. RELATED WORK

First attempts to deadlock detection considered the global state space of a centralized system (or rather its model) [4] by analysis of a graph of dependencies called "Wait-for Graph". This approach allows to predict a risk of deadlock statically. Alternate methods allow a snapshot-based on-line observation of a system, which lets to discover deadlocks at run-time [5]. This is useful especially in systems in which global behaviour is hard to predict, for example a set of independent user programs requesting shared resources.

The approach of Wait-for-Graphs is transferred to distributed systems with addition of locality obligation, since a global state does not exist in such systems generally. Locality means that there are no notions like precise time flow or simultaneity: a global decision on a deadlock state is made based on independent local circumstances, which are reported by system components [6, 7, 8]. The Wait-for Graph approach is successfully used till now, especially in run-time (dynamic) deadlock detection [9, 10].

Dynamic methods identify some properties of a distributed system, like deadlock or termination, by the system itself, during its operation in run-time. They require some sort of instrumentation, typically message exchange, to monitor local states of processes. A decision on the existence of a cycle of waiting processes, which denotes a deadlock, is made in distributed way or by distinguished component. The static methods operate off-line on a model of a verified system, in which a cycle of waiting processes is searched. A total deadlock is a special case in these methods: if all processes of a system wait, a deadlock obviously occurs and no waiting cycle should be identified.

Separate techniques are typically used for these two cases: resource deadlocks [7, 11] and communication deadlocks, without buffers [12, 13, 14, 15], or with finite or infinite buffers [16, 17]. There is a significant difference between resource deadlock and communication deadlock [18].

Special algorithms are addressed for data bases [10, 19, 20, 21]. An alternative method for data bases uses variables values: *Wait-for*, *Held-by* and *Request-Q* [22].

A particular case is Pulse [23]: a Linux deadlock finder, which creates "ghost" copies of waiting processes and lets them run in a limited environment (actions causing permanent results, like writing to output, are skipped) to test if any process fulfils a condition on which other processes wait. This approach is efficient, but may cause false positives and false negatives.

Dynamic methods find deadlocks in distributed systems, but if a deadlock is not reported, it does not guarantee the deadlock freeness of a system and some deadlocks may occur in the future.

Besides Wait-for-Graph deadlock detection methods, the distributed termination detection techniques evolved [11, 24, 25, 26]. The methods are based on observation of special features of distributed processes (sometimes defined specifically for termination detection) or control over message traffic.

Modern static verification techniques are based on exploration of a global state space of a system (or a part of it in some techniques) [27, 28]. Many methods are used, typically based on temporal state space verification (model checking) or Petri net analysis. These techniques are intensively developed in research and recently are used in the verification of commercial software [29]. Among the methods are graph-based (as deadlock detection in statecharts [30]) and language-based (as verification of Promela specification in Spin [31]).

In model checking, the activities of a system are expressed in terms of local features of its components, and the global state space of the system is constructed. The features of system components are expressed in a temporal logic and verified by the evaluation of temporal formulas. The deadlock is one of most important features. Model checkers are equipped with deadlock detection procedures [32, 33], for example in Spin and PathFinder [31, 34, 35]. Typically, the deadlock is identified as a "state with no future", i.e., a strongly connected subgraph containing one state only: the deadlock itself [30, 36]. The deadlock freeness is checked by a CTL temporal formula like `AG EX true` (there is always a next state) [37, 38, 39, 40, 41, 42, 43]. To differentiate a deadlock from a termination, the formula is refined: `AG ((! FINAL) -> EX true)` (there is always a next state, except for process terminating states [44], but no general notion of "FINAL" feature was given).

Similar results are obtained using succession relation $\rightarrow$ in CCS [45]. A CCS term is in deadlock if $\not\rightarrow$ [46], which is equivalent to $\neg$`(AG EX true)` temporal formula. This has the same disadvantages as model checking: not distinguishing of deadlock from termination and total deadlock finding only. Similar methods use CSP [47] or the evaluation of formulas which depend on component Finite State Machines [48].

Model checking is typically used to find total deadlocks, i.e., states in which all processes wait for each other and no progress is possible. Temporal formulas

can also be used to check partial deadlocks, i.e., states in which some processes are involved in deadlocks, but other processes can continue their execution. Generally, in the case of partial deadlock checking, temporal formulas are based on the structure of verified models, for example, they use *key states*, as in: `AG(LQ.Head.Occ ⇒ AF(¬LQ.Head.Occ))` [49] (whenever the head buffer of the link queue is occupied with a request, this will eventually be processed). This allows to identify a deadlock in individual processes. Similar methods are described in other papers [50, 51, 52, 53, 54], some of them using cyclic wait discovery [55]. The disadvantage of such an approach is that temporal formulas need to be developed individually for each analysed system.

Some other approaches to partial deadlock detection use universal temporal formulas (i.e., formulas not related to the structure of a specific model), however such universal formulas require the models to have specific properties. Inspect checker [56] is based on runtime model checking, which do not accept cyclic spate spaces. In pairwise model checking [57] the formula `AG EX true` is used for every pair of mutually related processes operating in shared memory. The "weak invariant simulation" [58] finds deadlocks by looking for circular wait configurations of processes in ring networks of isomorphic subprocesses.

If a system is non-terminating (cycling), a process discontinuation is obviously a deadlock [59]. Conversely, a method may be addressed to terminating processes only [60]. Some detection methods are used for specific architectures of systems. For example, WickedXmas approach uses nodes communicating by queues [61]. A specific method is designed for tree-like component architectures [62]. Our approach finds a discontinuation of individual processes, using universal temporal formulas, but it works for any type of system: looping, terminating or constructed over both types or processes. A termination is also a kind of a process discontinuation (see later for termination detection methods), and our method distinguishes both kinds: deadlock and termination.

Other set of static methods concern Petri nets. Some of them are based on analysis of reachability graph of a Petri net [63]. Total deadlock is a leaf in reachability graph – no outgoing transition is present. Thus, reachability graph analysis is similar to model checking techniques, and typically they are combined as temporal analysis of the graph [64, 65, 66]. In both approaches it is hard to distinguish a deadlock from a termination of a processes: these methods are addressed to endlessly looping systems mainly [67].

Reachability analysis is usually based on the exploration of the entire state space. In many models the state space can be very large. The combinatorial state explosion problem refers to models in which the size of state space grows exponentially with the number of concurrent components. Partial order reduction methods provide a way to avoid the state explosion problem by using equivalences of states. There are methods based on independence of transitions: for interleaving systems [68, 69, 70] and for coincidence-based systems [71]. For Petri nets, stubborn sets are invented which exploit commutativity over sequences of actions [72, 73]. All these methods preserve deadlocks in reduction process.

Alternatively, structural analysis of Petri nets can be used. Structural analysis determines properties of models on the basis of the model's structure, so no exploration of the state space is needed. Structural analysis of deadlocks is based on subnets called siphons [74]. It can be shown that if a model is deadlocked, the unmarked places constitute a siphon. Structural analysis of deadlocks systematically finds basic siphons (basic siphons are siphons from which other siphons are composed) and checks if the found siphons can become unmarked (usually linear programming is used for this purpose). If no basic siphon can be found that can become unmarked, the model is deadlock-free.

Many algorithms for deadlock detection based on finding siphons in a Petri net are proposed [74, 75, 76]. Also, partial deadlock may be found in Petri net, using a method comparable to a Wait-for Graph (finding processes waiting for resources held by other processes in a cycle [77]). Again, identification of siphons is helpful. Other methods use conversion of a Petri net to another graph [78].

Model checking techniques typically interrupt the verification when an erroneous state is found. As a result, only one error is reported and after a revision of the system, a new verification is necessary. The methods based on Petri net siphons are an improvement, because they identify all siphons in a given net. The strong point of model checking methods is counterexample generation: a trace leading from initial state to a deadlock state, suitable for analysis of sources of deadlocks.

In some methods, operations on matrices are used, typically to calculate reachability. A Resource Allocation Graph (RAG) is represented with an adjacency matrix. The solutions are based on the calculation of powers of the matrix [79, 80] or the reduction of some columns and rows [81]. In Petri nets used for modelling manufacturing systems, the analysis of the machine-job incidence matrix is applied [82].

The features of main deadlock detection techniques are collected in Table 1.

As many techniques of deadlock detection evolved, even the notion of a deadlock differs in various papers. Some of them define the deadlock as a feature of the model of the system (as often is the case in Wait-for-Graph or model checking techniques [32, 68, 83, 84, 85, 86]). A better formulation is in abstract terms, but imprecision is often the drawback of this approach, like: "Deadlock is a situation ... in which a system or a part of it remains indefinitely blocked and cannot terminate its task" [87]. This formulation is close to ours, as it

Technique / Feature	1	2	3	4	5	6	7	8	9	10
Dynamic WFG-resource	+	-	+	+	+	+	+	+	-	-
Dynamic WFG-communication	-	+	+	-	-	+	+	+	-	-
Pulse	+	+	-	+	+	+	+	+	-	-
Model checking `AG EX true`	+	+	-	+	-	+	-	-	+	-
Petri net siphons	+	+	-	+	+	+	+	-	+	+

1 - resource deadlocks found
2 - communication deadlocks found
3 - resource/communication deadlocks distinguishable
4 - deadlocks over reusable resources found
5 - deadlocks over consumable resources found
6 - total deadlocks (all processes involved)
7 - partial deadlocks (not all processes involved)
8 - deadlock distinguishable from termination
9 - deadlock freeness guaranteed if no deadlock found
10 - all possible deadlocks found

TABLE 1. Advantages and disadvantages of various deadlock detection techniques

covers resource and communication deadlock, total and partial deadlock. In some papers the infinite waiting of a single process is called a *stall* [88], while mutual waiting of at least two processes is called a deadlock. The latter definition comes from the formulation of the Coffman deadlock conditions: mutual exclusion, hold and wait, no pre-emption condition, and circular-wait [89].

Sometimes a deadlock is related to resources only: "Deadlocks arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group" [90] or communication only: "A global state contains a deadlock error when all of the communication channels are empty and all of the entities' states are receive states, where a receive state is a state whose outgoing transitions are all receive ones" [91].

For the purpose of this paper we define deadlock as a global state in which a process (will be defined in Section 3) waits for a condition that cannot be fulfilled. Total deadlock refers to all the processes in the system, while partial deadlock leaves some processes running. Resource deadlock and communication deadlock will be defined in terms of the model of the distributed system in Section 4. Intuitively, resource deadlock occurs when a process waits for a value (of a shared variable) that cannot be reached. Communication deadlock occurs when a process infinitely waits for a certain message which would let it continue (other messages may arrive, but with no effect).

In the detection of resource deadlocks, several semantic models are used [10]. The simplest is *One-resource* model, in which a process can have at most one outstanding resource request at a time. In the *AND* model, a process is permitted to request a set of resources. It is blocked until it is granted all the resources it has requested. An alternative model of resource requests is the *OR* model. A request for numerous resources is satisfied by granting any requested resource. The *AND-OR* model is a generalization of the two previous models, in which the requests may specify any combination of *AND* and *OR* in the resource request. The *k-out-of-n* model allows the specification of requests to obtain any k available resources out of a pool of size n. It is a generalization of the *AND-OR* model. In the most general model, no underlying structure of resource requests is assumed.

As in IMDS resources are not directly included into the formalism, any semantics may be modelled. The granting of resources is modelled as the servers' states. The most natural model is the *OR* model, because there may be many states matching the agent's message, and any of these states enables the agent. Yet, a server may be constructed in such a way that its state informs of fulfilment of a complicated condition over resources, for example in the *AND-OR* or *k-out-of-n* model.

In the analysis of distributed systems, two kinds of processes discontinuation are observed: undesired lack of progress (deadlock), which is an error, and expected stopping called *process termination*. Deadlock detection and termination detection methods must distinguish the two kinds of discontinuation [92], or simply prohibit one of them, it is the reason while many deadlock detection techniques are addressed to endlessly looping systems only [59, 67].

Just as in a case of deadlock detection, dynamic (run-time) methods of termination detection require some instrumentation of a system, which is typically sending messages reporting the states of individual processes, and a mechanism of combining them into a global decision on distributed termination [92, 93, 94]. There are methods differing in instrumentation, dealing with failed processes or link failures, acceptance of temporary network partitioning [95, 96, 97, 98].

Static termination detection methods are based on observation of terminal states of individual processes. Model checking techniques are suitable for this purpose, using either model-specific formulas [99] or universal

ones [100]. A construction of Counting Agent [101] may be applied bot dynamically and statically. Transition invariants [102] allow to check if every execution starting in an initial state is finite.

In the paper we propose the application of IMDS, which highlights locality of properties of verified system and exploits communication dualism (message passing/resource sharing). When combined with model checking, IMDS allows for the expression and finding of a partial deadlock and to differentiate it from process termination. Moreover, the new method allows to distinguish a resource deadlock from a communication deadlock.

The comparison with dynamic methods is included to give a wide look on deadlock detection methods. Our method is an improvement of static model checking methods. As typical static method, our approach requires calculating of total state space of a verified system. The novelty lies in specification of universal formulas (using IMDS and model checking) for partial deadlocks and to distinguish deadlock from distributed termination and communication deadlocks from resource deadlocks. The universal formulas allow a designer for a verification without knowledge of temporal logic. The Dedan program is built upon this paradigm and allows for a verification in "push the button" style, model checking techniques are hidden inside the program.

3. INTEGRATED MODEL OF DISTRIBUTED SYSTEMS

An IMDS system is composed over *servers*, each server represented by its *current state* as a pair: *(server_identifier, value)*. Each server offers a set of *services*. There is also a set of *agents* which represent distributed computations. Agents originate *messages* which invoke *services* at the servers. A message is a triple: *(agent_identifier, server_identifier, service)*). The execution of a service is called an *action* (of a server appointed by a message). An action changes the current state of the server and issues a next message of the agent. This new message can be directed to the same or a different server. In such a way the distributed computation is executed as a sequence of actions, changing the states of the participating servers.

An action (of a server) can be regarded as a relation mapping a state and a message into the next state and the next message. The current state of a server may allow the execution of a service (then the state and the message *match*) or not. If the service can be executed, the corresponding action is *prepared*. Messages waiting at a server are *pending*. Several actions can be prepared on the same server at the same time. The choice of an action for execution (firing the action) is nondeterministic.

The actions of different servers are executed using interleaving [103] and the choice of specific servers is again nondeterministic.

A special case of actions is when there is no next message: in such a case the agent *terminates*. Therefore, the number of agents may decrease during computation.

It is assumed that a system starts with *initial states* of all servers and *initial messages* of all agents.

Sequences of actions (in a system) are called *processes*. Sequence of actions executed by the same server is called *server process* and sequence of actions executing messages of the same agent is called *agent process*.

Each system (the complete computation) can be decomposed into server processes (constituting the *server view*) or alternatively into agent processes (constituting the *agent view*).

Many properties of systems can be verified using the model checking technique. This paper concentrates on deadlock and termination properties [5]). Different features may be observed in different views (i.e., server or agent view), as shown in Section 3.4.

The IMDS formalism will be defined as follows: first, the static elements conforming a configuration of a system will be defined. Then, the behaviour of a system by means of actions that cause the transition from one configuration to the other will be presented. The construction of a Labelled Transition System that defines a system semantics follows. Last, temporal formulas for deadlock and termination detection will be presented.

3.1. Static system structure

A typical model of a distributed system consists of three elements: servers, their states and messages exchanged. Our definition follows this scheme. It is assumed that an unfailing channel between any pair of servers exists.

The definition of the system follows, supported by a simple example of a bounded buffer with capacity 2 (presented informally in Fig. 2: the servers' states are nodes, initial states are surrounded by bold ellipses, input and output messages are on transitions). In the example, three servers cooperate: the `buffer` containing 0, 1 or 2 elements (states `elem_0...elem_2`), the producer `Sprod` (with its agent `Aprod`) spontaneously making new elements and then sending them to the buffer (service `put`) (provided that given operation is acceptable; for instance no putting into full `buffer` is acceptable), and the consumer `Scons` (with its agent `Acons`) retrieving the elements from the buffer (service `get`) and destroying them. The definitions and the description of IMDS static elements are included in Table 2: the notions are introduced in the first column, formal definitions are in the second, the third column contains a "tabled equation" number (Ti) for a definition, and an example is given in the fourth column.

A system is built over *servers*; any server has

its unique identifier (T1). Any server has its *value* belonging to the set of servers' values (T2). A *server state* is a pair *(server, value)* (T3). The *global system state* is the set of pairs such that any server has its exactly one state (T4). In a given *global system state*, the pair *(server, value)* is the *current state* of the server. The system contains the set of *services* to be invoked at the servers (T5). Valid servers' services are the pairs *(server, service)* (T6). The *agents* in the system represent distributed computations (T7). The agents originate service invocations called *messages* (T8). A message is an invocation of a service on a server by an agent, thus it is a triple: *(agent, server, service)*. Issued but unserved yet messages are *pending* at the servers (T9). At most one message may be pending with an agent identifier. A *global system configuration* (or simply a *configuration*) is a set of current states for all servers and pending agents of messages (all except terminated) (T10). An *initial configuration* contains an *initial state* for every server and an *initial message* for every agent (T11). States and messages together are called *items* (T12).

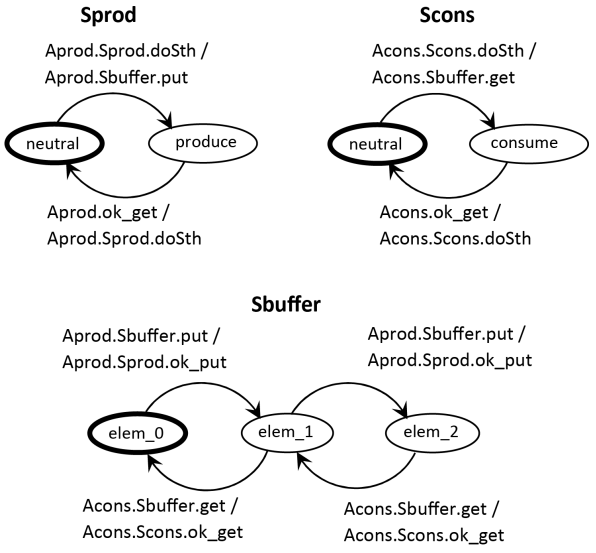

FIGURE 2. "Bounded buffer" example presented as automata for buffer, producer and consumer; states of automata are servers' states, input and output symbols on transitions made of IMDS messages

3.2. System behaviour (actions)

A static system configuration was defined above. We now define actions that change the global configuration. A message is an invocation of a server's service (T8). The execution of this service – the action – changes the sever state (replaces the state by a new state). Thus we are considering the two configurations: an input one (before the execution of the action) and the output one (after the execution). The *input configuration* contains the *input state* and the *input message* of the action, and a successive configuration not containing the input state and the input message (which both disappear in the action), but containing a new server state – the *output state*. The execution of the service produces a new message (the *output message*), which determines the continuation of the distributed computation. It may be a reply for the input message, a call of a service at another server or continuation of computation on the same server. Therefore, the input message and the output message originate from the same agent. The output message appears in the *output configuration* of the executed service.

We define an *action* (1) as the execution of the service a *state=(s,v)* by a *message=(a,s,r)*, such that the server component of the message is the same as the server component of the state. If an action is defined for a pair *(message, state)*, we say that they *match*. The action deletes this pair from the input configuration and replaces it with a new pair *(state', message')* in the new (output) configuration (2). The obvious condition imposed on the action is that the server component in the input state and the output state is the same. Dually, the agent component of the input message and of the output message is the same (the server component of the output message is usually not the same). We call the output state – *continuation state* and the output message – *continuation message*.

A special type of action is a *terminating action*: It has an output state but there is no output message. In such an action, the server remains (with a new state), but the agent terminates, as there is no message with the agent's identifier on output.

Formally, a set of actions is a relation over the set of pairs *(m,p)* and singletons *(p)*:

$$\Lambda = \{(m,p)\lambda(m',p')\} \cup \{(m,p)\lambda(p')\} \qquad (1)$$

$$m = (a,s,r) \in M, m' = (a',s'',r') \in M, a' = a;$$
$$p = (s,v) \in P, p' = (s',v') \in P, s' = s$$

The set $\{m,p\}$ of input items of the action will be denoted $I(\lambda)$ and the set of output items (continuation state and continuation message or continuation state only) will be denoted $O(\lambda)$:
$O(\lambda) = \{m',p'\} \vee O(\lambda) = \{p'\}$. A pair (m,p) *match* if for this pair an action exists.

An action λ transforms its input configuration $T_{inp}(\lambda)$ to its output configuration $T_{out}(\lambda)$. The output configuration is constructed from the input configuration in such a way that the input items are extracted and the output items replace them:

$$T_{out}(\lambda) = T_{inp}(\lambda) \setminus I(\lambda) \cup O(\lambda) \qquad (2)$$

In such a way, the server s gets a new state p' and the agent a generates a new message m' (or it terminates).

notion	**definition**	**eq.**	**example**
server	set of servers $S = \{s_1, ...\}$	(T1)	*S*={*Sbuffer, Sprod, Scons*}
server's value	set of values $V = \{v_1, ...\}$	(T2)	*V*={*elem_0, elem_1, elem_2, neutral, produce, consume*}
server's state *(server, value)*	set of all servers' states $P = \{(s, v) \mid s \in S, v \in V\}$	(T3)	*P*={*(Sbuffer, elem_0), (Sbuffer, elem_1), (Sbuffer, elem_2), (Sprod, neutral), (Sprod, produce), (Scons, neutral), (Scons, consume)*}; *elem_i* states valid for *Sbuffer*, *neutral* and *produce* states valid for *Sprod*, *neutral* and *consume* states valid for *Scons*
global system state - set of servers' states; current state	global system state $T_S = \{(s, v) \in P \mid$ $\forall_{s \in S} \exists_{p=(s,v) \in P}\ p \in T_S$; $\forall_{p,p' \in P}\ p = (s, v), p' = (s', v')$, $p \neq p' \implies s \neq s'\}$; *(s,v)* is *current* state of server s	(T4)	T_S=*{(Sbuffer, elem_1), (Sprod, neutral), (Scons, consume)*}
service	set of services $R = \{r_1, ...\}$	(T5)	*R*={*put, get, ok_put, ok_get, doSth*}
server's service *(server, service)*	set of all valid servers' services $U = \{(s, r) \mid s \in S, r \in R\}$	(T6)	*U*={*(Sbuffer, put), (Sbuffer, get), (Sprod, ok_put), (Sprod, doSth), (Scons, ok_get), (Scons, doSth)*}; *put* and *get* valid for *Sbuffer*, *ok_put* and *doSth* valid for *Sprod*, *ok_get* and *doSth* valid for *Scons* (services *put* and *get* consist in inserting/ retrieving element from/to *buffer*, respectively; services *ok_put* and *ok_get* are invoked on completion of inserting/retrieving operation to inform *Sprod*/*Scons* about the fact; *doSth* service causes *Sprod*/*Scons* to *produce*/*consume* an element)
agent	set of agents $A = \{a_1, ...\}$	(T7)	*A*={*Aprod, Acons*}, sequence of operations performed by *Aprod* is producing an element, putting it into *buffer*, obtaining confirmation and causing *Sprod* to rest in *neutral* state; analogous sequence concerns *Acons*
message	set of all messages $M = \{(a, s, r) \mid a \in A, (s, r) \in U\}$	(T8)	*M*={*(Aprod, Sbuffer, put), (Acons, Sbuffer, get), (Aprod, Sprod, ok_put), (Aprod, Sprod, doSth), (Acons, Scons, ok_get), (Acons, Scons, doSth)*}
pending message	set of messages pending $T_M = \{(a, s, r) \in M \mid \forall_{m,m' \in P}$ $m = (a, s, r), m' = (a', s', r')$, $m \neq m' \implies a \neq a'\}$; *(a,s,r)* is *pending message* of agent a at server s, calling service r	(T9)	T_M=*{(Aprod, Sbuffer, put), (Acons, Scons, ok_get)*}
global system configuration - set of states and messages;	global system configuration $T = T_M \cup T_P$ $\forall_{s \in S} \exists_{(s,v) \in T_S}\ (s, v) \in T$	(T10)	*T*={*(Sbuffer, elem_1), (Sprod, produce), (Scons, neutral), (Aprod, Sbuffer, put), (Acons, Scons, ok_get)*}; the former three pairs are current states of servers, the latter two triples are messages of agents, pending at some servers of the system
initial configuration	initial configuration T_0 $\forall_{a \in A} \exists_{(a,s,r) \in T_M}\ (a, s, r) \in T_0$	(T11)	T_0=*{(Sbuffer, elem_0), (Sprod, neutral), (Scons, neutral), (Aprod, Sprod, doSth), (Acons, Scons, doSth)*}
item	set of all items $H = M \cup P$ item $h \in H; h = (a, s, r) \vee h = (s, v)$; $(a, s, r) \in M; (s, v) \in P$	(T12)	*H*={*(Sbuffer, elem_0), (Sbuffer, elem_1),... (Aprod, Sbuffer, put), (Acons, Sbuffer, get),...*}; items are states and messages

TABLE 2. Definitions of IMDS static elements

The current state p in the input configuration T_{inp} is replaced by the new current state p' of the same server s in the output configuration T_{out}. The message m pending in the input configuration T_{inp} is replaced by the new message m' of the same agent a, pending typically at some other server s'' (but not necessarily; it may be the same server s) in the output configuration T_{out}. Alternatively, if an output message is not present in the action ($T_{out} = \{p'\}$), the agent a terminates.

From (T4, T10, 2), $T_{out}(\lambda)$ contains the set of states of the same set of servers as $T_{inp}(\lambda)$. From (T9, T10, 2), $T_{out}(\lambda)$ contains the set of messages of the same or smaller set of agents as $T_{inp}(\lambda)$ (except a terminated agent, if the action λ terminates it).

To sum up, a system is defined by the sets of states, messages and actions, and by its initial configuration.

In the example, we define the following set of actions performed in the ***Sprod*** server (the symbol $\rightarrow$ is used instead of λ):

```
((Aprod,Sprod,doSth),(Sprod,neutral)) →
  ((Aprod,Sbuffer,put),(Sprod,produce)),
((Aprod,Sprod,ok_put),(Sprod,produce)) →
  ((Aprod,Sprod,doSth),(Sprod,neutral))
```

The agent ***Aprod*** invokes the service ***doSth*** which causes the server to enter the ***produce*** state and to send the ***put*** message to the ***Sbuffer*** server. When the response message ***ok_put*** arrives, in the action the server switches to the ***neutral*** state and the agent issues the issuing the ***doSth*** message to the same server. The actions of the ***Scons*** server are similar:

```
((Acons,Scons,doSth),(Scons,neutral)) →
  ((Acons,buffer,get),(Scons,consume)),
((Acons,Scons,ok_get),(Scons,consume)) →
  ((Acons,Scons,doSth),(Scons,neutral))
```

The third server ***Sbuffer*** performs the following, obvious actions of inserting and retrieving elements:

```
((Aprod,Sbuffer,put),(Sbuffer,elem_0)) →
  ((Aprod,Sprod,ok_put),(Sbuffer,elem_1)),
((Aprod,Sbuffer,put), (Sbuffer,elem_1)) →
  ((Aprod,Sprod,ok_put),(Sbuffer,elem_2)),
((Acons,Sbuffer,get),(Sbuffer,elem_2)) →
  ((Acons,Scons,ok_get),(Sbuffer,elem_1)),
((Acons,Sbuffer,get),(Sbuffer,elem_1)) →
  ((Acons,Scons,ok_get),(Sbuffer,elem_0))
```

Note that getting an element from the empty buffer and putting an element to the full buffer is inhibited, since the ***get*** service does not match the ***elem_0*** state (is not accepted in the ***elem_0*** state) and the ***put*** service does not match the ***elem_2*** state.

The system starts at the initial configuration T_0, which assigns an *initial state* to every server and an *initial message* to every agent. In the example, the initial configuration is:

```
{ (Sprod,neutral),
  (Scons,neutral),
  (Sbuffer,elem_0),
  (Aprod,Sprod,doSth),
  (Acons,Scons,doSth) }
```

The example was prepared by grouping the actions on individual servers. This is an intuitive approach, where processes reside on servers and communicate by means of messages.

Formally, we define a *process* as arbitrary nonempty subset of H (a set of states and messages). A process is *sequential*, if for any action to which the process delivers an item (a state or a message or both):

- one of the continuation items (continuation state or continuation message) belongs to the process,
- or the process terminates in the action (from (2), termination may concern only an agent, not a server).

A process K is sequential iff:

$$K = \{h \in H | \forall_{m\in M} \forall_{p\in P} \forall_{\lambda\in\Lambda} \{m,p\} = I(\lambda), \quad \{m,p\} \cap K \neq \emptyset \Rightarrow O(\lambda) \cap K \neq \emptyset \vee K \textit{ terminates}\} \tag{3}$$

If the process is sequential, one of the output items of the action belongs to the process (or it terminates). The second output item (if present) typically belongs to some other process, therefore the communication between processes is performed by this second item.

The synchrony of communication of the processes is defined in terms of the input items: two processes communicate synchronously in an action, if one of them delivers the input state and the other one delivers the input message.

Two processes K_1, K_2 communicate synchronously in λ iff:

$$K_1 \cap I(\lambda) = p \in P \wedge K_2 \cap I(\lambda) = m \in M \vee K_1 \cap I(\lambda) = m \in M \wedge K_2 \cap I(\lambda) = p \in P \tag{4}$$

A process is *synchronous* if for every action in which it is involved (delivers an item), the communication with other processes is synchronous [2].

A process is *asynchronous* if the synchrony is encapsulated inside it, i.e., in every action in which it is involved, the process delivers both input items (the state and the message).

Two sequential processes K_1, K_2 communicate asynchronously in λ iff:

$$I(\lambda) \subset K_1 \wedge m \in O(\lambda) \cap M, m \in K_1 \wedge \quad (5a)$$
$$p \in O(\lambda) \cap P, p \in K_2$$
$$\vee$$
$$I(\lambda) \subset K_1 \wedge p \in O(\lambda) \cap P, p \in K_1 \wedge \quad (5b)$$
$$m \in O(\lambda) \cap M, m \in K_2$$
$$\vee$$
$$I(\lambda) \subset K_2 \wedge m \in O(\lambda) \cap M, m \in K_2 \wedge \quad (5c)$$
$$p \in O(\lambda) \cap P, p \in K_1$$
$$\vee$$
$$I(\lambda) \subset K_2 \wedge p \in O(\lambda) \cap P, p \in K_2 \wedge \quad (5d)$$
$$m \in O(\lambda) \cap M, m \in K_1$$

A process is *asynchronous sequential* iff it is sequential and in every action (to which it delivers items) it communicates asynchronously with other processes.

Note that one of two forms of asynchronous communication may be observed in an action: by state or by message. If the process (which delivers both input items to the action) takes the continuation state, sending the continuation message to some other process, communication is performed by message passing (5b,5d).

Otherwise, if the process (which delivers both input items to the action) takes the continuation message, leaving the continuation state to some other process, the communication is performed by resource sharing (5a,5c). The server occurring in input state and in continuation state is the resource and the communicating processes share the states of the server.

In the former case, the continuation state is the carrier of the process (as all states and continuation states in the actions of the process have the same server component) and thus resource sharing is hidden inside the process (5b,5d). In the latter case, the continuation message is the carrier of the process (as all messages and continuation messages in the actions of the process have the same agent component) and the message passing is hidden inside the process (5a,5c).

Note that a system may be decomposed either to asynchronous sequential processes communicating by messages (with resource sharing hidden) or to asynchronous sequential processes communicating by states (with hidden message communication). The decomposition is canonical, which is proved in [2]. The former decomposition is called *server decomposition* and such processes are called *server processes*. A server process contains all states of its server and all messages addressed to its server:

$$\textit{server process } B(s): \quad (6)$$
$$\forall_{(s',v') \in P}\ s' = s \Rightarrow (s', v') \in B(s) \wedge$$
$$\forall_{(a',s',r') \in M}\ s' = s \Rightarrow (a', s', r') \in B(s)$$

The latter decomposition is *agent decomposition* and such processes are called *agent processes*. An agent process contains all messages with its agent identifier and all visited servers' states:

$$\textit{agent process } C(a): \quad (7)$$
$$\forall_{(a',s',r') \in M}\ a' = a \Rightarrow (a', s', r') \in C(a) \wedge$$
$$\forall_{(s',v') \in P}(\exists_{(a'',s'',r'') \in M}\ a'' = a \wedge s' = s'') \Rightarrow$$
$$(s', v') \in C(a)$$

Note that agent processes overlap (they have common subsets of states) but their message sets are separate (7). Server processes are totally disjoint (6).

It should be stressed that the system is uniform, yet there are two projections onto asynchronous sequential processes: server decomposition and agent decomposition.

Static elements (items), actions and an initial configuration define a system completely. The processes imposed on a system can be used for the observation of its behaviour. Server processes and agent processes represent the two aspects of behaviour.

In the example, in server decomposition there are three server processes: *B(Sprod), B(Scons), B(Sbuffer)*:

```
B(Sprod)={ (Sprod, neutral),
  (Sprod, produce),
  (Aprod, Sprod, doSth),
  (Aprod, Sprod, ok_put) }
B(Scons)={ (Scons, neutral),
  (Scons, consume),
  (Acons, Scons, doSth),
  (Acons, Scons, ok_get) }
B(Sbuffer)={ (Sbuffer, elem_0),
  (Sbuffer, elem_1),
  (Sbuffer, elem_2),
  (Aprod, Sbuffer, put),
  (Acons, Sbuffer, get) }
```

In the agent decomposition there are two agent processes:

```
C(Aprod)={ (Aprod, Sprod, doSth),
  (Aprod, Sbuffer, put),
  (Aprod, Sprod, ok_put),
  (Sprod, neutral),
  (Sprod, produce),
  (Sbuffer, elem_0),
  (Sbuffer, elem_1),
  (Sbuffer, elem_2) }
C(Acons)={ (Acons, Scons, doSth),
  (Acons, Sbuffer, get),
  (Acons, Scons, ok_get),
  (Scons, neutral),
  (Scons, consume),
  (Sbuffer, elem_0),
  (Sbuffer, elem_1),
  (Sbuffer, elem_2) }
```

3.3. Semantics

The semantics of modelled system is defined by a Labelled Transition System (LTS [3]), in which the nodes are system configurations and transitions are actions.

If the items p and m match in a configuration, then an action having p and m on input may be executed. We say that the action is *prepared* in the configuration containing p and m. The execution of the action is called *firing* the action. We assume interleaving semantics, i.e., only one of the prepared actions is fired at a time [103].

Note that the actions prepared on the same server are always in conflict, as each "consumes" the current server's state.

On the other hand, the actions prepared on distinct servers are independent. The firing of an action on a given server does not influence prepared actions on other servers (they remain prepared, although the action may add a new prepared action on some server).

The Labelled Transition System ($LTS = (N, W)$) is defined by the set of *nodes* N and the set of *transitions* W. The nodes are the system configurations (with the *initial node* being the initial configuration) and the transitions are actions. Such an LTS contains all possible executions of the system.

IMDS provides three kinds of nondeterminism:

- nondeterminism between servers – if there are prepared actions on distinct servers, the action to fire is chosen in nondeterministic way,
- nondeterminism in server – if a current state matches more than one of pending messages – an action to fire is chosen nondeterministically,
- nondeterminism in agent – if more than one action is prepared with matching state and message – an action to fire is chosen nondeterministically.

The system is assumed to act fair, i.e., if an action is prepared an infinite number of times – it must be fired. Thus, no part of the LTS is left as "dead code".

3.4. Deadlock and termination

We define deadlock as the situation where a process that is not terminated, cannot continue (at the moment and in the future). Deadlock should be defined separately for server process and for agent process, since they have different characteristics.

First consider a server process. The following situations may occur:

- the current state matches some of messages pending at the server – at least one action is prepared and it may be fired; the server process runs;
- the current state does not match any message pending at the server (or no message is pending) – a matching message may occur at the server in the future; the server process waits;
- neither a message is pending at the server nor a message will occur at the server in the future; the server process is idle;
- the current state does not match any message pending at the server – but no matching message may occur at the server in the future; the server process is in deadlock.

The former two cases are normal operation of the server process. The third case means that the server will not change its state – the current state will be current forever. It is right, because no other server wants any service offered by the given server. The last case is a deadlock – some services wait to be executed but neither the current state matches any of them nor it will match in the future. The cases may be distinguished during the analysis of the LTS.

Servers in IMDS do not terminate, but a server may remain in a particular state infinitely, if neither any message is pending at this server nor any message will occur in the future. Therefore, we do not call such a situation termination, we prefer the term *idleness* of the server. Some author treat the termination and the idleness of cooperating servers equivalently [92, 94, 98].

Consider an agent process. The cases are:

- the agent's message is pending at a server and it matches the current state of this server – the action is prepared and it may be fired; the agent runs;
- the agent's message is pending at a server and it does not match the current state of this server, but a matching state may occur in the future; the agent waits;
- the agent's message is pending at a server and neither the current state of this server matches the message nor such a state may occur in the future; the agent fell into a deadlock;
- there is no message pending with the agent identifier – the agent process is terminated.

The former two cases are normal operation of the agent process. The third case is a deadlock – there is a message to be serviced but this will never occur. The last case means that the process is terminated. Again, the cases may be distinguished by the analysis of the LTS.

4. MODEL CHECKING OF IMDS MODELS IN DEDAN ENVIRONMENT

The LTS of a system may be analysed manually or using graph algorithms. Yet, the LTS can be easily interpreted as a Kripke Structure [3, 27]: a finite set of *nodes* (we do not call them states to avoid ambiguity) – configurations, *initial node* – initial configuration, a total *transition relation* – actions (with self-loops added for configurations in which no pair matches

– total deadlock), naturally assignable *labelling* (see later). This allows for the analysis of the graph by model checking techniques, especially to find deadlocks and check termination of processes.

A communication deadlock occurs when the current server's state does not match any message pending at the server, and no matching message may occur at the server in the future. In other words, some messages are pending and no action will be fired on the server. We label the LTS nodes using the Boolean formulas (only configurations with *true* labels are described, *false* is for all remaining configurations):

- D_s – *true* in all configurations where at least one message is pending at the server s,
- E_s – *true* in all configurations where at least one action is prepared at the server s.

The formulas that check if the server s falls into a communication deadlock or to an idle state are given in Table 3, features $dds(s)$ and $idle(s)$, receptively. The properties are given in both LTL and CTL temporal logics, as the Dedan program cooperates with two external model checkers: Spin (using LTL [31, 34]) and NuSMV (using CTL [104]).

A resource deadlock (or deadlock over resources) occurs when the agent process' message is pending at a server but it will never match any state of this server. We need the following labelling of the LTS (F_a is for finding agent termination):

- D_a – *true* in all configurations where a message of the agent a is pending,
- E_a – *true* in all configurations where an action is prepared with a message of the agent a,
- F_a – *true* in all configurations where a terminating action is prepared with a message of the agent.

The formula that checks if the agent a falls into a resource deadlock is given in Table 3, feature $dda(a)$. An agent termination $term(a)$ (no message of the agent pending) is also included. Of course, other properties may be added, provided that proper labelling is applied on nodes of the LTS.

Note that universal formulas finding deadlocks concern individual processes, not the system as a whole. This feature differs the presented technique from other deadlock detection methods using model checking.

5. EXAMPLE DEADLOCK DETECTION: THE DEDAN ENVIRONMENT AT WORK

The deadlock detection in IMDS formalism will be presented on the example of a system in which the two computations (agents), each one running on its own server, use two semaphores, each one residing on a separate servers.

```
A1:                 A2:
sem1.wait;          sem2.wait;
sem2.wait;          sem1.wait;
sem1.signal;        sem2.signal;
sem2.signal;        sem1.signal;
```

This system falls into a total deadlock when *A1* holds *sem1* and waits for *sem2* and *A2* holds *sem2* and waits for *sem1*. But the situation changes if we add another agent process *A3* which simply loops and does its own calculations. In deadlock, agents *A1* and *A2* cannot continue, but *A3* still works. We denote calculations made by *A3* symbolically as endless execution of *left* and *right* services on the server *r*:

```
A3:
loop  {
      r.left;
      r.right;
      }
```

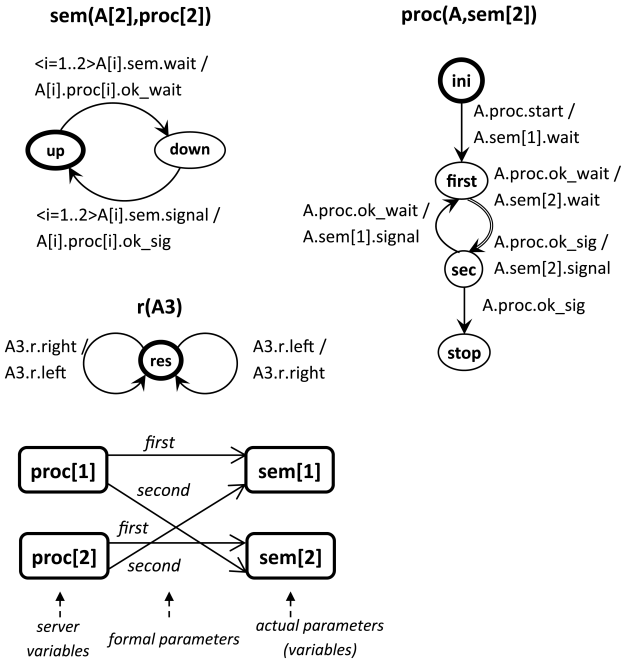

FIGURE 3. "2 semaphores" example: automata types (with formal parameters) and mapping of actual parameters (switched for proc[2])

The system is not in a total deadlock, it is a partial deadlock which is hard to identify using model checking techniques (no system configuration is a "configuration without future"). In the system, termination of *A1* and *A2* (which are designed to terminate after a number of calculations) is not inevitable (a deadlock is not a termination). Using IMDS, partial deadlock of *A1* and *A2* may be identified by the proposed universal formulas, and so the lack of termination of these agents. The agent *A3* does not terminate as well – but this is

property	LTL	CTL
communication deadlock in server s: $dds(s)$	$\Diamond\Box(D_s \wedge \neg E_s)$	$\mathbf{EF\ AG}(D_s \wedge \neg E_s)$
server s idle: $idle(s)$	$\Diamond\Box(\neg D_s)$	$\mathbf{EF\ AG}(\neg D_s)$
resource deadlock in agent a: $dda(a)$	$\Diamond\Box(D_a \wedge \neg E_a)$	$\mathbf{EF\ AG}(D_a \wedge \neg E_a)$
termination of agent a: $term(a)$	$\Diamond(F_a)$	$\mathbf{AF}(F_a)$

TABLE 3. Temporal formulas for finding various situations in processes

expected because it works in infinite loop.

The verification is performed in the Dedan (Deadlock analyser) environment which consists of a specification part and *TempoRG* model checker [71]. The two-semaphore system is shown graphically (informally) in Fig. 3 and presented below in IMDS notation, which is the input notation of Dedan. The specification is in the server view. It is simply a grouping of actions on individual servers. This is a sequence of server type specifications (enclosed by *server ...;*), server and agent instances (variables) declaration (*agents ...; servers ...;*) and an initial configuration phrase (*init* $\rightarrow$ {...}). A server type heading contains a set of formal parameters: agents and servers used in actions of the server type. Formal parameters may be vectors. Each server has a set of states, a set of services and a set of actions assigned (in arbitrary order). In initialization part, actual parameters are bounded to formal parameters. The initial states of servers and the initial messages of agents are also assigned.

```
system   two_sem_server_view;

server:  sem (agents A[2]; servers proc[2]),
services {wait, signal},
states   {up, down},
actions  {
<j=1..2> {A[j].sem.wait, sem.up} →
             {A[j].proc[j].ok_wait, sem.down},
<j=1..2> {A[j].sem.signal, sem.down} →
             {A[j].proc[j].ok_sig, sem.up},
<j=1..2> {A[j].sem.signal, sem.up} →
             {A[j].proc[j].ok_sig, sem.up},
}

server:  proc (agents A; servers sem[2]),
services {start, ok_wait, ok_sig},
states   {ini, first, sec, stop},
actions  {
         {A.proc.start, proc.ini} →
             {A.sem[1].wait, proc.first},
         {A.proc.ok_wait, proc.first} →
             {A.sem[2].wait, proc.sec},
         {A.proc.ok_wait, proc.sec} →
             {A.sem[1].signal, proc.first},
         {A.proc.ok_sig, proc.first} →
             {A.sem[2].signal, proc.sec},
         {A.proc.ok_sig, proc.sec} →
             {proc.stop},
};

server:  r (agents A3),
services {left, right},
states   {res},
actions  {
         {A3.r.left, r.res} →
             {A3.r.right, r.res},
         {A3.r.right, r.res} →
             {A3.r.left, r.res},
};

agents:  A[2], A3;
servers: sem[2], proc[2], r;

init →   {
<j=1..2> A[j].proc[j].start;
         A3.r.left;
<j=1..2> proc[j](A[j],sem[j],sem[3-j]).ini;
<j=1..2> sem[j](A[1..2],proc[1..2]).up;
         r(A3).res;
}.
```

Note that the system above is defined in the server view, as the actions concerning a given server are grouped. Server states: *up, down, first, sec* are hidden inside server processes (not used outside the containing servers), and the communication is performed via messages (invoking services *wait, ok_wait, ...*). In Dedan the system may be automatically converted to the agent view (the two decompositions are canonical [2]). In the agent view, the messages are hidden inside agent processes, they communicate via servers' states.

```
system   two_sem_agent_view;

server:  sem,
services {wait, signal},
states   {up, down};

server:  proc,
services {start, ok_wait, ok_sig},
states   {ini, first, sec, stop};

server:  r,
services {left, right},
states   {res};

agent:   A (servers proc,sem[2]),
```

```
actions  {
         {A.proc.start, proc.ini} →
             {A.sem[1].wait, proc.first},
         {A.sem[1].wait, sem[1].up} →
             {A.proc.ok_wait, sem[1].down},
         {A.proc.ok_wait, proc.first} →
             {A.sem[2].wait, proc.sec},
         {A.sem[2].wait, sem[2].up} →
             {A.proc.ok_wait, sem[2].down},
         {A.proc.ok_wait, proc.sec} →
             {A.sem[1].signal, proc.first},
         {A.sem[1].signal, sem[1].down} →
             {A.proc.ok_sig, sem[1].up},
         {A.proc.ok_sig, proc.first} →
             {A.sem[2].signal, proc.sec},
         {A.sem[2].signal, sem[2].down} →
             {A.proc.ok_sig, sem[2].up},
         {A.proc.ok_sig, proc.sec} →
             {proc.stop},
};

agent:   A3 (servers r),
actions  {
         {A3.r.left, r.res} →
             {A3.r.right, r.res},
         {A3.r.right, r.res} →
             {A3.r.left, r.res},
};

agents:  A[2], A3;
servers: sem[2], proc[2],r;

init →  {
<j=1..2> proc[j].ini,
<j=1..2> sem[j].up,
         r.res,
         A[1](proc[1],sem[1..2]).start,
         A[2](proc[2],sem[2],sem[1]).start,
         A3(r).left,
}.
```

The Dedan program prepares the global state space and then runs TempoRG verifier to evaluate CTL formulas. Alternatively, a specification in infput form of Spin or NuSMV is prepared for verification under an external model checker. The results of the verification are – communication deadlocks:

- server *sem[1]* – *true* (deadlock in *sem[1]*),
- server *sem[2]* – *true* (deadlock in *sem[2]*),
- server *proc[1]* – *false* (no deadlock in *proc[1]*),
- server *proc[2]* – *false* (no deadlock in *proc[2]*),
- server *r* – *false* (no deadlock in *r*).

Resource deadlocks:

- agent *A[1]* – *true* (deadlock in *A[1]*),
- agent *A[2]* – *true* (deadlock in *A[2]*),
- agent *A3* – *false* (no deadlock in *A3*),

Agent termination:

- agent *A[1]* – *false* (*A[1]* may not terminate),
- agent *A[2]* – *false* (*A[2]* may not terminate),
- agent *A3* – *false* (*A3* does not terminate, which is desired as it contains an endless loop).

To sum up, in our case agent processes *A[1]* and *A[2]* fall into resource deadlock while agent *A3* does not. Also, server processes *sem[1]* and *sem[2]* fall into communication deadlock while *r* does not. No agent necessarily terminates.

The Dedan program generates the counterexamples (in a case of a deadlock or a lack of termination) or witnesses (in a case of successful termination) and displays them in readable format. The deadlock or termination may be checked for individual processes of for a given set of processes at the same time (together). The example of a common communication deadlock of *sem[1]* and *sem[2]* in a server view is presented in Fig. 4. Resource deadlock of the agent processes in agent view is shown in Fig. 5 (agent identifiers are changed from *A[1],A[2]* to *A,A__1* during the conversion to the agent view). Then, a lack of termination of all three agent processes is shown in Fig. 6. The coloured versions of the Figs. 4, 5 and 6 are attached as a supplementary material, file Figs_Colour.zip.

A counterexample or a witness is shown as a sequence diagram-like chart, the heading shows the names of the servers (and agents in agent view), the upper part shows the initial states and the sequences of states and messages leading to a deadlock, termination or non-terminating state in a given process. A state is displayed simply as its identifier with light blue background. An agent causing the entering of a given state is shown on dark blue background. A message is shown with agent identifier on light yellow background on send, and with service name on yellow background on receive. The middle part is a sequence repeating infinitely (if any – the entering of states *left* and *right* by *A3* process in this case). The last part shows the finishing deadlock or termination configuration: all servers' states and all pending messages are displayed. A counterexample could be shown over a graphical form of our LTS as configuration sequence alternatively. The program also provides a system simulation and an LTS analysis tool.

It should be observed that different models at the same level of abstraction are equivalent, and this equivalence includes the same number of concurrent components and similar numbers of their states. In the IMDS model, the server and agent views provide additional insights in the behavior of the model rather than increase its complexity. These additional views are actually projections of the model behaviour that are supposed to simplify some analyses.

The views of the modelled system emphasize different aspects of the behaviour. For example, in the model of

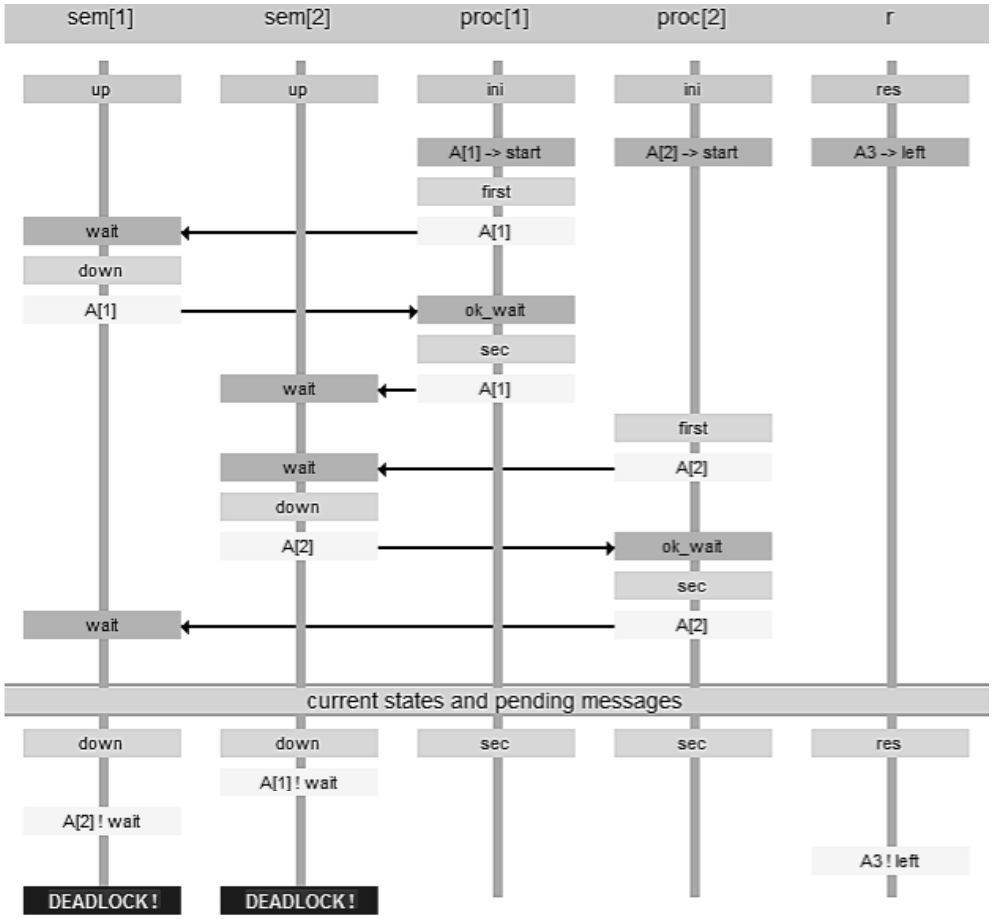

FIGURE 4. Common communication deadlock in *sem[1]* and *sem[2]* in the server view.

an automatic vehicle guidance system [105], the server view describes the behaviour from the point of view of road segment controllers. The agent view describes the behaviour from the vehicles' point of view. But the system is still uniform, the views are simply different groupings of actions.

Many examples were verified using Dedan, including several versions of bounded buffers (including deadlock-prone version), dining philosophers with deadlock and several deadlock-free solutions, systems with resource allocation errors and others. Several examples are available [106], commented shortly at the end of this paper. These are examples of primarily didactic nature, with state space generation time of several seconds and verification time less than 1 second. Much greater system, like four roads crossing, with 8 servers, 4 agents and 752 actions, exceeds the internal possibilities of Dedan due to large state space, which generation was cancelled having over 2.5 million configurations. Yet the road crossing example, exported to external model checker with symbolic state space representation and CTL verification, gives reasonable export time of 65 seconds and verification time less than 1 second. The example reduced to 3 roads (6 servers, 3 agents, 306 actions, over 107 thousand configurations) can be verified entirely under Dedan, with state space generation time about 8 hours and verification time of several seconds. This will be better when state space reduction and symbolic representation techniques are applied in Dedan.

Typically a communication deadlock in a server view manifests as a resource deadlock in an agent view. Yet, in some cases of systems with too little resources a server view may be free from communication deadlock, while some agents fall into a resource deadlock in an agent view (those for which there are not enough resources). It is illustrated in the Res_DeadlockS.txt example [106]. The system contains not enough resources. A resource deadlock takes place (one of requesting agents waits forever for a resource) but there is no communication deadlock (all servers work, including the server containing the resources). This comes from the different formulations of communication deadlock and resource deadlock: a communication deadlock concerns all agents whose messages are pending at the server, while a resource deadlock may concern a single agent (while other agents work, having resources allocated). Agents with

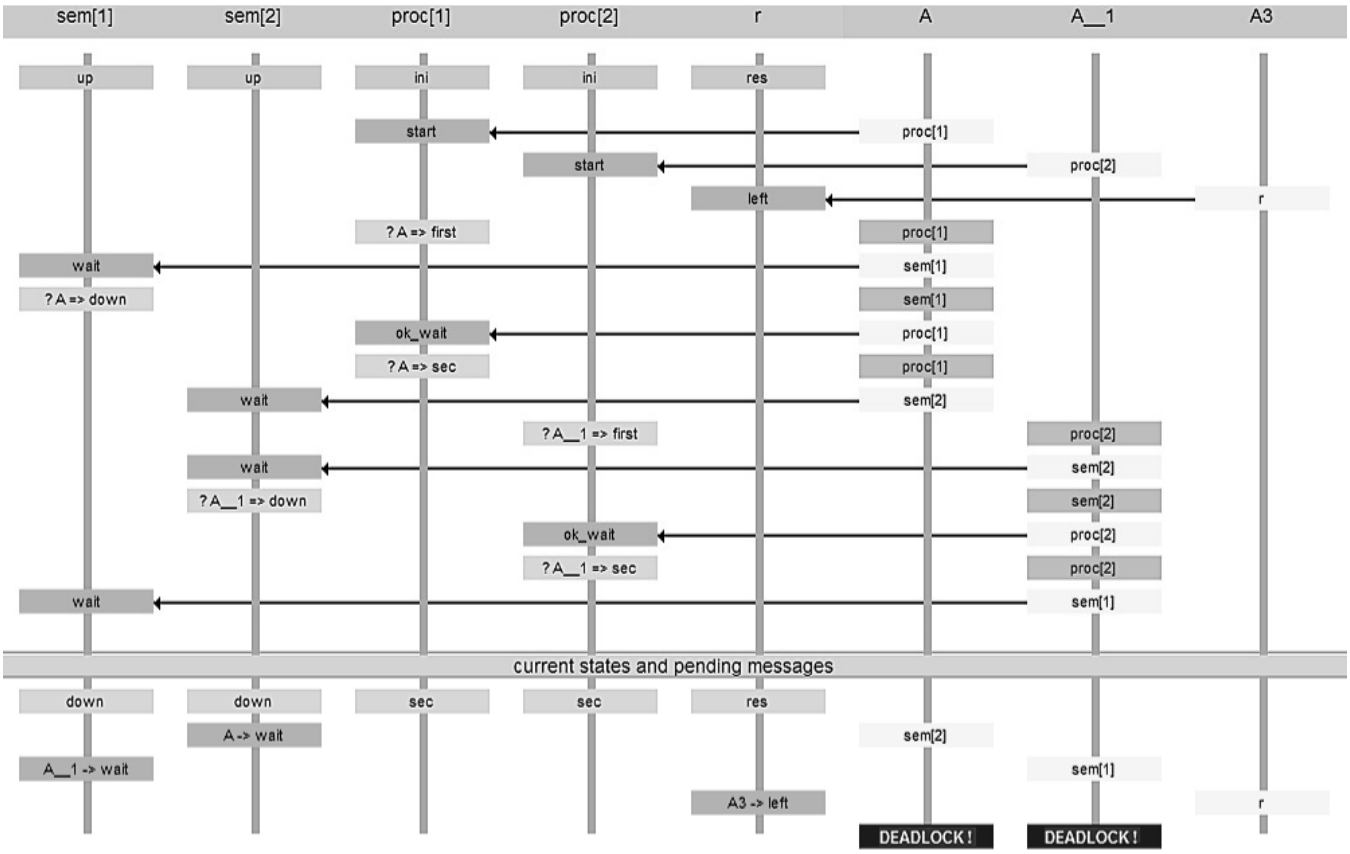

FIGURE 5. Resource deadlock of agents in the agent view.

insufficient resources fall into deadlocks individually, but checking for common deadlock of all agents does not show a deadlock.

6. CONCLUSIONS AND FURTHER WORK

The small repertoire of "general" behaviour features related to deadlock and termination in distributed systems may be verified by static methods. This disadvantage comes from the lack of formalisms in which some detailed local features may be expressed with universal formulas. Much attempt must be applied to express the features, like partial deadlocks, distinguishing deadlock from termination, differentiation between resource deadlock and communication deadlock, in terms of elements of the verified system. The features may be expressed in terms of a given model and require knowledge about temporal logic. The IMDS formalism for description of distributed systems liberates the designer from the simple "state with no future" verification which allows to find only total deadlock/termination states and gives no help how to distinguish them from each other.

The application of IMDS formalism allows to highlight locality of features and differentiates between features relative to agents behaviour from features relative to servers behaviour. IMDS (where no global system configuration is observed in execution of an action, only local state and local pending messages are observed) is especially destined to true distributed systems, like LANs, internet services, multiagent systems. The locality of "observations" allows to verify a part of a system (a subset of servers or a subset of agents) independently on the activities of the rest of the system, possibly changing in its various versions. Therefore, real negative and positive features of distributed systems may be verified using the proposed idea.

To check communication deadlocks, resource deadlocks (in total or partial form) and termination in an arbitrary system, using the methodology presented in the paper, the designer should express the system (or – better – its model; or best – the communication skeleton of the system) in terms of IMDS actions (of course messages and servers' states should be defines first); the proposed notation is presented informally in the paper – it is an intuitional form – simply input and output items of any action are listed. The rest of verification is due to the Dedan program which finds communication deadlocks, resource deadlocks and agents termination,

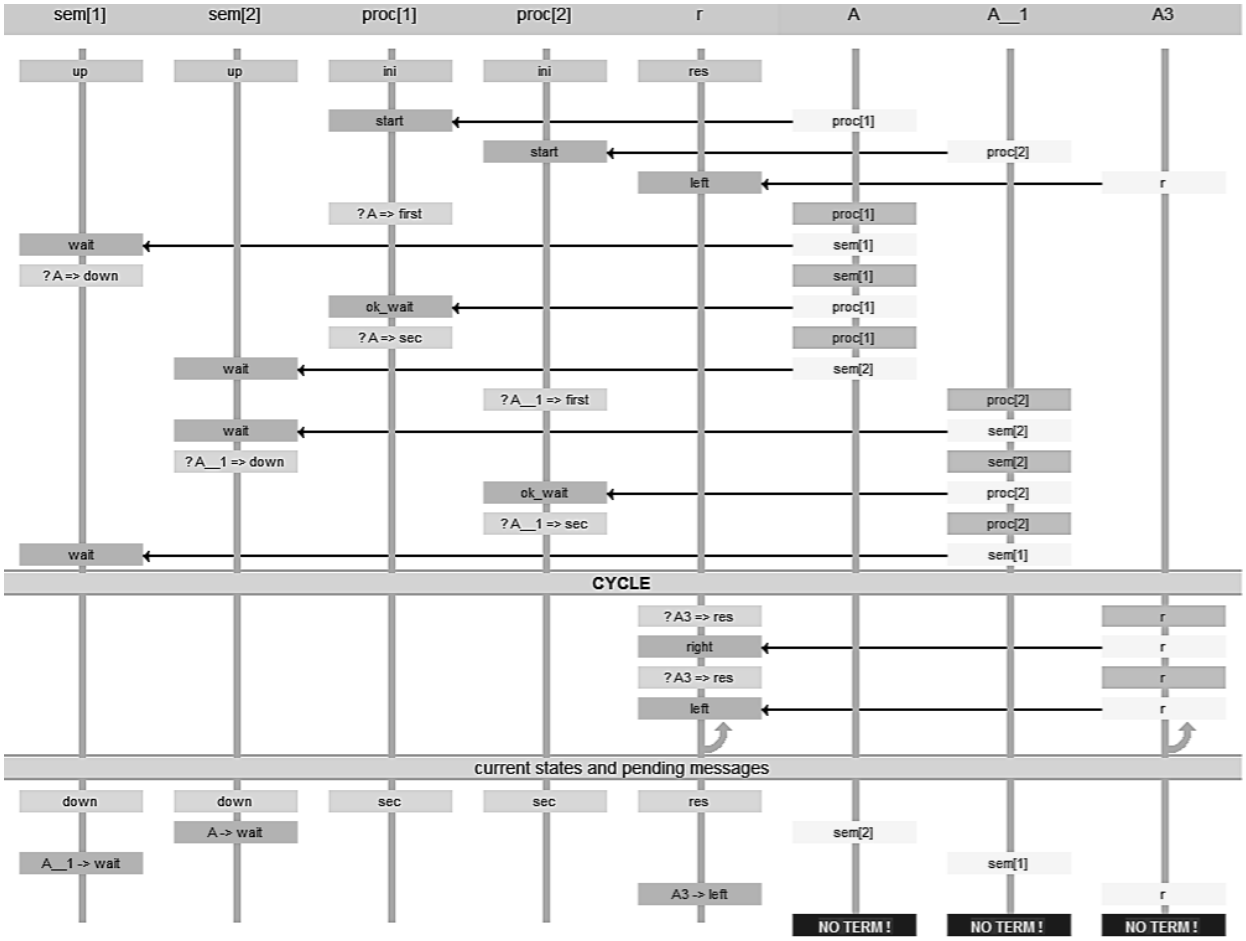

FIGURE 6. Lack of termination of all three agents in the agent view.

and the user need not know temporal logic to verify the model. Counterexamples and witnesses are generated automatically in readable form of sequence diagram or configuration sequence. Also, simulation of the verified system may be performed and observation of the shape of the LTS is possible.

Similar approach can be observed in the case of Delfin+ system [107, 108]. In this approach, processes are modelled in CCS and checked for deadlocks using heuristic search instead of model checking. This verification allows to find partial deadlocks just like in Dedan. However, communication deadlocks cannot be differentiated from resource deadlocks since communication dualism (message passing/resource sharing) is not modelled in Delfin. Also, we claim that IMDS is better destined for modelling of distributed systems than CCS, since in IMDS the execution of an action is not dependent on a whole (global) system configuration. Only the pairs *(state, message)* cause the firing of system actions. The advantage of Delfin is that it does not require constructing the complete transition model of a system.

The table of deadlock detection features of various methods now may be extended to cover Dedan approach (Table 4).

Although the IMDS formalism is general and allows for modelling many different kinds of systems, it has some limitations.

First, at most one message is sent in every action. Therefore, systems using broadcast or similar mechanisms cannot be modelled.

Second, the system to be statically model-checked must be finite. There is a version of IMDS with process creation (process creation and termination may be treated as *fork* and *join* operations), but this may lead to an infinite LTS if a processes are created in a loop. In general, fork is allowed if a Petri net corresponding to an IMDS system [2] is *k-bounded* [109] (which gives finite reachability graph corresponding to the LTS of IMDS system). Instead of a necessity of proving a finiteness of every system with process creation, we rejected the process creation mechanism. Therefore, dynamic task force cannot be used. If a designer needs a task force to perform a distributed computation, it should be created statically.

Dedan is now included in student laboratory equipment to let the students find errors in their concurrent programming solutions. The program is

Technique / Feature	1	2	3	4	5	6	7	8	9	10
Dedan	+	+	+	+	-	+	+	+	+	-

The columns have the same meaning as in Table 1

TABLE 4. Advantages and disadvantages of IMDS deadlock detection

available at [110].

The Dedan verification environment is under expansion. For model checking using built-in verifier TempoRG (which uses explicit state space) a reduction technique [69, 70] will be applied. In addition to TempoRG verifier, external model checkers are applied, including Spin [31, 34] and symbolic NuSMV [104] using BDD state space representation. The usage of external verifiers allows for verification of large systems. Verification with real time constraints is planned, with a connection to real-time model checker (not chosen now). The flow of time may be related to execution of actions and to passage of messages between servers. A conversion of IMDS to timed automata [111] is natural. This will allow for instance to model and verify urban transportation problems. Also, discrete time verification will be applied using EmLan [112].

The other directions of the Dedan program development are:

- checking for assertions expressed in terms of states and messages,
- invariant discovery – relations between states and messages held in loops of the LTS,
- graphical input – specification of server processes or agent processes in automata-like graphical form and graphical simulation of verified system over component automata,
- language-based input – elaboration of two languages for distributed systems specification: one for the server view (exploiting locality in servers and message passing) and the other one for the agent view (exploiting travelling of agents and resource sharing in distributed environment); a preliminary version of a declarative language-based preprocessor for a server view of verified systems is developed by the students of ICS, WUT,
- agent's own actions – equipping the agents with their own sets of actions, carried in their "backpacks", parametrizing their behaviour; this will allow for modelling of mobile agents (agents carrying their own actions model code mobility) and to avoid many server types in specification, differing slightly (as in the example of dining philosophers, with asymmetric solution – some philosophers take their left forks first and some of them take their right forks first, see the example phil3_asymS.txt [106]),
- dynamic creation of processes – new server's state and/or new agent's message created in an action [2], some problems with calculation of a state space of such a system must be solved first.

ACKNOWLEDGEMENTS

Włodzimierz Zuberek, Jerzy Mieścicki and Janusz Sosnowski gave important advice and helped to elaborate the final form of the paper.

SUPPLEMENTARY MATERIAL

The file FIGURES.zip contains coloured versions of the figures (strucla cake and three output files from Dedan program).

SIMPLE EXAMPLES

The file EXAMPLES.zip [106] contains several examples of parallel systems for verification. All examples are prepared in IMDS source code of the server view. The examples are converted by the Dedan program to the agent view. The source codes are included in files which names end with 'S'. The agent views of the examples are contained in files which names end with 'T'.

1. BufS.txt – bounded buffer
   It is a no-deadlock example, a bounded buffer with readers and writers. Macrogeneration constants defining numbers of elements in the buffer, readers and writers are used.
2. BufDeadlockS.txt – buffer with a deadlock
   Users act as readers-writers, therefore a deadlock occurs when all N users read from the empty buffer or all N users write to the full buffer.
3. BufNoDeadlockS.txt – buffer without a deadlock
   The solution uses Dijkstra's butlers which limit the number of reading and writing users. The butlers reject attempts to be the N-th reader or N-th writer.
4. phil3S.txt – 3 philosophers
   The Dijkstra's problem of dining philosophers, limited to 3 philosophers. Deadlock occurs.
5. phil3_asymS.txt – 3 philosophers, asymmetric solution
   The solution based on the order of obtaining the forks: one philosopher takes the left fork first and one takes the right fork first.
6. phil3_2orNo_S.txt – 3 philosophers, all-or-nothing solution
   The solution based on taking two forks if both are available, on no fork if at least one fork is taken.
7. phil3_butlersS.txt – 3 philosophers, butlers solution
   The two butlers are responsible for a prohibition of taking all three forks as least first and of taking all three forks as right first.

8. Res_DeadlockS.txt – not enough resources
   A common server containing 2 resources, ordered by 3 users. The server view shows no communication deadlock, as the server works. The agent view shows a resource deadlock, as one the users suffers from a lack of resources. Only one of the agents is deadlocked, test for common deadlock does not show a deadlock.
9. SemS.txt – two semaphores and a side process not using the semaphores (the example in Section 5)
   Two users use two semaphores, in crossed order: the first user issues `wait` on `sem[1]` then on `sem[2]`, the second uses issues `wait` in opposite order, the third process does something else.
10. sem_noDeadlockS.txt – two semaphores – no deadlock
    Similar to the previous example, yet the semaphores are taken in the uniform order.

REFERENCES

[1] Jia, W. and Zhou, W. (2005) *Distributed Network Systems. From Concepts to Implementations* Network Theory and Applications. Springer, New York. ISBN: 978-0-387-23839-5

[2] Chrobot, S. and Daszczuk, W. B. (2006) Communication Dualism in Distributed Systems with Petri Net Interpretation. *Theoretical and Applied Informatics*, **18**(4), 261–278. https://taai.iitis.pl/taai/article/view/250/taai-vol.18-no.4-pp.261 (7.07.2016)

[3] Reniers, M. A. and Willemse T. A. C. (2011) Folk Theorems on the Correspondence between State-Based and Event-Based Systems. *37th Conference on Current Trends in Theory and Practice of Computer Science*, Novy Smokovec, Slovakia, 22-28 January 2011. pp. 494–505. Springer-Verlag, Berlin Heidelberg. DOI 10.1007/978-3-642-18381-2_41

[4] Holt, R. C. (1972) Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, **3**(4), 179–196. DOI 10.1145/356603.356607

[5] Chandy, K. M. and Lamport, L. (1985) Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, **3**(1), 63–75. DOI 10.1145/214451.214456

[6] Chandy, K. M., Misra, J. and Haas, L. M. (1983) Distributed deadlock detection. *ACM Transactions on Computer Systems*, **1**(2), 144–156. DOI 10.1145/357360.357365

[7] Elmagarmid, A. K. (1986) A survey of distributed deadlock detection algorithms. *ACM SIGMOD Record*, **15**(3), 37–45. DOI 10.1145/15833.15837

[8] Mitchell, D. P. and Merritt, M. J. (1984) A distributed algorithm for deadlock detection and resolution. *Third annual ACM symposium on Principles of distributed computing – PODC '84*, Vancouver, Canada, 27-29 August 1984. pp.282–284. ACM Press, New York, NY. DOI 10.1145/800222.806755

[9] Agarwal, R. and Stoller, S. D. (2006) Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. *Proc. of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD-IV), ISSTA*, Portland, ME, 17-20 July 2006, pp.51–59. ACM Press, New York, NY. DOI 10.1145/1147403.1147413

[10] Knapp, E. (1987) Deadlock detection in distributed databases. *ACM Computing Surveys*, **19**(4), 303–328. DOI 10.1145/45075.46163

[11] Zhou, J., Chen, X., Dai, H., Cao, J. and Chen, D. (2004) M-Guard: A New Distributed Deadlock Detection Algorithm Based on Mobile Agent Technology. *2nd international conference on Parallel and Distributed Processing and Applications ISPA'04*, Hong Kong, China, 13-15 December 2004. pp.75–84. Springer-Verlag, Berlin Heidelberg. DOI 10.1007/978-3-540-30566-8_13

[12] Hilbrich, T., de Supinski, B. R., Schulz, M. and Müller, M. S. (2009) A graph based approach for MPI deadlock detection. *23rd international conference on Conference on Supercomputing - ICS '09*, Yorktown Heights, NY, 8-12 June 2009. pp.296–305. ACM Press, New York, NY. DOI 10.1145/1542275.1542319

[13] Hilbrich, T., de Supinski, B. R., Nagel, W. E., Protze, J., Baier, C. and Müller, M. S. (2013) Distributed wait state tracking for runtime MPI deadlock detection. *International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, Denver, CO, 17-21 November 2013, pp.1–12. ACM Press, New York, NY. DOI 10.1145/2503210.2503237

[14] Natarajan, N. (1986) A distributed scheme for detecting communication deadlocks. *IEEE Transactions on Software Engineering*, **SE-12**(4), 531–537. DOI 10.1109/TSE.1986.6312900

[15] Hosseini, R. and Haghighat, A. (2005) An Improved Algorithm for Deadlock Detection and Resolution in Mobile Agent Systems. *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, Vienna, Austria, 28-30 November 2005, Vol.2, pp.1037–1042. IEEE, New York, NY.

[16] Allen, G. E., Zucknick, P. E. and Evans, B. L. (2007) A Distributed Deadlock Detection and Resolution Algorithm for Process Networks. *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, 15-20 April 2007, Honolulu, HI, Vol.2, pp.II-33–II-36. IEEE, New York, NY. DOI 10.1109/ICASSP.2007.366165

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

[18] Singhal, M. (1989) Deadlock detection in distributed systems. *Computer*, **22**(11), 37–48. DOI 10.1109/2.43525

[19] Choudhary, A., Kohler, W., Stankovic, J. and Towsley, D. (1989) A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions on Software Engineering*, **15**(1), 10–17. DOI 10.1109/32.21721

[20] Yeung, C.-F., Huang, S.-L., Lam, K.-Y. and Law, C.-K. (1994) A new distributed deadlock detection algorithm for distributed database systems. *IEEE Region 10's 9th Annual International Conference on: 'Frontiers of Computer Technology'*, TENCON'94, Singapore, 22-26 August 1994, Vol. 1. pp.506–510. IEEE, New York, NY. DOI 10.1109/TENCON.1994.369249

[21] Park, Y. and Scheuermann, P. (1991) A deadlock detection and resolution algorithm for sequential transaction processing with multiple lock modes. *Fifteenth Annual International Computer Software & Applications Conference*, Tokyo, Japan, 11-13 September 1991. pp.70–77. IEEE Comput. Soc. Press, New York, NY. DOI 10.1109/CMPSAC.1991.170154

[22] Grover, H. and Kumar, S. (2013) Analysis of Deadlock Detection And Resolution Techniques in Distributed Database Environment. *International Journal of Computer Engineering & Science* , **2**(1), 17–25. http://www.ijsrp.org/research-paper-0915/ijsrp-p4581.pdf (7.07.2016)

[23] Li, T., Ellis, C. S., Lebeck, A. R. and Sorin, D. J. (2005) Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. *USENIX Annual Technical Conference*, Berkeley, CA, pp.31–44. USENIX, Berkeley, CA. https://usenix.org/legacy/publications/library/proceedings/usenix05/tech/general/full_papers/li/li.pdf (7.07.2016)

[24] Huang, S.-T. (1989) Detecting termination of distributed computations by external agents. *The 9th International Conference on Distributed Computing Systems*, Newport Beach, CA, 5-9 June 1989, pp.79–84. IEEE Comput. Soc. Press, New York, NY. DOI 10.1109/ICDCS.1989.37933

[25] Mattern, F. (1989) Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, **30**(4), 195–200.

[26] Peri, S. and Mittal, N. (2004) On Termination Detection in an Asynchronous Distributed System. *17th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS)*, San Francisco, CA, 15-17 September 2004. pp.209–215. International Society for Computers and Their Applications, Cary, North Carolina http://www.iith.ac.in/~sathya_p/index_files/PDCS-Transtd.pdf (7.07.2016)

[27] Clarke, E., Grumberg, O. and Peled, D. (1999) *Model Checking*. MIT Press, Cambridge, MA. ISBN:0-262-03270-8

[28] Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T. H. and Peled, D. (2011) Efficient deadlock detection for concurrent systems. *9th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011*, 11-13 July 2011, Cambridge, UK, pp.119–129. IEEE, New York, NY. DOI 10.1109/MEMCOD.2011.5970518

[29] Miller, S. P., Whalen, M. W. and Cofer, D. D. (2010) Software model checking takes off. *Communications of the ACM*, **53**, 58–64.

[30] Kaveh, N. (2000) Using Model Checking to Detect Deadlocks in Distributed Object Systems. In Emmerich, W. and Tai, S. (eds.), *2nd International Workshop on Distributed Objects*, Davis, CA, 2-3 November 2000, LNCS vol.1999. pp.116–128. Springer-Verlag, London, UK. DOI 10.1007/3-540-45254-0_11

[31] Holzmann, G. J. (1995) Tutorial: Proving properties of concurrent systems with SPIN. *6th International Conference on Concurrency Theory, CONCUR'95*, Philadelphia, PA, 2124 August 1995, pp.453–455. Springer-Verlag, Berlin Heidelberg. DOI 10.1007/3-540-60218-6_34

[32] Corbett, J. (1996) Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, **22**(3), pp.161–180. DOI 10.1109/32.489078

[33] Puhakka, A. and Valmari, A. (2000) Livelocks, Fairness and Protocol Verification. *16th World Conf. on Software: Theory and Practice*, Beijing, China, 21-25 August 2000, IFIP, pp.471–479. International Federation for Information Processing, Laxenburg, Austria. http://www.cs.tut.fi/ohj/VARG/publications/00-3.ps (7.07.2016)

[34] Holzmann, G. (1997) The model checker SPIN. *IEEE Transactions on Software Engineering*, **23**(5), 279–295. DOI 10.1109/32.588521

[35] Havelund, K. and Pressburger, T. (2000) Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, **2**(4), 366–381. DOI 10.1007/s100090050043

[36] Magee, J. and Kramer, J. (1999) *Concurrency: Models and Programs - From Finite State Models to Java Programs*. John Wiley, Chichester. ISBN 0470093552

[37] Cho, J., Yoo, J. and Cha, S. (2006) NuEditor – A Tool Suite for Specification and Verification of NuSCR. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, LNCS 3647, Springer, Berlin Heidelberg, 19–28. DOI 10.1007/11668855_2

[38] Royer, J. (2001) Formal specification and temporal proof techniques for mixed systems. *15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, San Francisco, CA, pp.1542–1551. IEEE Comput. Soc., New York, NY. DOI 10.1109/IPDPS.2001.925139

[39] Perna, J. and George, C. (2006) Model checking RAISE specifications. Technical report. United Nations University. http://i.unu.edu/media/unu.edu/publication/1550/report331.pdf (7.07.2016)

[40] Mazuelo, C. L. (2008) Automatic Model Checking of UML models. Master's thesis. Universitat Berlin. http://www.iam.unibe.ch/tilpub/2008/lar08.pdf (7.07.2016)

[41] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L. and Schnoebelen, P. (2001) *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer-Verlag, Berlin Heidelberg. ISBN 978-3-662-04558-9

[42] Kokash, N. and Arbab, F. (2008) Formal Behavioral Modeling and Compliance Analysis for Service-Oriented Systems. In de Boer, F. S. (ed.), *Formal Methods for Components and Objects - 7th International Symposium, FMCO 2008*, Sophia

Antipolis, France, 21-23 October, 2008, LNCS 5751, pp.21–41. Springer, Berlin Heidelberg. DOI 10.1007/978-3-642-04167-9_2

[43] Arcaini, P., Gargantini, A. and Riccobene, E. AsmetaSMV: a model checker for AsmetaL models – Tutorial. Technical report TR120. DTI Univ. of Milan. https://air.unimi.it/retrieve/handle/2434/69105/96882/Tutorial_AsmetaSMV.pdf (7.07.2016)

[44] Corbett, J. C. (1994) An empirical evaluation of three methods for deadlock analysis of Ada tasking programs. *International symposium on Software testing and analysis*, ISSTA'94, Seattle, WA, 17-19 August 1994, pp.204–215. ACM Press, New York, NY. DOI 10.1145/186258.187206

[45] Milner, R. (1984) *A Calculus of Communicating Systems*. Springer-Verlag, Berlin Heidelberg. ISBN 0387102353

[46] De Francesco, N., Santone, A. and Vaglini, G. (1998) State space reduction by non-standard semantics for deadlock analysis. *Science of Computer Programming*, **30**(3), 309–338. DOI 10.1016/S0167-6423(97)00017-8

[47] Shi, L. and Liu, Y. (2010) Modeling and verification of transmission protocols: A case study on CSMA/CD protocol. *SSIRI-C 2010 - 4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion*, Singapore, 9-11 June 2010. 143–149. IEEE Computer Society, Washington, DC. DOI 10.1109/SSIRI-C.2010.33

[48] Hiraishi, H. (2000) Verification of deadlock free property of high level robot control. *Ninth Asian Test Symposium ATS 2000*, Taipei, 4-6 December 2000. pp.198–203. IEEE Comput. Soc., New York, NY. DOI 10.1109/ATS.2000.893625

[49] Kern, C. and Greenstreet, M. R. (1999) Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, **4**(2), 123–193. DOI 10.1145/307988.307989

[50] Havelund, K. and Skakkebaek, J. U. (1999) Applying Model Checking in Java Verification *5th and 6th International SPIN Workshops*, Trento, Italy, 5 July 1999, Toulouse, France, 21 and 24 September 1999. pp. 216–231. Springer-Verlag, London, UK. DOI 10.1007/3-540-48234-2_17

[51] Chang, F. S.-H. and Jackson, D. (2006) Symbolic model checking of declarative relational models. *28th international conference on Software engineering - ICSE 06*, Shanghai, China, 20-28 May 2006. pp.312–320. ACM Press, New York, NY. DOI 10.1145/1134285.1134329

[52] Benes, N., Cerna, I., Sochor, J., Varekova, P. and Zimmerova, B. (2008) A Case Study in Parallel Verification of Component-Based Systems. *Electronic Notes in Theoretical Computer Science*, **220**(2), 67–83. DOI 10.1016/j.entcs.2008.11.014

[53] Cordeiro, L. (2010) SMT-based bounded model checking for multi-threaded software in embedded systems. *32nd ACM/IEEE International Conference on Software Engineering*, ICSE10, Cape Town, RSA, 2-8 May 2010. Vol. 2, pp.373–376. ACM Press, New York, NY. DOI 10.1145/1810295.1810396

[54] Inverso, O., Nguyen, T. L., Fischer, B., La Torre, S. and Parlato, G. (2015) Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, 9-13 November 2015. pp.807–812. IEEE, New York, NY. DOI 10.1109/ASE.2015.108

[55] Heussner, A., Poskitt, C. M., Corrodi, C. and Morandi, B. (2015) Towards Practical Graph-Based Verification for an Object-Oriented Concurrency Model. *Electronic Proceedings in Theoretical Computer Science*, 181, 32–47. DOI 10.4204/EPTCS.181.3

[56] Yang, Y., Chen, X. and Gopalakrishnan, G. (2008) *Inspect: A Runtime Model Checker for Multithreaded C Programs*, Report UUCS-08-004, University of Utah, Salt Lake City, UT. http://www.cs.utah.edu/docs/techreports/2008/pdf/UUCS-08-004.pdf (7.07.2016)

[57] Attie, P. C. (2015) Synthesis of large dynamic concurrent programs from dynamic specifications. *Formal Methods in System Design*, **47**(131), 1–54. DOI 10.1007/s10703-016-0252-9

[58] Zibaeenejad, M. H. and Thistle, J. G. (2014) Weak Invariant Simulation and Its Application to Analysis of Parameterized Networks. *IEEE Transactions on Automatic Control*, **59**(8), pp.2024–2037. DOI 10.1109/TAC.2014.2315311

[59] Baier, C. and Katoen, J.-P. (2008) *Principles of Model Checking*. MIT Press, Cambridge, MA. ISBN:9780262026499

[60] Fahland, D., Favre, C., Koehler, J., Lohmann, N., Voelzer, H., Wolf, K. (2011) Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data & Knowledge Engineering*, **70**(5), 448–466. DOI 10.1016/j.datak.2011.01.004

[61] Joosten, S. J. C., Julien, F. V. and Schmaltz, J. (2014) WickedXmas: Designing and Verifying on-chip Communication Fabrics. *3rd International Workshop on Design and Implementation of Formal Tools and Systems, DIFTS14*, Lausanne, Switzerland, 20 October 2014, pp.1–8. Technische Universiteit Eindhoven, The Netherlands https://pure.tue.nl/ws/files/3916267/889737443709527.pdf (7.07.2016)

[62] Martens, M. (2009) *Establishing Properties of Interaction Systems*. PhD Thesis, University of Mannheim, Germany. http://ub-madoc.bib.uni-mannheim.de/2629/1/DrArbeitMyThesis.pdf (7.07.2016)

[63] Duri, S., Buy, U., Devarapalli, R. and Shatz, S. M. (1994) Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Transactions on Software Engineering and Methodology*, **3**(4), 340–380. DOI 10.1145/201024.201038

[64] Manna, Z. and Pnueli, A. (1992) *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, NY. ISBN:978-1-4612-6950-2 DOI 10.1007/978-1-4612-0931-7

[65] Yoneda, T. and Schlingloff, H. (1992) On Model Checking for Petri Nets and a Linear Time Temporal Logic. *IEICE FTS'92 workshop*, Kyoto, Japan, June 1992. https://www2.informatik.hu-berlin.de/~hs/Publikationen/1992_IEICE-

FTS_Yoneda-Schlingloff_Model-Checking-for-Petri-Nets-and-Linear-Temporal-Logic-using-Partial-Orders.pdf (7.07.2016)

[66] Liu, S. (2014) *Formal Modeling and Analysis Techniques for High Level Petri Nets*. PhD Thesis, Florida International University, Miami, FL. http://digitalcommons.fiu.edu/cgi/viewcontent.cgi?article=2592&context=etd (7.07.2016)

[67] Guan, X., Li, Y., Xu, J., Wang, C. and Wang, S. (2012) A Literature Review of Deadlock Prevention Policy Based on Petri Nets for Automated Manufacturing Systems. *International Journal of Digital Content Technology and its Applications*, **6**(21), 426–433. DOI 10.4156/jdcta.vol6.issue21.48

[68] Godefroid, P. and Wolper, P. (1991) Using partial orders for the efficient verification of deadlock freedom and safety properties. *3rd International Workshop, CAV '91*, Aalborg, Denmark, 1-4 July, 1991, LNCS 575, pp.332–342. Springer, Berlin Heidelberg. DOI 10.1007/3-540-55179-4_32

[69] Penczek, W., Szreter, M., Gerth, R. and Kuiper, R. (2000) Improving Partial Order Reductions for Universal Branching Time Properties. *Fundamenta Informaticae*, **43**(1-4), 245–267. DOI 10.3233/FI-2000-43123413

[70] Gerth, R., Kuiper, R., Penczek, W. and Szreter, M. (1999) Partial Order Reductions Preserving Simulations. *Concurrency Specification and Programming (CS&P)*, Warsaw, Poland, 28-30 September 1999, pp.153–171. http://www.ipipan.waw.pl/~penczek/WPenczek/papersPS/IPI843-97.ps.gz (7.07.2016)

[71] Daszczuk, W. B. (2003) Verification of Temporal Properties in Concurrent Systems. PhD thesis, Warsaw University of Technology. DOI 10.13140/RG.2.1.2779.6565

[72] Valmari, A. (1994) State of the Art Report: STUBBORN SETS. *Petri Net Newsletter*, **46**, 6–14. http://www.cs.tut.fi/ohj/VARG/publications/94-1.ps (7.07.2016)

[73] Valmari, A. and Hansen, H. (2010) Can Stubborn Sets Be Optimal? *31st International Conference, PETRI NETS 2010*, Braga, Portugal, 21-25 June 2010, pp.43–62. DOI 10.1007/978-3-642-13675-7_5 (12.05.2016)

[74] Chu, F. and Xie, X.-L. (1997) Deadlock analysis of Petri nets using siphons and mathematical programming. *IEEE Transactions on Robotics and Automation*, **13**(6), 793–804. DOI 10.1109/70.650158

[75] Tricas, F., Colom, J. M. and Merelo, J. J. (2014) Computing minimal siphons in Petri Net models of resource allocation systems: An evolutionary approach. *CEUR Workshop Proceedings, Petri Nets and Software Engineering*, Tunis, Tunisia, 23-24 June 2014, pp.307–322. Univesität Hamburg, Germany http://ceur-ws.org/Vol-1160/paper18.pdf (7.07.2016)

[76] Wegrzyn, A., Karatkevich, A. and Bieganowski, J. (2004) Detection of Deadlocks and Traps in Petri Nets by means of Thelen's Prime Implicant Method. *Int. Journal of Ajpplied Math and Computer Science*, **14**(1), 113–121. https://www.amcs.uz.zgora.pl/?action=download&pdf=AMCS_2004_14_1_13.pdf (7.07.2016)

[77] Tricas, F., Garcia-Valles, F., Colom, J.M. and Ezpeleta, J. A Petri Net Structure Based Deadlock Prevention Solution for Sequential Resource Allocation Systems *2005 IEEE International Conference on Robotics and Automation*, Barcelona, Spain, 18-22 April 2005, pp. 271–277. DOI 10.1109/ROBOT.2005.1570131

[78] Bertino, E., Chiola, G. and Mancini, L. V. (1998) Deadlock Detection in the Face of Transaction and Data Dependencies. *19th International Conference ICATPN98*, 22-26 June 1998, Lisbon, Portugal, pp.266–285. Springer, Berlin Heidelberg. DOI 10.1007/3-540-69108-1_15

[79] Leibfried, T. F. (1989) A deadlock detection and recovery algorithm using the formalism of a directed graph matrix. *ACM SIGOPS Operating Systems Review*, **23**(2), 45–55. DOI 10.1145/858344.858348

[80] Ni, Q., Sun, W. and Ma, S. (2009) Deadlock Detection Based on Resource Allocation Graph. *5th International Conference on Information Assurance and Security*, Xian, China, 18-20 August 2009. pp.135–138. IEEE, New York, NY. DOI 10.1109/IAS.2009.64

[81] Nguyen, H. H. C., Le, V. S. and Nguyen, T. T. (2014) Algorithmic approach to deadlock detection for resource allocation in heterogeneous platforms. *International Conference on Smart Computing*, Hong Kong, 3-5 November 2014, pp.97–103. IEEE, New York, NY. DOI 10.1109/SMARTCOMP.2014.7043845

[82] Petrovic, T., Bogdan, S. and Sindicic, I. (2009) Determination of circular waits in multiple-reentrant flowlines based on machine-job incidence matrix. *European Control Conference (ECC) 2009*, Budapest, Hungary, 23-26 August 2009, pp.4463–4468. IEEE, New York, NY. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7075103 (7.07.2016)

[83] Kaveh, N. and Emmerich, W. (2001) Deadlock detection in distribution object systems. *8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-9*, Vienna, Austria, 10-14 September 2001, pp.44–51. ACM Press, New York, NY. DOI 10.1145/503209.503216

[84] Karatkevich, A. and Grobelna, I. (2014) Deadlock detection in Petri nets: One trace for one deadlock? *2014 7th International Conference on Human System Interactions (HSI)*, Costa da Caparica, Lisbon, Portugal, 16-18 June 2014. pp.227–231. IEEE, New York, NY. DOI 10.1109/HSI.2014.6860480

[85] Karacali, B., Tai K.-C. and Vouk, M. (2000) Deadlock detection of EFSMs using simultaneous reachability analysis. *International Conference on Dependable Systems and Networks*, DSN 2000, New York, NY, 25-28 June 2000. pp.315–324. IEEE Comput. Soc., New York, NY. DOI 10.1109/ICDSN.2000.857555

[86] Geilen, M. and Basten, T. (2003) Requirements on the Execution of Kahn Process Networks. *ESOP'03 the 12th European Symposium on Programming*, Warsaw, Poland, 7-11 April 2003, LNCS 2618, pp.319–334. Springer-Verlag, Berlin Heidelberg. DOI 10.1007/3-540-36575-3_22

[87] Longley, D. and Shain, M. (1986) *Dictionary of information technology*. Macmillan Press, London. ISBN 0-19-520519-7

[88] Masticola, S. and Ryder, B. (1990) Static Infinite Wait Anomaly Detection in Polynomial Time. *1990 Inter-*

national Conference on Parallel Processing, Urbana-Champaign, IL, 13-17 August 1990. Vol.2, pp.78–87. Penn State University Press, University Park, PA. ISBN: 978-0-271-00728-1, Volume 2: Software

[89] Coffman, E. G., Elphick, M. and Shoshani, A. (1971) System Deadlocks. *ACM Computing Surveys*, **3**(2), 67–78. DOI 10.1145/356586.356588

[90] Isloor, S. and Marsland, T. (1980) The Deadlock Problem: An Overview. *Computer*, **13**(9), 58–78. DOI 10.1109/MC.1980.1653786

[91] Huang, C.-M. and Huang, D.-T. (1993) A backward protocol verification method. *TENCON '93. IEEE Region 10 International Conference on Computers, Communications and Automation*, Beijing, China, 19-21 October 1993, Vol.1, 515–518. IEEE, New York, NY. DOI 10.1109/TENCON.1993.320039

[92] Sharma, N. K. and Bhargava, B. (1987) *A Robust Distributed Termination Detection Algorithm*, Report 87-726, Purdue University Press, West Lafayette, IN. http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1626&context=cstech (7.07.2016)

[93] Dijkstra, E. W. and Scholten, C. S. (1980). Termination detection for diffusing computations, *Information Processing Letters*, **11**(1), 1–4. DOI 10.1016/0020-0190(80)90021-6

[94] Kumar, D. (1985). A class of termination detection algorithms for distributed computations. *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, 1618 December 1985, pp.73–100. Springer-Verlag, Berlin Heidelberg DOI 10.1007/3-540-16042-6_4

[95] Mattern, F. (1987) Algorithms for distributed termination detection. *Distributed Computing*, **2**(3), 161–175. DOI 10.1007/BF01782776

[96] Matocha, J. and Camp, T. (1998) A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, **43**(3), 207–221. DOI 10.1016/S0164-1212(98)10034-1

[97] Chalopin, J., Godard, E., Metivier, Y. and Tel, G. (2007) About the Termination Detection in the Asynchronous Message Passing Model. *33rd Conference on Current Trends in Theory and Practice of Computer Science*, Harrachov, Czech Republic, 20-26 January 2007, pp.200–211. Springer, Berlin Heidelberg. DOI 10.1007/978-3-540-69507-3_16

[98] Kshemkalyani, A. D. and Mukesh, S. (2008) *Distributed Computing. Principles, Algorithms, and Systems.* Cambridge University Press, Cambridge, UK. ISBN:978-0-521-87634-6

[99] Ma, G. (2007) *Model checking support for CoreASM: model checking distributed abstract state machines using Spin*, MSc Thesis. Simon Fraser University, Burnaby, Canada. http://summit.sfu.ca/system/files/iritems1/8056/etd2938.pdf (7.07.2016)

[100] Ray, I. and Ray, I. (2001) Detecting Termination of Active Database Rules Using Symbolic Model Checking. *5th East European Conference on Advances in Databases and Information Systems*, ADBIS 01, Vilnius, Lithuania, 2528 September 2001, pp.266–279. Springer-Verlag, London, UK. DOI 10.1007/3-540-44803-9_21

[101] Garanina, N. O. and Bodin, E. V. (2014) Distributed Termination Detection by Counting Agent. *23th International Workshop on Concurrency, Specification and Programming*, CS&P 2014, Chemnitz, Germany, 29 September – 1 October 2014, pp.69–79. Humboldt University, Berlin, Germany. http://ceur-ws.org/Vol-1269/paper69.pdf (7.07.2016)

[102] Podelski, A. and Rybalchenko, A. (2003) *Software Model Checking of Liveness Properties via Transition Invariants*, Technical Report, Max Planck Institite fur Informatik, 2003

[103] Glabbeek, R. J. and Goltz, U. (1990) Equivalences and refinement. *LITP Spring School on Theoretical Computer Science* La Roche Posay, France, 2327 April 1990, LNCS 469, pp. 309–333. Springer-Verlag, Berlin Heidelberg. DOI 10.1007/3-540-53479-2_13

[104] Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M. (2000) NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, **2**, 410–425. DOI 10.1007/s100090050046

[105] Czejdo, B., Bhattacharya, S., Baszun, M. and Daszczuk, W. B. (2016) Improving Resilience of Autonomous Moving Platforms by real time analysis of their Cooperation. *Autobusy-TEST*, **2016**(6), 1294–1301. http://www.autobusy-test.com.pl/images/stories/Do_pobrania/2016/nr%206/logistyka/10_l_czejdo_bhattacharya_baszun_daszczuk.pdf (8.07.2016)

[106] Dedan Examples, http://staff.ii.pw.edu.pl/dedan/files/examples.zip (7.07.2016)

[107] Gradara, S., Santone, A. and Villani, M. L. (2005) Using heuristic search for finding deadlocks in concurrent systems. *Information and Computation*, **202**(2), 191–226. DOI 10.1016/j.ic.2005.07.004

[108] Gradara, S., Santone, A. and Villani, M. L. (2006) DELFIN+: An efficient deadlock detection tool for CCS processes. *Journal of Computer and System Sciences*, **72**(8), 1397–1412. DOI 10.1016/j.jcss.2006.03.003

[109] Cheng, A. (1995) *Complexity Results for Model Checking.* BRICS Report RS-95-18. University of Aarhus, Denmark. http://ojs.statsbiblioteket.dk/index.php/brics/article/view/19920/17574 (21.09.2016)

[110] Dedan, http://staff.ii.pw.edu.pl/dedan/files/DedAn.zip (7.07.2016)

[111] Alur, R. and Dill, D. L. (1994) A theory of timed automata. *Theoretical Computer Science*, **126**(2), 183–235. DOI 10.1016/0304-3975(94)90010-8

[112] Krystosik, A. (2006) Embedded Systems Modeling Language. *2006 International Conference on Dependability of Computer Systems*, DepCos-RELCOMEX '06, Szklarska Poreba, Poland, 25-27 May 2006, pp.27–34. Springer, Berlin Heidelberg. DOI 10.1109/DEPCOS-RELCOMEX.2006.21