A Petri Net-based Approach to Incremental Modelling of Flow and Resources in Service-Oriented Manufacturing Systems

Corina Popescu, Maria Cavia Soto, Jose Luis Martinez Lastra

A Petri Net-based Approach to Incremental Modelling of Flow and Resources in Service-Oriented Manufacturing Systems

Journal:	*International Journal of Production Research*
Manuscript ID:	TPRS-2010-IJPR-0601.R2
Manuscript Type:	Original Manuscript
Date Submitted by the Author:	22-Dec-2010
Complete List of Authors:	Popescu, Corina; Tampere University of Technology, Department of Production Engineering Cavia Soto, Maria; Universidad de Cantabria Martinez Lastra, Jose; Tampere University of Technology, Department of Production Engineering
Keywords:	PETRI NETS, MODELLING, DYNAMIC SCHEDULING, E-MANUFACTURING
Keywords (user):	Service Oriented Architecture, Timed Net Condition Event Systems

A Petri Net-based Approach to Incremental Modelling of Flow and Resources in Service-Oriented Manufacturing Systems

Corina Popescu[a], M. Cavia Soto[b], Jose L. Martinez Lastra[a]

[a] *Department of Production Engineering, Tampere University of Technology, Tampere, Finland;* [b] *Universidad de Cantabria, Santander, Spain*

(Received June 2010; final version received December 2010)

In order to overcome the myopia problem, routing strategies must be based on formal representations of flow that automatically account for modifications in the values of parameters of interest and in the model itself. This work addresses this problem and discusses how to automatically incorporate resources (e.g. workstations/ transportation devices/ storages) in a Petri Net derived model of flow that is modifiable at runtime to reflect and influence the routing in a manufacturing line. The modelling approach takes into consideration scalability needs and was experimentally validated. The applicability of the models is shown for PN-based dynamic scheduling.

1. Introduction

Frequent changes in production demand and the continuously increasing time-to-market pressure command manufacturing line modifications that are sometimes subject to critical deadlines. The required adjustments range from PLC-level program changes to machine/robot replacements and sometimes even reorganization of the entire line.

The clear separation between the set of actions that modify the state of the world (the process), the view that the outside world has of this set of actions (the encapsulation of the process as a service) and the physical equipment where a process executes (the resource) has been recognized to have a huge potential to address these problems (Delamer and Lastra, 2007). The technology of Web Services (WS) deploying Service Oriented Architecture (SOA) offers the necessary infrastructure to abstract a manufacturing system as a set of service encapsulations of provided and requested processes (equipment skills and product needs). This type of infrastructure allows both changes in the values of parameters of interest (online equipment

modifications) and the flow itself (variations in product type demand order) to be recognized and responded to in a natural way.

In an agile world, the capability to change rapidly is a desirable property of manufacturing systems. However, this capability alone does not guarantee a good overall performance. Good support of reconfigurability and adaptability through WS technology is ensured only if dynamic decision taking mechanisms rely on formal flow representations that continuously reflect the situation in the line.

The main objective of this research is to develop a methodology to automatically modify the model of Web Services orchestration itself in order to account for elements and events newly introduced to or retrieved from the represented world. The discussed results are an extension of previous work (Popescu et. al, 2009). This paper discusses how to automatically incorporate resources (e.g. workstations/ transportation devices/ storages) in a Petri Net derived model of flow that is modifiable at runtime to reflect and influence the routing in a manufacturing line. A method to automatically associate context information of each pallet with the elements of the formal model of flow is also presented. Context information is appended to elements in the model automatically, while the model is constructed, and the added data influences the firing rules of the transitions in the model, during the scheduling search.

The paper is organized as follows: Section 2 discusses the background of this work: the terminology used (‘service oriented manufacturing systems’), related work on modelling and scheduling, and the formalism chosen for modelling. Section 3 discusses the modelling of resources in a modular, typed and composable way. Section 4 introduces an incremental approach to automatically modify the model of flow in manufacturing systems that use Web Services to implement the Service-

Oriented-Architecture pattern. Section 5 discusses scheduling on variable-size PN models of flow and presents the proposed approach to embed context information in the model. Section 6 gives implementation details and describes the experiments to validate the proposed modelling approach. Section 7 draws the conclusions and discusses future challenges. Appendix A gives details on the conflict detection and resolution mechanisms utilized during the scheduling search presented in Section 5.

2. Background

2.1. Service Oriented Manufacturing Systems

Service-oriented manufacturing systems are manufacturing systems that use Web Services as a technology to implement the Service Oriented Architecture pattern. The term 'service-oriented' refers to the specific architecture and technology that are used to implement the middleware of this type of systems.

Dedicated Manufacturing Systems, Flexible Manufacturing Systems and Reconfigurable Manufacturing Systems can be encapsulated as services if a minimal set of *implementation constraints* is respected. This set is the Devices Profile for Web Services Specification (DPWS), which is an extension of the Web Services protocol suite. Initially published in May 2004 and submitted for standardization to OASIS in July 2008, DPWS defines a minimal set of implementation constraints to enable secure Web Service description, messaging, dynamic discovery and publish/subscribe eventing at device level. DPWS is built around a group of Web Services standards: WSDL, XML Schema, SOAP, WS-Addressing, WS-Metadata Exchange, WS-Transfer, WS-Policy, WS-Security, WS-Discovery and WS-Eventing.

The adoption of DPWS has been supported by several European research projects. Implementations of DPWS in embedded devices have been achieved in the ITEA-SIRENA (SIRENA), and ITEA-SODA (SODA) projects. DPWS-enabled

devices were implemented and tested in pilot prototypes in the industrial domain (FP6-SOCRADES).

Services are encapsulations of processes and can be thought of as interfaces. A service provides a clear separation between the way the encapsulated process is executed and the view other entities have of the process from the outside. Services are loosely coupled (e.g. the technical details of two collaborating applications are hidden from each other) and can be (de)composed to whichever level of granularity may be required (the highest level focuses on business processes). Additionally, if annotated semantically, a service may be automatically discovered, invoked and composed.

From an SOA perspective, a manufacturing system 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, i.e. the services that should operate upon the raw material to obtain a final product.

In a system there can be as many orchestrators as users with needs (i.e. as pallets with raw products that should circulate through the line to be processed). The orchestrated services are those needed (requested) by the users. In the case of a manufacturing line, the users are the pallets circulating through the line. The needs are the services that should be performed on the raw material to obtain a final product. Once the needs of a user are transferred to the system, the system will translate these needs into a desired orchestration.

Following the SOA pattern, pallets (service requestors) *search* and *locate* the needed services in the order specified by their corresponding orchestrators. The devices (the service providers) *publish* the processes they can offer. Selections of each

device to operate upon a pallet 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. These steps take place for each service specified in the orchestrator of a pallet, until all product needs are satisfied and the pallet exits the line.

2.2. Incremental Modeling of Flow

Wang and Deng (1999) present an incremental multi-levelled modelling and verification methodology based on Time Petri Nets (TPN) and Real Time Computational Tree Logic (RTCTL). TPNs are used to manually represent component behavior and connections. RTCTL is used to describe time critical constraints as formulas over communication ports (each port represents an atomic proposition, which is true at the moment that a token arrives in the port).

An incremental Petri Net-based modeling approach of production sequences for logic control design is presented in (Castelnuovo, A. et. al, 2007). Subnets are progressively added to a partial model until all specifications have been included. The approach is based on a generic feedforward connection rule and on specifications of the logic behavior of sets of transitions (Binary Firing Patterns). The used nets (feedforward Petri Nets - FFPNs) are very similar to the workflow nets of (Aalst). In order to obtain recipe models holding the minimum requirement for system correctness (boundedness, liveness and reversibility), the authors describe a well-definiteness property to be achieved at each stage of the modeling procedure, as the model is incremented (the property is similar to the concept of soundness (Aalst, 1999)). It is not clear whether it is possible to also easily remove blocks from an FFPN.

No approach known to the author exists that represents automatically a Petri Net - derived model of mixed sequences of operations that accounts for (additions/removals of) resources and possible disruptions (machine failures/ urgent orders / unload operations), to continuously and accurately reflect the situation in the line *during the scheduling search*.

2.3. Scheduling

Finding an optimal schedule in a distributed system is in almost all realistic scenarios an NP complete problem, i.e. computationally intractable (Kopetz, 1997). To account for the effects of the numerous factors influencing the factory floor, schedulers should aim to obtain a reasonable load on the shop rather than optimized sequences (Silva and Vallete, 1989). The experimental findings of Lawrence and Sewell (1997) support this claim: the study compares optimal seeking algorithms versus heuristic methods applied to 53 standard job shop scheduling problems, when processing times are uncertain. As processing time uncertainty increases, the results indicate convergence in the performances of fixed optimal sequences and fixed heuristic sequences. The best performance is obtained with dynamically updated heuristic schedules. Matsuura and colleagues (1993) refine the analysis by categories of uncertainties, and report a better performance of sequencing versus dispatching (for small machine breakdowns) and of dispatching versus sequencing (for specification changes and rush jobs). To take these findings into account, they propose a switching approach between sequencing and dispatching according to the manufacturing situation.

Research is needed to input real time information collected from the factory/plant to product routing /asset management algorithms, to assist the devices/resources cooperate (optimally) while reducing waste caused by loss of energy/material and inefficient processes.

2.4. Timed Net Condition Event Systems

The Timed Net Condition Event Systems (TNCES) (Rausch and Hanisch, 1995; Hanisch et al., 1997) formalism enhances the expression capabilities of Petri Nets (Murata, 1989) 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 (event inputs/outputs and condition inputs/outputs) and a modular hierarchy.

An example of a simple TNCES module of name 'Example TNCES Module' and type 'tnces_module_example' is depicted in Figure 1.

Figure 1 Example TNCES Module

Apart from sets of places ({p1, p2, p3, p4, p5}), transitions ({t1,t2,t3,t4}) and flowarcs ({(p1,t2), (t1,p1), (p2,t1), (t2, p2), (p4,t3), (t2, p4), (p5,t4), (t3, p5), (p3, t2), (t4, p3)}), which are present in any PN, this TNCES module has event inputs ({ei1}), event outputs ({eo1}) , condition inputs ({ci1}) and condition outputs ({co1}). Event arcs ({(ei1, t2), (t4, eo1),) link event inputs to transitions / transitions to event outputs. Condition arcs ({(ci1, t1), (p3, co1)}) link condition inputs to transitions / places to condition outputs.

Figure 2 presents a composition of two TNCES modules, of names and types B and R.

Figure 2. Example of condition arc (a) / event arc (b) interconnection between TNCES modules

Two possible types of module interconnections may exist in a TNCES model: condition arcs (e.g. Figure 2a: {(B.co1, R.ci1)}) and event arcs (e.g. Figure 2b: {(B.eo1, R.ei1)}).

Condition and event arcs influence the firing rules in a TNCES module. A transition that is *marking enabled* (i.e. has at least one token in each of its input places) may fire *at any point in time* in case it is also *condition enabled*. A condition enabled transition that is not also marking enabled may not fire. Considering the example in Figure 2a: transition R.t1 may fire at any point in time if there is at least one token in the place R.p1 (i.e. the transition is marking enabled) and if there is one token in the place B.p2 (i.e. the transition is condition enabled through the module condition arc (B.co1, R.ci1)). A transition that is marking enabled will fire *immediately* if it is also *event enabled*. An event enabled transition that is not marking enabled will not fire. In the module depicted in Figure 2b, transition R.t1 fires *immediately* if there is at least one token in place R.p1 and *once* transition R.t2 fires (change in state signalled through the module event arc (B.eo1, R.ei1)).

TNCES modules may be associated delay times with flowarcs outgoing from places.

3. Modelling Resources

This section gives details on the construction procedures of modular and composable TNCES models for three main types of resources: processing workstations (machines or robots), transportation devices (robots or conveyors) and storages.

Each resource is associated exactly one *status*-typed TNCES module, to describe its state (i.e. idle/busy/unloading). Additionally, a resource is associated as many *usage*-typed TNCES modules as necessary, to describe location and invocation

of the resource by service requestors, according to the Service-Oriented-Architecture pattern.

The interface of *status* (respectively *usage*) –typed modules is the same, irrespective of the type of modelled resource (Figure 3).

Figure 3 status and usage typed modules – interfaces

status –typed modules have three event inputs (*invoked/finished/transfer_finished*) and three condition outputs (*available/orch_for_transfer/to_next*). *usage*-typed modules have two condition inputs (*available* and *orch_for_transfer*) and two event outputs (*invoked* and *finished*). The inner elements are defined per resource type, and can be generated automatically (Popescu et al., 2009).

3.1. Processing Workstations(robots / machines)

The internal elements of the TNCES typed status and usage modules used to describe a processing workstation are shown in Figures 4 and 5.

Figure 4 Representing processing workstations: TNCES module of type 'status'

Figure 5 Representing processing workstations: TNCES module of type 'usage'

There can be exactly one *status* and as many *usage* modules as necessary per resource in the overall flow model. A *usage* module is added to the overall model each time a device is identified as potential provider for a requested service. Time constraints are associated to the (p2 t2) flowarc of the usage module, to account for multiple processing capabilities associated with a device.

Figure 6 illustrates the model of a resource that is located and possibly invoked once. In case the resource is idle (i.e., there is one token in place *resource_status.p1* (m(*resource_status.p1*)=1) and identified as potential provider of service for a particular requestor/pallet (m(*resource_usage.p1*)=1), transition

resource_usage.t1 may fire at any time (condition-enabled transitions may fire at any point in time as long as the condition(s) hold and they are marking enabled).

Figure 6 TNCES model of a resource located once

The information regarding the (un)availability of the resource is carried through the condition arc that links place *resource_status.p1* to transition *resource_usage.t1*. If *resource_usage.t1* fires, a token is placed in place *resource_usage.p2*. At the same time, the firing of *resource_usage.t1* is announced through the module event arc connecting the *resource_invoked* event output and input. The triggering of this event will immediately cause the firing of transition *resource_status.t1* (Only event-enabled transitions may have a triggered firing (if they are also marking enabled)). Consequently, a token is placed in place *resource_status.p2* as well, condition-enabling transition *resource_usage.t2*.

Figure 7 illustrates the situation in which the same device is searched for by two different requestors. In this case, the two separate *resource_usage* modules initialized at *m(p1)=1* reflect the case in which two pallets have discovered this particular device to be capable of responding to their current demands. Resource invocation can take place only once.

Figure 7 TNCES model of a resource located twice

3.2. Transportation Devices

Two types of transportation devices are considered here: robots and conveyors. The typed *status* and *usage* TNCES modules (Figures 4 and 5) are used to model robots and conveyors of one location (crossing points). For conveyors of more than one location, the *usage* typed module does not change. The *status* typed module remains the same with respect to the interface elements; however, its internal representation changes. Figure 8 illustrates the *status* typed module for unidirectional

conveyors of N locations. The models can be automatically generated if the number of locations per conveyor is known (Popescu, 2009).

Figure 8 TNCES module of type 'conveyor_status' (N locations)

The interconnections between *usage* and *status* typed modules follow the rules depicted in Figures 6 and 7: Module condition arcs link the *available/orch_for_transfer* condition output of the status module to the *available/ orch_for_transfer* condition input of each usage module. Module event arcs connect the *resource_invoked / finished* event outputs of each usage module to the *resource_invoked / finished* event input of each status module.

3.3. Output Storages(unloading services)

Load/unload operations are implicitly added to each standalone orchestrator model once it is added to the overall flow model (entry and exit points to/from the line are described semantically).

As a result of load operations, the overall flow representation is updated automatically with *status* and *usage* typed modules representations of the entry point (workstation/conveyor/etc.) of the newly added orchestrator. The update proceeds to include in the model the transportation devices needed to take the loaded pallet to the desired destinations. Input storages are not incorporated in the overall flow model. Output storages must be incorporated in the overall flow model, because knowledge of the amount of remaining available unload space is needed e.g. when seeking a feasible schedule.

Figure 9 illustrates two interconnected TNCES models of type status and usage representing an output storage. The interconnections between the modules include one module event arc (output_storage_status.*transfer_finished* X

output_storage_usage.*transfer_finished*) and two module condition arcs {((output_storage_status.*available* X output_storage_usage.*available*); (output_storage_usage.*transfer_to_storage* X output_storage_status. transfer_to_storage)}. Conflict resolution (e.g. for the conflict between transitions t1 and t3 in the output_storage_status module) relies on a matrix-based algorithm that is presented in Appendix A.

Figure 9 (Distinguishable) output buffer. Two part types.

4. Incremental Modelling of Flow

Each time a pallet is introduced into the line, its standalone orchestrator formal model must be generated and combined with the existing overall flow model into a final orchestrator mix model. Figure 10 illustrates the dynamic generation procedure for the orchestrator mix model through a small example:

Figure 10 Standalone orchestrators. Separation of flow –related representation from atomic services representation

A pallet with the required sequencing described by orchestrator O1 (Figure 10, top side) is first introduced into the line. The inner elements of each internal module are abstracted from for simplicity. Upon entering the line, the TNCES model of O1 is automatically generated. As there is only one pallet in the line, this model is a full formal representation of the current orchestrator mix.

Another pallet, characterized by O2 (Figure 10, bottom side), follows the first one after some time. O2 is a sequence of three atomic services: S1 (the same service searched by O1 initially), S3 and S4. The newly generated formal model (O2) must be added automatically to the already existing orchestrator mix model.

For each standalone orchestrator, the atomic services are first separated from the rest of the model. An example of the outcome of this procedure is illustrated in Figure 10 (right side). The top-left side of the figure depicts orchestrator O1 as generated automatically in the initial stage. After separation (top-right), orchestrator O1 consists of only the Sequence module. Additional condition and event input and output sets are created to enable the necessary links to the external modules Service S1 and Service S2. To ensure a correct separation, the order in which the atomic services are withdrawn from the initial orchestration module is tracked. This order must be consistent with the generation order of corresponding additional event input – event output pairs.

The TNCES orchestrator mix model obtained by adding the model of O2 to the original orchestrator mix model (i.e., O1) is illustrated in Figure 11.

Figure 11 Orchestrator mix model for the two orchestrators of Figure 10.

The generation procedure considers the orchestrator model to be added at this step (Figure 10, bottom-right side). A check is performed to see whether the atomic services of the newly introduced model are already part of the existing mix model or not. In this case, service S1 is found to already have been included in the orchestrator mix model. Therefore, only the necessary connections are added to the mix module (i.e., the event arcs (O2.*start_s1* X S1.*start*) and (S1.*end* X O2.*end_s1*) and the condition arc (S1.*available* X O1.*available_s1*)). The other two atomic services involved in the formal model of O2 - S3 and S4 - are not yet part of the current mix model. Therefore, the corresponding TNCES modules and the necessary connections are added to the main module.

The mapping of services to resources (i.e. to physical devices capable of performing the processes encapsulated by services) is done at runtime (continuously, while the pallets circulate through the line), to account for online equipment modifications or additions. Each TNCES module of type 'atomic service' is automatically replaced with corresponding *status* and *usage* modules for each of the resources (transportation/workstations/storages) involved in the processing. Each resource may be assigned multiple *usage* modules (one for each potential resource invocation), and exactly one *status* module. The model is then updated with the necessary orchestrator-resource and resource-resource connections.

The inter-module connections in the TNCES flow model are classified and explained as follows.

5.1. Status-Usage connections

For one resource, status typed modules are connected to each of the corresponding usage modules through four connection arcs (Figure 12):

Figure 12 status-usage connections

- A condition arc linking the '*available*' condition output of the *status* typed module to the '*available*' condition input of the *usage* typed module.
- A condition arc linking the '*orch_for_transfer*' condition output of the *status* typed module to the '*orch_for_transfer*' condition input of the *usage* typed module.
- An event arc connecting the '*finished*' event output of the *usage* typed module to the '*finished*' event input of the *status* typed module.
- An event arc connecting the '*invoked*' event output of the *usage* typed module to the '*invoked*' event input of the *status* typed module.

These rules are valid for all resource modules except the TNCES modules of the type '*output_storage*' (Figure 9).

5.2. Resource-to-Resource connections

Two connection arcs are needed to model transfer of orchestrators from a source resource to a destination resource (Figure 13):

Figure 13 Resource-to-resource connections

- A condition arc linking the '*to_next*' condition output of the *status* typed module of the source resource to the '*available*' condition input of the *usage* typed module of the destination resource.
- An event arc connecting the '*finished*' event output of the *usage* typed module of the destination resource to the '*transfer_finished*' event input of the *status* typed module for the start resource.

5.3. Orchestrator-Resource connections

The orchestrator-service connections are replaced with corresponding orchestrator-resource linkages, for each resource capable of performing a service (Figure 14):

Figure 14 Replacing orchestrator-service connections with orchestrator-resources connections

- event arc connecting the *start_s_j* event output of the orchestrator to the start event input of the '*atomic service*' typed TNCES module is replaced with a condition arc. The new connection links the *start_s_j* condition output of the orchestrator to the *available* condition input of the *usage* typed module associated with the first needed transportation device.

- The event arc connecting the *end* event output of the '*atomic service*' typed TNCES module to the *end* event input of the orchestrator module is replaced with another event arc. The new link connects the *finished* event output of the processing resource module associated with the replaced service to the *end_s_j* event input of the orchestrator.
- The condition arc connecting the *available* condition output of the '*atomic service*' typed TNCES module to the *available* condition input of the orchestrator module is replaced with another condition arc. The new link connects the *available* condition output of the *status* typed module of the processing resource module associated with the replaced service to the *available* condition input of the orchestrator.

5. Scheduling on variable-size models. Embedding context information in the flow model

The scheduling problem is formulated as the process of finding a production schedule for a given set of orchestrators (i.e. pallets associated with product needs - the order of services that are needed to operate on the raw material to obtain a final product) and resources (i.e. devices, the physical equipment of the line), by searching a feasible transition firing order (i.e. from the current marking to the goal marking) in a PN-based formal model that represents accurately the situation in the line.

For production scheduling, Petri nets are used either in conjunction with heuristic rule base systems (Hu et al, 1995; Chincholkar and Krishnaiah Chetty, 1996; Krishnaiah Chetty et al., 1996; Lin and Lee, 1997; Kattan et al., 2003; Wu and Zhou, 2007; Gradisar and Music, 2007) or with search algorithms (Lee and DiCesare, 1994; Zhou and Jeng, 1998; Shih and Sekiguchi,1991; Xiong and Zhou, 1998; Venkatesh and Zhou, 1998; Jeng et al., 1998; Kis et al, 2000; Hu and Li, 2009; Huang et al. , 2008). It is remarkable that the seeking of schedules is *generally conducted offline, on*

manually constructed PN models. The search for a schedule is performed on PN-based models that are constructed *prior* to the initiation of the search.

The scheduling search proposed here operates on a Petri Net model that is modified automatically to account for arrival/exit of new orchestrators to/from the line. The automatically updated orchestrator mix may be subjected to scheduling search *while the model itself is changing*, provided that:

(1) the differences in the structure of the underlying TNCES model are accounted for (addition/removal of an orchestrator to the orchestrator mix will result in addition/reset of corresponding rows and columns in the incidence /condition/event matrixes characterizing the net) and

(2) context information is embedded in the model so that the physical location of each orchestrator is taken into account when selecting firable transitions in the flow model during the scheduling search. This second point is further explained as follows.

Several orchestrators (pallets) may be using the same transportation device (e.g. a multi-location conveyor). The identification of enabled outgoing condition arcs at the output point (i.e. pTransfer in the *status* typed module associated with the transportation device) relies solely on the orchestrator situated at the exit point and its intended destination. Such a case is illustrated in Figure 15, where two different orchestrators (O1 and O2) may use the same conveyor (the modules conveyor_status, *conveyor_usage1*, *conveyor_usage2* and *conveyor_usage3*) for transportation: O1 must be directed towards *resource2* for processing (*conveyor_usage1*), while O2 must head towards resource1 (*conveyor_usage2*). This scenario is not clearly reflected in the structure of the flow model: a token in conveyor_status.*transfer* (place *p9* of the

module) enables both outgoing condition arcs (and therefore, possibly, both destination '*usage*' typed modules).

Figure 15 Two orchestrators use the same conveyor for transportation

Confusion can also occur when one orchestrator might use the same transportation device more than once. In Figure 15, this situation is sketched for orchestrator O1: After visiting *resource2*, transportation to the next processing unit (*resource3*) relies solely on the same device (*conveyor*) that is used for transfer to *resource2* (*conveyor_usage3*). Without additional information, this situation is not immediately distinguishable at the *conveyor_status.orch_for_transfer* condition output.

Track must be kept of each orchestrator's location and intended destination in the '*status*' typed module associated to the conveyor to ensure correct firing of the transitions at the output point. This is achievable through additional information conferred to places and defined inter-dependencies between this information and the firing rules governing the dynamics of the net.

The tokens in a place P may be associated information Info(P) regarding the id of the calling orchestrator and the associated services. This information is stored in four fields:

$$\text{Info}(P) = \{\text{Info}_{\text{Orchestrator}}(P), \text{Info}_{\text{Service}}(P), \text{Info}_{\text{ServicesFrom}}(P), \text{Info}_{\text{ServicesTo}}(P)\},$$

attached to the element corresponding to P in the marking vector. The initialization of these four fields is done when *usage* typed modules are first added to the overall flow model:

- For each *usage* typed module associated with a workstation, the default value of the $Info_{Service}$ field is modified to reflect the particular workstation process that the added module is being associated with.
- For each *usage* typed module associated with a transportation device, the default value of the $Info_{ServicesFrom}$ and $Info_{ServicesTo}$ fields is modified accordingly.

The inter-dependencies between place-related information and the firing rules governing the dynamics of the net are explained as follows.

- The first rule concerns firing rules in *status* and *usage* modules that are all associated to the same resource (Figure 12). In case transition t_1 in a '*usage*' typed module fires, the Info(p1) value of the module is copied into the Info(p2) field of the associated '*status*' module. The rule is generally expressible as:

USAGE.t1 fires ⇨ Info (STATUS.p2):= Info (USAGE.p1)

(where USAGE and STATUS are the names and types of the discussed modules). For example, in Figure 15, after the firing of transition conveyor_usage1.t1 the marking vector of the flow model is updated accordingly:

Info (conveyor_status.L1_busy):= Info (conveyor_usage1.t1)

- The second rule concerns firing rules inside *status* typed modules. The information associated with the tokens of 'busy' places in *status* typed modules is propagated with the firing of the local transitions of the module. The propagation rules for *status* typed modules of unidirectional conveyors of n locations (Figure 8) are:

  • n≥2:

    if t_j fires (j=2:n) ⇨

      Info(l_j_busy) := Info(l_(j-1)_busy);

      Info (l_(j-1)_busy):=NULL

if t_{n+1} fires ⇨

Info ($p_{Transfer}$) := Info (l_n_busy);

Info (l_n_busy):=NULL

if t_{n+2} fires ⇨

Info ($p_{Transfer}$):=NULL

- n=1:

if t_{n+1} fires ⇨

Info($p_{Transfer}$) := Info(l_n_busy);

Info(l_n_busy):=NULL

if t_{n+2} fires ⇨ Info ($p_{Transfer}$):=NULL

The above specified rules are generally applicable to *status* typed modules.

- The third rule concerns resource-to-resource module interconnections that model transfer of orchestrators from a source resource (represented by one $status_{SOURCE}$ module and one or more $usage_{SOURCE}$ modules) to a destination resource (represented by one $status_{DESTINATION}$ module and one or more $usage_{DESTINATION}$ modules). Resource-to-resource transfer is enabled (i.e. transition $usage_{DESTINATION}$.t1 is firable) only for the $status_{SOURCE}$ - $usage_{DESTINATION}$ connections that are characterized by matching token-related information. For the example of Figure 13, the rule depends on the nature of the transfer and is summarized in Table 1.

Table 1 Match check to test information-enabling of a place

In Figure 15, transition resource_usage.t1 is firable iff:

m(conveyor_status.*transfer*)==1 AND

m(resource1_usage.*located*)==1 AND

$$Info_{Orchestrator}(conveyor\_status.\textit{transfer}) == Info_{Orchestrator}(resource1\_usage.\textit{located})$$

AND

$$Info_{ServicesTo}(conveyor\_status.\textit{transfer}) == Info_{Service}(resource1\_usage.\textit{located})$$

6. Implementation

6.1. Case Study

Figure 16 illustrates the line used as a case study for the proposed approach. The line consists of six cells. The robots situated in four of the cells are in charge of drawing model parts of mobile phones (frame, screen and keyboard), of three different types each. The pallets carry the drawing boards. Two of the cells are responsible for load and respectively unload activities. The unload cell contains a static buffer that may be used both as an output storage and for temporarily buffering parts undergoing processing.

Figure 16 Line case study

When entering the line, each pallet's orchestrator has knowledge of the services that need to act upon it to obtain the desired final drawing. Examples of orchestrators for this setting are:

- O_1: *Sequence*[$Frame_1$, *AnyOrder*($Screen_1$,$Keyboard_1$)]
- O_2: *Sequence*[$Frame_2$, *Choice*(*AnyOrder*($Screen_2$,$Keyboard_2$); *AnyOrder*($Screen_3$,$Keyboard_3$))]
- O_3: *Sequence*[$Frame_3$, *SplitJoin*($Screen_3$, $Keyboard_3$)]

The procedures to incrementally construct and update the flow model and to dynamically search for a schedule in the model are implemented in JAVA.

7.2. Experiments

The goals of the conducted experiments were:

1. To validate that the proposed modelling methodology is feasible from the viewpoint of safeness. That is, that no tokens are generated unnecessarily (i.e. falsely representing the situation modeled) within the net once the building blocks of the model are interconnected and the model is executed.
2. To validate that it is feasible (from the viewpoint of timing constraints) to interrupt a scheduling search on a PN model in order to update the model itself (i.e. add/remove elements to/from the model), and then continue the scheduling search (on the updated model) from the point it was initially interrupted on the previous version of the model. Comparison with other scheduling methods is beyond the scope of this paper.

To validate the first point, flow models were constructed statically (prior to the scheduling search) for mixes including 2 to 15 orchestrators of types O_1, O_2 and O_3 (as described in Section 7.1). Backtracking search was subsequently performed on these models to find a schedule. During the search, feasible groups of transitions were identified and conflicts resolved as discussed in Appendix A.

The program was run on an AMD Athlon 64 Processor at 1.99 GHz and 1Gb of RAM. The construction of an initial model takes in average 3 to 5 seconds (CPU time). Table 2 reports the durations (in seconds) of the scheduling search performed on static orchestrator mix (i.e. that remains unmodified during the search), obtained by firing together feasible groups of transitions (see Apendix A for details).

Table 2 Scheduling search duration – in CPU time / seconds (static orchestrator mix)

Several simulation runs were conducted for each orchestrator mix. Throughout each run, the number of tokens remained either 0 or 1 in all places except in *status*-typed modules corresponding to representations of output storages (Figure 9), whose

capacity is greater than 1. The sum of tokens remained constant per typed module (including representations of output storages) throughout each run.

7.3. Proposed implementation architecture

The proposed implementation architecture is shown in Figure 17.

Figure 17 SOA-based implementation schema

When entering a manufacturing line, an orchestrator requests a Scheduling Service. Through the Service Broker, it identifies the alternatives and invokes the Scheduling Service Provider. Its flow description is automatically translated to the TNCES formal model and added to the orchestrator mix model(s) to be considered in the search.

Scheduling Service Providers are requestors of Machining/Transportation services information, obtainable through the Service Broker. In addition to this, Scheduling Service Providers may request Monitoring services to account for machine breakdowns or equipment additions/removals while executing.

The invocations of Scheduler Service Providers could be done in two ways:

(1) Each new orchestrator invokes each of the available Scheduling Service Providers when entering a manufacturing line. In this way, all available Scheduling Service Providers operate on the same underlying formal model.

(2) Each new orchestrator invokes exactly one available Scheduling Service Provider. The feasibility of this scenario remains to be studied, as it implies the need for synchronization between all Scheduling Service Providers (the firing of transitions in *status*-typed modules must be made visible to all Scheduling Service Providers to avoid inconsistencies between the found schedules).

7. Conclusions

This work's primary research objective is to construct formal models that will reflect at all times what is happening to product flow in a service-oriented manufacturing system (i.e. that uses Web Services to deploy SOA). The intention is not only to modify parameters of a model, but the model itself, to account for changes in equipment (additions / breakdowns) and in the order of the product requests input to a line. Model updates may be done continuously, while products enter/exit the line and are being processed at various workstations.

A line consisting of six robotic cells was the case study for validation of the proposed modelling methodology. Formal models of individual orchestrators (flows of processes) associated to various product types were automatically constructed and mixed into one final flow model. Scheduling search was subsequently successfully performed on each of the final models. For each search, the TNCES models subjected to scheduling were monitored to check two hypotheses: first, that the number of tokens in each place does not exceed 1 and second, that the sum of tokens in some of the modules remains constant irrespective of the token game. The first hypothesis held true for all typed modules, except representations of output storages (here the number of tokens in places associated to such modules never exceeded the buffer's capacity – as expected). The second hypothesis was validated: the sum of the tokens in each typed module remained constant throughout each simulation run.

Simulations were also run to check the feasibility of adding/removing elements to/from a flow model *during* the scheduling search performed on it. In each case, the scheduling procedure was successfully paused in a point so that the underlying model is modified, and then continued from the same point on. Provided obtained scheduling data is preserved in-between model modifications, it may be concluded that it is feasible to have formal models automatically constructed based on

myopic, local views of individual product needs, in order to obtain a global view of the situation in a line and act accordingly.

Appendix A Sources of Conflict. Matrix-based Conflict Resolution Algorithm

Sources of Conflict

Conflict may result from three main sources:

First, unless it is a multiple location conveyor or a storage, a resource cannot perform two or more services simultaneously. . If the transitions of multiple processing resource *usage* modules associated with the same *status* module are condition-enabled at the same time, only one of the transitions of the *usage* typed modules can fire.

Second, a service cannot be performed on the same orchestrator by two different resources at the same time.). In the TNCES flow model, this situation may appear with multiple *usage* typed modules connected to the same *end_s_j* event input of a calling module. This type of conflict is resolved by requiring that outgoing arc connections from '*to_next*' condition outputs of status typed modules are mutually exclusive. In case such connections condition-enable transitions in more than one *usage* typed module, exactly one of the firable transitions should be allowed to fire.

An additional source of conflicts that must be considered when constructing the search space is the typed nature of the composing modules in the TNCES flow model. Module types dictate the type of internal transition firing, when building the search space. For instance, in Split or Split+Join modules, all transitions that are enabled simultaneously should fire concurrently. In Choice and AnyOrder typed modules, the same scenario requires that only one of the eligible transitions fires. In modules of type 'output storage_status' (illustrated in Figure 9), all transitions connected to the transfer_to_storage condition input are conflicting.

Matrix-based Conflict Resolution Algorithm

At each step of a scheduling search, a decision must be made to select - out of the set of enabled transitions - a group of transitions that may fire together (a firable transition group). To keep consistency with the physical meaning of a transition firing, transition conflicts must be taken into consideration when this selection is made. The firing of some of the enabled transitions must be prevented to account for the semantics of the connection arcs of the TNCES model. As described earlier in this section, such conflicts are automatically detectable based on the structure of the TNCES flow model and information on the current marking.

Figure 18 illustrates the procedure to search for a feasible firable group of transitions via a small case study. The example is given for an input vector of enabled transitions T_{EN} = {39, 43, 57, 59, 69, 71} (where the numbers denote the flat numbers of the transitions in the overall TNCES model). The total number of transitions in the considered model is 71.

Figure 18 Search firable feasible transition group. Example.

The conflict matrix C (partially shown in Figure 19) is automatically obtainable from the structure of the flow model. C[i,j]=1 if there is a conflict between transitions ti and tj; otherwise C[i,j]=0.

Figure 19 Conflict Matrix: example

The purpose of the T_{EN} vector (see Figure 18) is to record all enabled transitions in the model at one step. $T_{EN}[k]=1$ if transition t_k is enabled, otherwise it is 0.

To identify a group of transitions that may fire together, successive subtractions are made from T_{EN}. The steps documented in Figure 18 are explained as follows:

1. The conflict matrix row corresponding to transition 39 is deducted from T_{EN}. The index 39 is selected randomly from the list of non-zero elements of T_{EN}. The index 39 is registered to have already been searched in a searched_indexes list.
2. The newly obtained TEN vector (denoted by $T_{EN}^{(1)}$) has nonzero elements corresponding to indexes 39, 59 and 71. The conflict matrix row corresponding to transition 59 is subtracted from $T_{EN}^{(1)}$. The index 59 is selected randomly from the list of non-zero elements of $T_{EN}^{(1)}$ that are not already contained in the searched_indexes list (i.e. 59 and 71 - since index 39 was already selected at step 1 it is no longer considered for random selection). The index 59 is registered to have already been searched in searched_indexes list.
3. The newly obtained T_{EN} vector (denoted by $T_{EN}^{(2)}$) has nonzero elements corresponding to indexes 39 and 59. As both indexes have been previously considered for subtraction (and therefore all indexes that may be in conflict with the two have been removed from the list of firable transitions), the search ends here. The resulting firable transition group is {39, 59}.

Nodes of the reachability graph may be visited more than once during a scheduling search. In case a search path is abandoned, then alternative paths need to be explored. This requires that a feasible firing transition group is reselected from the set of enabled transitions in the considered start state of the paths. In order to ensure that already selected transition groups are no longer considered when a state is revisited, each selection startpoint should be recorded per node so that it is discarded as startpoint in future path computations.

The general procedure to select a feasible firable transition group in the TNCES flow model at each search step is illustrated in Table 3.

Table 3 Identification of feasible firable transition groups based on knowledge of conflicts in the entire flow model

References

Castelnuvo, A., Ferrarini, L., Piroddi, L., 2007. An Incremental Petri Net-Based Approach to the Modeling of Production Sequences in Manufacturing Systems. *IEEE Transactions on Automation Science and Engineering*, vol. 4 , no. 3.

Chincholkar, A.K., Krishnaiah Chetty, O.V., 1996. Stochastic Coloured Petri Nets for Modelling and Evaluation, and Heuristic Rule Base for Scheduling of FMS. *Int. J. Adv. Manuf. Technol*., vol. 12, pp.339-348.

Delamer, I.M., Martinez Lastra, J.L., 2007. Loosely-coupled Automation Systems using Device-level SOA, *Proceedings of the 5th IEEE International Conference on Industrial Informatics*, Vienna, Austria, pp. 743-748.

DPWS Specification, Available online: http://docs.oasis-open.org/ws-dd/dpws/1.1/pr-01/wsdd-dpws-1.1-spec-pr-01.html, Accessed 11.10.2010

FP6-SOCRADES, http://www.socrades.eu/

Gradisar, D., Music, G., 2007, Production-process modelling based on production-management data: a Petri-net approach. *International Journal of Computer Integrated Manufacturing*, vol. 20, no. 8, pp. 794-810.

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

Hu, G.H., Wong, Y.S., Loh, H.T., 1995. An FMS Scheduling and Control Decision Support System Based on Generalised Stochastic Petri Nets. *Int. J. Adv. Manuf. Technol*., vol. 10, pp. 52-58.

Hu, H., Li, Z., 2009. Modeling and scheduling for manufacturing grid workflows using timed Petri nets. *Int. J. Adv. Manuf. Technol.* , vol. 42, pp. 553-568.

Huang, B., Sun, Y., Sun, Y.M., 2008. Scheduling of flexible manufacturing systems based on Petri nets and hybrid heuristic search. *International Journal of Production Research*, vol. 46(16), pp.4553-4565.

Jeng, M.D., Lin, C.S., Huang, Y.S., 1998. Petri net dynamics-based scheduling of flexible manufacturing systems with assembly, *Journal of Intelligent Manufacturing*, vol.10, pp. 541-555.

Kattan, I., Mikolajczak, B., Kattan, K., Alqassar, B., 2003. Minimizing cycle time and group scheduling, using Petri nets a study of heuristic methods. *Journal of Intelligent Manufacturing*, 14, pp. 107-121.

Kis, T., Kiritsis, D., Xirouchakis, P., 2000. A Petri net model for integrated process and job shop production planning. *Journal of Intelligent Manufacturing*, vol. 11, pp. 191-207.

Kopetz, H., 1997. Real Time Systems: Design Principles for Distributed Embedded Applications. *Kluwer Academic Publishers*, Chapter 11

Krishnaiah Chetty, O.V., Gnanasekaran, O. C., 1996. Modelling, Simulation and Scheduling of Flexible Assembly Systems with Coloured Petri Nets. *Int. J. Adv. Manuf. Technol*., vol.11, pp. 430-438.

Lawrence S.R., Sewell, E.C., 1997, Heuristic, optimal, static and dynamic schedules when processing times are uncertain. *Journal of Operations Management* 15 (1997), pp.71-82.

Lee, D.Y., DiCesare, F., 1994 . Scheduling Flexible Manufacturing Systems Using Petri Nets and Heuristic Search. *IEEE Transactions on Robotics and Automation.* vol. 10, no.2, pp.123 -132.

Lin, J.T., Lee, C.C., 1997. Petri net-based integrated control and scheduling scheme for flexible manufacturing cells. *Computer Integrated Manufacturing Systems*, vol. 10, no.2, pp.109-122.

Matsuura, H., Tsubone, H., Kanezashi, M., 1993, Sequencing, dispatching and switching in a dynamic manufacturing environment. *Int.J. Prod.Res.*, vol.31, no.7 , 1671-1688.

Maturana, F., Shen, W., Norrie, D.H.,1999. MetaMorph: an adaptive agent-based architecture for intelligent manufacturing, *Int. J. Prod. Res.*, vol. 37 , no. 10, pp.2159-2173.

Murata T., 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, vol.77 (4),pp. 541-580.

Popescu, C., Cavia Soto, M., Martinez Lastra, J.L., 2009. Runtime Modeling of Flow for Dynamic Deadlock-Free Scheduling in Service-Oriented Factory Automation Systems. *IETE Technical Review*,vol.26 (3), pp.203-212.

Popescu, C., 2009, 'An Approach to Incremental Modelling of Web Services Orchestration: An Application to Deadlock-free Scheduling in Automated Systems', Dr. Tech., Tampere University of Technology

Rausch M., Hanisch, H.-M., 1995. Net condition/event systems with multiple condition output, *Proceedings of the Symposium on Emerging Technologies and Factory Automation*, vol.1, pp. 592-600.

Shen, W., Maturana, F., Norrie, D.H., 2000, MetaMorph II: an agent-based architecture for distributed intelligent design and manufacturing, *Journal of Intelligent Manufacturing*, vol. 11, pp.237-251.

Shih, H., Sekiguchi, T.,1991. A timed Petri net and beam search based online FMS scheduling system with routing flexibility. in *Proceedings of the International Conference on Robotics and Automation*, pp.2548-2553.

Silva, M., Vallete, R.,1989. Petri Nets and Flexible Manufacturing, *Lecture Notes in Computer Science*, Advances in Petri Nets

SIRENA, http://www.sirena-itea.org/

SODA, http://www.soda-itea.org/

Van der Aalst, W.M.P., 'The Application of Petri nets to workflow management' *Journal of Circuits, Systems and Computers*, vol.8, pp. 21-66.

Van der Aalst, W.M.P., 1999. Interorganisational workflows: An approach based on message sequence charts and Petri nets. *Systems Analysis, Modelling, Simulation* 34 (3), pp.335-367.

Venkatesh, K., Zhou, M.C., 1998. Object-oriented design of FMS control software based on Object Modeling Technique diagrams and Petri Nets. *Journal of Manufacturing Systems*, vol. 17(2), pp.118-136.

Wang, J., Deng, Y., 1999 . Incremental modeling and verification of flexible manufacturing systems. *Journal of Intelligent Manufacturing*, vol. 10, pp.485-502.

Wu, N., Zhou, M.C., 2007 . Real-time deadlock-free scheduling for semiconductor track systems based on colored timed Petri nets. *OR Spectrum*, vol. 29, pp.421-443

Xiong, H.H., Zhou, M.C., 1998. Scheduling of Semiconductor Test Facility via Petri Nets and Hybrid Heuristic Search. *IEEE Transactions on Semiconductor Manufacturing*, vol.11, no. 3, pp. 384-393.

Zhou, M., Jeng, M.D., 1998. Modeling, Analysis, Simulation, Scheduling and Control of Semiconductor Manufacturing Systems: A Petri Net Approach. *IEEE Transactions on Semiconductor Manufacturing,* vol.11, no. 3, pp.333-357.

Table 1 Match check to test information-enabling of a place

Source type	Destination type	Match check
Workstation	Transportation	$Info_{Orchestrator}(status_{SOURCE}.transfer) == Info_{Orchestrator}(usage_{DESTINATION}.located)$ $Info_{Service}(status_{SOURCE}.transfer) == Info_{ServicesFrom}(usage_{DESTINATION}.located)$ $Info_{ServicesTo}(status_{SOURCE}.transfer) \supset Info_{ServicesTo}(usage_{DESTINATION}.located)$
Transportation	Workstation	$Info_{Orchestrator}(status_{SOURCE}.transfer) == Info_{Orchestrator}(usage_{DESTINATION}.located)$ $Info_{ServicesTo}(status_{SOURCE}.transfer) == Info_{Service}(usage_{DESTINATION}.located)$ $Info_{ServicesFrom}(usage_{DESTINATION}.located) \supset Info_{ServicesFrom}(status_{SOURCE}.transfer)$
Transportation	Transportation	$Info_{Orchestrator}(status_{SOURCE}.transfer) == Info_{Orchestrator}(usage_{DESTINATION}.located)$ $Info_{ServicesFrom}(status_{SOURCE}.transfer) == Info_{ServicesFrom}(usage_{DESTINATION}.located)$ $Info_{ServicesTo}(status_{SOURCE}.transfer) == Info_{ServicesTo}(usage_{DESTINATION}.located)$
-	output_storage	$Info_{Orchestrator}(status_{SOURCE}.transfer) == Info_{Orchestrator}(usage_{DESTINATION}.located)$ $Info_{Service}(status_{SOURCE}.transfer) == Info_{Service}(usage_{DESTINATION}.located)$

Table 2 Scheduling search duration – in CPU time / seconds (static orchestrator mix)

			Run Number					
Number of Orchestrators in the (Static) Mix[1]	**Number of Places**	**Number of Transitions**	1	2	3	4	5	6
2	508	484	2	2	1	2	2	2
3	806	780	4	4	2	4	3	3
4	903	876	6	4	3	5	4	3
5	1201	1172	11	7	6	9	8	7
6	1298	1268	11	9	9	10	9	10
7	1596	1564	14	39	13	14	14	13
8	1693	1660	14	16	14	15	14	14
9	1991	1956	18	20	21	21	22	24
10	2088	2052	22	42	29	24	25	52
11	2386	2348	28	26	20	27	21	28
12	2483	2444	31	27	31	32	30	30
13	2781	2740	29	26	25	32	31	27
14	2878	2836	28	27	27	27	29	33
15	3176	3132	39	39	41	40	37	39
20	4063	4012	49	63	62	60	66	62
25	5151	5092	151	130	87	134	94	89

[1] Approx. 20 operations – on average - to be scheduled per orchestrator.

Table 3 Identification of feasible firable transition groups based on knowledge of conflicts in the entire flow model

Procedure SEARCH_FEASIBLE_FIRABLE_TRANSITION_GROUP ($T_{EN}^{(i)}$, C, searched_indexes)

Needed data structures:

- **T_{EN}** vector, contains the flat numbers of all enabled transitions in the TNCES flow model at one step
- **C** conflict matrix, automatically obtained from the structure of the flow model. C[i,j]=1 if there is a conflict between transitions t_i and t_j; otherwise C[i,j]=0
- **index** vector, contains the integer indexes of the elements of T_{EN} that are equal to 1
- **k** integer
- **seeds** vector of integers, the start flat number of a new search. A different start **seed** ensures a new transition group is selected each time a new SEARCH_FEASIBLE_FIRABLE_TRANSITION_GROUP search starts from the same initial T_{EN} vector. At the beginning of the search for a schedule **seeds** = Φ
- **searched_indexes** vector, stores the flat numbers of the transitions already investigated for conflicts; initially **searched_indexes** = Φ

```
do
        index:= find (T_EN^(i)==1);
        if i==0 then do
                select k ∉ seeds randomly from index;
                seeds:=[seeds, k]
                od.
        else do
                select k randomly from index; od. fi.
        T_EN^(i+1)=T_EN^(i) – C(k, :);
        searched_indexes := [searched_indexes; k]
        SEARCH_FEASIBLE_FIRABLE_TRANSITION_GROUP (T_EN^(i+1), C)
while searched_indexes ≠ find (T_EN^(i)==1) od.
```

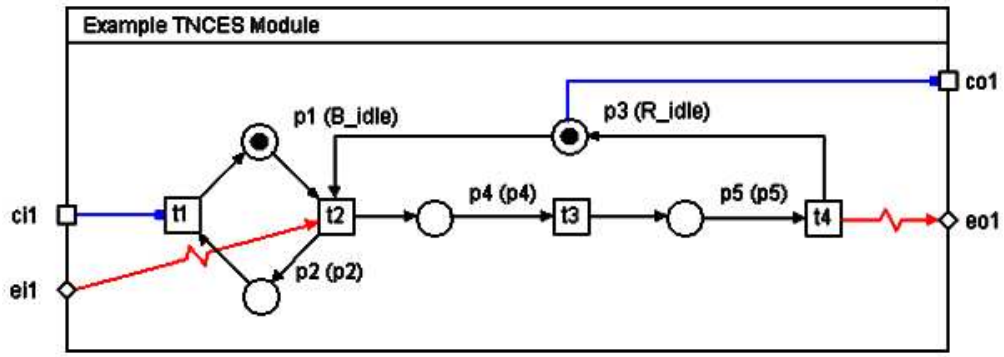

Figure 1. Example TNCES module
211x76mm (72 x 72 DPI)

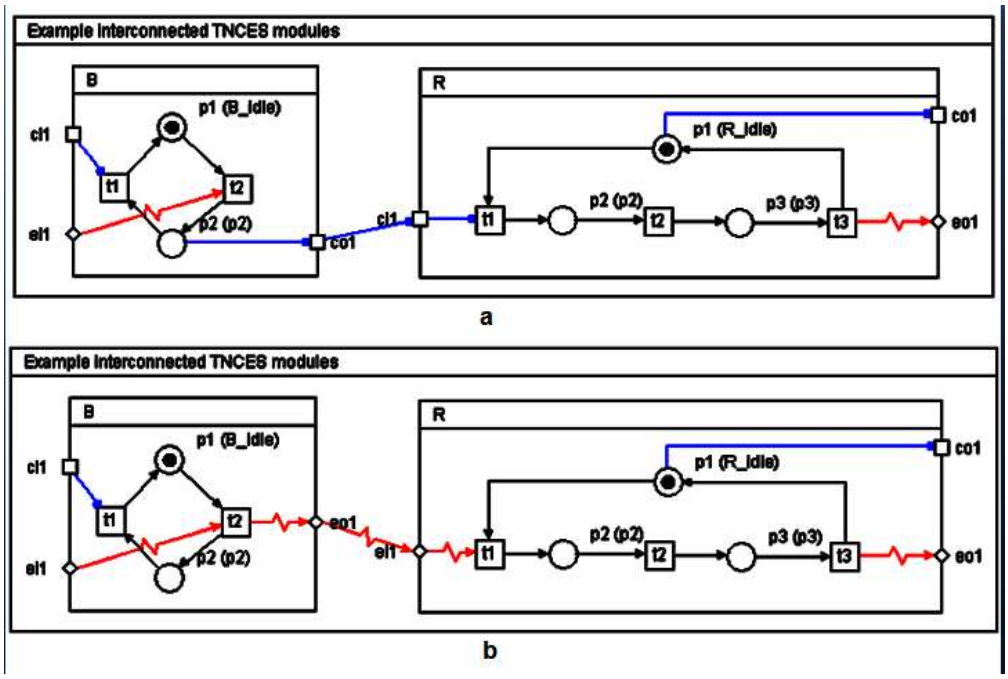

Figure 2. Example of condition arc (a) / event arc (b) interconnection between TNCES modules
255x171mm (72 x 72 DPI)

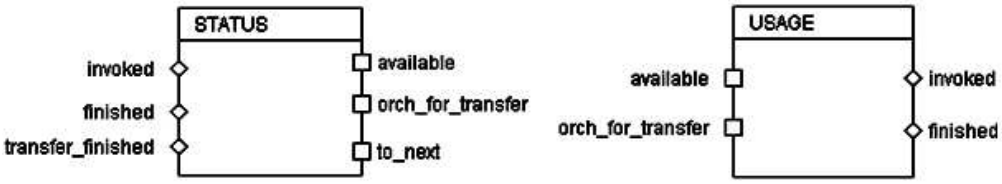

Figure 3. status and usage typed modules – interfaces
214x40mm (72 x 72 DPI)

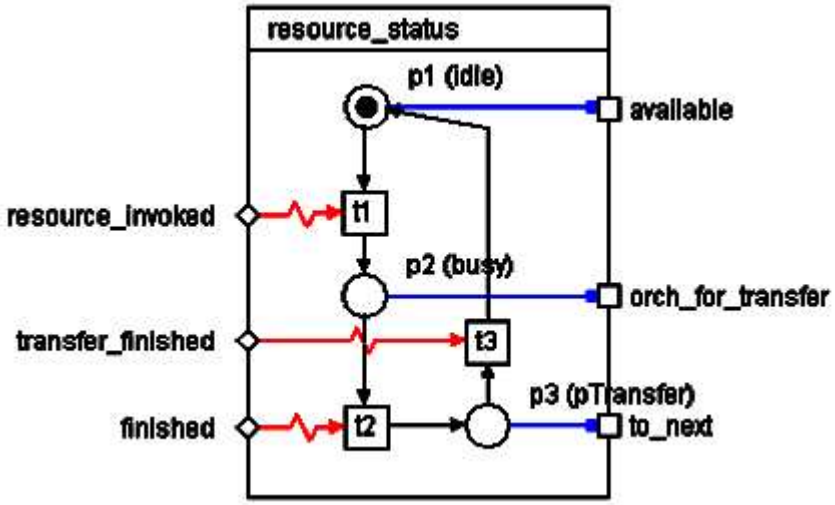

Figure 4. Representing processing workstations: TNCES module of type 'status'
147x89mm (72 x 72 DPI)

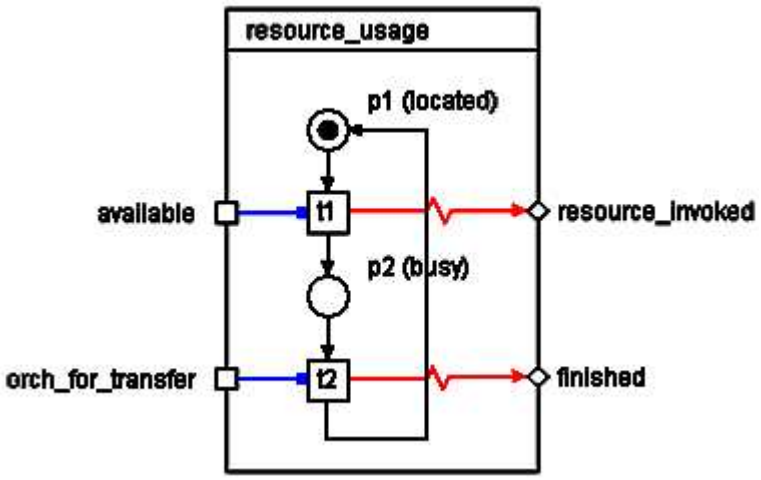

Figure 5. Representing processing workstations: TNCES module of type 'usage'
134x85mm (72 x 72 DPI)

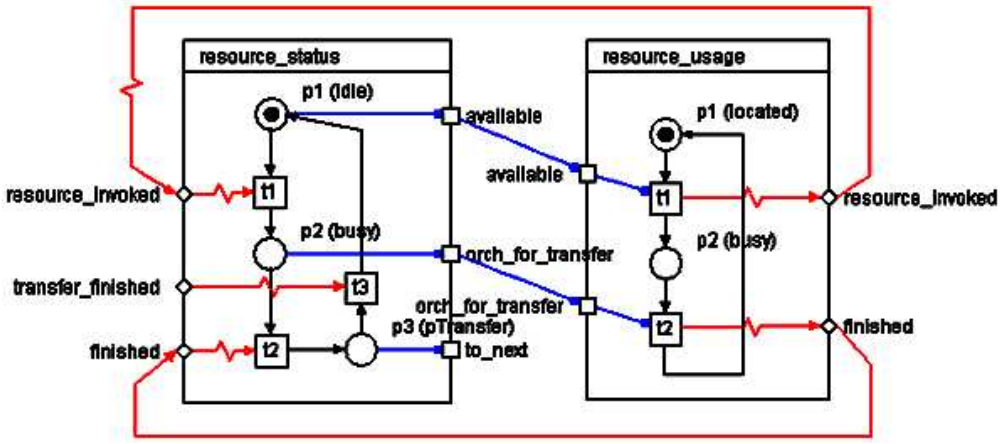

Figure 6. TNCES model of a resource located once
209x93mm (72 x 72 DPI)

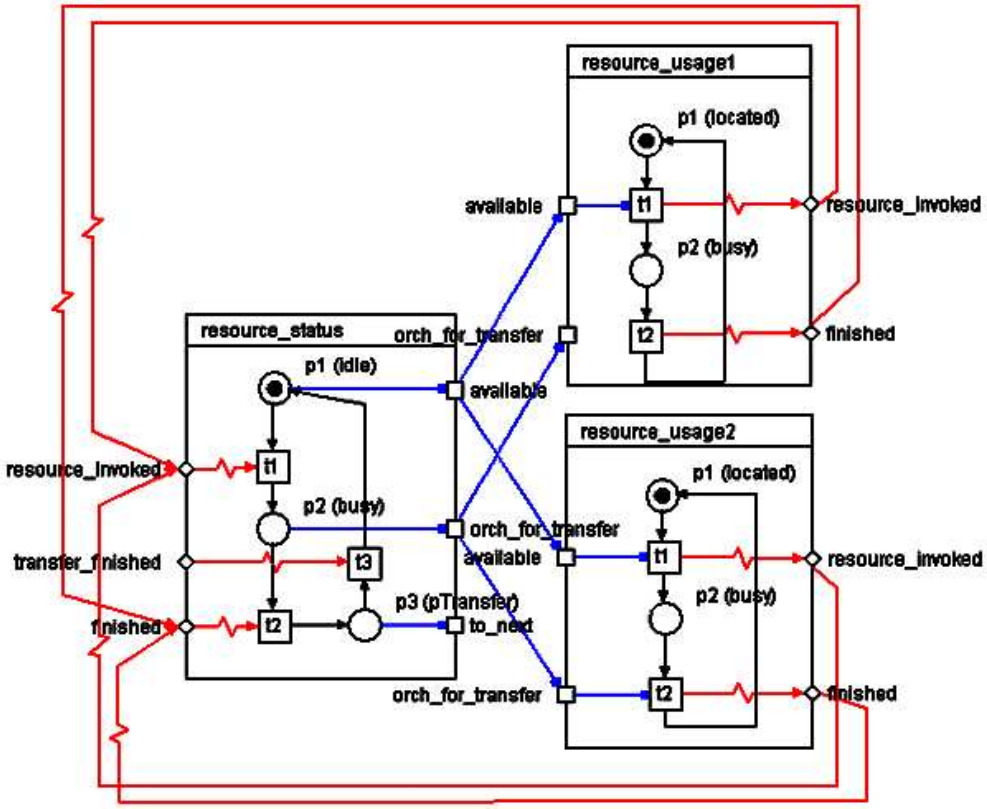

Figure 7. TNCES model of a resource located twice
213x173mm (72 x 72 DPI)

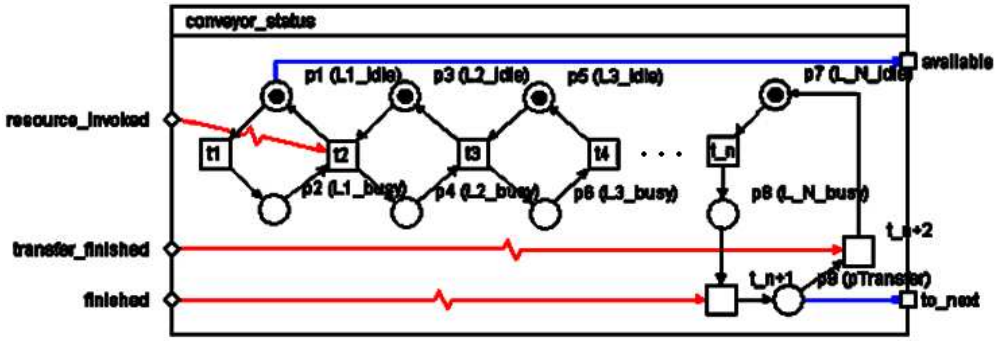

Figure 8. TNCES module of type 'conveyor_status' (N locations)
262x89mm (72 x 72 DPI)

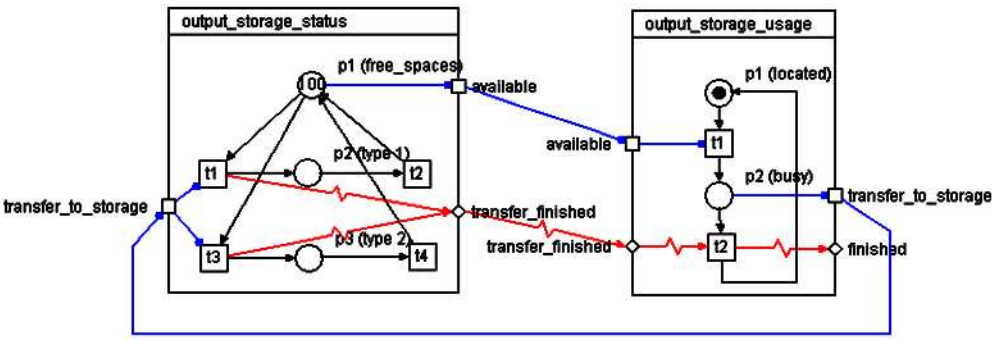

Figure 9. (Distinguishable) output buffer. Two part types.
275x92mm (72 x 72 DPI)

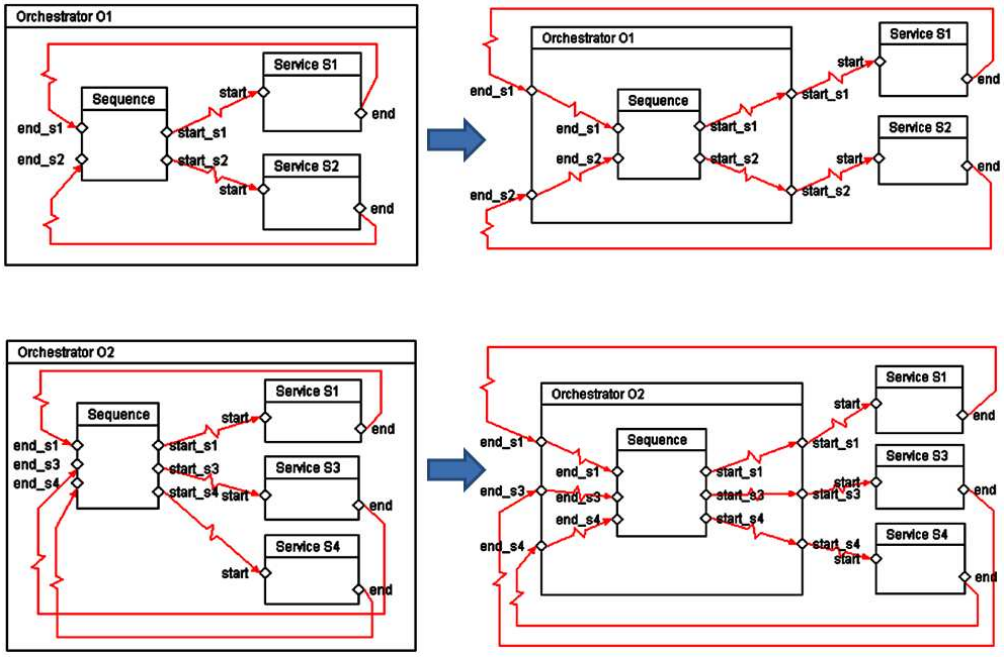

Figure 10. Standalone orchestrators. Separation of flow-related representation from atomic services representation
349x231mm (72 x 72 DPI)

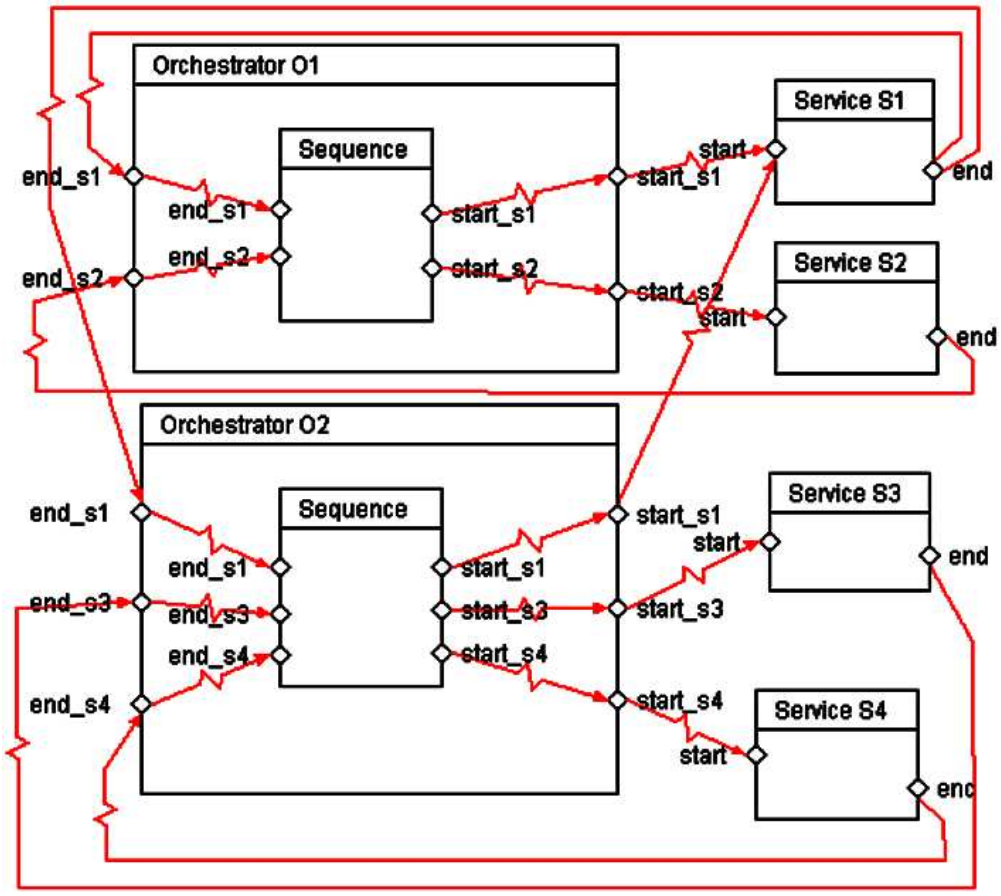

Figure 11. Orchestrator mix model for the two orchestrators of Figure 10
249x221mm (72 x 72 DPI)

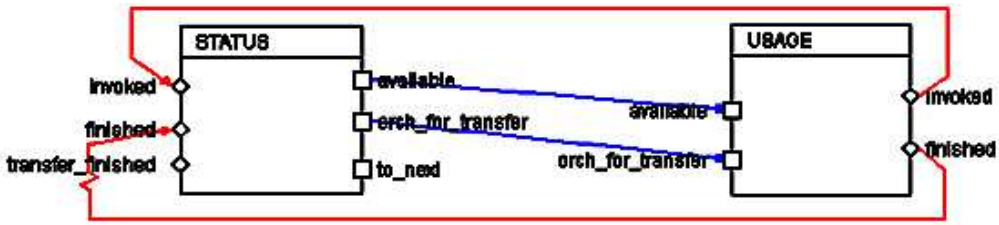

Figure 12. status-usage connections
194x44mm (72 x 72 DPI)

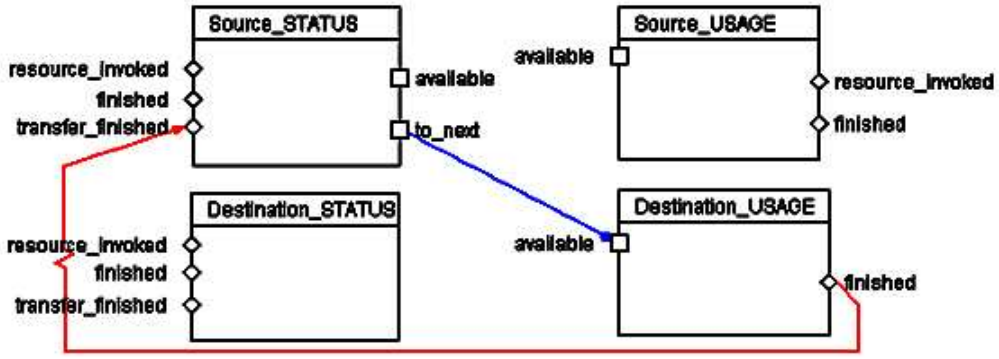

Figure 13. Resource-to-resource connections
196x74mm (72 x 72 DPI)

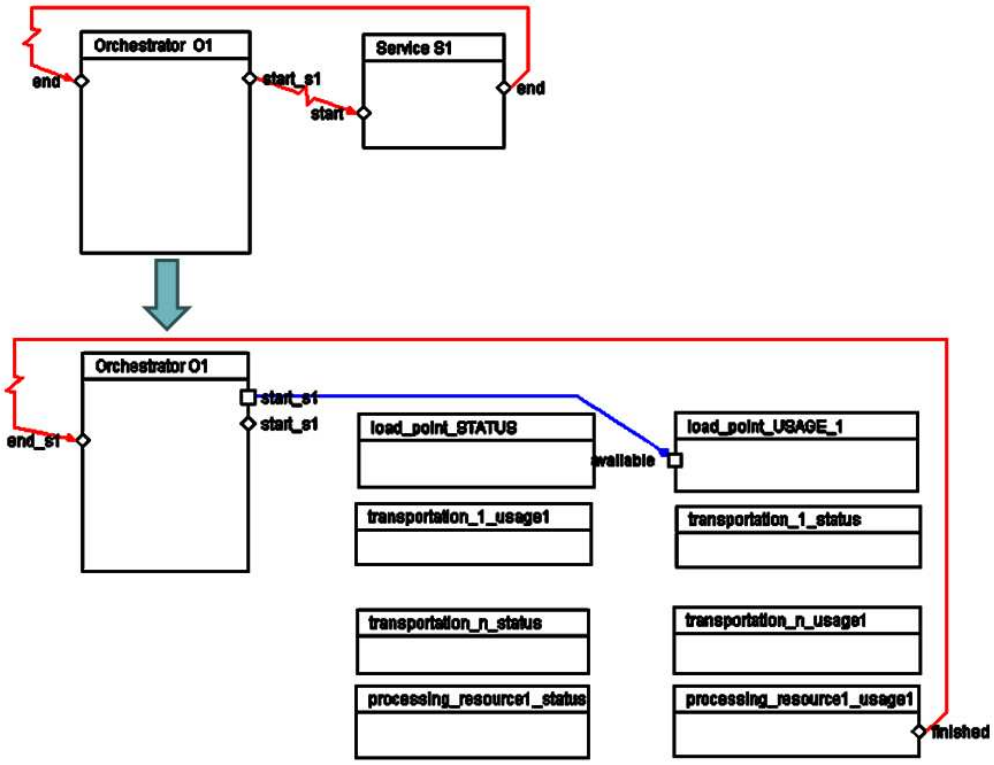

Figure 14. Replacing orchestrator-service connections with orchestrator-resources connections
306x238mm (72 x 72 DPI)

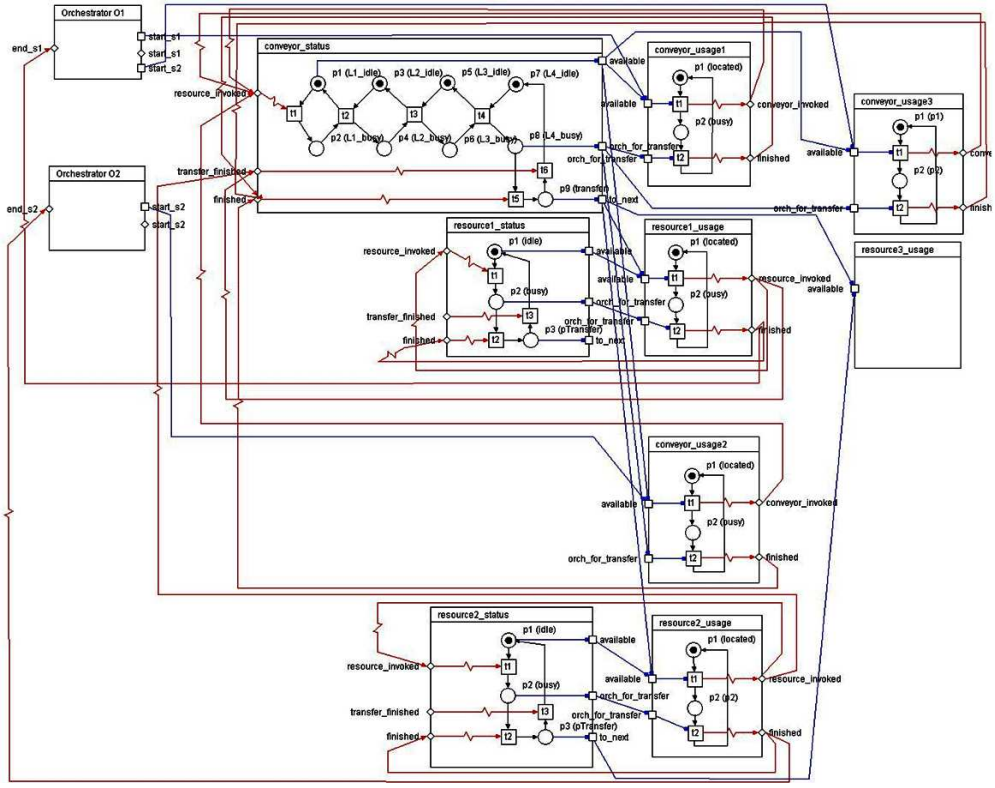

Figure 15. Two orchestrators use the same conveyor for transportation
412x322mm (72 x 72 DPI)

Figure 16. Line case study
451x254mm (72 x 72 DPI)

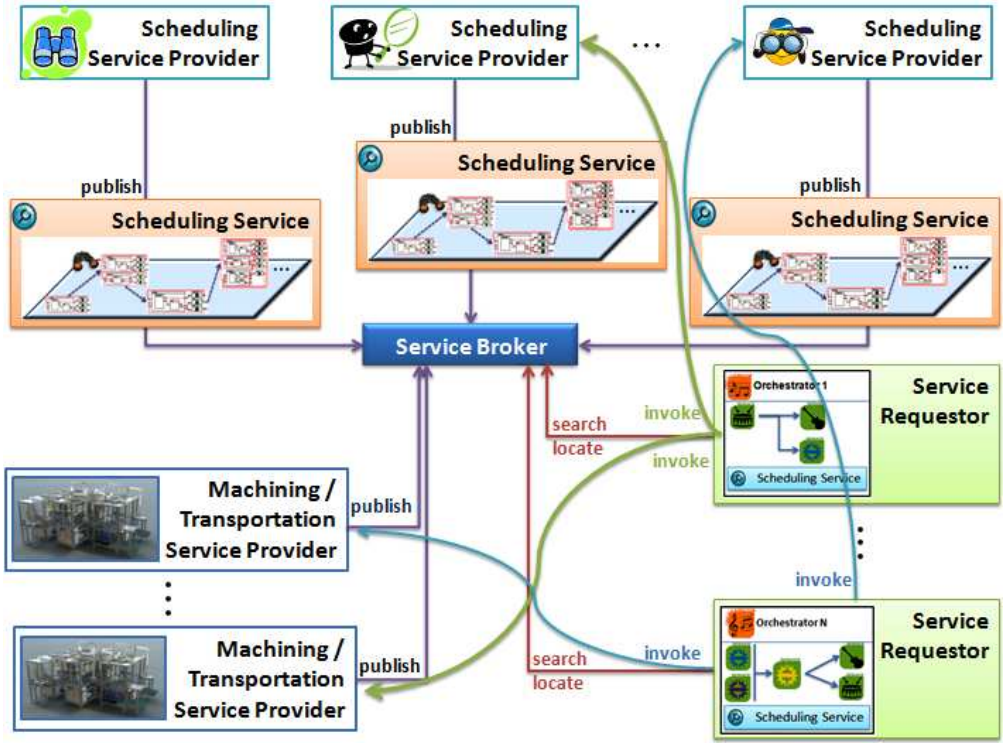

Figure 17. SOA-based implementation schema
249x185mm (72 x 72 DPI)

index(T_{EN})	...	39	...	43	...	57	...	59	...	69	...	71	Comment
$T_{EN}^{(0)}$	...	1	...	1	...	1	...	1	...	1	...	1	initial vector of enabled transitions
$T_{EN}^{(1)} = T_{EN}^{(0)} - C[39, :]$		**1**		0		0		**1**		0		**1**	t_{39} **possibly** firable together with t_{59} and t_{71}
$T_{EN}^{(2)} = T_{EN}^{(1)} - C[59, :]$		**1**		-1		0		**1**		0		0	t_{39} **is** firable together with t_{59}

Figure 18 Search firable feasible transition group. Example.
303x67mm (72 x 72 DPI)

Conflict Matrix (C)	...	39	...	43	...	57	...	59	...	69	...	71
⋮												
39		0		1		1		0		1		0
⋮												
43		1		0		0		1		0		0
⋮												
57		1		0		0		1		1		0
⋮												
59		0		1		0		0		0		1
⋮												
69		1		0		1		0		0		1
⋮												
71		0		1		0		1		1		0

Figure 19 Conflict Matrix: example
212x117mm (72 x 72 DPI)