Recognizing finite repetitive scheduling patterns in manufacturing systems

RECOGNIZING FINITE REPETITIVE SCHEDULING PATTERNS IN MANUFACTURING SYSTEMS

Martijn Hendriks[1*], Barend van den Nieuwelaar[2†], Frits Vaandrager[1*]

[1]*Department of Computing Science, University of Nijmegen, The Netherlands*
martijnh@cs.kun.nl, fvaan@cs.kun.nl

[2]*Department of Mechanical Engineering,*
Eindhoven University of Technology, The Netherlands
N.J.M.v.d.Nieuwelaar@tue.nl

Abstract Optimization of timing behaviour of manufacturing systems can be regarded as a scheduling problem in which tasks model the various production processes. Typical for many manufacturing systems is that (collections of) tasks can be associated with manufacturing entities, which can be structured hierarchically. Execution of production processes for several instances of these entities results in nested finite repetitions, which blows up the size of the task graph that is needed for the specification of the scheduling problem, and, in an even worse way, the number of possible schedules. We present a subclass of UML activity diagrams which is generic for the number of repetitions, and therefore suitable for the compact specification of task graphs for these manufacturing systems. The approach to reduce the complexity of the scheduling problem exploits the repetitive patterns. It reduces the original problem to a problem containing the minimum amount of identical repetitions, and after scheduling of this much smaller problem the schedule is expanded to the original size. We demonstrate our technique on a real-life example from the semiconductor industry.

Keywords: Manufacturing system scheduling, finite repetitive behaviour.

*Supported by the European Community Project IST-2001-35304 (AMETIST)
†Part-time software architect at ASML.

1. Introduction

Scheduling in manufacturing systems has received much attention in the literature. The basic scheduling issue, the assignment of mutually exclusive resources to tasks, is addressed in the Job Shop Scheduling literature [15, 19]. In addition, in some cases also the order of tasks for a single resource influences temporal behaviour of the manufacturing system [3], analogous to the Traveling Salesman Problem [13]. Both of these problems are $\mathcal{NP}$ hard to solve. Combination of these optimization problems and the size and diversity of practical manufacturing cases makes scheduling of manufacturing systems an interesting challenge.

In [14], an overview of specific scheduling issues playing a role in manufacturing systems can be found. One of them is the fact that the same manufacturing processes have to be executed repetitively for several instances of manufacturing entities. Often, the relations between the manufacturing entities are hierarchical. Consider for example an assembly system. A final product in such a system, called an assembly, can consist of sub-assemblies, which in turn can consist of sub-sub-assemblies. Products are manufactured in batches and manufacturing orders consist of multiple batches. The relationships between the manufacturing entities of this system can be expressed by the Entity-Relationship Diagram (ERD) of figure 1.

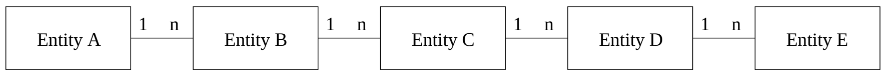

Figure 1: Hierarchical structure of manufacturing entities.

For the assembly system example, entities A through E can be associated with order, batch, product, sub-assembly, and sub-sub-assembly, respectively. Another example with hierarchical manufacturing entity relations concerns packaging. For instance, a manufacturing order of a beer brewery consists of several pallets, containing several crates with several bottles of beer. A third example concerns a wafer scanner manufacturing system from the semiconductor industry [1]. Wafers are also produced in batches (lots). A wafer scanner projects a mask on a wafer, using light. Eventually, the projected masks result in Integrated Circuits (ICs). On one wafer, multiple ICs and types of ICs are manufactured. Multiple types of ICs involve multiple masks, and multiple masks are placed on a reticle. The manufacturing entities can be modeled as in figure 1, where entity A through E can be associated with lot, wafer, reticle, mask, and IC, respectively. As this example concerns not only entities that end up in the product but also other entities required for manufacturing, this example is considered in the remainder of this paper.

Repetitive execution of manufacturing processes for several instances of manufacturing entities leads to finite repetitive patterns in manufacturing schedules. In practice, execution of the first few instances and last few instances of a manufacturing entity differ slightly from the rest. This is a large difference with unlimited repetitive behaviour of manufacturing systems, which has received much attention in literature [20, 11]. Furthermore, the hierarchical structure of the manufacturing entities leads to patterns on several granularity levels. The purpose of this paper is describe an approach to identify exactly identical repetitive scheduling patterns in order to reduce the complexity of the scheduling problem. With this information, a (sub-) optimized manufacturing schedule can be determined by concatenation of the optimized sub-schedules of the patterns. Without this information, combination of the possible sub-schedules for these recurrent patterns blows up the number of possible overall schedules dramatically.

Concretely, our contribution is twofold. First, we introduce a subclass of UML activity diagrams [10, 17] for the compact specification of task graphs which contain finite repetitive behaviour. The second part of our contribution consists of a method for finding repetitive subgraphs in these task graphs. This information can easily be exploited to speed up the scheduling process in a dramatic way. We show this by applying our technique to a real-life example from the semiconductor industry.

Related work. In [16] UML activity diagrams that specify scheduling problems are translated to timed automata models. Schedulability of the activity diagram is translated to a reachability property, which is checked by the model checker UPPAAL [12]. If the property is satisfied (the activity diagram is schedulable), then a trace that proves the property is equivalent to a schedule for the activity diagram. Although the explosion of the scheduling effort due to hierarchical, finite repetitions is recognized in [16], no solution is provided.

Related work w.r.t. the semantics of UML activity diagrams includes the verification of workflow models specified by these diagrams [5]. In contrast to our work, the semantics of [5] associates a transition system to each activity diagram, using some form of "token game". The transition system semantics of [5] can serve as a basis for verification using model checking but, unlike our task graph (partial order) semantics, it cannot be used as a starting point for solving scheduling problems. Another semantics for UML activity diagrams is provided by [7], using a straightforward translation to Petri nets. However, this semantics does not address the evaluation of conditionals, and as a result it is unclear how to extract a task graph in order to address scheduling issues.

Related work w.r.t. the second part includes computer-aided design of video processing algorithms [18]. Most video algorithms consist of repetitive executions of operations on data, which can be described by using nested loops and multidimensional arrays. The scheduling problem in this case is to minimize a particular cost function while satisfying certain timing, resource and precedence constraints. However, it seems that no exploitation of equality of loop instances takes place. Our work probably relates more to widening and acceleration techniques (e.g., [4] and [2, 8]), which try to accelerate the fixed-point computation of reachable sets. At least the approaches in the latter two use static analysis of the control graph (the syntax) to detect interesting cycles, of which the result of iterated execution can be computed by one single meta-transition. These meta-transitions are then added to the system and favored by the state space exploration algorithm, resulting in faster exploration of the state space. Our technique also exploits cyclic structures, yet not in the syntax, but in the semantics of the activity diagrams, to derive future behaviour. When this is done during the actual scheduling, it can be regarded as a form of acceleration. We are not aware of any other related work.

Outline. The paper is structured as follows. In section 2, activity graphs – a subclass of UML activity diagrams – are introduced, which are suited to model finite repetitive behaviour of manufacturing systems. Subsequently, we formally define the syntax of activity graphs, and we define the semantics by association of task graphs. Section 3 discusses an approach to recognize repetitive patterns in task graphs associated with activity graphs. Moreover, we show how this information can be used in a real-life, industrial, example. Finally, concluding remarks are presented in section 4.

2. Activity graphs

Task graphs are basic objects for the specification of scheduling problems. However, they are less suited for the specification of manufacturing systems with finite repetitive behaviour. Consider figure 1 and assume that we need to produce 5 units of each entity to be able to produce its parent entity. A task graph describing such a problem then consists of 5^5 sub graphs for the production of the needed quantity of entity E. In other words, the task graph may be exponentially large (or even worse!) in the number of different entities, which makes specification using a task graph and scheduling inconvenient.

In this section we introduce a subclass of UML activity diagrams for the compact specification of task graphs which contain limited repetitive behaviour [10, 17]. Activities can be associated with manufacturing processes. Limited repetition of (collections of) activities can be obtained using the so-called conditionals, that start the next repetition if the limit is not reached, and proceed if the limit has been reached.

2.1 Formal definition of activity graphs

Activity graphs are directed graphs with different types of vertices (a.k.a. nodes), which correspond to the types in UML activity diagrams, and with an annotation of the *conditional nodes*, which is used to specify finite repetitions of subgraphs[1].

Definition 1 (Activity graph) *An activity graph is defined by a tuple* $(N, n^0, \rightarrowtail, c)$*, where*

- N *is a finite set of* nodes*, partitioned into the sets* C, F, J, A, M *and* E *which are sets of* conditional*,* fork*,* join*,* activity*,* merge *and* exit *nodes respectively,*
- $n^0 \in F \cup A \cup M$ *is the* initial *node,*
- $\rightarrowtail \subseteq N \times N$ *is the set of* precedence edges *such that:*
  - *exit nodes have no successors, fork nodes have at least two successors, conditional nodes have two successors, and other nodes have one successor, and*
  - *join and merge nodes have at least one predecessor, and other nodes have one predecessor. The initial node is an exception since it may have no predecessors.*
- $c : C \to N \times \mathbb{N} \times N \times 2^C$ *is the* conditional function *such that: if* $c(v) = (v', n, v'', R)$*, then* $v \rightarrowtail v'$ *and* $v \rightarrowtail v''$*. We call* $v \rightarrowtail v'$ *the* true transition *of* v*,* n *the* upper bound *of* v*,* $v \rightarrowtail v''$ *the* false transition *of* v*, and* R *the* reset set *of* v*.*

We can explain the conditional function c as follows. Assume that $c(v) = (v', n, v'', R)$. This means that initially the true transition of v is enabled and the false transition of v is disabled. After n executions of the true transition it becomes disabled and the false transition becomes

[1]In this paper we do not need to perform the actual scheduling. Therefore, we omit the durations and resource requirements of the activities/tasks in all definitions.

enabled. The enabledness can be reset to the initial situation by taking a false transition of a conditional w such that v is in the reset set of w.

We use the regular UML conventions for the graphical representation of activity diagrams to represent our activity graphs. Summarizing, forks and joins are represented by bars, merges by diamond shapes with one outgoing arrow, activities by boxes with a name inside, exits by circled black dots, and conditionals by diamonds with two guarded outgoing edges. The initial node is preceded by a black dot. In our representation we use the conditionals as “counters” to keep track of the number of executions of the true edge of the conditional.

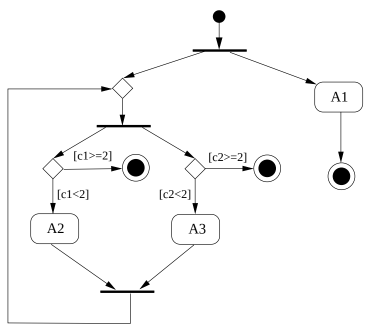

Figure 2: A small activity graph.

For instance, figure 2 depicts a small activity graph that has three activities and uses two conditionals, c1 and c2. There is one cycle, controlled by the conditionals, that is executed twice and in which activities A2 and A3 can run in parallel. Activity A1 must be run once, and this can happen in parallel with the cycle.

2.2 From activity graphs to task graphs

We define the semantics of an activity graph by unfolding it (which means resolving the conditional choices) to obtain all reachable instances of nodes and their precedence relation. For instance, figure 3 depicts the intended unfolding of the activity graph in figure 2.

Although the unfolding operation is intuitively quite clear, there is a snag in it. Consider, for instance, the activity graph of figure 4. We can unfold this graph in two ways, since we can choose when to reset c1. The first unfolding contains one instance of both activities whereas the second unfolding contains two instances of activity A1 and one instance of activity A2. This example shows us that an activity graph may contain race conditions in which two parallel branches of a fork use the same imaginary conditional counter, which results in a non unique unfolding.

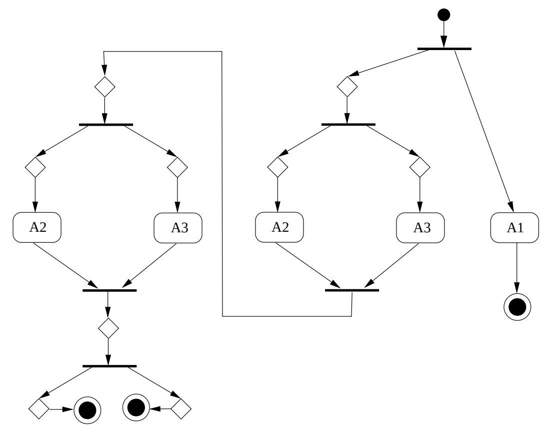

Figure 3: The intended unfolding of the activity graph of figure 2.

Such race conditions are undesirable and we want to restrict ourselves to a subclass of activity graphs that do not contain race conditions and which have a unique unfolding.

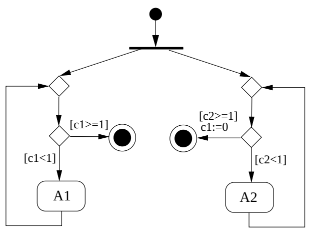

Figure 4: A non-deterministic activity graph.

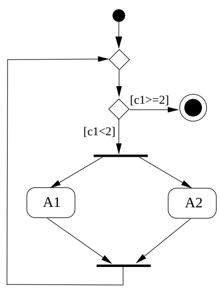

Figure 5: An activity graph which requires the use of additional counters.

In order to recognize the situation in which an activity graph can be unfolded in more than one way, we compute a distribution of the privileges of using the conditionals over the various parallel branches in the activity graph. We obtain such a distribution, a *relevancy mapping*, by assigning a set of relevant conditional nodes to each node in an activity graph. In order to exclude race conditions, we forbid parallelism of nodes that have overlapping relevancy sets. For instance, the conditional c1 is relevant for both parallel branches of the fork node in figure 4, which renders a unique unfolding impossible.

To avoid problems with, for instance, the unfolding of the activity graph in figure 5, we extend the range of the relevancy mapping to all nodes[2]. The rationale behind the formal definition of the relevancy mapping is as follows. First, we require that a node is relevant for itself, and the reset set of a conditional is relevant for that conditional. Second, we require that relevancy is passed on to neighboring nodes, except not forward through forks and not backward through joins (we consider these last two situations separately). This is necessary to give an inductive definition of the unfolding operation. Third, if a node is relevant for the successor of a fork, then it is also relevant for the fork. Moreover, the relevancy sets of any two different successors of a fork are disjoint. The first part is necessary for the inductive definition, and the second part is to avoid race conditions. Fourth, we require that the set of relevancy sets of the predecessors of a join is a partitioning of the relevancy set of the join, which, again, is necessary for the inductive definition. These four points are formalized as follows:

Definition 2 (Relevancy mapping) *A relevancy mapping for an activity graph* $(N, n^0, \rightarrowtail, c)$ *is a function* $X : N \rightarrow 2^N$ *such that:*

(i) If v *is a conditional node with* $c(v) = (v', n, v'', R)$*, then* $\{v\} \cup R \subseteq X(v)$*. Otherwise,* $v \in X(v)$*.*

(ii) If $v \rightarrowtail v'$*,* v *is not a fork node and* v' *is not a join node, then* $X(v) = X(v')$*.*

(iii) If v *is a fork node with successors* $v_1, ..., v_n$*, then* $\cup_{i=1}^{n} X(v_i) \subseteq X(v)$ *and* $X(v_i) \cap X(v_j) = \emptyset$ *for all* $1 \leq i \neq j \leq n$*.*

(iv) If v *is a join node with predecessors* $v_1, ..., v_n$*, then we require* $X(v) = \cup_{i=1}^{n} X(v_i)$ *and* $X(v_i) \cap X(v_j) = \emptyset$ *for all* $1 \leq i \neq j \leq n$*.*

We can show that the problem whether a general activity graph has a relevancy mapping is $\mathcal{NP}$-complete by a reduction from 3-SAT without negation and with exactly one true literal per clause [6]. However, for the more restricted class of activity graphs for which holds that every node is reachable from the initial node – which is not a limiting assumption – we cannot find a reduction, yet we also cannot find a polynomial algorithm.

[2]Assume that the nodes in a unfolding are tuples (v, γ), where v is a node and γ is a valuation of the conditionals which are relevant for that node ($\gamma(v)$ counts the number of executions of the true edge of v since its last reset). Now consider figure 5. When we construct the set of relevant conditionals for each node, we see that either T1 must be labeled with c1 or T2 must be labeled with c1 (otherwise we cannot give a clean inductive definition of the unfolding operation). If we assume that T1 is labeled with c1, and we unfold the activity graph, then we see that only *one* instance of T2 appears, namely $(\text{T2}, \emptyset)$, since T2 has an empty relevancy set. This, of course, is not what we expect from the unfolding.

In practice, one can easily translate the problem of finding a relevancy mapping for an activity graph to $|N|$ SAT problems in $|N|$ literals (where N is the set of nodes), which can then be solved by state of the art algorithms [9]. We can also give an ad hoc algorithm for finding a relevancy mapping or showing that such a mapping does not exist. Our algorithm makes $\mathcal{O}(|N| \cdot d^n)$ calls to a balance function which is polynomial in the size of the activity graph. In this formula d is the maximal amount of predecessors of a join node and n is the total number of join nodes. Experimental research should be able to clarify which of the two approaches is the best.

In order to define the semantics of an activity graph, we define Γ_N for an activity graph with nodes N as the set of partial functions with type $N \hookrightarrow \mathbb{N}$. We call a $\gamma \in \Gamma_N$ a node valuation and we use the following abbreviations: $\gamma[v := v + 1]$ maps every node not equal to v to the same value as γ, and it maps v, if it is defined by γ, to the value $\gamma(v) + 1$. Similarly, $\gamma[R := 0]$ agrees with γ on the value of every node not in R and it maps every node in R, if it is defined by γ, to zero. If $\gamma, \gamma' \in \Gamma_N$ and they both are defined for disjoint sets of nodes, then we let $\gamma \cup \gamma'$ denote the node valuation that is defined for the union of these counter sets according to γ and γ'. Finally, if $\gamma \in \Gamma_N$ and S is a subset of counters, then we let $[\gamma]_S$ denote the partial counter valuation that is obtained by projecting γ to S.

For simplicity we make two assumptions about our activity graphs: (1) a conditional node is not immediately followed by a join node, and (2) a fork node is not immediately followed by a join node. Note that we can easily eliminate these constructions in an activity graph by adding “dummy merges” with only one predecessor. Therefore, these assumptions can be made without loss of generality, yet they make the following definition much shorter.

Definition 3 (Unfolding) *Let $\mathcal{A} = (N, n^0, \rightarrowtail, c)$ be an activity graph (N is partitioned in the usual way) with relevancy mapping X. The unfolding of $\mathcal{A}$ is a directed graph $(V, \mapsto)$, where $V \subseteq N \times \Gamma_N$ is the set of node instances, and $\mapsto \subseteq V \times V$ is a set of directed edges, inductively defined as follows:*

(i) The base clause is: $\{(n^0, \gamma^0)\} \in V$, where $\gamma^0(v) = 0$ if $v \in X(n^0)$ and it is undefined otherwise.

(ii) The inductive clauses are[3]*:*

$$\frac{(v,\gamma) \in V \quad v \in J \cup M \cup A \quad v \rightarrowtail v' \quad v' \notin J}{(v',\gamma') \in V \quad (v,\gamma) \mapsto (v',\gamma') \quad \textit{where } \gamma' = \gamma[v := v+1]} \tag{1}$$

$$\frac{(v,\gamma) \in V \quad v \in F \quad v \rightarrowtail v_1, \cdots, v \rightarrowtail v_n}{(v_i,\gamma_i) \in V \quad (v,\gamma) \mapsto (v_i,\gamma_i) \quad \textit{where } \gamma_i = [\gamma]_{X(v_i)}[v := v+1]} \tag{2}$$

$$\frac{\begin{array}{c}(v_1,\gamma_1), \cdots, (v_n,\gamma_n) \in V \quad v_1 \rightarrowtail v, \cdots, v_n \rightarrowtail v \\ \gamma_1(v_1) = \cdots = \gamma_n(v_n) \quad v \in J\end{array}}{(v,\gamma) \in V \quad (v_i,\gamma_i) \mapsto (v,\gamma) \quad \textit{where } \gamma = \cup_{i=1}^{n} \gamma_i[v_i := v_i + 1]} \tag{3}$$

$$\frac{(v,\gamma) \in V \quad v \in C \quad c(v) = (v', n, v'', R) \quad \gamma(v) < n}{(v',\gamma') \in V \quad (v,\gamma) \mapsto (v',\gamma') \quad \textit{where } \gamma' = \gamma[v := v+1]} \tag{4}$$

$$\frac{(v,\gamma) \in V \quad v \in C \quad c(v) = (v', n, v'', R) \quad \gamma(v) \geq n}{(v'',\gamma'') \in V \quad (v,\gamma) \mapsto (v'',\gamma'') \quad \textit{where } \gamma'' = \gamma[R := 0]} \tag{5}$$

With rules (2) and (3) we consider all successors or predecessors.

The inductive definition of the unfolding of an activity graph has a unique solution. For instance, the unfolding of the activity graph in figure 2 indeed is the one in figure 3 (we omitted the node valuations, since that unnecessarily complicates the picture).

An activity graph for which holds that its unfolding has no "loose ends" (which are non exit instances with no successors) is called *well-defined*. For instance, the unfolding in figure 3 is well-defined. However, if we construct the unfolding for the activity graph in figure 2 in which we have replaced the upper bound of conditional c1 with 3, then we see that the third instance of activity A2 is a loose end, since there will be only two instances of activity A3.

Note that the unfolding of a well-defined activity graph is a new activity graph in which all conditionals have been replaced by merges. Compare, for instance, the activity graph in figure 2 and its unfolding in figure 3.

Let a and b be two instances in some unfolding. If there is a path from a to b, then we denote this by $a \mapsto^* b$. A path consisting of at

[3]Essentially this is a parameterized definition. Thus, for each activity graph, we can find a finite set of inductive clauses, which are "instances" of the parameterized clauses, that are used for the construction of the unfolding of that particular activity graph.

least one edge is denoted by $a \mapsto^+ b$. We define parallelity of instances as follows:

$$(v,\gamma) \parallel (v',\gamma') \quad \iff \quad ((v,\gamma) \not\mapsto^* (v',\gamma') \land (v',\gamma') \not\mapsto^* (v,\gamma))$$

We say that an instance a is non-parallel if there is no instance b such that $a \parallel b$. We now informally state some useful properties of unfoldings. (For the formal statement and proof of these items we refer to the appendix.)

- Many different relevancy mappings may exist for an activity graph, but they all lead to essentially the same unfolding (lemma 1).
- Race conditions do not appear in activity graphs which have a relevancy mapping. Thus, node instances which use the same nodes are not parallel (lemma 3).
- The instances in a well-defined unfolding satisfy the same requirements on the number of successors and predecessors as their nodes in the activity graph, except that merge instances have one predecessor and conditional instances have one successor (lemmas 4 and 5).
- An unfolding is acyclic, but it might be infinite (lemma 6).
- We state a sufficient syntactical condition on activity graphs that ensures finiteness of the unfolding. We can check this condition in time polynomial in the size of the activity graph (lemma 7).

By "stripping away" the control structure instances (all instances but those of activity nodes) in an unfolding we obtain the task graph. Hence, we call instances of activity nodes tasks.

Definition 4 *Let $(V, \mapsto)$ be the unfolding of activity graph $\mathcal{A}$. The task graph of $\mathcal{A}$ is the tuple $(T, \rightarrow)$, where*

- $T = \{ (v,\gamma) \in V \mid v$ *is an activity node*$\}$, *and*
- $\rightarrow \subseteq T \times T$ *defined as:* $a \rightarrow b$ *iff*
  - $a \mapsto^+ b$ *without passing through other activity instances, and*
  - *no* $c \in T \setminus \{b\}$ *exists such that* $a \mapsto^+ c$ *without passing through other activity instances and* $c \mapsto^+ b$.

The fact that there are no cycles in an unfolding implies that the task graph is acyclic too. For instance, figure 6 depicts the task graph of the unfolding in figure 3. (Again, we did not depict the node valuations of the instances.)

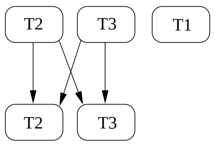

Figure 6: The task graph of the unfolding in figure 3.

3. An approach to exploit repetitive structures in activity graphs

In the previous section we have used activity graphs for the specification of scheduling problems for manufacturing systems with a finite repetitive control structure. The semantics of these activity graphs is defined in terms of unfoldings, which on their turn define (possibly infinite) task graphs.

Apart from the known $\mathcal{NP}$-hardness of the task graph scheduling problem, we also face a possible blow-up in size of the task graph due to nested cycles. This makes the approach in which we straightforwardly unfold the activity graph and feed it to an regular scheduler infeasible. Instead, we need to determine the repetitive structures in an activity graph to be able to exploit these during scheduling of the task graph.

Our approach consists of three steps. First, we lower the upper bounds of conditionals to values as small as possible, which means that we are just able to recognize repetitive structures in the condensed activity graph. Second, we compute the task graph of the condensed graph, and use regular scheduling tools to find a solution for this relatively small problem. Third, using repetitive structures and the schedule, we construct a schedule for the original, generally much larger, activity graph.

3.1 Formalizing our approach

In this section we formalize our three step plan introduced above for scheduling activity graphs. The first step involves decreasing the upper bounds of the conditionals that control the repetitions in the activity graph such that they are minimal w.r.t. to detecting the repetitions. At this moment, our approach for finding the minimal activity graph is as follows:

- The activity graph has been constructed with a clear view of what it should mean. Therefore, it is known which conditionals (or sets of conditionals) specify the repetitions of the manufacturing process. The first step is to set the upper bounds of all these conditionals to the value such that at least one regular instance of the manufacturing entity is present. E.g., all leading manufacturing entities (that differ slightly from normal ones) are present, plus a single regular entity. If all entities are the same, then the upper bound is set to one.

- Increment the lower bounds of conditionals that control a single repetition, until all (sets of) conditionals are "extendable" (below, we formally explain what extendability means). It seems that the order of this search process can be arbitrary.

Next, we explain what extendability of a set of conditionals means. First, we define what we exactly mean with the increment (and decrement of upper bounds).

Definition 5 ((G, n)-extension) *Let $\mathcal{A}$ be an activity graph, let G be a subset of conditionals of $\mathcal{A}$, and let $n \in \mathbb{N}$. We define the (G, n)-extension of $\mathcal{A}$, denoted by $\mathcal{E}(\mathcal{A}, G, n)$, as the activity graph in which the upper bounds of the conditionals in G have been incremented with n.*

A relevancy mapping for an activity graph is also a relevancy mapping for any extension of that activity graph. In the general case, however, the unfolding of such an extension does not need to be well-defined, as we already have sketched in the previous section. Next, we define what we exactly mean with a "repetitive structure" in an activity graph.

Definition 6 (Repetitive structure) *Let $\mathcal{A}$ be an activity graph and let $\mathcal{A}' = (N', \rightarrowtail')$ be a subgraph of $\mathcal{A}$. We call $\mathcal{A}'$ a* repetitive structure *of $\mathcal{A}$ iff there exists more than one* isomorphic embedding *of $\mathcal{A}'$ into the unfolding of $\mathcal{A}$, denoted by $(V', \mapsto')$, i.e., an injective function $i : N' \to V'$ satisfying*

- $i(v) = (v', \gamma') \Rightarrow v = v'$
- $v \rightarrowtail v' \Leftrightarrow i(v) \mapsto i(v')$

This definition carries easily over to task graphs. We think that it is quite useful to be able to point out the repetitive patterns in an unfolding (or task graph), since such information can be exploited during scheduling, e.g., by copy-pasting the sub schedules of the repetitive patterns to obtain a schedule for an extension of the activity graph.

Note that repetitions in an unfolding appear due to a cycle in the activity graph. The scope of this paper is finite repetitive behaviour, which implies that the cycles in the activity graph are controlled by conditionals. Therefore, we try to grasp repetitive structures using subsets of conditionals.

Let $\mathcal{A}$ be a well-defined activity graph with a finite unfolding $(V, \mapsto)$ and let G be a subset of conditionals of that graph. The algorithm depicted in figure 7 computes a pair of sets, $\mathbb{R}$ and $\mathbb{B}$, called the cyclic structure of G (we use $succ(R)$ to denote the set of direct successors of instances in set R).

```
(1)   $i := 0$
(2)   $R_0 := \{(n^0, \gamma^0)\}$
(3)   while $(R_i \neq \emptyset)$
(4)           $B_i :=$ trans closure of $succ(R_i)$ under $\mapsto \setminus \{ ((v,\gamma),(v',\gamma')) \mid v \in G \}$
(6)           $R_{i+1} := \{ (v,\gamma) \in B_i \mid v \in G \}$
(7)           $B_i := B_i \setminus R_{i+1}$
(8)           $i := i + 1$
(9)   od
(10)  return $(\mathbb{R}, \mathbb{B})$ where $\mathbb{R} = \{R_1, ..., R_{i-1}\}$ and $\mathbb{B} = \{B_1, ..., B_{i-1}\}$
```

Figure 7: Computation of the cyclic structure of the set of conditionals G.

The while loop eventually terminates since we assumed that the unfolding is finite, and we have proven that it is acyclic. Also note that R_i only contains instances of conditionals in G and all edges leading outside B_i lead to R_{i+1}. Figure 8 gives a graphical representation of a cyclic structure where all pairs R_i and R_j are disjoint.

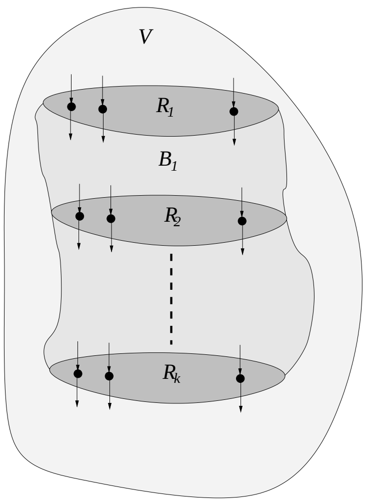

Figure 8: The cyclic structure associated with the set of conditionals G.

We can define Γ_i as the union of all valuations of the instances in R_i projected to the domain of the conditionals of the activity graph. If all instances in R_i are pairwise parallel, then Γ_i is a partial function, since lemma 3 tells us that the relevancy mapping must then assign disjoint sets to the conditional nodes of R_i. Next, we define the situation in which we are able to recognize repetitive structures.

DEFINITION 7 (EXTENDABILITY) *Let G be a subset of conditionals of an activity graph $\mathcal{A}$ and let $\mathbb{R} = \{R_1, ..., R_k\}$ and $\mathbb{B} = \{B_1, ..., B_k\}$ be the associated cyclic structure. We call G extendable, if*

- *the set $\mathbb{R}$ is a partitioning of the set of all instances of conditionals in G, such that $|R_i| = |G|$,*
- *if $(v, \gamma) \mapsto (v', \gamma')$ and $(v', \gamma') \in B_i$, then $(v, \gamma) \in B_i \cup R_i$,*
- *the outgoing edges of R_i either are all true edges or all false edges, and*
- *if R_i exits with true edges and R_{i+1} exits with false edges, then $\Gamma_{i+1} = \Gamma_i[G := G + 1]$. We call the set $R_i \cup B_i$ a* repetitive set*.*

This definition concludes the first step of our approach. We now can decide which condensed activity graph is the smallest such that the conditionals which specify repetitions are still extendable. This means, as we see during the third step, that we can detect repetitive patterns for all sets of extendable conditionals.

The second step of our approach consists of using regular scheduling tools to find a (optimal) solution for the condensed activity graph that we have found in step one. The actual scheduling falls outside the scope of this paper. Therefore, we just assume that we can obtain such a solution.

The third step consists of using repetitive structures of the condensed activity graph of step one and the schedule of step two to construct a schedule for the original activity graph. The next lemma formalizes our thought that the unfolding (task graph) of the condensed activity graph is just large enough to contain subgraphs which can be copy-pasted to obtain the unfolding (task graph) of the original problem.

LEMMA 8 *If an activity graph $\mathcal{A}$ is extendable for G, then $\mathcal{E}(\mathcal{A}, G, n)$ is well-defined for any $n \in \mathbb{N}$. Moreover, the subgraphs of $\mathcal{A}$ associated with the repetitive sets are repetitive structures of the extension.*

PROOF. Let $\mathbb{R} = \{R_1, ..., R_k\}$ and $\mathbb{B} = \{B_1, ..., B_k\}$ be the cyclic structure of G for an activity graph A. Let $\{i_1, ..., i_m\}$ be a set of indices of the repetitive sets. We assume that the indices are strictly increasing, that is $j < k \Rightarrow i_j < i_k$.

The first item of definition 7 says that any set R_i contains exactly one instance of every conditional node in G. Since $\mathbb{R}$ is a partitioning, every instance of a conditional node in G is included in some R_i. Moreover, the instances in R_i are pairwise parallel, since otherwise $\mathbb{R}$ would not be a partitioning, and therefore we know that every Γ_i is a partial function for all R_i.

Now consider the (G, n)-extension of $\mathcal{A}$, denoted by $\mathcal{A}'$. The relevancy mapping for $\mathcal{A}$ also is a relevancy mapping for $\mathcal{A}'$ and we use that relevancy mapping to construct the unfolding of $\mathcal{A}'$.

Since only the upper bounds of the conditionals in G are increased, we can use exactly the same inductive clauses from definition 3 to construct the unfolding up to the conditionals in the set R_{i_1+1}. Now instead of taking the false edges of the conditionals, the true edges must be taken, since the upper bounds of all counters in G have been increased with n. Since we required that $\Gamma_{i_1+1} = \Gamma_{i_1}[G := G + 1]$, we know that in this situation exactly the same conditional edges are enabled as from the conditionals in R_{i_1}. Moreover, the second item of definition 7 tells us that the structure $R_{i_1} \cup B_{i_1}$ is independent from the rest of the unfolding (especially it contains no join instances that have a predecessor that is not in $R_{i_1} \cup B_{i_1}$). Finally, the third item tells us that *every* instance of a conditional in G takes its true edge from R_{i_1} and its false edge from R_{i_1+1}. Therefore, we can say that the subgraph $R_{i_1} \cup B_{i_1}$ exactly defines the *last* execution of the cycle that is controlled by the set of conditionals G. In other words, we can apply the same inductive clauses that we used to show that $R_{i_1} \cup B_{i_1}$ is part of the unfolding to show that a "copy" of this sets also is part of the unfolding. Thus, we can copy $R_{i_1} \cup B_{i_1}$ n times before we proceed with the set R_{i_1+1}.

If we do this copy-pasting for all the indices $\{i_1, ..., i_m\}$, then we have extended all conditional instances, since by the first item of definition 7 we know that there are no conditionals in G that are not in some R-set. The resulting unfolding is the unfolding of the (G, n)-extension of A. Moreover, it is well-defined since $\mathcal{A}$ is well-defined and the second item of the definition ensures that copy-pasting introduces no loose ends.

Finally, it is clear that the subgraphs of A associated with the sets $R_{i_j} \cup B_{i_j}$ are repetitive structures, since we have shown that repetitive patters in the (G, n)-extension of $\mathcal{A}$ appear by applying the copy-past method sketched above to these sets. ■

The previous lemma only covers the extension of a single set of conditionals, whereas in general we need the extension of several sets of conditionals. The next definition covers hierarchy (or nesting) between cyclic structures of different sets of conditionals, which is needed for such a parallel extension.

Definition 9 (Hierarchy) *Let $\mathcal{A}$ be an activity graph and let G and G' be a disjoint sets of conditionals of $\mathcal{A}$, which both are extendable for $\mathcal{A}$. We say that $G \prec G'$ iff for all repetitive sets $R_i \cup B_i$ of G we can find a B'_j in the $\mathbb{B}$-set of G' such that $R_i \cup B_i \subset B'_j$.*

The next lemma states that we can extend an activity graph for two sets of conditionals, if they are hierarchical. (Note that we can easily generalize this lemma to an arbitrary number of hierarchical sets of conditionals.)

Lemma 10 *If an activity graph $\mathcal{A}$ is extendable for G and for G' and $G \prec G'$, then $\mathcal{E}(\mathcal{E}(\mathcal{A}, G, n), G', n')$ is well-defined for any $n, n' \in \mathbb{N}$. Moreover, the subgraphs of $\mathcal{A}$ associated with the repetitive sets of G and G' are repetitive structures of the extension.*

Proof (sketch). The idea is to construct the extension of $\mathcal{A}$ inside out. This means that we first apply the method sketched in the constructive proof of lemma 8 to G, and then to G'.

By definition 9 we can find a B'_j in the $\mathbb{B}$-set of the cyclic structure of G' such that $R_i \cup B_i \in B'_j$ for every repetitive set $R_i \cup B_i$ of G. Thus, we copy-paste the set $R_i \cup B_i$ n times and also add the new instances to the set B'_j. In the proof of lemma 8 we have sketched that this works out fine. Moreover, it does not disturb the validity of the cyclic structure of G', that is, G' is extendable for the extension $\mathcal{E}(\mathcal{A}, G, n)$ and the tuple $(\mathbb{R}, \mathbb{B}')$, where $\mathbb{B}'$ has been updated as described above, is the associated cyclic structure. ■

3.2 Example application of our approach

In this subsection we apply our approach to part of a scheduling problem from a wafer scanner. This example covers three out of five manufacturing entities that were discussed in the introduction: reticles, masks, and ICs.

- The number of reticles involved is 15 in this case. The conditional set that can be associated with this $G_C = \{$c0, c1$\}$. Obviously, the upper bounds of these conditionals equal 15.
- The number of masks involved is 8. The conditional set that can be associated with this $G_D = \{$c2, c3$\}$. For the first mask of

every reticle, some additional activities must be executed, which is controlled by conditionals c4, c5, c6, and c8.

- The number of ICs involved is 43, which is controlled by conditional set $G_E = \{c7\}$. Therefore, the total amount of ICs in the specified schedule will be $15 \times 8 \times 43 = 5160$.

It is clear that the task graph associated with this activity graph is quite large and not easily schedulable. We have implemented the theory in this paper using JAVA, and we can show in a few minutes that the task graph consists of 11655 tasks (construction of the relevancy mapping takes 85 calls to the balance function).

We try to apply the technique explained in the previous section. The first step consists of finding smallest upper bounds of the conditionals that specify the repetitions for the recticles, masks and ICs. We start by setting the upper bounds of G_C and G_E to 1, since every repetition is equal. However, we set the upper bounds of G_D to 2, since the first image of every reticle differs from the rest. Next, we check the extendability of G_C, G_D and G_E for this condensed graph which we call $\mathcal{A}_0$: they are all extendable. Moreover, $G_E \prec G_D \prec G_C$ as expected, which enables the "parallel" extension of the conditional sets (see lemma 10). Computation of the cyclic structures and checking the extendability takes – for this particular example – fractions of a second using our tool.

For the second step we use a regular scheduling tool to find an optimal schedule for the condensed activity graph. One such a optimal schedule is shown in figure 10 as a Gantt chart.

In the third step we use the repetitive structures of G_C, G_D and G_E to construct a schedule for the original activity graph. These repetitive structures are given during computation of the extendability of the sets of conditionals (see definition 7 and lemma 8). Furthermore, during the scheduling of the activity graph in step 2, tasks that share resources are put in a certain order to satisfy the mutual exclusion property of resources that plays a role in this kind of scheduling problems. This corresponds to adding additional precedences to the activity graph, such that the task order is forced to be the same as in the schedule.

For our example, step 3 of our approach can be described using figure 10 as follows. The repetitive structure that can be associated with G_C is the entire Gantt chart. The tasks in the time interval [b, d] are associated with G_D. Finally the tasks in the intervals [a, b] and [c, d] are associated with G_E. According to lemma 10 we need to construct the schedule for the original problem from inside out. First, we increase the upper bound of the conditional in G_E. Therefore, we copy-paste the interval [a, b] 42 times, and then we copy-paste the second interval also 42 times. Next,

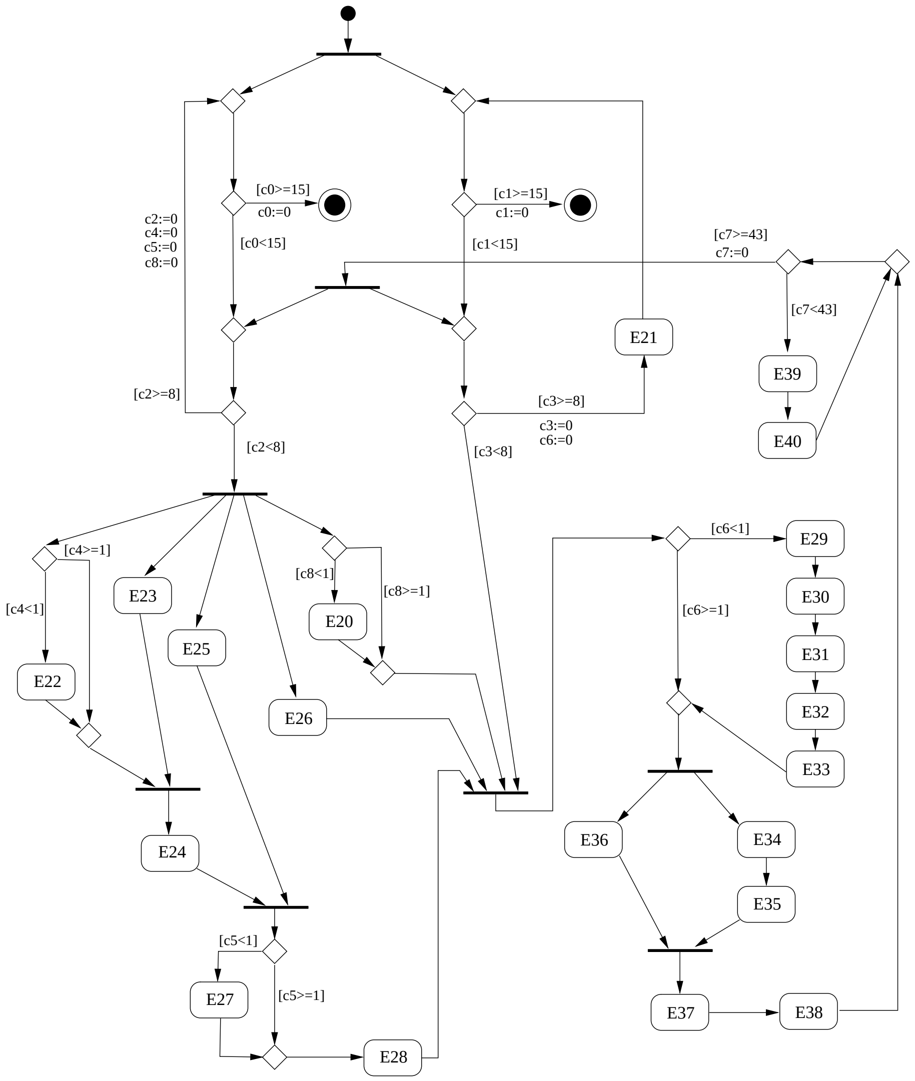

Figure 9: An activity graph which specifies part of a real-life scheduling problem for a wafer scanner.

we proceed with copy-pasting the interval that can be associated with G_D 6 times to increase the upper bounds of conditionals in G_D to their original values. Finally, we copy-past all tasks in the updated Gantt chart 14 times to increase the upper bound of the conditionals in G_C. Note that this copy-pasting does not concern a time interval, but a sub graph of the task graph. The tasks on resource 7 and 9 that are shown at the left of figure 10 will succeed the task that ends at d, and the

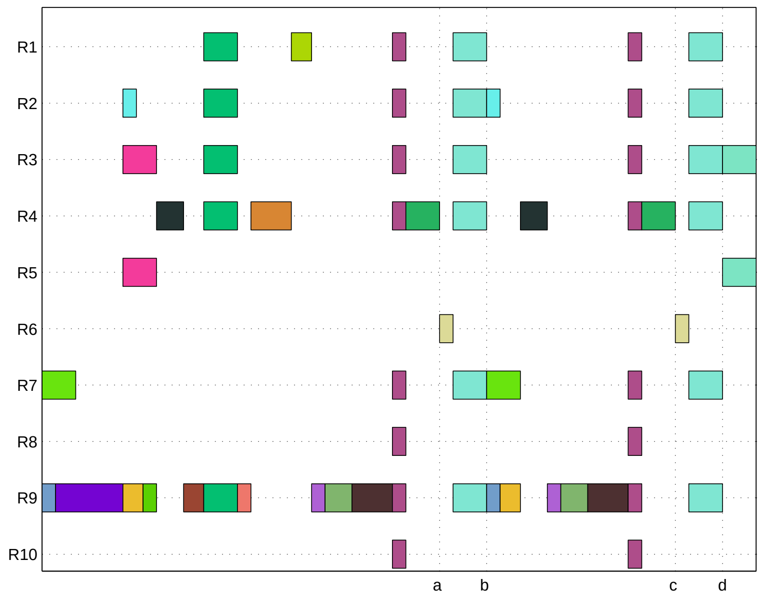

Figure 10: An optimal schedule for the condensed activity graph.

precedences admit that it is executed in parallel with the task starting at d on resource 3 and 5.

It is clear that our method only involves the scheduling of the relatively small task graph of the condensed problem. This renders it in many case much more suitable than the straightforward approach of scheduling the original, very large, task graph. It is necessary to quantify the (sub) optimality of the generated schedule, and we regard this as an important subject for future work.

4. Conclusions

The idea of this work is to reduce the complexity of scheduling problems being faced in many manufacturing systems by exploitation of the repetitive patterns that can be recognized in them. The task graph that usually forms a basis for description of a scheduling problem is extended with additional modeling features to describe this finite repetitive behaviour. This extended model follows the UML activity diagram standard, and is called an activity graph. In fact, an activity graph is a folded-in equivalent of a task graph with repetitive patterns, which is generic for the number of pattern repetitions. The activity graph is formally defined, and so is its equivalence with a task graph. An important issue is the unambiguousness of the activity graph, which is proven statically by construction of a ”relevancy mapping”.

The expressivity of the activity graphs is sufficient for a lot of practical cases from industry. It is possible to model parallelity of different instances of one manufacturing entity by introduction of multiple conditionals controlling execution of activities that can run in parallel for these different instances in the system. As a consequence of the fact that conditionals are not hierarchical, i.e., a conditional that can be associated with a lower level is not a child of a conditional of a higher level, it is not possible to describe a system in which manufacturing entities can “overtake” each other. This means that processing order must be first in, first out, which is fine for most practical cases. Extension of activity graphs for hierarchical conditionals could be considered for future work. The same goes for the ad-hoc algorithm to determine a relevancy mapping, which seems to be acceptable for practical cases, but a polynomial one would be preferable.

The approach for reduction of the complexity of the scheduling problems exploits the hierarchical manufacturing entity structure that results in nested patterns in the schedule. First, the scheduling problem is reduced with respect to the number of repetitions. Subsequently, the reduced problem in the form of an activity graph is converted to the usual form based on a task graph and can be scheduled using appropriate tooling. Finally, the schedule of the reduced problem is extended up to the size of the original problem using repetitive structures. This extension algorithm is in general much more efficient than scheduling the original task graph. We believe that preservation of (make span) optimality is ensured for a subset of problems that can be described. This means that for cases in which instances of manufacturing entities are processed sequentially, recurrent TSP-alike problems are recognized and therefore are to be scheduled only once while preserving optimality. This issue is an important subject for future research.

References

[1] Available through URL `http://www.asml.com/`.

[2] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *6th International Conference on Computer Aided Verification*, number 808 in LNCS, pages 55–67. Springer–Verlag, 1994.

[3] J.B.M. Melissen C.M.H. Kuijpers, C.A.J. Hurkens. Fast movement strategies for a step-and-scan wafer stepper. *Statistica Neerlandica*, 51(1):55–71, 1997.

[4] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of

Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[5] R. Eshuis and R. Wieringa. Verification support for workflow design with uml activity graphs. In *Proceedings of International Conference on Software Engineering*, 2002.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability. A guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[7] T. Gehrke, U. Goltz, and H. Wehrheim. The dynamic models of UML: Towards a semantics and its application in the development process. In *Hildesheimer Informatik-Bericht 11/98*. Institut für Informatik, Universität Hildesheim, 1998.

[8] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In E. Asarin, O. Maler, and S. Yovine, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, April 2002.

[9] E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.

[10] G. Booch J. Rumbaugh, I. Jacobson. *The Unified Modeling Language Reference Manuals.* Addison-Wesley, 1999.

[11] T.G. de Kok J.I. van Zante-de Fokkert. The simultaneous determination of the assignment of items to resources, the cycle times, and the reorder intervals in repetitive pcb assembly. *Annals of Operations Research*, 92:381–401, 1999.

[12] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, pages 134–152, 1998.

[13] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys (eds.). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York, 1985.

[14] J.E. Rooda N.J.M. van den Nieuwelaar, J.M. van de Mortel-Fronczak. Design of supervisory machine control. Submitted to European Control Conference 2003.

[15] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, 1995.

[16] S. Roels. Applicability of model-checking methods to scheduling in machines. Master's thesis, University of Nijmegen, The Netherlands, October 2002.

[17] OMG Unified Modeling Language Specification – version 1.4, September 2001. Available through URL `http://www.omg.org/uml/`.

[18] W. F. J. Verhaegh. *Multidimensional Periodic Scheduling.* PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.

[19] M. Wennink. *Algorithmic Support for Automated Planning Boards.* PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.

[20] X. Xinhe X. Puquan, L. Changyou. On sequencing problems of repetitive production systems. *IEEE Transactions on Automatic Control*, 38(7), 1993.

Appendix: Formal proofs

LEMMA 1 *If X and X' are relevancy mappings for an activity graph, then the two resulting unfoldings are bisimilar.*

PROOF. Consider the two resulting unfoldings $(V, \mapsto)$ and $(V', \mapsto')$. Bisimilarity in our case means that a relation $R \subseteq V \times V'$ exists such that:

- the two initial instances are related, and
- $((v, \gamma_1), (v', \gamma_1')) \in R$ and $(v, \gamma_1) \mapsto (w, \gamma_2)$ implies that there exists a $(w, \gamma_2') \in V'$ such that $(v', \gamma_1') \mapsto' (w, \gamma_2')$ and $((w, \gamma_2), (w, \gamma_2')) \in R$. (The reverse implication is covered by the symmetric definition).

We define R as follows: $((v, \gamma), (v', \gamma')) \in R$ if and only if $v = v'$ and γ and γ' agree on the intersection of their domain.

Since the node valuation of the initial instance maps a subset of nodes to 0 by definition, we see that the initial instances are related by R indeed.

Now let us assume that $((v, \gamma_1), (v', \gamma_1')) \in R$ and $(v, \gamma_1) \mapsto (w, \gamma_2)$. By definition of our relation, we know that $v = v'$. Moreover, γ_1 and γ_1' agree on the intersection of their domains (thus, $q \in X(v) \cap X'(v')$ implies that $\gamma_1(q) = \gamma_1'(q)$). We now consider the 5 possibilities for existence of the edge $(v, \gamma_1) \mapsto (w, \gamma_2)$ (these are the 5 different inductive rules of definition 3):

(1) We prove that the edge $(v', \gamma_1') \mapsto' (w, \gamma_2')$, where $\gamma_2' = \gamma_1'[v' := v' + 1]$, exists, by using *exactly the same* inductive clause as for $(v, \gamma_1) \mapsto (w, \gamma_2)$. We thus can derive that $\gamma_2 = \gamma_1[v := v+1]$ and $\gamma_2' = \gamma_1'[v := v+1]$. By definition of the relevancy mapping, the domain of γ_2 equals the domain of γ_1 and the domain of γ_2' equals the domain of γ_1'. Moreover, since $((v, \gamma_1), (v', \gamma_1')) \in R$, γ_1 and γ_1' agree on the intersection of their domain. Therefore, γ_2 and γ_2' also agree on the intersection of their domain, which proves that $((w, \gamma_2), (w, \gamma_2')) \in R$.

(2) The argument for this rule is similar to the argument of (1), except, that the domain of γ_2 is a subdomain of γ_1 and the domain of γ_2' is a subdomain of γ_1'. However, this does not change the validity of the argument.

(4) The argument for this rules is similar as in (1).

(5) The argument for this rules is similar as in (1).

(3) In this case, (w, γ_2) is a join instance. Since it exists, we know that all its predecessors exist (one of them is (v, γ_1)). We prove with induction on the number of inductive clauses of type (3) that is needed to prove that these predecessors are in V, that we can find a related $(w', \gamma_2') \in V'$.

The base of the induction is formed by 0 inductive clauses of type (3). Thus, for every predecessor of (w, γ_2) , say (p^i, γ^i) we can prove with the inductive clauses (1), (2), (4) and (5) that it is in V. Moreover, in the items above we argued that we can find related instances, say $(p^{i'}, \gamma^{i'})$ in V', by using exactly the same inductive clauses. Without loss of generality we can say that $p^i = p^{i'}$, and therefore γ^i and $\gamma^{i'}$ agree on the intersection of their domain.

By definition of inductive rule (3) we know that $\gamma^i(p^i) = \gamma^j(p^j)$. Since $p^i = p^{i'} \in X'(p^{i'})$, we know that $\gamma^{i'}(p^{i'}) = \gamma^{j'}(p^{j'})$. Therefore we can add the edge $(v', \gamma_1') \mapsto (w, \gamma_2')$, where $(v', \gamma_1') = (p^{i'}, \gamma^{i'})$ for some i. So, this γ_2' is the union of all (partially incremented, see definition 3) predecessor valuations. However, all these predecessor valuations $\gamma^{i'}$ agree on their domains with their counterparts γ^i that are used to construct the valuation γ_2. Therefore, γ_2 and γ_2' also agree on their domains, which proves that $((w, \gamma_2), (w, \gamma_2')) \in R$.

The proof of the induction step is equal to the proof of the base case.

■

The next lemma is used as a basis for some other lemmas.

LEMMA 2 *For every instance (v, γ) in an unfolding $(V, \mapsto)$ such that $q \in X(v)$ a path $(n^0, \gamma^0) \mapsto^* (v, \gamma)$ exists of which all nodes are labeled with q by X.*

PROOF. We prove the lemma by induction on the number of aplications of the base and inductive clauses of definition 3 that is needed to show that $(v, \gamma) \in V$. The base of the induction is formed by one application of a clause. Since the first application must be the base clause, we can only show that $(n^0, \gamma^0) \in V$. Therefore, $(v, \gamma) = (n^0, \gamma^0)$, which clearly proves that we can find the path.

Now let us assume that we can find the desired path for all instances for which we can show that they belong to V with n applications of the base and inductive clauses. Let us consider an instance (v', γ') for which we can show in $n+1$ applications of the base and inductive clauses that it belongs to V. Then obviously $(v, \gamma) \mapsto (v', \gamma')$ and we can show with n applications of the base and inductive clauses that (v, γ) belongs to V. We now distinguish two cases:

- The edge $(v, \gamma) \mapsto (v', \gamma')$ is added by inductive clause 1,2,4, or 5. We can conclude that v' is not a join, and by definition 2 that $q \in X(v')$ implies that $q \in X(v)$. By applying the induction hypothesis we can conclude that the desired path exists.
- The edge is added by inductive clause 3. Then we know that $(v_1, \gamma_1) \mapsto (v', \gamma')$, ..., $(v_n, \gamma_n) \mapsto (v', \gamma')$ such that $(v, \gamma) = (v_i, \gamma_i)$ for some $1 \leq i \leq n$. Moreover, we can show with n applications of the base and inductive clauses that (v_i, γ_i) belongs to V for all $1 \leq i \leq n$. By definition 2 we know that $q \in X(v')$ implies that $q \in X(v_i)$ for some i. Thus, by applying the induction hypothesis we can conclude that the desired path exists.

■

The next lemma ensures that race conditions do not appear in activity graphs which have a relevancy mapping.

LEMMA 3 *If (v, γ) and (v', γ') are two different instances and $X(v) \cap X(v') \neq \emptyset$, then $(v, \gamma) \not\parallel (v', \gamma')$.*

PROOF. First, we choose one particular $w \in X(v) \cap X(v')$. Then, using lemma 2 we can say that two paths that are completely labeled with w exist:

$$(n^0, \gamma^0) \mapsto (u_1, \gamma_1) \mapsto (u_2, \gamma_2) \mapsto ... \mapsto (u_m, \gamma_m) \mapsto (v, \gamma)$$

$$(n^0, \gamma^0) \mapsto (u'_1, \gamma'_1) \mapsto (u'_2, \gamma'_2) \mapsto ... \mapsto (u'_n, \gamma'_n) \mapsto (v', \gamma')$$

We now assume that $m < n$ (we fix the other cases later). We prove that $(u_i, \gamma_i) = (u'_i, \gamma'_i)$ for all $1 \leq i \leq m$ by an inductive argument:

- Base: $i = 1$. The initial instance either is not a fork, or is a fork. In the former case, its successor is fixed, and clearly $(u_m, \gamma_m) = (u'_n, \gamma'_n)$. In the latter case, both paths take the same successor node due to the disjunction part of requirement (iii) on the relevancy mapping X and the fact that $w \in X(u_1) \wedge w \in X(u'_1)$ (by construction of the paths). Therefore, the equivalence holds.

- Induction: assume that it holds for $i = k$. We need to distinguish three cases, namely (1) u_k and u'_k are forks, (2) they are conditionals, or (3) they have an other type. We can prove the equivalence for the first and third case with a similar argument as in the previous item. As for the second case, we know that $\gamma_k = \gamma'_k$ and therefore they choose the same successor: $u_{k+1} = u'_{k+1}$ and also $\gamma_{k+1} = \gamma'_{k+1}$.

Thus, we know that $(u_m, \gamma_m) = (u'_m, \gamma'_m)$. We can use the same argument as in the inductive proof above to show that $(v, \gamma) = (u'_{m+1}, \gamma'_{m+1})$ and therefore $(v, \gamma) \mapsto^+ (v', \gamma')$. With a similar argument we can prove that if $m > n$, then $(v', \gamma') \mapsto^+ (v, \gamma)$. However, if $m = n$, then $(v, \gamma) = (v', \gamma')$, which cannot occur since we assumed that they are different. ■

The next two lemmas state properties of the number of predecessors and successors of the instances in the unfolding.

Lemma 4 *Instances of an unfolding satisfy the same restrictions as their counterparts in the activity graph w.r.t. to the number of predecessors except that merge instances have exactly one predecessor.*

Proof. Let us consider an instance (v, γ) in the unfolding. We distinguish three cases. First, v is a activity, conditional, exit or fork node. First observe that this instance has at least one predecessor, since otherwise it would not be in the unfolding. Now assume that $(v', \gamma') \mapsto (v, \gamma)$ and $(v'', \gamma'') \mapsto (v, \gamma)$ and $(v', \gamma') \neq (v'', \gamma'')$. According to definition 1 $v' = v''$ and therefore, the domain of γ' equals the domain of γ''. Moreover, $\gamma = \gamma'[v' = v' + 1]$ and $\gamma = \gamma''[v'' = v'' + 1] = \gamma''[v' = v' + 1]$. Therefore, we may conclude that $\gamma' = \gamma''$ and thus, $(v', \gamma') = (v'', \gamma'')$ which is a contradiction.

Second, v is a join node. According to definition 3 there is only one inductive clause to show that a join node is part of the unfolding, namely rule (3). So, if we can show that a join instance, say $(v, \gamma) \in V$, then it has at least the correct number of predecessors, say $(v_1, \gamma_1), ..., (v_n, \gamma_n)$. Now suppose that another application of the inductive rule (3) adds the edge $(v', \gamma') \mapsto (v, \gamma)$ (this is the only way to add edges to join instances). Then, $v' = v_i$ for some $1 \leq i \leq n$ by definition. Moreover, γ' must be equal to γ_i, since otherwise (v', γ') cannot lead to (v, γ). Therefore, v has the desired number of predecessors.

Finally, we prove that an instance of a merge node, say (v, γ), has exactly one predecessor. By definition 3 it has at least one predecessor. If we assume that it has more than one predecessor, then we can pick two of them, say (v_1, γ_1) and (v_2, γ_2). By requirement (ii) of definition 2 we know that $v \in X(v_1)$ and $v \in X(v_2)$. In lemma 3 we can read that a path

$$(n^0, \gamma^0) \mapsto^* (v_1, \gamma_1) \mapsto (v'_1, \gamma'_1) \mapsto^* (v_2, \gamma_2) \mapsto (v, \gamma)$$

exists such that for all nodes v_i that occur in this path holds that $v \in X(v_i)$. Observe that $v'_1 = v$. Since if it was an instance of another node, then v_1 is a fork (because we assumed $(v_1, \gamma_1) \mapsto (v, \gamma)$). In that case, however, the relevancy mapping X does not satisfy the disjunction part of requirement (iii) for this fork, since $v \in X(v)$ by definition and $v \in X(v'_1)$ by construction of the path.

Now we distinguish two cases:

- $v_1 = v$. We now know that $\gamma(v) = \gamma_1(v) + 1$ (from our assumption that $(v_1, \gamma_1) \mapsto (v, \gamma)$ and rule (1)) and $\gamma'_1(v) = \gamma_1(v) + 1$ (from the path and rule

(1)). Since $v'_1 = v$ we see that the value of v also is increased on exit of (v'_1, γ'_1) and therefore necessarily $\gamma(v) > \gamma_1(v) + 1$ (from the path), which is a contradiction.

- $v_1 \neq v$. We now know that $\gamma(v) = \gamma_1(v)$ (from our assumption that $(v_1, \gamma_1) \mapsto (v, \gamma)$ and rule (1)) and $\gamma'_1(v) = \gamma_1(v)$ (from the path and rule (1)). Since $v'_1 = v$ we see that the value of v also is increased on exit of (v'_1, γ'_1) and therefore necessarily $\gamma(v) > \gamma_1(v)$ (from the path), which is a contradiction.

From these contradictions we conclude that the merge instance can have at most one predecessor. ■

LEMMA 5 *Instances of an unfolding either have the same number of successors (or 1 successor in case of conditional instances) as their counterparts in the activity graph, or they have zero successors.*

PROOF. Suppose that we have an instance (v, γ) which has more than its allowed number of successors. In other words, we can find two successors (v', γ') and (v'', γ'') such that $v' = v''$ (by definitions 1 and 3). Note that the inductive definition preserves the property: for all $(v, \gamma) \in V$ holds that $\gamma(v')$ is defined if and only if $v' \in X(v)$ for all nodes $v' \in N$. Thus, the domain of γ equals the domain of γ'. Now we distinguish two cases:

- $v' = v''$ is not a join node. By definition, γ undergoes the same transformation, since both edges must be added by the same inductive clause. Therefore we can conclude that $\gamma' = \gamma''$, which contradicts our assumptions.
- $v' = v''$ is a join node. Then, of course, (v', γ') and (v'', γ'') have their other necessary predecessors (as shown in lemma 4), say $(v'_1, \gamma'_1), ..., (v'_{n-1}, \gamma'_{n-1})$ and $(v''_1, \gamma''_1), ..., (v''_{n-1}, \gamma''_{n-1})$ respectively. Without loss of generality we may assume that $v'_i = v''_i$. It is clear that $\gamma'_i \neq \gamma''_i$ for at least one $1 \leq i < n$, since otherwise $\gamma' = \gamma''$, which is a contradiction of our initial assumption.

  Moreover, we know that $\gamma'_i(v'_i) = \gamma''_i(v''_i)$ by the combination of inductive clause (3) and the fact that $\gamma(v) = \gamma'_i(v'_i)$ and $\gamma(v) = \gamma''_i(v''_i)$ according to this clause. Now we apply lemma 3 to the different (as argued above) instances (v'_i, γ'_i) and (v''_i, γ''_i) (remember that $v'_i = v''_i$). Thus, we see that $(v'_i, \gamma'_i) \mapsto^+ (v''_i, \gamma''_i)$ (or the other way around, but that case is similar). Moreover, $v'_i = v''_i$ is not a conditional node (since we assumed that conditionals cannot directly lead to a join in activity graphs) and its value is therefore never reset. In other words, $\gamma''_i(v''_i) = \gamma'_i(v'_i) + 1$ which is in contradiction with knowledge derived in the previous paragraph.

Of course, due to failed synchronizations in join nodes, predecessors of join nodes may have zero successors. The resulting unfolding is not well-defined. ■

The next lemma states a very useful property of our unfoldings, namely that they are acyclic. This means that an unfolding defines a, possibly infinite, partial ordering, which is exactly what we intended.

LEMMA 6 *An unfolding is acyclic.*

PROOF. First, we prove that if there is a cycle in an unfolding, then the initial instance (n^0, γ^0) is on that cycle. Assume that there is a cycle, say $(v_1, \gamma_1), ..., (v_n, \gamma_n), (v_1, \gamma_1)$, such that the initial instance is not part of the cycle. In order for this cycle to be part of the unfolding, at least a path from the initial instance to this cycle must exist (see the extremal clause of definition 3).

From lemma 4 we conclude that one of the instances in the cycle, say (v_i, γ_i), is a join instance that connects the initial instance to the cycle (since only join instances can have multiple predecessors and the initial instance is not on the cycle). However, by definition 3 we know that $\gamma_{i+1}(v_i) = \gamma_i(v_i) + 1$. Since the value of a join node cannot be reset, it is impossible that $(v_{i+1}, \gamma_{i+1}) \mapsto^* (v_i, \gamma_i)$. From this contradiction we conclude that the initial instance must be part of the cycle.

Next, we prove the lemma by contradiction. Therefore, let us assume that a cycle exists in the unfolding. Above we have shown that (n^0, γ^0) is on the cycle. With the knowledge that the initial instance cannot be a conditional instance by definition 1 (and its value thus is never reset), we can use a similar argument as above to show that the cycle cannot be a cycle. Therefore, no cycles exist in unfoldings! ■

The next lemma states a sufficient condition on the syntax of activity graphs for finiteness of their unfoldings.

LEMMA 7 *Let $G = (N, n^0, \rightarrowtail, c)$ be an activity graph. If for all cycles in G, say $v_1, ..., v_n$, such that $n \leq 4 \cdot | \rightarrowtail |$ holds that they contain a conditional node, say v_i, such that (v_i, v_{i+1}) is the true edge and v_i is not reset on the cycle, then the unfolding of G is finite.*

PROOF. We prove the lemma by contradiction and therefore assume that the premises hold, but the unfolding is infinite. This means that there exists an infinite path, and since an activity graph is finite, at least one node, say v must appear infinitely often in this path. This can only occur, if v is on a cycle. Since this cycle satisfies the precondition, the counter of the "exit" conditional of this cycle must be reset infinitely often (otherwise the cycle is not infinitely often enabled). This means that there must be another cycle involving node v that resets the counter. Thus, connecting these cycles gives us a larger cycle that contradicts our assumption about the cycles of the activity graph (namely that the counter of the exit conditional is not reset). The question now is how long these two cycles can be.

We first consider the cycle which contains the true edge of the exit conditional, say $v' \rightarrowtail v''$. The path from v to v' can be bounded by $| \rightarrowtail |$, since any path from v to v' that is longer than $| \rightarrowtail |$, can easily be transformed to a path with length bounded by $| \rightarrowtail |$. The same holds for the path from v'' to v, with the result that the length of this first cycle can be bounded by $2 \cdot | \rightarrowtail |$. (More precisely, if there is a cycle in the activity graph involving v and $v' \rightarrowtail v''$ with a length greater than $2 \cdot | \rightarrowtail |$, then there exists a cycle also involving v and $v' \rightarrowtail v''$ with a length less or equal to $2 \cdot | \rightarrowtail |$.)

We can use the same argument to show that the cycle from v to the resetting conditional of v' and back also can be bounded by $2 \cdot | \rightarrowtail |$. Combination gives the required upper bound. ■