Matrix-based Conflict Resolution Algorithm: Application to Path Selection in PN-based Scheduling Search

Corina Popescu, Jose L. Martinez Lastra

Abstract—This paper presents an algorithm that selects a group of independent events out of a given set of possibly conflicting actions. The procedure is matrix-based and was implemented in JAVA to guide backtracking scheduling search based on a Petri Net-derived model of flow.

The manufacturing systems particularly addressed in the implementation are using Web Services to implement the Service-Oriented-Architecture pattern. However, the conflict resolution method is applicable to more general settings, provided that conflicts are automatically detectable and expressible in matrix-form.

I. INTRODUCTION

Local goals of resources in a factory automation system are frequently in conflict. It is not possible that all individual interests are satisfied optimally. Widely adopted solutions to the conflict resolution problem are generally based on myopic negotiation mechanisms and heuristics.

This paper presents a matrix-based algorithm that was successfully utilized for conflict resolution to guide the scheduling search in a Petri Net-derived model of flow. Based on automatically obtainable information concerning conflicts, the resolution procedure chooses groups of firable transitions out of the set of all enabled transitions at one search step.

Although the manufacturing systems addressed here are using Web Services to implement the Service-Oriented-Architecture pattern, the proposed method is applicable to more general settings, provided that conflicts are automatically detectable and expressible in matrix-form.

The paper is organized as follows: Section II presents a short overview of conflict resolution strategies. Section III summarizes the background of this research. Details are given on the modeling procedure, the scheduling search and the hypotheses that serve as startpoint to automatically identify physical conflicts in the state space of the formal representation. Section IV presents the proposed conflict resolution algorithm. Section V draws the conclusions.

II. SHORT OVERVIEW OF CONFLICT RESOLUTION STRATEGIES

Conflict resolution strategies have been proposed in the past for automation systems and manufacturing scenarios. Zeng et al. [1] focus on the conflict resolution problem in workflows constrained by resources and non-determined duration. The proposed solutions include short-priority and key-activity strategies. Chun and Wong [2] propose a distributed negotiation algorithm extended from the A* search procedure to achieve dynamic and optimal scheduling of events. A Problem Analysis, Resolution and Ranking (PARR) problem resolution function to support the handling of complex traffic patterns in free flight operations is described in [3]. Mireles and Lewis [4] and Huang [11] focus on matrix/rule-based discrete event controller design and implementation. Similar to the work presented in [4], Lee et al. [6] proposed liveness-enforcing PN supervisors for semiconductor manufacturing systems. Fractal manufacturing systems (i.e. composed of sets of similar agents) are addressed in [7], and an agent-based negotiation mechanism is implemented to resolve conflicts that may appear during the process of goal formation. The work of [8] addresses the real-time dynamic control problem for FMS. The conflict problem is addressed in an on-the-fly manner by Jawahar et al. [9], who use heuristics such as shortest/longest operation/travel times to guide AGVs in case of disagreement or blockage. Finally, Camurri and colleagues [12] make a clear distinction between the decision support system and the execution of a Petri Net model of the system. The real time scheduling subsystem is based on particular heuristic sub-optimal multi-criteria algorithms. A general-purpose conflict strategy is utilized during execution in case tailored strategies (that depend on the problem class) fail at scheduling stage.

III. BACKGROUND OF THE RESEARCH

Service Oriented Architecture deployed by Web Services could make it possible for industrial players to respond in an agile manner to changes in production demand and to achieve fast enterprise integration.

Several research projects have focused on bringing the SOA paradigm to the production setting. Implementations of DPWS in embedded devices have been achieved in the ITEA-SIRENA [13], and ITEA-SODA [14] projects. The FP6-SOCRADES [15] project focused on tests of DPWS-enabled devices in pilot prototypes in the industrial domain.

In this work, formal models of flow are constructed automatically and used as a basis for scheduling search. The idea is to overcome the myopia problem traditionally associated with dispatching rules by combining heuristics with the use of a formal model that will mirror at all times the flow of pallets in a (part of) a manufacturing line.
A. Terminology

A process is a set of actions that modifies the state of the world.

A resource is a physical device (processing machine, transportation device, storage).

A service is an interface that encapsulates a process (an abstraction of how the process is seen in terms of inputs and outputs from the outside world). The details of how a service is executed are hidden from the outside world. A service can be dynamically published, located and invoked.

Resources (the service providers) publish/un-publish the service encapsulations of the processes they are capable/no longer capable of executing to a Service Broker. This allows rapid modifications in equipment.

Pallets are service requestors. Each pallet has knowledge of its service orchestration (or orchestrator): the order of execution (the flow) of its needs (i.e. of the services that should operate upon the raw material to obtain a final product). Each pallet searches and locates (using the Service Broker) the feasible service providers and subsequently invokes one of the found resources (i.e. the resource starts the execution of the sought process). The search/locate-invoke sequences are done for each needed service, in the order specified by the orchestrator of the pallet. This allows rapid modifications of flow.

An orchestrator (or service orchestration) refers to the desired flow (the order of execution) of services in a line, for a user.

B. Modeling

Service orchestration is represented through a modular, typed and composable Petri Net [16] derived formalism called Timed Net Condition Event Systems (TNCES) [17]. The translation procedure relies on a library of defined typed TNCES modules to represent atomic services, flow descriptors (e.g. Sequence, Any Order, Split Join ...), Boolean connectors and resource location and invocation. The TNCES model of an orchestrator is obtained through interconnecting automatically generated modules of this library. The global orchestration model is dynamically updated whenever new orchestrators are added to/exit the line. Fig. 1 illustrates the general concept.

Fig. 1 Incremental modeling of service orchestration: general approach.

Fig. 2 illustrates a simplified version of a flow mix model of two orchestrators, O1 and O2. The internal elements of the composing TNCES modules are detailed in [19] and are not shown. Orchestrator O1 specifies desired flow as a sequence of two services, S1 and S2. Orchestrator O2 specifies a needed sequence of services S1, S3 and S4. Service S1 may be performed by either resource5 or resource10. The shown TNCES models of the atomic services S2, S3 and S4 are not yet replaced with corresponding resource-typed modules.

C. Incremental model-based dynamic scheduling

The runtime constructed TNCES model of flow mix is input to search procedures to find feasible deadlock-free schedules. The net marking obtained after each update of the entire model is the initial marking of the new search.

The update of the flow mix model and its parameters is event-driven. The entire model changes as a result of machine failure or machine addition. Model parameters (e.g. time intervals) are updated in case of machine replacement.

The search for a schedule does not have to start all over again each time a new orchestrator is inputted to the line (schedules for new orchestrators can be searched to fit in the existing overall schedule). The model update preserves the current situation of the orchestrators already involved in the mix model, while adding a new module.

The goal marking of the search is automatically obtainable due to the typed and modular nature of the model TNCES blocks.

Both backtracking (BT) and best-first (BF) search [18] can be used to find feasible schedules based on the runtime-constructed TNCES models of flow. Backtracking maintains in storage a single path leading to the current marking, without considering optimality.
Therefore there is no need to needlessly increase the size of the model input to the search procedure: for each service requested by each orchestrator, it is enough to incorporate only one of the found resource possibilities in the model. The filtering of unnecessary paths in the state space of the complete flow model can thus already be assisted in the modeling stage. Complementary resource allocation policies can be used during model construction to guide a feasible modeling phase.

**Best-first search** examines before each decision the entire set of available alternatives (both newly generated and suspended in the past). Therefore, if optimality is desired, all resource possibilities must be included in the overall flow model to be input to the search procedure (as illustrated in Fig. 2).

### D. Conflicts in scheduling search space

To search for a feasible schedule, the marking update procedure must be driven by semantically-enriched firing rules associated with the connecting arcs of the flow model. Two main policies must direct state evolution:

1. First, unless it is a multiple location conveyor or a storage, a resource cannot perform two or more services simultaneously.

2. Second, a service cannot be performed on the same orchestrator by two different resources at the same time.

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.

### IV. 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 that are generated by the rules described in Section III.D 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.

Such conflicts are automatically detectable based on the structure of the TNCES flow model and information on the current marking. The conflict detection procedure is tailored to the modeling methodology and the formalism used for representation, and is not described as it is considered to be out of the scope of this paper.

Fig. 5 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. The conflict matrix $C$ (partially shown in Fig. 6) is automatically obtainable from the structure of the flow model. $C[i,j]=1$ if there is a conflict between transitions $t_i$ and $t_j$; otherwise $C[i,j]=0$.

To identify a group of transitions that may fire together, successive subtractions are made from $T_{EN}$. The steps documented in Fig. 5 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 $\text{searched_indexes}$ list.

2. The newly obtained $T_{EN}$ 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 $\text{searched_indexes}$ 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 $\text{searched_indexes}$ 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
index 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 I.

<table>
<thead>
<tr>
<th>Procedure SEARCH_FEASIBLE_FIRABLE_TRANSITION_GROUP ((TEN^0, C), searched_indexes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Needed data structures:</strong></td>
</tr>
<tr>
<td>- TEN vector, contains the flat numbers of all enabled transitions in the TNCES flow model at one step</td>
</tr>
<tr>
<td>- 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)</td>
</tr>
<tr>
<td>- index vector, contains the integer indexes of the elements of (TEN) that are equal to 1</td>
</tr>
<tr>
<td>- k integer</td>
</tr>
<tr>
<td>- 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 TEN vector. At the beginning of the search for a schedule seeds = (\Phi)</td>
</tr>
<tr>
<td>- searched_indexes vector, stores the flat numbers of the transitions already investigated for conflicts; initially searched_indexes = (\Phi)</td>
</tr>
</tbody>
</table>

```java
do
  index := find (TEN^0, C) = 1;
  if i=0 then do
    select k \(\notin\) seeds randomly from index;
    seeds := [seeds, k]
  od.
else do
  select k randomly from index; od. fi.

TEN^((i+1)) := TEN^0 - C[k, ];
searched_indexes := [searched_indexes; k]
SEARCH_FEASIBLE_FIRABLE_TRANSITION_GROUP (TEN^((i+1)), C)

while searched_indexes \(\neq\) find (TEN^0, C) od.
```

V. CONCLUSIONS

This paper presents a procedure to select feasible firable transition groups in a Petri Net-derived model of flow. The resolution procedure assumes that conflicts are automatically detectable at runtime, based on the structure of the model. The method was implemented in JAVA and utilized to search for feasible schedules on a Petri Net derived model of flow. The proposed approach can be used for general conflict resolution problems as alternative to negotiation-based mechanisms, provided identified conflicts are expressible in a matrix form.

REFERENCES