International Journal of Computational Intelligence in Control
Vol. 12 No. 2 (July-December, 2020)

Graph-based Selection of Orchestrator Paths in Manufacturing Lines

Corina Popescu and Jose L. Martinez Lastra

Tampere University of Technology, Korkeakoulunkatu 6, Tampere 33101

Abstract: *Service encapsulation of processes in manufacturing should facilitate a natural and rapid response to machine failure or replacement and to changes in required product types and quantities. Such situations are traditionally addressed by offline modifications in the schedule of the entire line; the search space of the optimal solution varies with each change. Alternative answers are needed for the cases characterized by unknown input job mix and online equipment changes. This paper proposes that the device selection be based on a search in the model of the physical system. Transportation services can be selected based on a search in the state space of the physical system. This way the search space remains independent from the process needs of the pallets entering the line. Information about the cycles of the physical layout graph and their average cycle time, together with real time data of the line will assist the decision taking process. A method to compute the node sequences of the cycles in a graph is presented to serve as a tool for quantification of possible pallet routes in the line.*

Keywords: *Service oriented architecture, Factory automation, Scheduling, Cycles in a graph, Petri Nets*

1. INTRODUCTION

The bridging of the Service Oriented Architecture (SOA) paradigm and the factory automation world is envisioned to address frequent changing market demands and time to market pressure [3]. The loose coupling provided by services ensures changes in one part of the system do not affect other parts of the system. Dynamic discovery of new services not known beforehand is attainable with SOA. Moreover, ontologies provide computer interpretable descriptions of services that make it possible to achieve automatic composition.

From a SOA perspective, a manufacturing line is seen as a set of service encapsulations of provided and requested processes. The provided processes are the equipment skills. The requested processes are the product needs. Each product can be described in terms of its orchestrator. The orchestrator specifies the order of execution (the flow) of its needs – the services that should operate upon the raw product to get it to a finished status. When entering the line, a pallet discovers the devices that offer the services requested by its orchestrator. Selections of each device to execute upon it are made gradually, as the orchestrator executes. Each time a device is selected for execution, the transportation services needed to carry the pallet to its chosen destination are subjected to discovery and selection as well.

The production process in a line should be highly adaptable to changes. Machine failures or replacements and changes in lot sizes should be recognized and responded to naturally. These goals can be reached if factory automation is seen from a SOA perspective. However, the loose coupling provided by services will not fully support fast reconfigur ability and adaptability unless satellite issues such as scheduling and planning are re-considered from this viewpoint.

Traditionally, the scheduling problem is formulated as the finding of an optimal input sequence of jobs and resource usage for a given job mix [1] [2]. To schedule a system it is absolutely necessary to have beforehand knowledge of all due product types and device capabilities. Each time a change occurs, a new schedule has to be derived offline. A wide variety of methods can be used for this purpose [1]. Among other methods, Petri Net (PN) based scheduling has been successfully used for manufacturing systems, because the formalism can finely describe shared resources, synchronization and lot sizes. A PN based schedule is heuristically searched in the state space of the complete model of the system. This type of scheduling is deadlock free and event driven. However, with this approach it can happen that the search space becomes too large for complex systems. The scheduling speed strongly depends on the selected heuristic search function. The optimality of the obtained schedules is also influenced by this choice, and cannot be always guaranteed.

Traditional scheduling approaches rely on the assumption of offline equipment changes and known input job mix. The limitations of the traditional vision are clear: First, it is not possible to build a new schedule, on-the-fly, in case of machine breakdown. The same applies in case a machine is reconfigured to provide an enriched or slightly different set of operations, or when the line is added equipment. Second, variations in the input job sequence are unavoidable, mainly due to human error. The implemented

schedule cannot deal with this type of situations. The input sequence of the jobs must be the assumed input sequence when initially building the schedule. Indirectly, this limitation is related to sizable multiple lot size scheduling problems, which are especially difficult to solve optimally in a reasonable amount of time and space.

The encapsulation of processes within services achieves the objective of having a running production line with online device modifications and unknown sequencing of requested product types. This evolution from a traditional system is desirable, yet it imposes significant constraints on the assumptions that scheduling techniques rely on. To meet these constraints, new methods for finding optimal routes for each pallet entering the system must be evaluated.

PN scheduling techniques work on the state space of the entire system. The system is modelled by merging the PN sub-models of each (possibly) requested job (sequence of operations). All possible allocations of resources for each operation of a job and for material handling have to be incorporated in its model. Multiple lot sizes are represented by the amount of tokens held within the start places of each sub-model. Flexible routes can be conveniently expressed through choice structures, at the expense of an increased search space. This approach imposes (possibly large) modifications in the search space each time any type of change occurs.

The notion of 'job' differs from the earlier defined concept of 'orchestrator'. A job is assigned all possible devices that may perform its composing operations before being input to the line. An orchestrator discovers eligible devices while it executes. An orchestrator does not impose temporal constraints on the activities within a line, and can re-adjust in case of machine failure or online replacement. If one machine is no longer able to offer a certain service, an orchestrator will search and discover other devices compatible with its needs. Unless this type of situation is explicitly incorporated in the initial schedule, a job will not adapt to such a situation.

In the new, service oriented context, resource allocation (device selection) is a problem of finding the best of all possible paths within the model of the physical layout of the system, for each orchestrator. The search for transportation devices should be performed on the state space of the physical system. This way the search space remains the same no matter what are the orchestrators within the line, unless additions to the line occur. The search should take into consideration existing possibilities for the other orchestrators in the line, in addition to the makespan of each pallet.

This paper analyzes the mind shift required to work with service encapsulations of manufacturing processes. The usage of formal representations of each orchestrator and the physical layout of the system to optimally allocate resources and select transportation devices is investigated.

The paper is structured as follows: Section 2 describes the modelling assumptions of this discussion and the used formalism. Section 3 highlights the relations between the state space of each orchestrator and the physical layout of the system. Section 4 explores possible solutions to the problem of finding the best routes of each orchestrator in the line - a new algorithm to compute the cycles in a graph is defined in its general form. Section 5 addresses related work, and Section 6 presents the conclusions.

2. MODELING ASSUMPTIONS

This work uses for modelling a Petri Net (PN) [4] derived formalism called Timed Net Condition Event Systems (TNCES) [5][6]. TNCES enhances the expression capabilities of PNs with typed modularity, and adds to the originally defined elements of a PN the notions of event arcs and condition arcs. Event arcs report changes in the state of the system, while condition arcs carry state information. TNCES can model simultaneous start, has a clear notion of interfaces and a modular hierarchy. An example of a simple TNCES module is depicted in Figure 1.

TNCES may be defined by the following tuple:

$$\text{TNCES} = \{\text{name, type, } P, T, F, m_0, \psi, CN, EN, DC\} \quad (1)$$

where:

- name—a unique string id of a module used to differentiate the modules at the same hierarchical level of the composite TNCES module.
- type—a unique string id of a module to allow reusing the module in hierarchical models.
- $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places;
- $T = \{t_1, t_2, \ldots, t_m\}$ is a set of transitions disjoint with P;
- $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of flow arcs between places and transitions;
- m_0 is an initial marking;
- ψ is input/output structure of TNCES module;
- $CN \subseteq (P \times T)$ is a finite set of condition arcs;
- $EN \subseteq (T \times T)$ is a finite set of event arc.

The input/output structure of TNCES module is represented by the following tuple [6]:

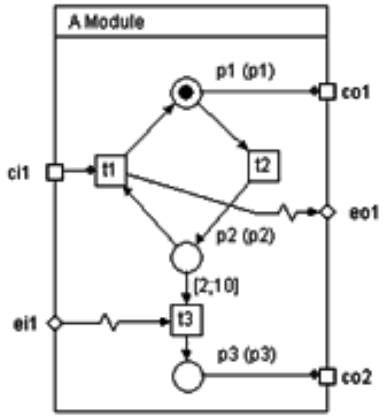


Figure 1: TNCES Module example

$$\psi = \{C^{in}, E^{in}, C^{out}, E^{out}, Bc, Be, Cs, Dt\} \quad (2)$$

where:

- C^{in} is a finite set of TNCES module condition input signals;
- E^{in} is a finite set of TCNES module event input signals;
- C^{out} is a finite set of TNCES module condition output signals;
- E^{out} is a finite set of TCNES module event output signals;
- $Bc \subseteq C^{in} \times T$ is a set of TNCES module input condition arcs;
- $Be \subseteq E^{in} \times T$ is a set of TNCES module input event arcs;
- $Cs \subseteq P \times C^{out}$ is TNCES module output condition arcs;
- $Dt \subseteq T \times E^{out}$ is a set of TNCES module output event arcs.

Time intervals may be assigned to the pre-transition flow arcs ($F^{-} \subseteq P \times T$), which imposes time constrains to the firing of the transition [6]:

$$DC = \{DR, DL, D_0\} \quad (3)$$

defines a set of delay times:

- DR representing the minimum times that the token should spent at particular place before the transition can fire;
- DL is the set of limitation time that defines the maximum time that the place may hold a token (if all the other conditions for transition firing are met);
- D_0 is the initial set of the clocks associated with the places.

The main elements of the tuple of the module in Figure 1 are: name = type = "A Module"; $P = \{p1, p2\}, p3\}$; $T = \{t1, t2, t3\}$; $F = \{(p1, t2), (t2, p2), (p2, t1), (t1, p1), (p2, t3), (t3, p3)\}$; $C^{in} = \{ci1\}$; $E^{in} = \{ei1\}$; $C^{out} = \{co1, co2\}$; $E^{out} = \{eo1\}$; $Bc = \{(ci1, t1)\}$; $Be = \{(ei1, t3)\}$; $Cs = \{(p1, co1), (p3, co2)\}$; $Dt = \{(t1, eo1)\}$; $D_0(P) = \{0, 0, 0\}$; $DR(P) = \{0, 2, –\}$; $DL(P) = \{\infty, 10, –\}$. The formalism is hierarchical, modular and composable. Its typed nature facilitates the tracking of the blocks associated with each module in the markings of the state space. Condition and event arcs enrich Petri Nets with possibilities to carry information about states and changes of states. These extensions can be fully expressed mathematically, so verification techniques are not traded for higher modelling power.

The physical layout of the system is modelled following the general guidelines for modelling flexible manufacturing systems [1][2]. The modelling of service orchestration is

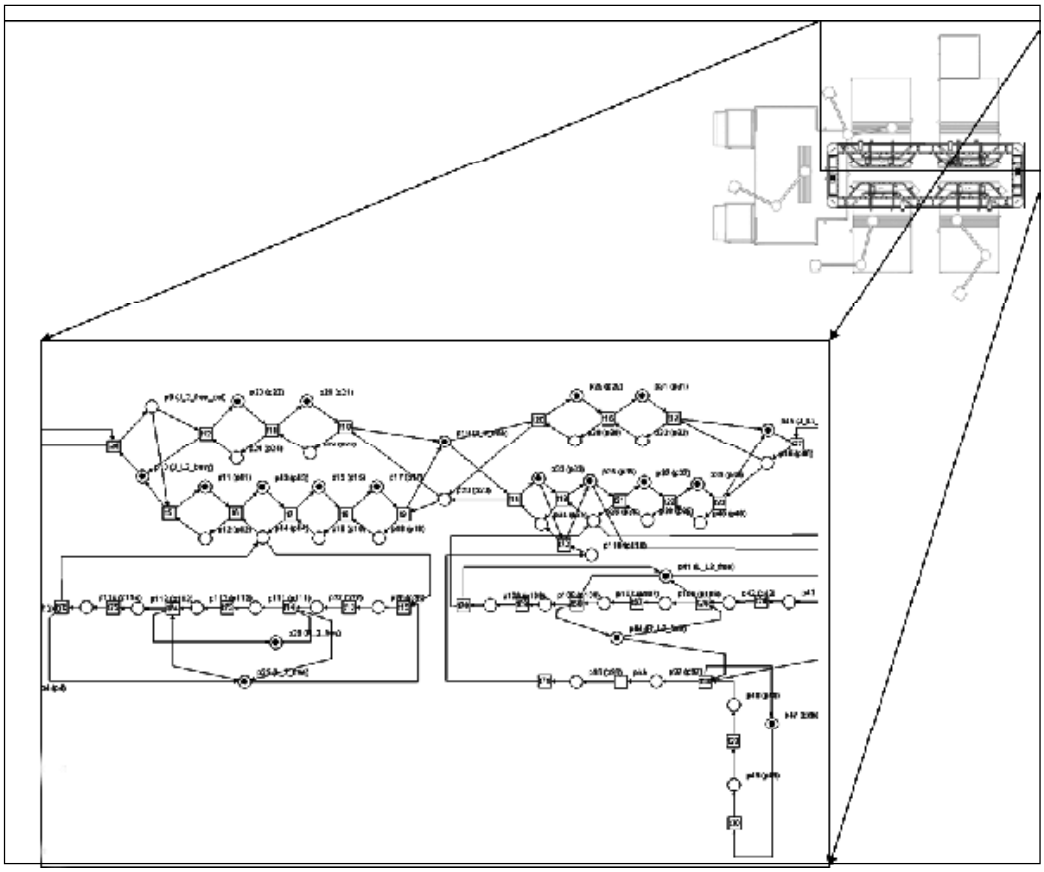


Figure 2: Fragment of the Petri Net model (bottom-left) of the Physical layout of a Line (Top-right)

approached in a modular manner: a set of TNCES models has been defined to cover eight flow descriptors capable of expressing multithreading, synchronization, looping and sequencing. Separate modules to address the formal models of the Boolean conditions that come in conjunction with looping constructs have been created as a satellite set [13]. The set of basic flow descriptors was taken from the list of control constructs specified by the OWL-S W3C Note [7]. The end atomic services are treated as black boxes characterized by a time interval, to specify the upper and lower boundaries of the admissible execution of the process. The formal model of each orchestrator can be expressed as interconnected TNCES modules.

Figure 2 illustrates a fragment of the PN model of the physical layout of a system consisting of five robotic cells and a conveyor system. Pallets can either occupy a workstation of a cell or bypass it through an auxiliary conveyor. If the workstation is occupied, lifters situated beneath its main conveyor assist the robot in reaching the pallet. The shown fragment is the formal representation of the layout of two of the cells within the line. The right-most cell can be input pallets by the adjacent cells or through an extra conveyor. Examples of possible orchestrators that are input to this line are shown in Figs. 3 & 7.

The models of the orchestrators and the equipment offer an explicit state view of the services to be requested/to finish in future. The markings of each orchestrator map to corresponding groups of state possibilities in the reachability graph of the equipment model. An update of all markings as the pallets go through the line should make inferences on potential evolutions of line activity possible. These inferences are valuable because the actual mapping to the physical devices is done gradually, as each orchestrator executes. Decisions have to be taken in case several orchestrators compete for the same device, or whenever the same service is offered by two different devices. Knowledge of the set of path possibilities within the state space of the equipment, for all orchestrators, can assist the optimal selection of devices.

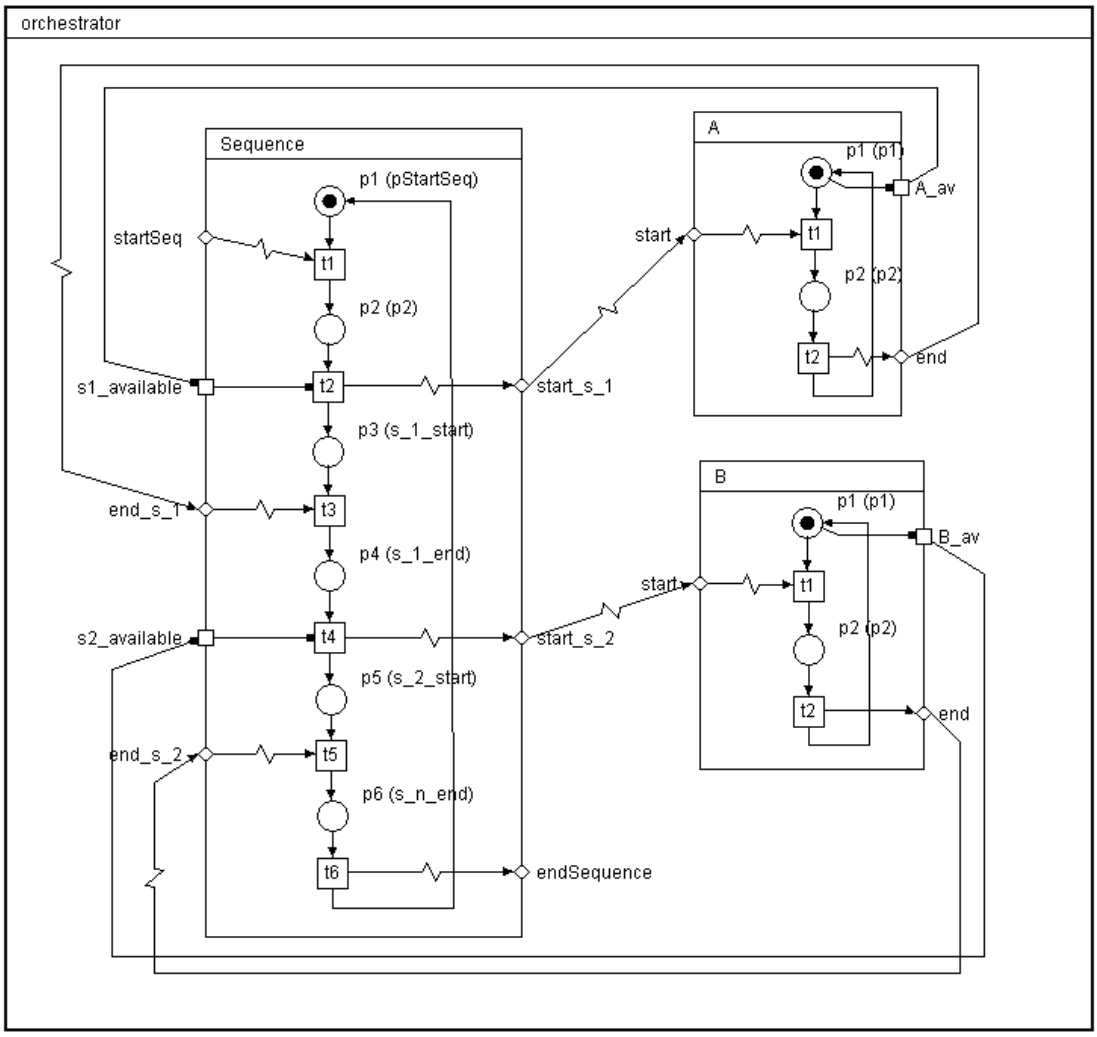


Figure 3: TNCES Orchestrator Model: Sequence of Two Atomic Services

3. RELATIONS BETWEEN THE STATE SPACES OF ORCHESTRATOR MODELS AND THE PATHS OF THE PHYSICAL SYSTEM

Based on the nature of the basic flow representations used at modelling stage, it is possible to identify relations between pairs of states in the resulting state space of each orchestrator. A group of such relations for one orchestrator is denoted here by the term 'stamp'. Timing can be associated with each stamp if the knowledge about the equipment capable to provide the needed services is taken into consideration. Each path/cycle in the reachability graph (RG) of an orchestrator corresponds to several path/cycle possibilities in the model of the physical layout of the system, depending on the related number of possible device-process mappings. A stamp can assist in detecting the changing, over time, of the degree of desirability of each ongoing path. Thus it can support decision making about the evolution of the system.

This section analyzes, first, the stamps of TNCES models of three basic flow descriptors. Further on, a system resulting from interconnecting elements of the basic set is analyzed to generalize the initial statements.

3.1. State Spaces of Basic Modules

An example of a **Sequence** TNCES module engaging two participant services A and B is illustrated in Figure 3. Transition t_1 of the **Sequence** module is enabled and may fire only upon receiving the event *startSeq*. There are two atomic services A and B, whose internal functionality is not included in the model. The only information concerning the atomic services that is available to the other TNCES modules is related to their busy/idle status and possibly their time intervals. The TNCES representation of the **Sequence** specifies the entire orchestration of this simple system.

Figure 4 depicts the reachability graph of the model shown in Figure 3. Rows numbered 1 to 6 host the marking vectors corresponding to each state of the logical RG. The row labeled '*P.nr.*' keeps a record of the flat numbers of the places within the overall model. There are 12 places in the flat model. The first 6 belong to the **Sequence** TNCES module. Places 7 and 8 (respectively 9 and 10) correspond to the functionality within the atomic TNCES representation of service A (respectively B). Finally, places 11-12 are part of an '*init*' TNCES module that is part of the overall model but is not shown as it is technically irrelevant for the present discussion (its purpose is to ensure the start of the execution of the **Sequence** module when building the reachability graph). The first element of the marking vector is the first element of the **Sequence** block: a token in p_1 corresponds to the initiation of activity in the **Sequence** module. The sixth element of the Sequence block corresponds to the last place of the **Sequence** module, so $m(p_6) = 1$ marks the end of activity within the module. The relation imposed by the **Sequence** on the state space of the final orchestrator is $1\Re 6$. Time is imposed on this relation by the equipment capable to provide services A and B. If there is more than one piece of equipment capable to provide the same service, the annotation with time may be governed by the slowest device.

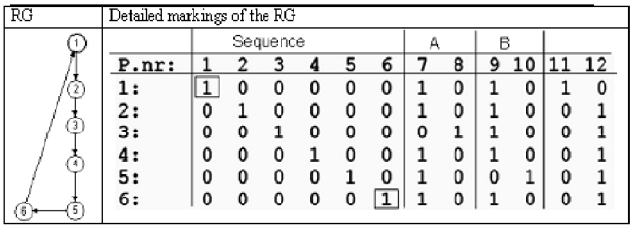


Figure 4: RG of the Model of Figure 3

An optimal scenario from the viewpoint of the orchestrator is a situation in which the pallet does not wait for a long time to have the devices operate on it. This corresponds to a full execution of the **Sequence** in the amount of time that is necessary for all needed services to complete, in case the slowest eligible device is chosen in each case. This corresponds, in the above described situation (the reasoning is similar if a number of participants that is greater than 2 is considered), to the temporal distance between state 6 and state 1

$$T_{SEQ} \in [6l - 1h;\ 6h - 1l] \qquad (2)$$

where l and h denote the lower and respectively higher time instants at which the corresponding state may be reached in an optimal situation from the orchestrator's perspective. Once state 1 is visited, $6l - 1h$; $(6h - 1l)$ is the lower (respectively higher) boundary of the needed optimal time range for the orchestrator to reach state 6.

Transportation services depend on the position of the pallet in the physical system and of the chosen device-process mappings, and are not considered in this reasoning. Temporally, the **Sequence** typed module is equivalent to the **AnyOrder** typed module. Consequently, this analysis can be extended to **AnyOrder** similarly.

The detailed RG of a model of an orchestrator involving a **Split + Join** construct in conjunction with two participating atomic services A and C is depicted in Figure 5. The first five places correspond to the block describing the activity in the **Split + Join** module. An example of a **Split + Join** model with two participants can be seen in Figure 7, as part of a more complex model. The beginning of the activity in this module is marked by the presence of tokens in places p_2 and p_3 (state 2). Termination of activity in the module is announced by the presence of tokens in p_4 and p_5.The relation

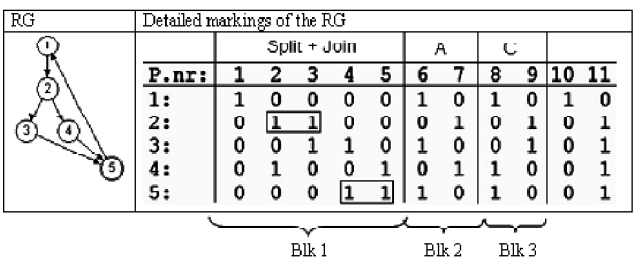


Figure 5: RG of a TNCES Orchestrator Model: SJ of Two Atomic Services A and C

imposed by the **Split + Join** on the state space of the final orchestrator is $2\Re 5$. Time is imposed on this relation by the equipment capable to provide services A and C in the line.

Based on knowledge about the equipment, a temporal distance may be defined quantitatively between states 5 and 2:

$$T_{SJ} \in [5l - 2h;\ 5h - 2l] \tag{4}$$

where l and h have the same meanings as above.

Figure 6 illustrates the RG of a model of an orchestrator involving a **Choice** between two participating atomic services A and B. Places 7 to 10 in each marking vector correspond to the activity of the **Choice** module in the overall system model. The relation imposed by Choice on the state space of the orchestrator is $7\Re 8$. The temporal distance between states 2 and 1 (in this order on the time line) is given by:

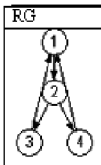
$$T_{CH} \in [1l - 2h;\ 1h - 2l] \tag{5}$$

Quantitatively this distance is given as:

$$[min(T_{min}(A),\ T_{min}(B)),\ max(T_{max}(A),\ T_{max}(B))] \tag{6}$$

Each of the constructs occupies a clearly delimited segment of the marking vector. For instance, the activity of the **Split + Join** module (Figure 5) is specified by the first 5 places of the vector. The activity of the **Choice** module (Figure 6) is described by the last 4 places in the vector. The identification of the places that mark start and end of activity in each of the typed TNCES module within the basis can be performed automatically. This is achievable if the (flat) numbers of the places are known. For instance, each **Sequence** of N services used in the overall model of the system requires $2N + 2$ elements in its corresponding block in the vector of markings. The first element in the block is the start of that particular **Sequence**. Element $2N + 2$ in the block corresponds to the end of the activity of the module. For each **Split + Join** of N services, a block of $1 + 2N$ elements is necessary in the marking vector. In this case, tokens in elements 2 to $N + 1$ of the block mark the start of activity of the construct. The end of the **Split + Join** activity is identified by presence of tokens in elements $N + 2$ to $2N + 1$ of the block. This result is obtained in case the numbering is performed first for the places $s\_\ j\_$ start, and then for the places $s\_\ j\_$ end. This reasoning can be easily extended to all TNCES models within the basis.

RG



Detailed markings of the RG

	A		B				Choice			
P.nr:	1	2	3	4	5	6	7	8	9	10
1:	1	0	1	0	1	0	**1**	0	0	0
2:	1	0	1	0	0	1	0	**1**	0	0
3:	1	0	0	1	0	1	0	0	0	1
4:	0	1	1	0	0	1	0	0	1	0

Figure 6: RG of a TNCES Orchestrator Model: Choice of Two Services A and B

The stamp of an orchestrator may be defined by inspecting the typed blocks of its marking vectors that correspond to the basic modules forming it.

3.2. State Spaces of Orchestrator Models Obtained as Interconnections of Basic Modules

Consider a production system involving three atomic services A, B and C, which are to be completed in order to obtain a final product (Figure 7). The order in which the three tasks must take place involves first initiation of both tasks A and B. After task A is completed task C must be initiated. The final product is obtained when both B and C are finished.

Figure 8 illustrates the logical flow between the markings of the reachability graph and the detailed marking vectors describing each state within the RG. Places 1 to 5 define the segment in the marking vector that characterizes the activity within the **Split + Join** construct. To infer the states that should be in a temporal relation in the optimal case, the **Split + Join** block should be analyzed to define the groups of states (the regions) characterizing the start and end of activity within the block. A possible initiation of activity is thus indicated in five separate cases (region R_1^{SJ}, corresponding to states 2, 3, 5, 7 and 9). Only one state (region R_2^{SJ}, i.e. state 13) is a possible termination of activity within the construct. The relation imposed by **Split + Join** on the state space of the orchestrator is $R_1^{SJ}\ \Re\ R_2^{SJ}$. In an optimal scenario, the temporal distance between the two found regions is a function of the temporal constraints associated with all possible physical participants to the flow module.

$$T_{SJ} \in [R_2^{SJ}l - R_1^{SJ}h;\ R_2^{SJ}h - R_1^{SJ}l] \tag{7}$$

This relation may be refined to finer levels if each region is investigated separately to look for its possible source/ sink states. In this case, the source state of the region concerned with the start of activity - R_1^{SJ} - is state 2. R_2^{SJ} does not need to be refined to its sink state as it contains only one state. The relation imposed by **Split + Join** on the state space of the orchestrator becomes $13\Re 2$, and its temporal distance:

$$T_{SJ} \in [13l - 2h;\ 13h - 2l] \tag{8}$$

The logical regions marking the start and end of the activity in the **Sequence** module are $R_1^{SEQ} = \{1, 11, 13\}$ and $R_2^{SEQ} = \{9, 12\}$. The temporal distance between the regions is defined similarly:

$$T_{SEQ} \in [R_2^{SEQ}l - R_1^{SEQ}h;\ R_2^{SEQ}h - R_1^{SEQ}l] \tag{9}$$

and may be further refined to states 11 (the source state of R_1^{SEQ}) and 12 (the sink state of R_2^{SEQ}). The relation becomes:

$$T_{SEQ} \in [12l - 11h;\ 12h - 11l] \tag{10}$$

The temporal relations defined in Eq. 8 (between states 2 and 13) and Eq.10 (between states 11 and 12) are inferred through inspecting the basic modules used to build the final model of the orchestrator. The temporal stamp associated with the model of Figure 7 is represented by the group of relations $12\Re 11$ and $13\Re 2$.

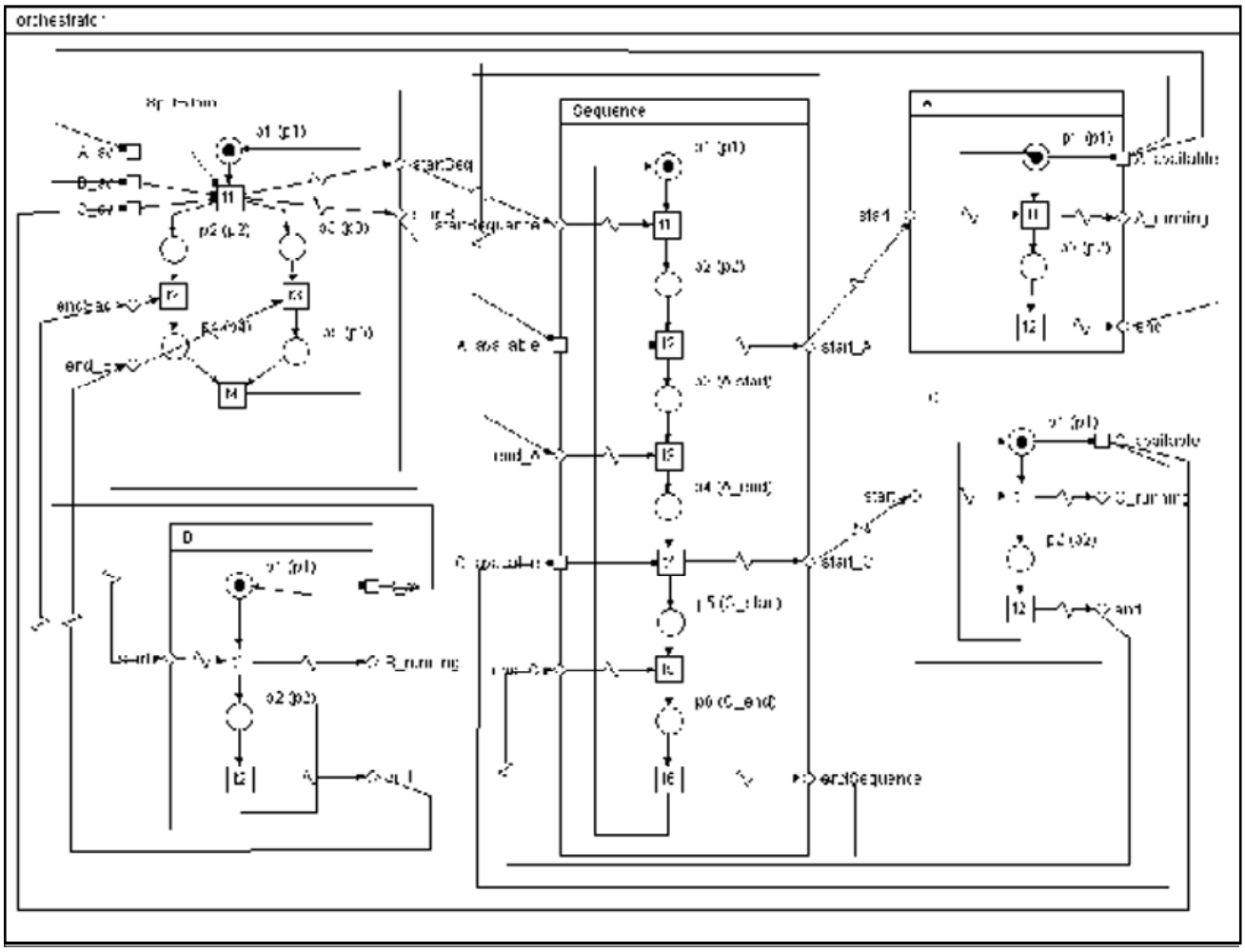
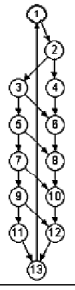


Figure 7: TNCES Orchestrator Model: Interconnections of Split + Join and Sequence Constructs

Once an orchestrator is introduced in the system it should work towards achieving/minimizing its temporal stamp. Increases of the inferred temporal distances are either correlated with a malfunction of one of the atomic services, or with the waiting for one of the services to become available for execution. The first case is not of concern here, as it is assumed that all atomic services incorporate an inner fault handling mechanism, characterized by time boundaries as well. The second case is analogous to the situation in which a controlled value remains within control limits, but the overall process is out of control: each atomic process is executing correctly once started, and yet the overall orchestrator does not follow its optimal temporal constraints accurately.

RG | Detailed markings of the RG



	Split + Join					Sequence						A		C		B	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	**1**	0	0	0	0	0	1	0	1	0	1	0
2	0	**1**	**1**	0	0	0	1	0	0	0	0	1	0	1	0	0	1
3	0	**1**	**1**	0	0	0	0	1	0	0	0	0	1	1	0	0	1
4	0	**1**	**0**	0	1	0	1	0	0	0	0	1	0	1	0	1	0
5	0	**1**	**1**	0	0	0	0	0	1	0	0	1	0	1	0	0	1
6	0	**1**	**0**	0	1	0	0	1	0	0	0	0	1	1	0	1	0
7	0	**1**	**1**	0	0	0	0	0	0	1	0	1	0	0	1	0	1
8	0	**1**	**0**	0	1	0	0	0	1	0	0	1	0	1	0	1	0
9	0	**1**	**1**	0	0	0	0	0	0	0	**1**	1	0	1	0	0	1
10	0	1	0	0	1	**0**	0	0	0	1	0	1	0	0	1	1	0
11	0	0	1	1	0	**1**	0	0	0	0	0	1	0	1	0	0	1
12	0	1	0	0	1	**0**	0	0	0	0	**1**	1	0	1	0	1	0
13	0	0	0	**1**	**1**	**1**	0	0	0	0	0	1	0	1	0	1	0

Figure 8: Markings for the Model of Figure 7

Once an orchestrator is introduced in the system it should work towards achieving/minimizing its temporal stamp. Increases of the inferred temporal distances are either correlated with a malfunction of one of the atomic services, or with the waiting for one of the services to become available for execution. The first case is not of concern here, as it is assumed that all atomic services incorporate an inner fault handling mechanism, characterized by time boundaries as well. The second case is analogous to the situation in which a controlled value remains within control limits, but the overall process is out of control: each atomic process is executing correctly once started, and yet the overall orchestrator does not follow its optimal temporal constraints accurately.

The stamp of each orchestrator may be reflected temporally once or several times in the state space of the physical layout of the system.

This depends on the amount of devices that are capable to provide the needed services. The inferred temporal relations can be used together with real time observations of the running system to detect the situations in which the system loops in a state cycle whose degree of desirability is decreasing with time.

A trivial example to illustrate this point is the situation of Figure 7: a certain temporal distance characterizes the orchestrator state cycles containing states 2 and 13 (and respectively 12 and 11). This temporal distance is dictated by the equipment that is able to perform the services requested by the orchestrator. The cycles in question can be mapped to cycle possibilities in the state space of the physical layout. This evaluation can be performed for each orchestrator, and intersections between the possible groups

of paths inferred. Future evolutions of the physical layout can be thus evaluated to select the routes that are most likely to satisfy the optimal time needs of each orchestrator. In this way situations that are found to be potentially critical in the future can be resolved at an early stage.

4. QUANTIFICATION OF POSSIBLE ORCHESTRATOR PATHS—AN APPROACH TO COMPUTE THE CYCLES OF A GRAPH

The main steps of the proposed algorithm are depicted in Figure 9. The graph is first subjected to structural reduction. The reduced graph serves as an input to an encoder who performs a search on the graph and outputs a vector encoding an entire set of cycles. Second, the coded vector is input to a decoder. The output of the decoder consists of an array of vector representations of both complete and partially covered (incomplete) cycles. The complete cycles are retained, whereas each partially covered cycle will be input once more to the encoder for further processing. The procedure ends when there are no more incomplete cycles to be processed. Details on each of the steps described above are given below.

4.1. Structural Reduction of the Reachability Graph

To illustrate the structural reduction procedure, a small example of an RG containing 7 states (Figure 10a) shall be considered. The graph is reduced to so-called 'crosspoints' (i.e. states with more than one outgoing state) and the corresponding transitions.The results of cycle computation would not be affected by this type of reduction. Figure 10 illustrates on the right side the reduced graph. A record of all sequences of states emerging from one crosspoint and ending in another is kept.

For instance, crosspoint number 2 in Figure 10b should keep a record of the sequence {2, 5, 2}, although node 5 is removed in the reduced graph. The cycles for node 5 may be easily computed based on the cycles obtained for the start crosspoint (crosspoint 2) of the sequence. In order to retrieve the actual cycle {2, 5, 2} from the crosspoint cycles,

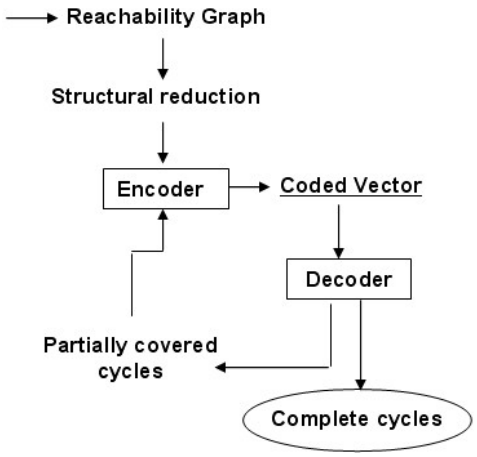


Figure 9: Computation of Cycles in a Graph. General Approach

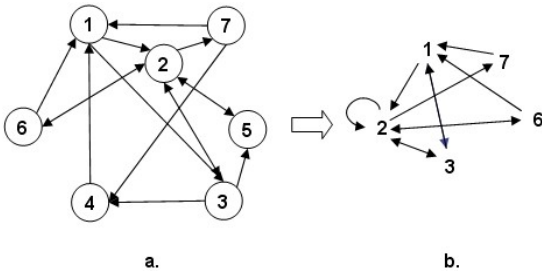


Figure 10: (a) Initial Graph (Example); (b) The Corresponding Structurally Reduced Graph

trimming may be performed for the first and last branches in the cycle. The state space may be safely reduced this way.

Further cleanup is performed in two steps, in order to eliminate all crosspoints that are sure to result in a deadlock. First, the crosspoints with no incoming crosspoints are eliminated (as they are sure to be part of no cycle). The crosspoints found to have all incoming crosspoints among the ones already eliminated are removed from the reduced graph as well, iteratively. Second, the reduced graph is checked with respect to the sets of outgoing crosspoints of each crosspoint. Once a set is found to be fully included in the set of previously eliminated crosspoints, the owner of the set is removed as well from the state space. This cleanup is performed for all remaining crosspoints iteratively each time eliminations are detected to have occurred, until no such detections are signaled. The graph is thus reduced to the set of crosspoints prone to be a part of one or more cycles.

4.2. Encoding Algorithm

In order to find all cycles starting and ending in a particular crosspoint, a Depth First Search is performed. Each element in the coded vector keeps a record of two data structures: the number of successors and the incomplete cycles. The number of successors of an element denotes the number of outgoing crosspoints of the element that appear further in the coded vector. An incomplete cycle is a search direction that was stopped for reasons related to the consistency of the coded vector, but needs further investigation.

The general steps of the encoding algorithm are shown in Table 1. The needed data structures are outlined below:

- a vector **codedVector** that stores the elements output by the encoding procedure. Each element in the coded vector is represented by:
- an integer **codedElementNo**: the number of the corresponding crosspoint in the reduced reachability graph.
- an integer **noSuccessors**: the number of outgoing crosspoints of the crosspoint with the number codedElementNo, that are stored further in the codedVector.
- a Boolean variable **final**: to mark the noSuccessors as non-modifiable in the future

Table 1
Encoding Algorithm–General Steps

Procedure ENCODE (Reduced RG, CrossPoint cp)

1. currentElement.codedElementNo ← cp.no,
   currentelement.incompleteCycles ←
   NULL,currentElement.nosuccesors ← –1;
2. **if** codedVector.length > 1 and cp.no = sourceNo
   then append currentElement to the codedVector **fi**, return;
3. **if** cp.no tempFinal **then** append cp.no to the incompleteCycles of the last element in temp **fi**;
4. **if** cp.no ∈ temp then return; **fi**;
5. **if** cp.no ∉ codedVector **then**
6. append currentElement to the codedVector;
7. increment noSuccessors for the last element in temp;
8. **for** childCp ∈ outgoingCrosspoints **do**
9. temp ← NULL, tempFinal ← NULL;
10. **for** i = codedVector.length **downto** 0 **do**
11. **if** codedVector(i). final **then**
12. append codedVector(i) to tempFinal;
13. **else** append codedVector(i) to temp **fi**; **od**;
14. ENCODE(RG, childCp); // apply the algorithm for the child
15. **if** childCp = the last element in outgoingCrosspoints **then**
16. currentElement.final ← true **fi od**; **fi**.

Besides the codedVecor some auxiliary data structures are used:

- a list **incompleteCycles** containing integers that store the search directions that are yet to be explored.
- a set **tempFinal** of the elements in the coded vector with the number of successors set to final.
- a set **temp** of all elements within the coded vector that do not have the number of successors set to final yet.
- a structure representation **reducedRG** of a reduced reachability graph.
- an integer **sourceNo**: the number of the first element in the coded vector, for which the cycle computations are to be performed.
- a list **outgoingCrosspoints**: the outgoing crosspoints of the crosspoint with the number equal to codedElementNo.

The small example of Figure 10 shall be used henceforth to illustrate the steps of the encoding procedure. Table 2 shows the sets of outgoing crosspoints for each of the 5 crosspoints of the reduced graph of Figure 10b. The steps of the encoding algorithm applied to searching all cycles emerging and ending in crosspoint number 2 are outlined below. Table 3 shows the encoded vector output after the algorithm is applied once.

Depth First Search is started on crosspoint 2. The crosspoint is found to have 4 outgoing crosspoints: {3, 6, 2, 7}. The first outgoing state number (3) is added to the coded vector, and its children are searched further.

Table 2
Sets of Outgoing Crosspoints for the Example Graph of Figure 9

Crosspoint No	*Outgoing Crosspoints*
1	2 3
2	3 6 2 7
3	2 1
6	1 2
7	1

Table 3
Example of a Coded Vector

Coded element	2	3	2	1	2	6	2	2	7
No. successors	4	2	4	1	4	1	4	4	0
Incomplete cycle numbers	–	–	–	–	–	1	–	–	1

Crosspoint 3 is found to have the set of children {2, 1}. 2 is added to the coded vector, and its children are not searched further as its number is equal to the number of the first element in the coded vector (a cycle end was detected). The number of successors of element 3 is incremented and its next child (i.e. 1) is searched for.

Crosspoint 1 is detected to not have been added previously in the vector, so it is added and the number of successors of element 3 is incremented. Crosspoint 1 has two outgoing crosspoints: 2 and 3. 2 is added to the coded vector, and the number of successors of element 1 in the coded vector is incremented. The next child of element 1 is 3. Crosspoint 3 is not added to the coded vector nor searched further, as inspection on the structure of the coded vector results in detecting the fact that an inner 2 – {3 – 1 – 3} loop was reached. Details on detection of loops are given later in this subsection. As there are no more children to inspect for element 1, its number of successors of element 1 is marked to be final.

Similarly, the number of successors of element 3 is marked to be final. The next child of the source crosspoint (i.e. 6) is investigated. Element 6 is added to the coded vector and the set of its outgoing crosspoints ({1, 2}) is investigated. Element 1 is detected to be part of the coded vector already, and verification of the existence of a possible inner loop is performed. 1 is added to the list of incomplete cycles for element 6, and the next child of element 6 is searched for. Since it is a 2, it is added to the coded vector, the number of successors for element 6 is incremented and also set as final (there are no more children to investigate).

The number of successors of the source crosspoint is incremented, and its next children (2 and 7) are taken into consideration for processing. Element 2 is added to the coded vector and the number of successors is incremented. Finally, element 7 is added to the coded vector, and since its only

child (1) has already been added to the coded vector, its set of incomplete cycles is added element 1.

The number of successors of the source crosspoint is incremented and set as final, and the procedure terminates.

Detection of the loops is done based on the structure of the encoded vector, by means of a simple verification on whether the element to be added can be found among the elements already existing in the coded vector, with the number of successors not set to be final yet.

4.3. Decoding Algorithm

Decoding is performed backwards on the coded vector. The general steps of the decoding algorithm are outlined in Table 4. The needed data structures are as follows:

- Input: a vector **codedVector** storing the encoded elements.
- Output: an array of vectors **allCycles**: to store the decoded cycles.
- An integer **sourceNumber**: the number of the first element in the input codedVector.

Table 4
Decoding Algorithm–General Steps

Procedure DECODE (codedVector)

1. allCycles ← NULL, sourceNumber ← the number of the first element in the input codedVector, counter ← 0;
2. **for** i = codedVector.length **downto** 0 **do**
3. decodedElement ← codedVector(i);
4. **if** decodedElement.codedElementNo = sourceNumber OR decodedElement.noSuccessors = 0 **then**
5. append decodedElement.codedElementNo to allCycles [counter];
6. increment counter; **fi**;
7. **if** decodedElement.codedElementNo ≠ sourceNumber and decodedElement.no Successors ≠ 0 **then**
8. counterSuccessors ← 0,
9. noSuccessors ← decodedElement.noSuccessors,
10. cycleIndex ← counter – 1,
11. previousNumber ← – 1.
12. **while** counter Successors < noSuccessors + 1 **do**
13. int elementNo ← the codedElementNo of the last element in allCycles[cycleIndex];
14. **if** previousNumber elementNo OR elementNo = sourceNumber **then**
15. increment counterSuccessors **fi**
16. append decodedElement to allCycles[cycleIndex];
17. decrement cycleIndex; **od**; **fi**;
18. **if** size of decodedElement.incompleteCycles ≠ 0 **then**
19. append decodedElement to allCycles[counter];
20. increment counter; **fi**;
21. **od**.

- An integer **counter** for iterating through allCycles array.
- Local integer variables **counterSuccessors**, **noSuccessors**, **cycleIndex**, **previousNumber**.

The decoder reads the coded vector backwards. The allCycles array, whose role is storage of the vectors representing decoded cycles, is initialized.

A new vector is initialized and added to the allCycles array each time the decoder encounters either an element with the number equal to the number of the source crosspoint for which the encoding was performed or an element with the final number of successors set to 0. The former represents the end of a coded cycle, whereas the latter represents a partially covered path which was coded until an inner loop was found. If the decoder encounters an element with a final number of successors greater than 0 and an element number different from the source crosspoint's number, then this element has to be added to the decoded cycles. The number of successors of the element (noSuccessors) is the decision maker for how many cycles in the allCycles array the element has to be appended to. The allCycles array is read backwards, and a counter is incremented from 0 up to noSuccessors whenever a change in the number of the last element of the vector in allCycles array is detected, or when this number is found to be equal to the number of the source crosspoint. The element is added to the vectors in the allCycles array as long as the counter does not exceed noSuccessors.

Initialization and addition of a new vector to the set of all cycles occurs also when the decoder encounters an element with a non void set of incomplete cycles.

The final step is reversing the order of the elements in each vector found in the array of all cycles.

The coded vector of Table 3 will be used to illustrate the decoding procedure. The output of the decoder is illustrated in Table 5. The decoded cycles are denoted here by the letters A to G. The first element encountered by the decoder is element 7, which has a final number of successors set to 0 and a set of incomplete cycles containing only one element–element 1.

The first vector to be added to the allCycles array contains the number of the element 7. The second vector retains the information related to the incomplete cycles attached to element 7 as well.

Table 5
Decoded Cycles for the Coded Vector of Table III

	'All Cycles' Array			
A			2	7
B			2	7 (1)
C			2	2
D		2	6	2
E			2	6 (1)
F	2	3	1	2
G		2	3	2

The second element read by the decoder is element 2, whose number is equal to the number of the first element in the coded vector (the source crosspoint). Therefore a new vector is initialized and added to the array of all cycles.

The third element encountered by the decoder is processed similarly.

The fourth element to be decoded is 6, which has one successor and one incomplete cycle, ending in element 1. Crosspoint 6 is appended only to the last vector added to the array of allCycles, since the last element found in this vector is 2 (the end of a cycle). To retain the information related to the incomplete cycle of 6, a new vector is subsequently initialized and added to the array of all cycles.

The fifth element encountered in the decoding process is 2, which is treated the same as the second and third.

Next item to be read is element 1, who is found to have one successor, so it is appended to the last decoded vector in the array.

A new vector is initialized and added to the array of all cycles to retain the seventh element read by the decoder.

Element 3 has 2 successors, so the array of allCycles is read backwards and element 3 is appended to the vectors {2} and {2, 1}.

The last encountered element (the source) is appended similarly to obtain the full output of the decoder.

Finally, all decoded cycles are reversed to obtain the results shown in Table 5.

4.4. Separation between Complete and Incomplete Cycles. Feeding the Encoder with Incomplete Cycles

The output of the decoder is further separated into arrays of complete and respectively incomplete cycles. Complete cycles are the vectors whose first and last element numbers are equal to the number of the source crosspoint (e.g. cycles C, D, F and G of Table 5). Incomplete cycles are decoded vectors ending in an element whose set of incomplete cycles is non void (e.g. cycles B and E of Table 5). The encoder is fed again with the partially covered cycles until none are found in the pool of incomplete cycles.

Auxiliary parameters (i.e. noSuccessors and incompleteCycles) for the elements in an incomplete cycle are reset before feeding the encoder again with the cycle. Each element's number of successors is set to be final and 1, whereas each element's set of incomplete cycles is set to be void. The last element in the incomplete cycle retains the information related to the search directions that are yet unexplored, so this information has to be temporarily stored before resetting the parameters.

The coded vector is initialized to be the decoded incomplete cycle, and the encoder is fed the new coded vector and the number of the unexplored search direction. In case of the first decoded incomplete cycle in Table 5, the coded vector becomes {2, 7}, and the encoder starts adding coded elements to the coded vector by appending element 1 and searching its outgoing crosspoints.

5.5. Retrieval of the Actual Cycles from the Crosspoint Cycles

The complete cycles output by the decoder contain the information related to the sequence of crosspoints encountered in that cycle. Each crosspoint stores all sequences of states (branches) emerging from it and ending in the same or another crosspoint. The actual cycles can be retrieved from the decoded complete cycles as sequences of combinations $\bigcup_i B_{m,n}^{k_{m,n}}$, where $i = 0 : L-2$ (L = the length of the complete cycle) is the index iterating over the elements in the complete cycle, m is the number of the crosspoint element at index i in the complete cycle, n is the number of the crosspoint element at index $i + 1$ in the complete cycle and $k_{m,n}$ is the index of the selected branch connecting crosspoints m and n.

Computation of all cycles for a node that is not a crosspoint is slightly different from the procedure described above, in the sense that first identification of the crosspoints containing branches that include the state of interest is to be performed. The search is thus reduced to finding all cycles for each of the identified crosspoints. Finally, trimming is to be performed for the first and last branches in the cycle at the stage of retrieving the actual cycles from the crosspoint cycles.

4.6. Benefits of the Proposed Algorithm

If actual vertex sequences forming a cycle are needed, methods for calculating all cycles in a graph generally rely on the filling of a tree based on the graph, together with either breadth first search or depth first search algorithms applied on the tree. Filling a tree based on the graph is subject to memory constraints, as it is impossible to know beforehand how many nodes the tree will have in the end (the resulting tree may include the same node several times on different branches). Additionally, all nodes in the tree have to keep track of all their ancestors, to make detection of loops possible while filling the tree.

Using the proposed approach is beneficial with respect to the memory size needed to keep track of the elements within a cycle. By structurally reducing the state space, and by the nature of the encoding procedure, a smaller number of objects is needed in order to represent the information describing a set of cycles. The length of the coded vector will always be less than the number of encoded complete cycles, plus the number of crosspoints in the reduced RG.

The actual cycles obtained from the decoded complete crosspoint cycles may simply be stored in arrays of integers. Partially covered paths leading to a loop are no longer considered for storage after each encoding-decoding step takes place. The encoded elements do not need to have knowledge of all their ancestors (or descendants), as detection of loops is done online, based on the structure of the coded vector. Therefore the amount of information to be retained by each encoded element is smaller.

6. RELATED WORK

Farrow and colleagues [9] were interested in quantifying the information regarding both cycle and transient structure of a network. This was achieved by means of developing the scalar equation approach to Boolean network models. The authors proposed investigations on designing an algorithm to systematically find all possible types of scalar equations of Boolean network equations as future research.

Manivannan [10] designed an algorithm for detection of knots and cycles in a distributed graph. The method relies on exchange of messages between the vertices within the graph. If the initiator vertex is part of a knot, the exact nodes that are involved in the knot will be found. Otherwise, the algorithm outputs the set of nodes which are sure to be in a cycle with the initiator.

In order to deal with the state space explosion problem, several graph reduction techniques were proposed in the literature. Muhanna [11] mentioned the recursive removal of all source and sink nodes from the graph (since they cannot be involved in a cycle). Further reduction in the size of the graph was achieved by partitioning the graph into its strong connected components. Koppol [12] suggested grouping certain observational equivalent states into a single condensed state, followed by removal of the states found to be satisfying specified conditions. Elimination of redundant paths, while preserving observational equivalence, was also taken into consideration.

5. CONCLUSION

Service encapsulation of processes can achieve a high level of reconfigurability and adaptability of manufacturing lines. Alternatives to scheduling should be researched to ensure natural response to machine failures or replacements and to changes in required product types and quantities. Possible solutions should take advantage of the relations between the state spaces of each orchestrator and the physical layout of the system. An explicit state view of the line and its participants is obtained through formally representing all orchestrators and the physical layout. It is thus possible to have knowledge of executed/pending processes for each orchestrator, their positioning in the line and the activities of the neighboring devices. Quality criteria can be elaborated based on the updated formal models to assist decision taking.

For each orchestrator, the selection of the next device can be redefined as a search problem in the model of the physical layout of the system. The decision on the needed transportation services is a problem of search in the state space of the physical layout of the model. Traditional scheduling approaches rely on known input job mix and models that represent all possible device-process mappings for the given jobs. A different model and search space is constructed every time a change occurs in the line. The proposed solution structure is beneficial with respect to the search space, which is the same for every possible orchestrator entering the line, unless additions to the equipment are made.

Each orchestrator faces several options at the point of device selection for its requested processes. Although the selection is done gradually, the selection criteria may and should take into consideration (some of) the subsequent needed services, and the locations where they can be physically performed in the line. These options are the potential orchestrator routes in the physical layout of the system. Device selection may be performed for one device at a time only, but each time this selection is done the orchestrator can register its intention to follow a certain route in the physical system that may best satisfy its needs. In case of machine failure or replacement, the orchestrator will recognize the new surrounding situation, because the discovery process is performed continuously. Therefore it may unregister from the initially planned route and the evaluation of new routes may begin once more. The registering of an orchestrator's intentions is valuable as each path may support physically a certain amount of pallets only. The number of orchestrators that intend to use a certain itinerary together with the physical constraints of the itinerary constitute important decision criteria when selecting the devices.

For path quantification, an algorithm to compute all cycles in a graph is presented in its general form. The method extracts the exact sequences of nodes within each cycle at the expense of a lower memory cost than other approaches known by the authors. The cycles of interest for the presented problem are the paths in the physical layout of the system that correspond to each orchestrator's requirements. Information about these cycles and their average cycle time, together with real time data from the physical model will help the system adjust itself to incoming pallets, no matter what process sequencing is requested by each.

Cycle computation can be performed offline, as the search space remains the same unless the machines are physically removed or added. In the case of machine removal, the corresponding cycles may be deleted from the existing cycle set. Machine addition can be addressed by analyzing the newly imposed physical constraints (the relationship of the introduced equipment to the already existing devices), and making the necessary modifications on the cycle set directly (without the need to compute once more the entire set). Machine replacements will not affect the existing cycle set–the routes will be the same, however they will be considered only by the orchestrators that need the newly advertised services.

REFERENCES

[1] M. Zhou, K. Venkatesh, Modeling, Simulation and Control of Flexible Manufacturing Systems–A Petri Net Approach, World Scientific Publishing, 1999.

[2] M. Zhou, Petri Nets in Flexible and Agile Automation, Kluwer Academic Publishers, 1995.

[3] J. L. Martinez Lastra, I. M. Delamer, Semantic Web Services in Factory Automation: Fundamental Insights and Research Roadmap, *IEEE Transactions on Industrial Informatics*, **2**, 1-11, Feb. 2006.

[4] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, **77**(4), 541-580, April 1989.

[5] M. Rausch, H.-M. Hanisch, Net Condition/Event Systems with Multiple Condition Outputs. In: Proceedings of the Symposium on Emerging Technologies and Factory Automation, 3069-3094. IEEE Press, Paris (1995).

[6] H. M. Hanisch, J. Thieme, A. Luder and A. Wienhold, Modeling of PLC Behavior by Means of Timed Net Condition/Event Systems. In Proceedings of 6th International Conference on Emerging Technologies and Factory Automation, 391-396. IEEE Press, Los Angeles (1997).

[7] D. Marin, M. Paolucci, S. A. McIraith, M. Burstein, D. McDermott, D. McGuiness, B. Parsia, *et al.*: Bringings Semantic to Web Services: The OWL-S Approach. In Proceedings of SWSWPC, 26-42, San Diego (2004).

[8] J. Gross, J. Yellen, Graph Theory and its Applications, CRC Press, NJ: 1999, 31-33, 126.

[9] C. Farrow, J. Heidel, J. Maloney and J. Rogers, "Scalar Equations for Synchronous Boolean Networks with Biological Applications," *IEEE Trans. Neural Networks*, **15**(2), March 2004, 348-354.

[10] D. Manivannan and M. Singhal, "An Efficient Distributed Algorithm for Detection of Knots and Cycles in a Distributed Graph," *IEEE Trans. Parallel and Distributed Systems*, **14**(10), October 2003, 961-972.

[11] W. A. Muhanna, "Composite Programs: Hierarchical Construction, Circularity and Deadlocks", *IEEE Trans. Software Engineering*, **17**(4), April 1991, 320-332.

[12] P. V. Koppol, R. H. Carver and Kuo-Chung Tai, "Incremental Integration Testing Of Concurrent Programs", *IEEE Trans. Software Engineering*, **28**(6), June 2002, 607-623.

[13] C. Popescu, J. L. Martinez Lastra, "A Method for the Formal Representation of the Boolean Conditions of Orchestrated Services", AINAW 2008, 22nd International Conference on, 25-28 March 2008, 1410-1415.