Siphon-based deadlock detection in Integrated Model of Distributed Systems (IMDS)

Wiktor B. Daszczuk
Institute of Computer Science,
Warsaw University of Technology,
Nowowiejska Str. 15/19, 00-665 Warsaw, Poland
e-mail wbd@ii.pw.edu.pl

Abstract—**Integrated Model of Distributed Systems (IMDS) is a formalism for specification and verification of distributed systems, especially following IoT (Internet of Things) paradigm. The formalism emphasizes such features as asynchrony of actions and communication, locality of decisions, and autonomy in executing actions. In conjunction with model checking, IMDS allows to analyze such features of distributed systems as deadlocks or distributed termination. However, the nature of model checking allows to find one deadlock in a single run of the verifier, which produces a counterexample.**

The conversion of IMDS specification to a Petri net is used to identify multiple deadlocks in one verification, using siphons. Model checking is used to verify if a siphon can become empty, which denotes a true deadlock in a purely cyclic system, like FMS (Flexible Manufacturing Systems). The extension of the verification by temporal checking allows to cover systems with any structure: cyclic, terminating, or with a more complex scheme. In addition, the proposed procedure allows to easily identify processes participating in partial deadlocks. Two types of deadlock can be identified: communication deadlocks and resource deadlocks.

I. Introduction

IMDS (Integrated Model of Distributed Systems [1][2]) is a formalism for describing the behavior of distributed systems, especially for finding deadlocks. A system is modeled as a set of actions, having servers' states and agents' messages on input and on output. In IMDS, a communication dualism is exploited, since the modeled system is represented as server processes that communicate by messages, or alternatively by travelling processes (agents) that communicate by means of servers' states. A model of a distributed system is uniform (that is, it has a single form), but it can be decomposed ("cut") to a set of server processes or a set of agent processes. System actions are combined in sequences to form the processes. An action has a current server's state and an agent's current message on input, and it produces a similar pair (a new server's state and a new agent's message) on output.

The two views of a system (server view and agent view) are obtained by the two possible groupings of a set of all actions into sequences. In the server view, actions of an individual server are grouped into a process (the definition of processes is included in Sect. IIIB). The server's states are the carrier of the server process, and the messages are the communication means between server processes. In the agent view, actions concerning an individual agent conform a process. Messages are internal to a process: they are the carrier of the process. The agent processes communicate via servers' states.

The IMDS formalism was used, together with model checking technique [3], to develop the Dedan program which finds various kinds of deadlock in a verified system [4]. These are: communication deadlock (in the server view), resource deadlock (in the agent view), partial deadlock (in which a subset of system's processes participate) and total deadlock (concerning all processes). A counterexample is generated if a deadlock is found. A counterexample is a path leading from the start of the system to the deadlock.

In Dedan, automatic conversion between the server view and the agent view is performed. Also, observation of a global transition graph and simulation on this graph are possible.

Dedan is built in such a way that the specification of temporal formulas and temporal verification are hidden to a user. The reason is that model checking techniques are seldom known by the engineers. Therefore, the program is constructed in such a way that a user specifies a system and simply "pushes the button" to check for the existence of a deadlock.

The model checking technique has a disadvantage: the evaluation of temporal formula consists in finding a single global configuration (will be defined in Section IIIA) which causes the *false* result, which denotes a deadlock. A *counterexample* is a sequence leading from initial configuration to the deadlock. The designer may repair the erroneous specification and run the verification again. The scheme should be repeated multiple times, until all deadlocks are found and repaired.

The other technique of deadlock identification is finding siphons in a Petri net corresponding to a verified IMDS specification. A siphon is a Petri subnet, which cannot restore tokens if it is emptied [5][6][7]. If an empty siphon is

This work was not supported by any organization

reachable, it denotes a deadlock. The deadlock concerns the processes (server processes and/or agent processes) that take part in the siphon. Therefore, it may be total or partial deadlock. Siphon analysis can find multiple deadlocks in a system, because multiple siphons may exist in a net and the algorithms find all siphons in a single run [7].

Siphon-based deadlock detection is used in some purely cyclic classes of Petri nets, used to model FMS (Flexible Manufacturing Systems) [8][9]. However, many systems cannot be modeled as a cyclic Petri net. Some examples of such systems, for example an IoT (Internet of Things) distributed systems with multiple terminating processes, are mentioned in Section VI.

The previous paper [10] concerned identification of deadlocks in IMDS specifications which correspond to purely cyclic systems, like a class of FMS systems. The contribution of this paper is an application of siphon-based deadlock detection to systems of arbitrary schemes: cycling, terminating or intermediate (some processes are cyclic while other ones terminate). For this purpose an IMDS model is converted to a Petri net. Siphon detection is done in the Petri net, while identification of deadlocks and finding processes involved (partial deadlocks and total deadlocks are identified) is performed using reachability verification and temporal analysis in IMDS specification.

As a siphon may be emptied in different ways, thus it may lead to more than one deadlock. Model checking identifies one example of siphon emptying in a single run. Therefore our procedure does not guarantee identification of all deadlocks in a single run, one deadlock is found per reachable empty siphon. Still, a possibility of identification of multiple deadlocks (one for every emptied siphon) in a single procedure is a benefit. Additionally, the described procedure liberates from constraining siphon-based deadlock detection from purely cyclic systems only.

The described procedure gives a possibility of identification of multiple deadlocks in distributed systems specified in IMDS formalism, preserving communication duality, locality and autonomy of distributed components, and asynchrony of actions and communication.

In this paper, a background of static deadlock detection methods is given in Section II. A definition of IMDS is given in Section III. The definition is formulated differently from the paper [10], where a distributed system was defined using four basic sets: servers, state values, services and agents. The present definition is much easier for readers, because it uses two basic sets: states and messages. The previously used four sets are used in IMDS implementation, mentioned in Section IV, where an example of a bounded buffer is presented. The conversion of IMDS specification to a Petri net, and deadlock detection using siphons and reachability is described in Section V. Section VI presents the application of proposed method to systems with various structures, including acyclic and hybrid ones. An example of a not purely cyclic system containing deadlock siphons and no-deadlock siphons is described in Section VII. A practical example is presented in Section VIII. Section IX concludes the paper.

II. Related Work on Deadlock Detection

Many deadlock detection techniques are described in the literature. Dynamic methods typically use some kind of wait-for graph [11] to discover a deadlock (ant typically to prevent a deadlock or to escape from it).

Static methods use a model of a system and explore it to find deadlocks. Model checking techniques are based on temporal reachability space verification. The activities of the system are expressed in terms of local features of its components, and the global reachability space of the system is constructed. The features of system components are given as temporal formulas and verified by the evaluation of them. Model checkers are often equipped with automatic deadlock detection procedures. Typically, deadlock is identified as "a state with no future", i.e., a strongly connected subgraph containing one state only: the deadlock itself [12]. Deadlock freeness is checked by a CTL temporal formula **AG EX** *true* (for any state a next state exists) [13]. Yet, total termination seems to be analogous state: no future exists. In cyclic system, where termination is not expected, the above formula identifies a deadlock. In terminating systems a deadlock should be distinguished from termination.

Temporal formulas can also be used to check partial deadlocks, in which some processes are involved in a deadlock, but other processes continue their run. Generally, in the case of partial deadlock detection, temporal formulas are based on the structure of verified models to identify deadlocks in individual processes [14]. The disadvantage of such an approach is that temporal formulas need to be developed individually for each analyzed system, using its specific features.

Some other approaches to partial deadlock detection use temporal formulas that are not related to the structure of a verified model. However, such model-unrelated formulas require the system to have specific properties [15][16]. If a system is non-terminating (cycling), a discontinuation of a process is obviously a deadlock [3]. Conversely, a method [17] may be ascribed to terminating processes only. Some detection methods are used for specific architectures of systems. For example, WickedXmas approach uses nodes communicating by queues [18].

Other set of static methods concern Petri nets. Some of them are based on analysis of reachability graph of a Petri net [19]. 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. In both approaches it is hard to distinguish a deadlock from distributed termination: these methods are addressed to endlessly cycling systems [20].

Alternatively, structural analysis of Petri net can be used. Structural analysis determines properties of models on the basis of their structure, so no exploration of the reachability

space is needed. Structural analysis of deadlocks is based on subnets called siphons [6][7]. It can be shown that if a model is deadlocked, the unmarked places constitute a siphon. Structural analysis of deadlocks systematically finds elementary siphons. Elementary siphons are ones from which other siphons are composed. After siphons identification, they are checked for unmarking possibility. The advantage of these methods is that multiple deadlocks are found in a single verification and both total and partial deadlocks are identified.

The deadlock detection procedure presented in this paper, based on combining siphon identification with temporal analysis, joins the advantages of the two methods and frees from their disadvantages:

- Identification of multiple deadlocks in single verification run.
- Finding both total and partial deadlocks.
- Distinguishing deadlocks from termination.
- Distinguishing between communication deadlocks and resource deadlocks.
- Automated verification, as deadlocks are expressed as formulas not related to specific features of a verified system.
- Verification of systems having arbitrary shape, without limitation to cycling, terminating or other schemes.

III. Integrated Model of Distributed Systems (IMDS)

A. Basic definition

IMDS is defined in [1][2]. In the present paper, the simplified version of IMDS is used, without dynamic process creation, which is suitable for static model checking. The formalism is founded on a basic observation: nodes on a distributed system (which are called *severs* in IMDS) receive messages, execute some actions changing their *states* upon accepted messages and finally send consecutive *messages*. Thus a distributed system may be defined as a relation between a finite set states of the servers $P=\{p_1,p_2,\ldots\}$ and a finite set of messages $M=\{m_1,m_2,\ldots\}$. The relation Λ defining the actions is:

$\Lambda \subset (M \times P) \times (M \times P)$

For an action $\lambda\in\Lambda$, $\lambda=((m,p),(m',p'))$ the first pair (m,p) is its *input* while the second pair (m',p') is its *output*.

A *configuration* T of a distributed system is a set of *current* states and *pending* messages. The messages in a configuration are current as well, but they are called pending to emphasize the fact that an action extracts from the configuration exactly one message, replacing it with a next message, and all other messages addressed to the action's server remain pending at the server. The system starts from its *initial configuration* T_0, containing initial set of states and messages.

Every action $\lambda=((m,p),(m',p'))$ converts a configuration $T_{inp}(\lambda)$ to a new configuration $T_{out}(\lambda)$ by replacing $\{m,p\}\subset T_{inp}(\lambda)$ with $\{m',p'\}\subset T_{out}(\lambda)$. Behavior of a distributed system is described by a Labeled Transition System LTS [21], containing all executions of the system. *Nodes* of the LTS (not called *states* for unambiguousness) are configurations and *transitions* are actions:

$LTS = \langle Q,q_0,W\rangle \mid$
$Q = \{T_0,T_1,\ldots\}$ (nodes);
$q_0 = T_0$ (initial node);
$W = \{(T,\lambda,T') \mid \lambda \in \Lambda,\ T=T_{inp}(\lambda),\ T'=T_{out}(\lambda)\}$ (transitions)

The interleaving way of executing the action is assumed (one action at a time [22]). Since all transitions in the LTS are instantaneous, it is assumed that message passing and actions take zero time. A timed version of IMDS, in which message passing and actions execution takes some periods of time, is also developed. This feature goes beyond the scope of this paper.

To differentiate between messages sent in different purposes (in a context of separate distributed computations), the autonomous sequential computations in a distributed system are extracted. The messages passed in a context of a given computation conform an *agent*. Thus the set of states P is split into subsets for the servers $1..n$ and the set of messages M into subsets for the agents $1..k$:

$P=\bigcup_{i=1..n} P_i,\ M=\bigcup_{j=1..k} M_j$

The subsets are pairwise disjoint:

$i\neq j\Rightarrow P_i\cap P_j=\emptyset,\ i\neq j\Rightarrow M_i\cap M_j=\emptyset$

The initial configuration contains *initial states* of all servers, one state for every server, and *initial messages* of all agents, one message for every agent:

$T_0\cap P_i=\{p_{0i}\},\ T_0\cap M_i=\{m_{0i}\}$

The input and output state of an action concern the same server and the input and output message of an action concern the same agent:

$\lambda=((m,p),(m',p')),\ \{m,m'\}\subset M_i,\ \{p,p'\}\subset P_j$

Agents may terminate in special actions of the form $\lambda=((m,p),(p'))$, where an output message is absent.

B. Processes

Is it useful to identify processes in a system, especially for verification purposes. Two "classical" models of distributed processes are used: client-server and RPC [23]. In the former model servers communicate by messages while in the latter one processes migrate forth and back. IMDS contains both models in a single specification, the models are extracted as the two perspectives: *server view* and *agent view*. A server process B in the server view is a sequence of actions connected by states of a server, as input and output states of an action concern the same server. Some actions may be unreachable and thus they would become "orphaned" (not included in any process), so the definition is extended to a set of all actions of a given server (rather than a sequence):

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

In the agent view, an agent process C is a sequence of actions connected by messages of an individual agent, because input and output messages of an action concern the same

agent. As for server processes, the definition is extended to a set of all actions of a given agent:

$C_j=\{\lambda\in\Lambda \mid \lambda=((m,p),(m',p')) \vee \lambda=((m,p),(p')),\ m,m'\in M_j\}$

As a result, a distributed system may be decomposed to server processes or to agent processes, giving the server decomposition **B** and agent decomposition **C**:

$\mathbf{B}=\{B_i \mid i=1..n\}$

$\mathbf{C}=\{C_j \mid j=1..k\}$

C. Locality, Asynchrony and Autonomy

An important feature of a distributed system is locality. In IMDS, locality means that no message may cause actions in distinct servers (for a message $m\in M$ and two actions $\lambda_1,\lambda_2\in\Lambda$, $\lambda_1=((m,p_1),(m_1',p_1'))$, $\lambda_2=((m,p_2),(m_2',p_2'))$, $p_1,p_2\in P_i$). Thus, a function *target*: $M\rightarrow S$ assigns a target server for every message. A server component of a message and a state in the input pair of an action must match: $target(m)=s_i$, $p\in P_j$, $i=j$. A configuration contains exactly one state for every server, as this is required for initial configuration, and every action replaces its input state with a next state of the same server. Yet, multiple messages may be pending at given server in a configuration, which is natural. A set of actions in a distributed system determines which messages may be accepted in individual states. If a server allows a message to be accepted, an action is defined for this message together with the current state of the server. If the acceptance of a message is prohibited in a given state, no action is defined for this pair.

Note that every server performs its action autonomously (only the current server's state and the messages pending at this server are considered). Also, the communication is asynchronous: a server process sends a message to some other server process (or in the agent view, an agent sets the server's state for some other agent) regardless of the current situation of a process with which it communicates (and every other process). As a result, the process may be called *autonomous* and *asynchronous*.

D. Deadlocks in IMDS

A deadlock in IMDS is defined as a discontinuation of a process (with an exception of process termination). As there are two views of processes, different type of deadlocks concern server processes communicating by messages and agent processes communicating by states of servers. There may be a communication deadlock that is not a resource deadlock [2].

- a *communication deadlock* of a server process – when there are messages pending at the server, but no matching pair of any message with a current server state will occur;
- a *resource deadlock* of an agent process – when an agent's message is pending at a server but it will never match any current or future state of this server.

For the identification of deadlocks, universal temporal formulas were elaborated [2]. Universality of the formulas means that they are independent on a structure of a given distributed system – only the two facts are concerned: if an action in a process is enabled and if it is executed. Therefore, temporal logic is built inside the verification tool and the user need not know temporal logic nor model checking technique.

The paper [2] presents a terminating distributed system in which two servers, every one containing a semaphore, are used by two agents (the third agent performs some other work to show a detection of a partial deadlock). This shows a communication deadlock in the server view and a resource deadlock in the agent view. The system is presented briefly in Sect VII. Another example [24] is the Automatic Vehicle Guidance System: in the server view the cooperation of the road segment controllers during the piloting of a vehicle is shown, while in the agent view the traffic from the vehicles perspective is presented. The deadlocks in both views are shown. Similar system is mentioned in Sect. VIII. In the IMDS specification of Karlsruhe Production Cell [25], the controllers of individual devices are modeled as servers and the metal plates traveling through the cell are agents. Additional agents server for performing some actions without the plates, for example return to a rest position.

IV. Example – Bounded Buffer

An example of IMDS system is a buffer with producer and consumer agents (each of them starting at its own server). In IMDS notation, sets of servers *S,* agents *A,* state values *V* and services *R* are introduced explicitly. The services are used to distinguish between messages sent in given purposes to the servers (like operations *wait* and *signal* on a semaphore). The messages are triples (a,s,r), $a\in A$, $s\in S$, $r\in R$. The states of servers are pairs (s,v) , $s\in S$, $v\in V$, where v is a value that represents a given state. Thus an action $(m,p)\Lambda(m',p')$ is denoted $((a,s,r),(s,v))\Lambda((a,s',r'),(s,v'))$. Note that the same agent *a* is used in an input and output message, in such a way a computation is continued. Likewise, the same server *s* is used in an input and output state. Also, the same server *s* is in input state and input message, which models an acceptance of a message on a server with its current state.

The server view of the system in Dedan notation is presented below. As in typical programming language, server types are introduced (lines 3, 12), with formal parameters that specify agents and other servers used in actions. Every server has sets of its states (l.4, 13), services (l.5, 14) and actions (l.7-10, 16-19). An action in Dedan is denoted `{a.s.r, s.v}->{a.s'.r', s.v'}`. Servers and agents are declared as variables (l.21-22, server types are omitted in the declaration because they have names equal to variables in this example). Lastly, actual parameters are passed to the servers and initial states are assigned to every server and initial messages are assigned to every agent (l.24-26).

```
1.   #DEFINE N 2
2.   #DEFINE K 1

3.   server: buf(agents A[N];servers S[N]),
4.   services{put, get},
5.   states{elem0,elem[K]},
6.   actions {
7.   <i=1..N>{A[i].buf.put, buf.elem0}
        -> {A[i].S[i].ok_put, buf.elem[1]},
8.   <i=1..N><j=1..K-1>{A[i].buf.put, buf.elem[j]}
        -> {A[i].S[i].ok_put, buf.elem[j+1]},
9.   <i=1..N><j=2..K>{A[i].buf.get, buf.elem[j]}
        -> {A[i].S[i].ok_get, buf.elem[j-1]},
10.  <i=1..N>{A[i].buf.get, buf.elem[1]}
        -> {A[i].S[i].ok_get, buf.elem0}
11.  };

12.  server: S(agents A;servers buf),
13.  services{doSth,ok_put,ok_get}
14.  states{neutral,prod,cons}
15.  actions {
16.  {A.S.doSth, S.neutral}
        -> {A.buf.put, S.prod}
17.  {A.S.doSth, S.neutral}
        -> {A.buf.get, S.cons}
18.  {A.S.ok_put, S.prod}
        -> {A.S.doSth, S.neutral}
19.  {A.S.ok_get, S.cons}
        -> {A.S.doSth, S.neutral}
20.  };

21.  servers buf,S[N];
22.  agents A[N];

23.  init  ->{
24.  <j=1..N>S[j](A[j],buf).neutral,
25.          buf(A[1..N],S[1..N]).elem0,
26.  <j=1..N>A[j].S[j].doSth,
27.  }.
```

Obviously, one can expect two deadlocks in the example: two agents both trying to get from an empty buffer and two agents trying to put to a full buffer. In model checking, a verifier typically shows the former deadlock because it requires two `get` operations to be reached. However, the latter deadlock is reached after three `put` operations, which lengthens a counterexample. A model checker searches for first counterexample and then it stops the evaluation. Therefore the latter deadlock may be reported in a second verification run, after a modification of the system to avoid the former deadlock (if the modification does not repair both deadlocks).

V. Deadlock Detection in a Petri Net Equivalent to IMDS Model

The main task of the Dedan program is identification of deadlocks and distributed termination. The Conversion of an IMDS system to a Petri net offers the designer new possibilities:

- identification of some structural properties: structural conflicts, dead code, etc.,
- temporal properties expressed in terms of Petri net,
- observation of the system in a graphical form,
- graphical simulation of a system run.

For this purpose a possibility of export of a model to a Petri net is included in Dedan. A format of ANDL (Abstract Net Description Language [26]) is used, which is the input of Charlie Petri net analyzer [27][28].

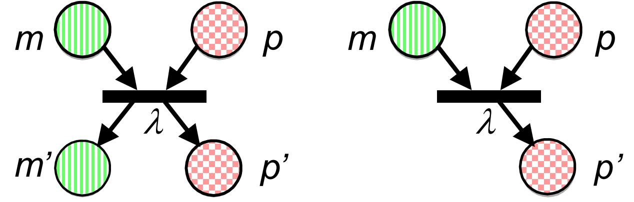

Fig. 1 IMDS actions converted to Petri net transitions: regular action (left) and agent-terminating action (right)

Fig. 1 shows a transition of a Petri net, which corresponds to an IMDS action. The input message and the input state are input places. The output message and the output state are output places (or only the output state in the case of a terminating action, Fig. 1b). The initial marking of the Petri net has tokens in the initial places of server states and initial messages of agents. The graph of reachable markings is equivalent to the LTS of the IMDS system, where states and messages correspond to the places, actions correspond to the transitions and markings correspond to the configurations.

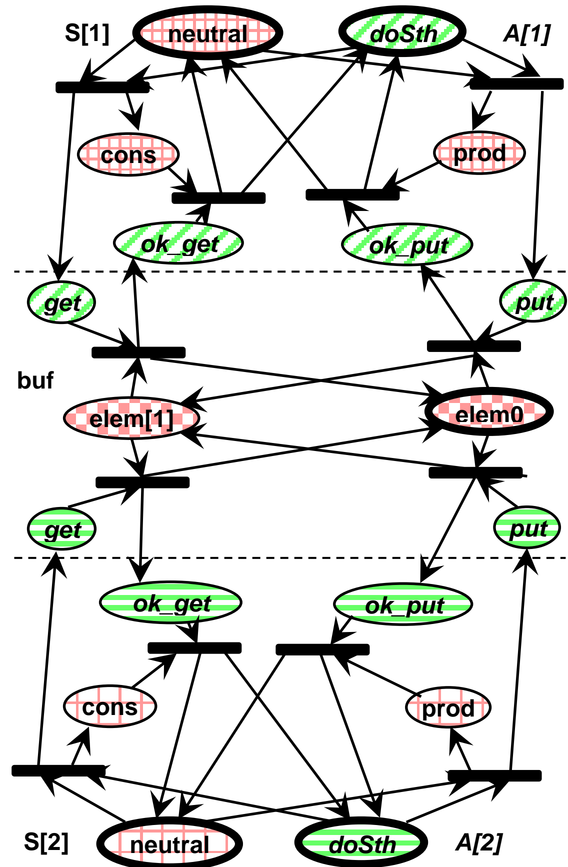

Fig. 2 Petri net representation of the *bounded buffer* system: servers *S*[1..2], *buf*, agents *A*[1..2]

The Petri net corresponding to the *bounded buffer* example is illustrated in Fig. 2. The states and messages in individual servers are grouped and separated by dashed lines. The states of servers are filled red, with patterns individual to every server. The messages are filled green, with patterns individual to every agent. Initial states and initial messages are bold. The servers and their states are in regular font, while the agents and their messages are in italics.

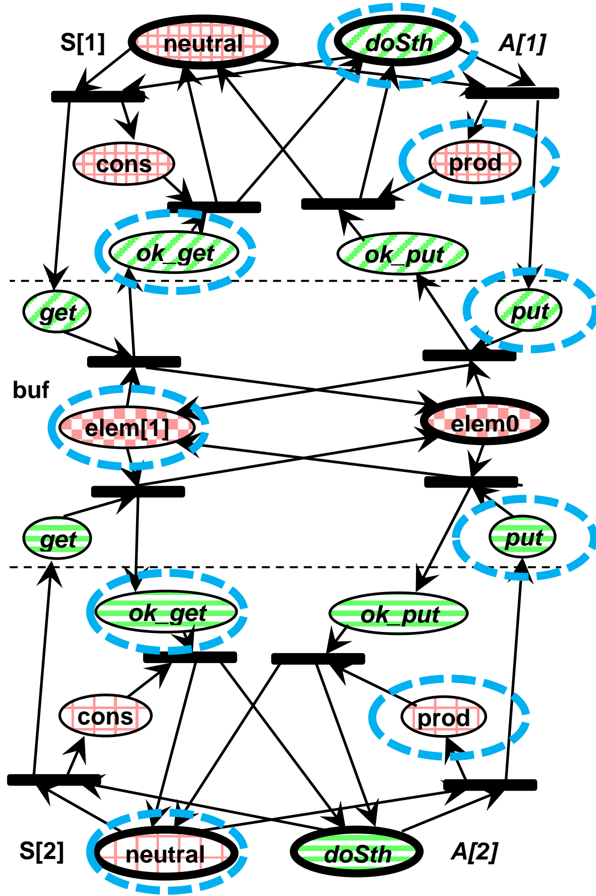

Fig. 3 One of elementary siphons found in the system

As the system falls into a deadlock, there should be siphons that may be emptied. A Petri net may contain a large number of siphons, but some of them are elementary, i.e., they do not contain other siphons [29][30]. Therefore, only elementary siphons need to be analyzed. The Charlie program reports 49 elementary siphons in the net. Every siphon should be tested for a reachability of its emptying. As an empty state place denotes a state which is absent in a configuration, and an empty message place denotes an absent message, an IMDS configuration should be found in which the siphon's states and messages are absent. A siphon may concern not all of the servers (and/or not all of the agents), in such a way partial deadlocks are found.

One of the siphons found in our example is presented in Fig. 3. It contains states (*S*[1],*prod*), (*buf,elem*[1]), (*S*[2],*prod*), (*S*[2],*neutral*) and messages (*A*[1],*S*[1],*ok_get*), (*A*[1],*S*[1],*doSth*), (*A*[1],*buf,put*), (*A*[2],*buf,put*), (*A*[2],*S*[2],*ok_get*).

The siphon emptying is verified by model checking. To do this, the CTL formula **AG** (¬ φ) (it reads: always not φ) is used, where φ is a set of states and messages in a configuration corresponding to a siphon's complement. Often a siphon represents a class of configurations, for example a siphon in Fig. 3 represents all configurations in which server *S*[1] is not in a state (*S*[1],*prod*), and thus it may be in one of a subset of states {(*S*[1],*neutral*), (*S*[1],*cons*)}. The formula for checking if the siphon cannot be emptied has the form **AG** (¬ (((*S*[1],*neutral*)∨(*S*[1],*cons*)) ∧ (*buf,elem0*) ∧ (*S*[2],*cons*) ∧ ((*A*[1],*S*[1],*ok_put*)∨ (*A*[1],*buf,get*)) ∧ ((*A*[2],*S*[2],*doSth*)∨(*A*[2],*S*[2],*ok_put*)∨ (*A*[2],*buf,get*))). For verification, internal Dedan model checker is used for typical cases (as it uses explicit state space) and Uppaal [31] for large cases.

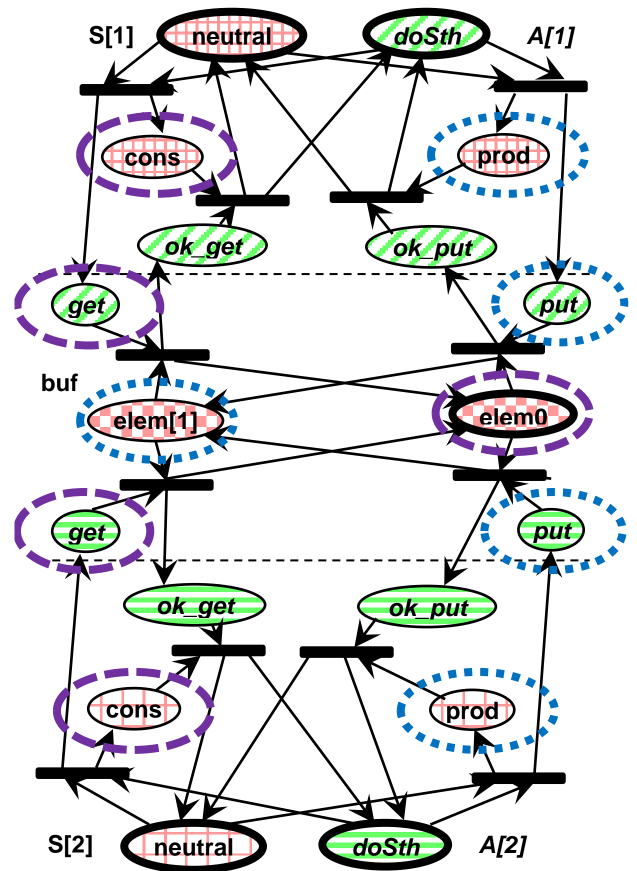

Fig. 4 The two deadlocks identified in the example. The dashed one is associated with the siphon in Fig. 3

The verification results in *false*. This means that the empty siphon is reachable, which denotes a deadlock. States of all three servers *S*[1],*S*[2] and *buf* take part in the siphon, so it is a total communication deadlock. Also, both agents *A*[1] and *A*[2] take part, denoting a total resource deadlock. Model checker generates a counterexample, in which both agents perform *get* on empty buffer (state (*buf,elem0*)). Charlie

reports multiple siphons that may be emptied by two *get* operations on the empty buffer. All these situations constitute a single deadlock (but the counterexamples for individual temporal formulas may differ in the order of issuing *get* by the two agents). The configuration finishing all counterexamples leading to this deadlock is exactly the same. For a partial deadlock, configurations finishing the counterexamples may differ, but only in states/messages of servers/agents not taking part in the deadlock. In the example, all tests for emptying of the siphons either finish in one of the two deadlock configurations, or emptying occurs unreachable (such siphons do not denote a deadlock).

Fig. 4 shows the two possible deadlocks that finish all reachable emptying of siphons. The ovals surround places of the two identified deadlocks: dashed ovals are associated with the siphon in Fig. 3, which it is caused by two *gets* on an empty buffer. The other deadlock (dotted ovals) is caused by two *puts* on a full buffer. Distinguishing between the two deadlocks is based on the two configurations finishing the counterexamples.

VI. Verification of Systems with Various Structures

Various systems can be modeled in IMDS, not only those having a shape of purely cyclic FMS. Fig. 5 shows some examples of shapes of not purely cyclic systems. In the figure “Ending Strongly Connected Subgraph” is a cycle from which no escape is possible. The pictures are schematic, showing general shape of a system. In IMDS specification every transition has two input places and two output places (or one in the case of agent-terminating action), see Fig. 1.

- “linear” systems (like the example of “two semaphores” described in [2]: users issue *wait* on two semaphores, then they issue *signal*),
- a system with a “*leader*” (initial part) and a main loop, sometimes called “lasso-shaped” [32],
- terminating system with a main loop, for example WF-net system [33],
- similar to lasso-shaped, but with an initial loop.

In some cases, for example in the acyclic system in Fig. 5 (on the top), the system may be easily converted to a cyclic one by connecting initial and terminating places. However in the analysis of distributed systems, especially those following the IoT paradigm, in which autonomous nodes agree their coordinated behavior. Such system may have multiple leaders (for every node) and multiple terminating places, where the nodes reach their goals. An example is Automatic Vehicle Guidance System presented in [24]. In IMDS model, servers implement road segment controllers while agents implement the vehicles. Such a system may be additionally complicated if endlessly-looping nodes are added, for example charging stations where serving agents run in cycles.

It is obvious that most of the leaders contain siphons that may be emptied. Yet, emptying of such a siphon does not denote a deadlock because the system runs further. This is the main difference in deadlock detection between purely cyclic and differently shaped systems.

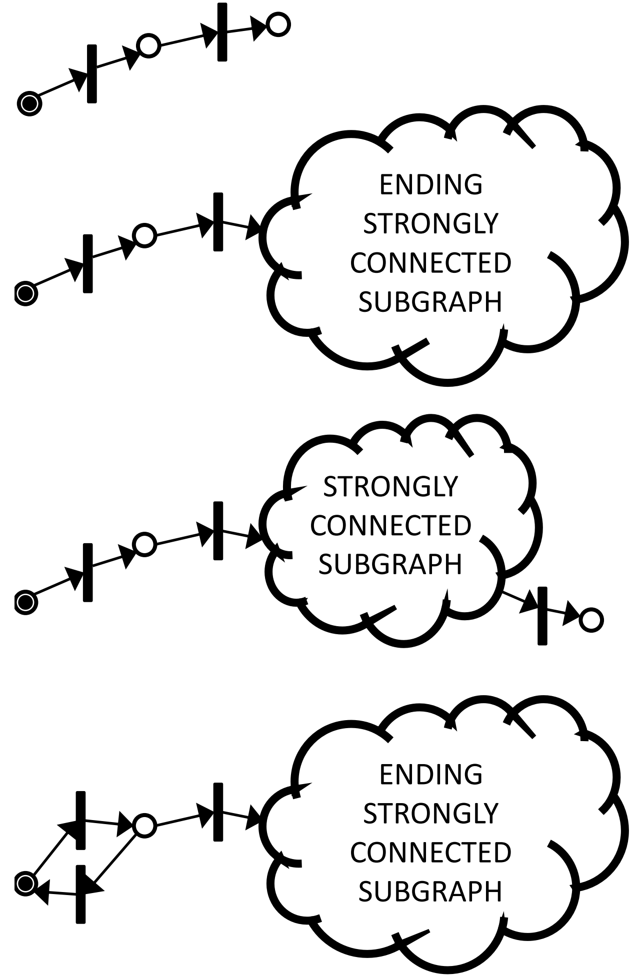

Fig. 5 Examples of acyclic and not purely cyclic systems containing emptyable and reachable siphons which are not deadlocks

A solution of this problem is quite simple: an emptied siphon should be verified if it is a real deadlock or not. This is done using an additional temporal formula, which uses the same subformula φ of an emptied siphon. A deadlock prevents a process from doing any move. Thus, the evaluation of $\mathbf{AG}(\neg\varphi)$ to *false* should be followed by application for every process (server and agent) the formula $\mathbf{AG}(\varphi \Rightarrow \mathbf{EF}\ \neg\varphi$/restricted to a process). The formula reads: always φ is inevitably followed by not φ. Of course, this procedure may be applied to a system of any shape, cycling or not. For the example of emptyable siphon in Fig. 3 (we can pretend that we do not know that the system is purely cyclic) the verification should be performed as follows:

- φ=((*S*[1],*neutral*)$\vee$(*S*[1],*cons*)) $\wedge$ (*buf,elem0*) $\wedge$ (*S*[2],*cons*) $\wedge$ ((*A*[1],*S*[1],*ok_put*)$\vee$(*A*[1],*buf,get*)) $\wedge$ ((*A*[2],*S*[2],*doSth*)$\vee$(*A*[2],*S*[2],*ok_put*)$\vee$(*A*[2],*buf,get*)),
- check $\mathbf{AG}(\varphi \Rightarrow \mathbf{EF}$ (*S*[1],*prod*)) for server *S*[1],

- check $\mathbf{AG}(\varphi \Rightarrow \mathbf{EF}\ (buf,elem[1]))$ for server *buf*,
- check $\mathbf{AG}(\varphi \Rightarrow \mathbf{EF}\ ((S[2],prod) \vee (S[2],neutral)))$ for server $S[2]$,
- check $\mathbf{AG}(\varphi \Rightarrow \mathbf{EF}\ ((A[1],S[1],doSth) \vee (A[1],S[2],ok_get) \vee (A[1],buf,put)))$ for agent $A[1]$,
- check $\mathbf{AG}(\varphi \Rightarrow \mathbf{EF}\ ((A[2],S[2],ok_get) \vee (A[2],buf,put)))$ for agent $A[2]$.

The result depends on a value of a formula for every process:

- *true* for every involved server – no communication deadlock,
- *false* for all involved servers, and all servers are involved – total communication deadlock,
- *false* for some involved servers, or for all servers involved but not all servers are involved – partial communication deadlock,
- *true* for every involved agent – no resource deadlock,
- *false* for all involved agents, and all agents are involved – total resource deadlock,
- *false* for some involved agents, or for all agents involved but not all agents are involved – partial resource deadlock.

In the last three cases concerning agents, only non-terminated agents are taken under consideration, i.e. the agents which messages are present in the configuration corresponding to the emptied siphon.

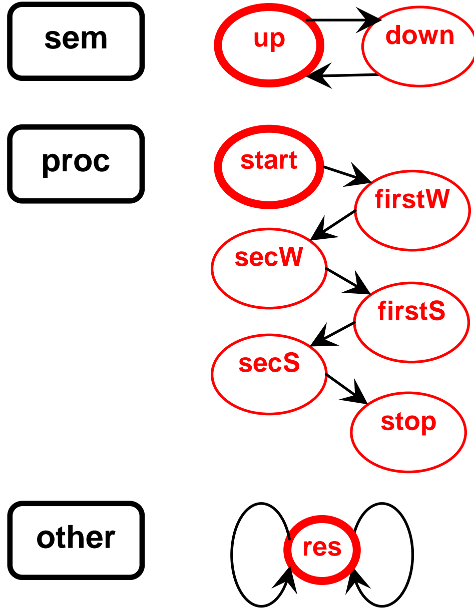

Fig. 6 Automata-like model of servers in *two-semaphores* system, agents not shown

The verification procedure uses several well-known and widely used algorithms. Computational aspects of finding siphons are discussed in [34]: elementary siphons may be found in linear time for a large class of Petri nets (and needs some preprocessings in general case). Also, parallel solutions exist [35].

The complexity of CTL model checking is P-Complete [36], which means that the time of temporal formula evaluation is $|LTS| \times |\psi|$, where $|LTS|$ denotes the number of nodes in a Kripke structure (it is the LTS of a verified system) and $|\psi|$ is the length of a formula ψ. Every formula $\mathbf{AG}(\varphi_1 \Rightarrow \mathbf{EF}\ \varphi_2)$ contains two temporal operators, so this evaluation cost is fixed. The verification should be repeated for every siphon, and according to each siphon for every server and every agent, i.e., the complexity is a number of elementary siphons $\times\ (n+k) \times |LTS|$, where n is a number of servers and k is a number of agents.

VII. Example System with Leaders

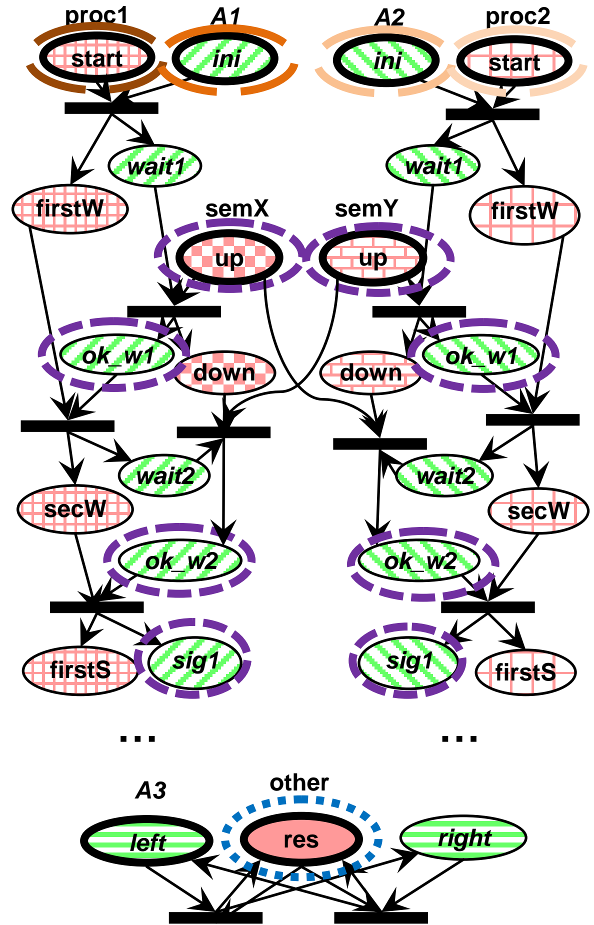

Fig. 7 Petri net representation of the *two-semaphores* system

As an example of system with leaders, we present a *two-semaphores* system consisting of two agents *A1* and *A2*, each one running on its own server (*proc1* and *proc2*). The agents use two semaphores *semX* and *semY*. They use the semaphores "crosswise", i.e., *A1* issues operation *wait* to *semX* than to *semY*, while *A2* does it in opposite order. To show a partial deadlock (not concerning all the servers/agents), the third agent *A3* is added, running on its own server *other* and

performing some looping calculations. The automata-like view of the system is presented in Fig. 6 (only servers' states and actions are shown, input and output messages in actions are omitted).

The system converted to a Petri net if shown in Fig. 7 (a part after second *wait* in the agents is suppressed). The general shape of the system consists in sequences of actions in agents *A1* and *A2*, leading from their start to their termination, and a separated Ending Strongly Connected Subgraph of agent *A3* (depicted in Fig. 8).

A verification in Charlie shows a siphon of an obvious deadlock, shown as places surrounded with denser dashed ovals (violet). This siphon is emptyable and the temporal formulas identifying processes involved show:

- Both agents *A1* and *A2* fall into resource deadlock.
- Both servers *semX* and *semY* fall into communication deadlock.

There is also a siphon containing the place *res* (dotted oval, dark blue), but it is not emptyable - this does not denote a deadlock. There are four such siphons in the system (three of them are not indicated in the figure).

Four siphons formed by places *proc1.start, proc2.start, A1.proc1.ini* and *A2.proc2.ini* are emtyable (they are depicted as sparsely dashed ovals on the top of Fig. 7), but the temporal formulas show that these siphons do not denote deadlocks. They are typical leader siphons.

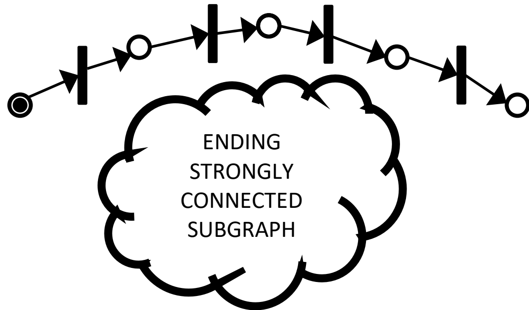

Fig. 8 General shape of the *two-semaphores* system. The chain - servers *proc1, proc2, semX, semY* and agents *A1, A2*. The cycle – server *other* and agent *A3*.

Summing up:

- A partial communication deadlock of processes *semX* and *semY* is identified, in which processes *proc1, proc2* and *other* are not involved.
- A partial resource deadlock of processes *A1* and *A2* is identified, in which process *A3* is not involved.
- Four not emptyable siphons are found (no-deadlock siphons).
- Four emptyable leader siphons are found (no-deadlock siphons).

VIII. Example Application to Automatic Vehicle Guidance System

We chose an Automatic Vehicle Guidance System (AVGS) verification to illustrate an application of our method. The AVGS system consists of a set of road segments (identifiers are taken from cardinal directions) with their controllers modeled as servers, for example on a crossing depicted in Fig. 9. The controllers are very simple: they allow or deny a vehicle to take up a road segment, depending on its freeness or occupation. Vehicles are modeled as agents. When more than one vehicle approaches the crossing, routes for all the vehicles are prepared, for instance using a genetic algorithm. A route connects an entrance segment (*A…*) and a target segment (*T…*) of vehicle's travel. Obviously, in the model every route terminates. The routes are tested for deadlock freeness using siphon detection and temporal verification, described in this paper. This can be performed automatically, because in our methodology deadlock detection formulas are independent on the structure of a verified system. Deadlock-free route sets are executed while routes exposed to deadlocks are rejected. If a new vehicle appears, the whole procedure is repeated from current positions of all the vehicles on the crossing and approaching ones.

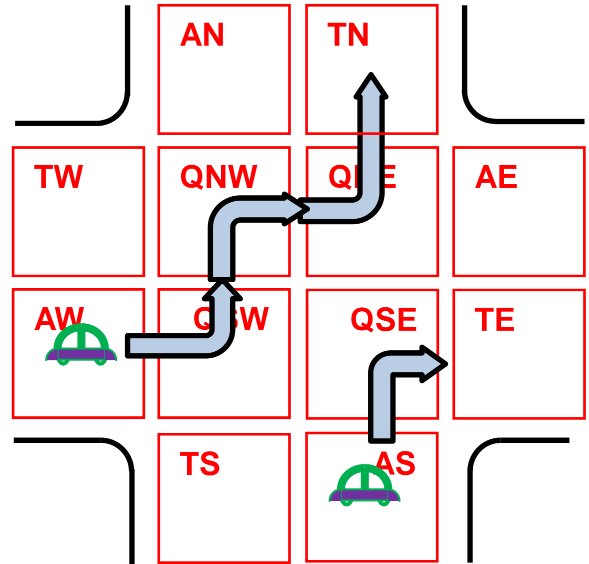

Fig. 9 Automatic Vehicle Guidance System

The described procedure may find solutions unusual in ordinary vehicle traffic, for example one of the routes shown in Fig. 9 causes a vehicle to apply left side traffic for a while. Also, safe routes may be fond even if one road segment is blocked, for example by a broken vehicle. In verification, AVGS is similar to the *two-semaphores* system, without server *other* and agent *A3*.

Other examples of systems that are based on terminating processes, which may be modeled using our approach, is taxi

service [37], business processes [38] or multi-agent systems based on Belief-Desire-Intention paradigm [39]

IX. Conclusions

An approach to deadlock detection is presented which is based on coupling IMDS formalism with Petri net structural analysis and model checking. The methodology allows to find total and partial deadlocks in two perspectives: servers communicating by messages and agents communicating by states of servers. As a result, communication deadlocks and resource deadlocks are identified, which highlights communication duality in distributed systems. Also, the specification in IMDS clearly identifies processes running in a system, which is sometimes difficult in ordinary Petri nets.

The methodology finds multiple deadlocks in a distributed system, preserving locality of decisions, autonomy of servers, and asynchrony of behavior and communication. In some rare cases, in which siphons can be emptied in various ways, not all deadlocks are identified in a single run, but the procedure may be repeated after correction of found deadlocks. However, we have not found any real-life example of a system with such feature, typically all deadlocks are found in a single run.

The presented deadlock detection procedure may be applied for system of arbitrary shape: cycling like a class of FMS systems, terminating like WF-nets, lasso-shaped, or IoT systems compound of multiple terminating and looping autonomous nodes.

A collection of programs is used for the described procedure: Dedan for specification, Charlie for siphon identification, internal Dedan model checker and Uppaal for verification. In the future, the whole procedure will be integrated in Dedan. This will allow to check for deadlocks and to perform other types of structural analysis in a uniform environment.

In the future, a timed version of the prosed procedure is planned, with application of UPPAAL timed automata [40].

Acknowledgment

Extensive discussions with Wlodek Zuberek helped to significantly improve the article, especially in the new formulation of IMDS.

References

[1] S. Chrobot and W. B. Daszczuk, “Communication Dualism in Distributed Systems with Petri Net Interpretation,” *Theor. Appl. Informatics*, vol. 18, no. 4, pp. 261–278, 2006. arXiv: 1710.07907

[2] W. B. Daszczuk, “Communication and Resource Deadlock Analysis using IMDS Formalism and Model Checking,” *Comput. J.*, vol. 60, no. 5, pp. 729–750, 2017. doi: 10.1093/comjnl/bxw099

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

[4] Dedan, http://staff.ii.pw.edu.pl/dedan/files/DedAn.zip

[5] W. Reisig, *Petri Nets - An Introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985. doi: 10.1007/978-3-642-69968-9

[6] D. C. Craig and W. M. Zuberek, “Two-stage siphon-based deadlock detection in Petri nets,” in *Current Advances in Computing, Engineering and Information Technology*, P. Petratos and P. Dandapami, Eds. Palermo, Italy: Int. Society for Advanced Research, 2008, pp. 317–330.

[7] F. Chu and X.-L. Xie, “Deadlock analysis of Petri nets using siphons and mathematical programming,” *IEEE Trans. Robot. Autom.*, vol. 13, no. 6, pp. 793–804, 1997. doi: 10.1109/70.650158

[8] M. Uzam, “An Optimal Deadlock Prevention Policy for Flexible Manufacturing Systems Using Petri Net Models with Resources and the Theory of Regions,” *Int. J. Adv. Manuf. Technol.*, vol. 19, no. 3, pp. 192–208, Feb. 2002. doi: 10.1007/s001700200014

[9] J. Ezpeleta, J. M. Colom, and J. Martinez, “A Petri net based deadlock prevention policy for flexible manufacturing systems,” *IEEE Trans. Robot. Autom.*, vol. 11, no. 2, pp. 173–184, Apr. 1995. doi: 10.1109/70.370500

[10] W. B. Daszczuk and W. M. Zuberek, “Deadlock Detection in Distributed Systems Using the IMDS Formalism and Petri Nets,” in *12th International Conference on Dependability and Complex Systems, DepCoS-RELCOMEX 2017*, Brunów, Poland, 2-6 July 2017. AISC vol 582, W. Zamojski et al., Eds, Cham, Switzerland: Springer International Publishing, 2018, pp. 118–130. doi: 10.1007/978-3-319-59415-6_12

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

[12] N. Kaveh, “Using Model Checking to Detect Deadlocks in Distributed Object Systems,” in *2nd International Workshop on Distributed Objects*, Davis, CA, 2-3 November 2000, LNCS vol.1999, 2001, pp. 116–128. doi: 10.1007/3-540-45254-0_11

[13] J. Cho, J. Yoo, and S. Cha, “NuEditor – A Tool Suite for Specification and Verification of NuSCR,” in *Lecture Notes in Computer Science vol. 3647*, Berlin Heidelberg: Springer, 2006, pp. 19–28. doi: 10.1007/11668855_2

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

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

[16] P. C. Attie, “Synthesis of large dynamic concurrent programs from dynamic specifications,” *Form. Methods Syst. Des.*, vol. 47, no. 131, pp. 1–54, Jun. 2016. doi: 10.1007/s10703-016-0252-9

[17] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf, “Analysis on demand: Instantaneous soundness checking of industrial business process models,” *Data Knowl. Eng.*, vol. 70, no. 5, pp. 448–466, May 2011. doi: 10.1016/j.datak.2011.01.004

[18] S. J. C. Joosten, F. V. Julien, and J. Schmaltz, “WickedXmas: Designing and Verifying on-chip Communication Fabrics,” in *3rd International Workshop on Design and Implementation of Formal Tools and Systems, DIFTS'14*, Lausanne, Switzerland, October 20, 2014, 2014, pp. 1–8. https://pure.tue.nl/ws/files/3916267/889737443709527.pdf

[19] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz, “Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada,” *ACM Trans. Softw. Eng. Methodol.*, vol. 3, no. 4, pp. 340–380, Oct. 1994. doi: 10.1145/201024.201038

[20] X. Guan, Y. Li, J. Xu, C. Wang, and S. Wang, “A Literature Review of Deadlock Prevention Policy Based on Petri Nets for Automated Manufacturing Systems,” *Int. J. Digit. Content Technol. its Appl.*, vol.

6, no. 21, pp. 426–433, Nov. 2012. doi: 10.4156/jdcta.vol6.issue21.48

[21] M. A. Reniers and T. A. C. Willemse, "Folk Theorems on the Correspondence between State-Based and Event-Based Systems," in *37th Conference on Current Trends in Theory and Practice of Computer Scienc*e, Nový Smokovec, Slovakia, January 22-28, 2011, 2011, pp. 494–505. doi: 10.1007/978-3-642-18381-2_41

[22] W. Penczek, M. Szreter, R. Gerth, and R. Kuiper, "Improving Partial Order Reductions for Universal Branching Time Properties," *Fundam. Informaticae*, vol. 43, no. 1–4, pp. 245–267, 2000. doi: 10.3233/FI-2000-43123413

[23] W. Jia and W. Zhou, *Distributed Network Systems. From Concepts to Implementations*. New York: Springer, 2005. doi: 10.1007/b102545

[24] B. Czejdo, S. Bhattacharya, M. Baszun, and W. B. Daszczuk, "Improving Resilience of Autonomous Moving Platforms by real-time analysis of their Cooperation," *Autobusy-TEST*, vol. 17, no. 6, pp. 1294–1301, 2016. arXiv: 1705.04263

[25] W. B. Daszczuk, "Asynchronous Specification of Production Cell Benchmark in Integrated Model of Distributed Systems," in *23rd International Symposium on Methodologies for Intelligent Systems, ISMIS 2017*, Warsaw, Poland, 26-29 June 2017, Studies in Big Data, vol. 40, Bembenik, R. et al., Eds, Cham, Switzerland: Springer International Publishing, 2019. pp. 115-129. doi: 10.1007/978-3-319-77604-0_9

[26] M. Schwarick, M. Heiner, and C. Rohr, "MARCIE - Model Checking and Reachability Analysis Done EffiCIEntly," in *2011 Eighth International Conference on Quantitative Evaluation of SysTems*, Aachen, Germany, 5-8 Sept. 2011, 2011, pp. 91–100. doi: 10.1109/QEST.2011.19

[27] Charlie, http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie

[28] M. Heiner, M. Schwarick, and J.-T. Wegener, "Charlie – An Extensible Petri Net Analysis Tool," in *36th International Conference, PETRI NETS 2015*, Brussels, Belgium, 21-26 June 2015, 2015, pp. 200–211. doi: 10.1007/978-3-319-19488-2_10

[29] Z. Li and M. Zhou, "Elementary Siphons of Petri Nets and Their Application to Deadlock Prevention in Flexible Manufacturing Systems," *IEEE Trans. Syst. Man, Cybern. - Part A Syst. Humans*, vol. 34, no. 1, pp. 38–51, Jan. 2004. doi: 10.1109/TSMCA.2003.820576

[30] M. H. Abdul-Hussin, "Elementary Siphons of Petri Nets and Deadlock Control in FMS", *J. of Comput. Commun*., vol.3, No.7, pp. 1–12, Jul 2015. doi: 10.4236/jcc.2015.37001

[31] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Developing UPPAAL over 15 years," *Softw. Pract. Exp.*, vol. 41, no. 2, pp. 133–142, Feb. 2011. doi: 10.1002/spe.1006

[32] T. Latvala, A. Biere, K. Heljanko, and T. Junttila, "Simple Bounded LTL Model Checking," in *International Conference on Formal Methods in Computer-Aided Design*, Austin, TX, 15-17 Nov 2004, LNCS 3312, 2004, pp. 186–200. doi: 10.1007/978-3-540-30494-4_14

[33] W. M. P. van der Aalst, "Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques," in *Business Process Management, LNCS vol.1806*, W. van der Aalst, J. Desel, and A. Oberweis, Eds. Berlin Heidelberg: Springer, 2000, pp. 161–183. doi: 10.1007/3-540-45594-9_11

[34] M. Yamauchi and T. Watanabe, "Time Complexity Analysis of the Minimal Siphon Extraction Problem of Petri Nets," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E82–A, no. 11, pp. 2558–2565, 1999.

[35] F. Tricas and J. Ezpeleta, "Computing minimal siphons in Petri net models of resource allocation systems: a parallel solution," *IEEE Trans. Syst. Man, Cybern. - Part A Syst. Humans*, vol. 36, no. 3, pp. 532–539, May 2006. doi: 10.1109/TSMCA.2005.855751

[36] P. Schnoebelen, "The complexity of temporal logic model checking," in *4th Conference Advances in Modal Logic (AiML'2002), Toulouse, France, 30 Sept - 2 Oct 2004, Advances in Modal Logic vol. 4*, 2003, pp. 437–459. http://www.aiml.net/volumes/volume4/Schnoebelen.ps

[37] R. Klimek and P. Szwed, "Verification of ArchiMate proc*ess specifications based on deductive temporal reasoning," in FEDCSIS 2013 - Federated Conference on Computer Science and Information Systems*, Kraków, Poland, 8-11 Sept 2013, 2013, pp. 1109–1116. https://ieeexplore.ieee.org/document/6644153/

[38] P. Szwed, "Efficiency of formal verification of ArchiMate business processes with NuSMV model checker," in *FEDCSIS 2015 - Federated Conference on Computer Science and Information Systems*, Łódz, Poland, 13-16 Sept 2015, 2015, pp. 1427–1436. doi: 10.15439/2015F44

[39] A. T. E. Dib and Z. Sahnoun, "Model Checking of Multi Agent System Architectures Using BigMC," in *FEDCSIS 2015 - Federated Conference on Computer Science and Information Systems*, Łódź, Poland, 13-16 Sept 2015, 2015, pp. 1717–1722. doi: 10.15439/2015F300

[40] F. Cicirelli, A. Furfaro, L. Nigro, and F. Pupo, "Modelling Java Concurrency: An Approach and a UPPAAL Library," in *FEDCSIS 2013 - Federated Conference on Computer Science and Information Systems*, Kraków, Poland, 8-11 Sept 2013, 2013, pp. 1373–1380, https://ieeexplore.ieee.org/document/6644196