RECOGNIZING FINITE REPETITIVE SCHEDULING PATTERNS IN MANUFACTURING SYSTEMS

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

[1] *Nijmeegs Instituut voor Informatica en Informatiekunde, 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 tasks or 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 extracted from the activity diagrams. It reduces the original problem to a problem containing some minimal number 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: Scheduling, manufacturing systems, UML activity diagrams, 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 [13, 19]. In addition, in some cases also the order of tasks for a single resource influences temporal behaviour of the manufacturing system [8], analogous to the Traveling Salesman Problem [10]. Both of these problems are 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 [12], 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. For example, consider 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.

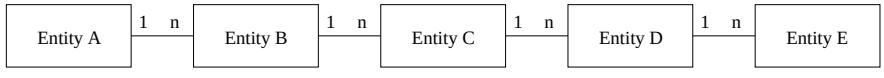

Figure 1: Hierarchical structure of manufacturing entities.

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 final 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 [14, 20]. Furthermore, the hierarchical structure of the manufacturing entities leads to patterns on several granularity levels. The purpose of this paper is to 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 created 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 [16, 17] for the compact specification of task graphs which may contain finite repetitive behaviour. Essentially, the same type of activity diagrams is used within ASML to specify scheduling problems. The key technical difficulty that we face when giving a formal definition of this subclass of activity diagrams is to rule out certain undesired race conditions. The second part of our contribution consists of a method for finding repetitive subgraphs in these task graphs using the activity diagrams. This information can 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 [15] 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 UPPAAL model checker [9]. 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 [15], 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 [4]. In contrast to our work, the semantics of [4] associates a transition system to each activity diagram, using some form of "token game". The transition system semantics of [4] 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 [6], 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 of our contribution includes the CABINS system [11]. It is "an integrated framework of iterative revision integrated with knowledge acquisition and learning for optimization in ill-structured domains" (quoted from the CABINS web-site). In this approach knowledge about task patterns that already have been scheduled before is used to improve schedules. There are, however, numerous difference between the CABINS approach and ours. For instance, we do not use learning, and CABINS supports run-time scheduling whereas our approach is static.

Also related is the computer-aided design of video processing algorithms of [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, apparently no exploitation of equality of loop instances takes place. Our work also relates to widening and acceleration techniques (e.g., [3] and [2, 7]), 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 using the activity diagram it is associated with. 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. 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 Section 2.1 we introduce a subclass of UML activity diagrams for the compact specification of task graphs which may contain limited repetitive behaviour [16, 17]. Section 2.2 associates task graphs with activity diagrams using the so-called *relevancy mapping*, which is needed to exclude the possibility that an activity diagram is ambiguous (i.e. can be associated with multiple task graphs) due to race conditions.

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. The actual scheduling of task graphs is not considered in the present paper. Therefore, we omit the durations and resource requirements of the activities/tasks in all definitions.

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.*

  We write $v \rightarrowtail v'$ *for* $(v, v') \in \rightarrowtail$*.*

- $c : C \to N \times (\mathbb{N} \setminus \{0\}) \times N \times 2^C$ *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 edge *of* v*,* n *the* upper bound *of* v *denoted by* $ub(v)$*,* $v \rightarrowtail v''$ *the* false edge *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 edge of v is enabled and the false edge of v is disabled. After n executions of the true edge it becomes disabled and the false edge becomes enabled. The enabledness can be reset to the initial situation by taking a false edge 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 [16]. 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.

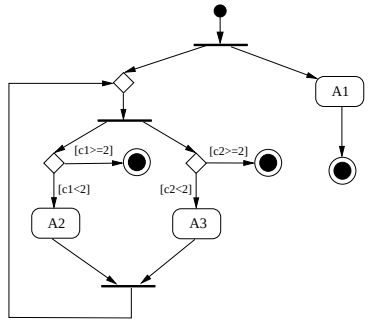

Figure 2: A small activity graph.

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.

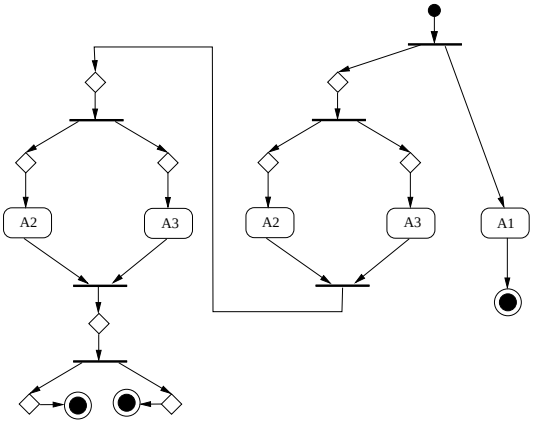

Figure 3: The intended unfolding of the activity graph of Figure 2.

Although the unfolding operation is intuitively quite clear, a difficulty concerning race conditions exists. 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 that an activity graph may contain race conditions in which two parallel branches of a fork use the same conditional counter, which results in a non-unique unfolding. 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.

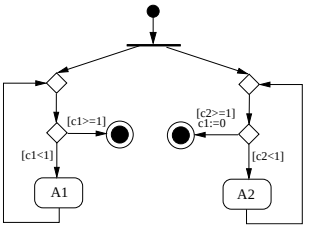

Figure 4: A non-deterministic activity graph.

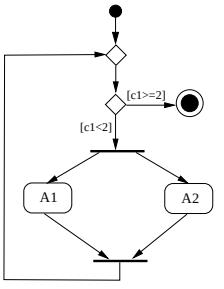

Figure 5: An activity graph which requires the use of additional counters.

In order to forbid situations in which an activity graph can be unfolded in more than one way, we require the existence of 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[1]. 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* $X : N \to 2^N$ *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* $1 \leq i \neq j \leq n$*.*

[1] 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 A1 must be labeled with c1 or A2 must be labeled with c1 (otherwise we cannot give a clean inductive definition of the unfolding operation). If we assume that A1 is labeled with c1, and we unfold the activity graph, then we see that only *one* instance of A2 appears, namely $(A2, \emptyset)$, since A2 has an empty relevancy set. This, of course, is not what we expect from the unfolding.

We can show that the problem whether a general activity graph has a relevancy mapping is NP-complete by a reduction from 3-SAT without negation and with exactly one true literal per clause [5]. 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. In practice we use an ad hoc algorithm to find a relevancy mapping that works fine for all activity graphs that appear in the present paper.

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 node sets according to γ and γ'. Finally, if $\gamma \in \Gamma_N$ and S is a subset of nodes, then we let $[\gamma]_S$ denote the partial node 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[2]:

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

[2]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.

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

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

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

$$\frac{(v,\gamma) \in V \quad v \in C \quad c(v) = (v', n, v'', R) \quad \gamma(v) \geq n}{(v'',\gamma'') \in V \quad (v,\gamma) \mapsto (v'',\gamma'') \quad \textit{where } \gamma'' = \gamma[v := v+1][R := 0]} \tag{5}$$

All successors or predecessors are considered in rules (2) and (3).

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

$$a \parallel b \quad \Longleftrightarrow \quad (a \not\mapsto^* b \wedge b \not\mapsto^* a)$$

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 isomorphic unfoldings (Lemma 12).

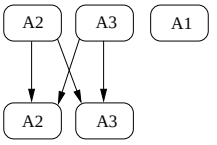

Figure 6: The task graph of the unfolding in Figure 3.

- 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 5).
- 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 8 and 6).
- An unfolding is acyclic, but it might be infinite (Lemma 9).
- There exists 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 10).

The control structure instances (all instances but those of activity nodes) can be "stripped away" as conditional and merge instances have no function anymore (see the third bullet above). The resulting structure is 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*

- $T = \{\, (v, \gamma) \in V \mid v \text{ is an activity node} \,\}$*, and*
- $\rightarrow \subseteq T \times T$ *defined as:* $a \rightarrow b$ *if and only if*
  - $a \mapsto^+ b$*, and*
  - *no* $c \in T \setminus \{b\}$ *exists such that* $a \mapsto^+ c$ *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, the node valuations of the instances are not depicted.)

3. An Approach to Exploit Repetitive Structures in Activity Graphs

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

Apart from the known 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 a scheduler infeasible. Instead, we find so-called *cyclic structures* in the activity graph, which are subgraphs that appear more than once in the unfolding. These cyclic structures can be exploited during scheduling, since they also define reappearing subgraphs in the task graph. Hence, only one such a subgraph needs to be scheduled.

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 cyclic 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 the cyclic structures and the schedule, we construct a schedule for the original, generally much larger, activity graph.

3.1 Formalizing the 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 cycles in the activity graph such that they are minimal w.r.t. to detecting the cyclic structures. 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 cycles 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 the activities to be done for all entities are the same, then the upper bound is set to one.

- Increment the lower bounds of all conditionals that control a single cycle, until the activity graph is "extendable" for the conditionals (below we formally explain what extendability means). The order of this search process can be arbitrary.

We define what we exactly mean with incrementing 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.*

Clearly, 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 shown in the previous section. Next, we define what we exactly mean with a "cyclic structure" in an activity graph.

DEFINITION 6 (CYCLIC 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* cyclic structure *of $\mathcal{A}$ iff there exists more than one* isomorphic embedding *of $\mathcal{A}'$ into the unfolding $(V, \mapsto)$ of $\mathcal{A}$, i.e., an injective function $i : N' \rightarrow V$ satisfying*

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

This definition carries easily over to task graphs. Note that repeated patterns in an unfolding are 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. First, we define the set of direct successors of a subset of instances N, and a subset of the $\mapsto$ edge relation that excludes incoming edges of a set of nodes G:

$$next(N) = \{b \mid a \mapsto b \wedge a \in N\} \quad (6)$$

$$\mapsto_{\overline{G}} = \{((v,\gamma),(v',\gamma')) \in \mapsto \mid v' \notin G\} \quad (7)$$

We say that $a \mapsto_{\overline{G}}^{+} b$ if and only if there exists a path from a to b which consists of one or more edges in $\mapsto_{\overline{G}}$. Using these two definitions we can define the notion of *repetitive structure* within an unfolding.

Definition 7 (Repetitive structure) *Let $\mathcal{A}$ be a well-defined activity graph with a finite unfolding $(V, \mapsto)$ and let G be a subset of conditionals of $\mathcal{A}$. The repetitive structure induced by $\mathcal{A}$ and G is inductively defined as follows:*

(i) Base: $R_0 = \{(n^0, \gamma^0)\}$ *and* $B_0 = \{(v, \gamma) \mid (n^0, \gamma^0) \mapsto^+_G (v, \gamma)\}$.

(ii) Induction: $R_i = \mathit{next}(B_{i-1}) \setminus B_{i-1}$ *and* $B_i = \{b \mid a \mapsto^+_G b \wedge a \in R_i\}$.

A repetitive structure consists of only a finite number of non-empty sets 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 for $i > 0$, and all edges leading outside B_i lead to R_{i+1}. Figure 7 gives a graphical representation of a repetitive structure where all sets R_i and R_j are disjoint.

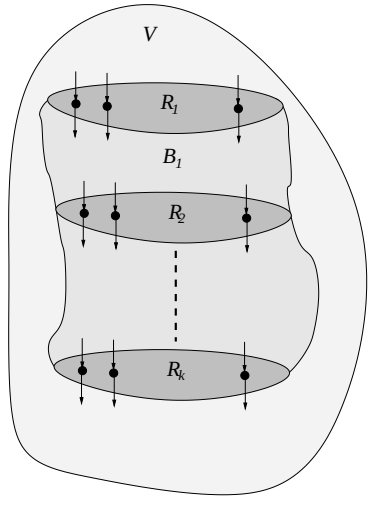

Figure 7: A repetitive structure.

If all instances in R_i are pairwise parallel, then we define γ_i^* as the union of all node valuations in R_i projected to the domain of the conditionals (the set C) of the activity graph as follows (it is well-defined by Lemma 5 of the appendix):

$$\gamma_i^*(v) = \begin{cases} \gamma(v) & \text{if } \exists_{v'}\ (v', \gamma) \in R_i \wedge \gamma(v) \neq \uparrow \wedge\ v \in C \\ \uparrow & \text{otherwise} \end{cases}$$

Next, we define the situation in which we are able to recognize cyclic structures from the repetitive structure. The idea is that we state conditions on the repetitive structure of a set conditional G such that certain

sets $R_i \cup B_i$ identify cyclic structures of the activity graph. As a result, we can copy-paste these sets to increase the upper bounds of the conditionals in G to extend the activity graph.

DEFINITION 8 (EXTENDABILITY) *Let G be a subset of conditionals of an activity graph $\mathcal{A}$ and let $R = \{R_1, ..., R_k\}$ and $\{B_1, ..., B_k\}$ be the associated repetitive structure. We call G extendable, if*

- *the set R is a partitioning of the set of all instances of conditionals in G, such that $|R_i| = |G|$ (hence, γ_i^* is defined: see the proof of the lemma below),*
- *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:*

$$c \in G \Rightarrow \big(\gamma_{i+1}^*(c) = \gamma_i^*(c) + 1\big) \\ \wedge \\ c \notin G \Rightarrow \big(\gamma_i^*(c) = \gamma_{i+1}^*(c) \vee (\gamma_i^*(c) \geq ub(c) \wedge \gamma_{i+1}^*(c) \geq ub(c))\big)$$

We call the set $R_i \cup B_i$ a repetitive set.

The ugly-looking formula in the fourth item of the definition informally means that (i) conditionals in G have taken their true-edges once in B_i, and (ii) the status of conditionals not in G has not changed: if the true-edge (false-edge) is enabled according to γ_i^*, then the true-edge (false-edge) is enabled according to γ_{i+1}^*.

LEMMA 9 *If an activity graph $\mathcal{A}$ is well-defined and 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 cyclic structures of the extension.*

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

First we prove that the instances in R_i are pairwise parallel. From Definition 7 follows that (i) every element of R_i is an instance of a conditional in G, (ii) every element of R_i has a predecessor in B_{i-1}, and (iii) $B_{i-1} \cap R_i = \emptyset$. Now assume that $v \nparallel v'$ and $v, v' \in R_i$. Then there exists a path from v to v'. Since the unfolding is acyclic, and conditional instances have 1 predecessor, this path must pass through B_i. Since the

B-sets do not contain instances of conditionals in G v' must be in the set R_j such that $i \neq j$. Thus, R is not a partitioning, which we assumed. Therefore, $v \parallel v'$.

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. By the fourth item in Definition 8 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 8 tells us that the structure $R_{i_1} \cup B_{i_1}$ is independent from the rest of the unfolding (in particular 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 8 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 $\mathcal{A}$. 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 $\mathcal{A}$ associated with the sets $R_{i_j} \cup B_{i_j}$ are cyclic structures, since we have shown that these subgraphs reappear in the (G, n)-extension of $\mathcal{A}$ due to the copy-pasting sketched above. □

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 10 (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* $G \prec G'$ *if and only if for all repetitive sets* $R_i \cup B_i$ *of* G *we can find a* B'_j *in the* 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 11 *If an activity graph* $\mathcal{A}$ *is well-defined and 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 cyclic 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 9 to G, and then to G'. Assume that the repetitive structure of G is given by $R = \{R_1, \ldots, R_m\}$ and $B = \{B_1, \ldots, B_m\}$, and that the repetitive structure of G' is given by $R' = \{R'_1, \ldots, R'_n\}$ and $B' = \{B'_1, \ldots, B'_n\}$.

By Definition 10 we can find for every repetitive set $R_i \cup B_i$ of G a B'_j such that $R_i \cup B_i \in B'_j$. Thus, we copy-paste the set $R_i \cup B_i$ n times and also add the new instances to the set B'_j. Lemma 9 then says that this creates the extension $\mathcal{E}(\mathcal{A}, G, n)$. Next, we must prove that G' still is extendable for this extension. Therefore, we observe that the copy-pasting does not change the status of conditionals not in G, due to the fourth item of Definition 8. Thus, the "new" set R'_j that follows the enlarged set B'_j is still equivalent to the old one w.r.t. the status of the conditionals. Therefore, $\mathcal{E}(\mathcal{A}, G, n)$ is still extendable for G', and the possibly enlarged repetitive sets $R'_j \cup B'_j$ are cyclic structures of $\mathcal{E}(\mathcal{E}(\mathcal{A}, G, n), G', n')$. □

This concludes the first step of our approach. We now can lower the upper bounds of a set of conditionals in an activity graph $\mathcal{A}$ such that these conditionals are just extendable (Definition 8). Lemma 9 proves that the subgraphs of $\mathcal{A}$ associated with the *repetitive sets* indeed are cyclic structures of the extension of $\mathcal{A}$. Definition 10 and Lemma 11 generalize this to multiple, hierarchical sets of 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. We assume that we can obtain such a solution, since the actual scheduling falls outside the scope of this paper. It is noteworthy, however, that scheduling a task graph consists of introducing extra precedence edges between tasks that are parallel and share resources to ensure mutual exclusion of such tasks.

The third step consists of using the repetitive sets of the condensed activity graph of step one and the schedule of step two to construct a schedule for the original activity graph. There are many different strategies to do this. For example, one could only consider the schedule of the repetitive sets. The already partially scheduled unfolding (and task graph) of the original activity graph can then be obtained using the method sketched in the proof of Lemma 9. At this moment we are happy with such a (hopefully!) good and non-trivial starting point for the scheduling activity. Of course, matters of optimality are important in this step and we regard it as an important subject of future work.

3.2 Example Application of the Approach

In this subsection we apply our approach to part of a scheduling problem from a wafer scanner. Its activity graph is depicted in Figure 8. In order to achieve maximal utilization, a wafer scanner has a work queue of recipes to do. This queue is extended with extra recipes every once in a while. On such occasion the schedule is extended, which implies a scheduling action of a (static) activity graph. The conditional upper bounds of the graph are instantiated in accordance with the number of entities of the recipes in the queue. 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 $G_C = \{\text{c0, c1}\}$. 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 $G_D = \{\text{c2, c3}\}$. 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 $G_E = \{\text{c7}\}$. 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 in JAVA, and we can show in a few minutes that the task graph consists of 11655 tasks (construction of the relevancy mapping takes virtually no time).

We try to apply the technique explained in the previous section. The first step consists of finding smallest upper bounds of the conditionals

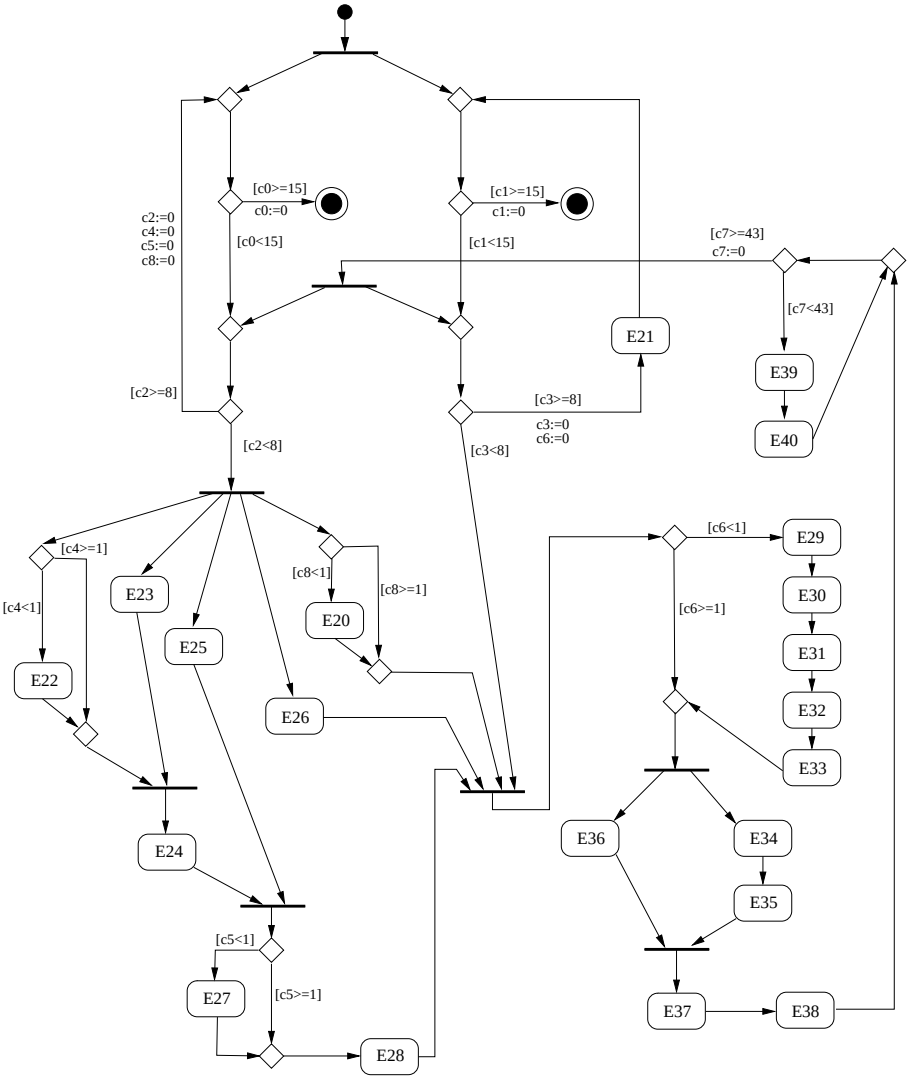

Figure 8: An activity graph which specifies part of a real-life scheduling problem for a wafer scanner.

that specify the repetitions for the reticles, 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 11). Com-

putation 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 an optimal schedule is shown in Figure 9 as a Gantt chart.

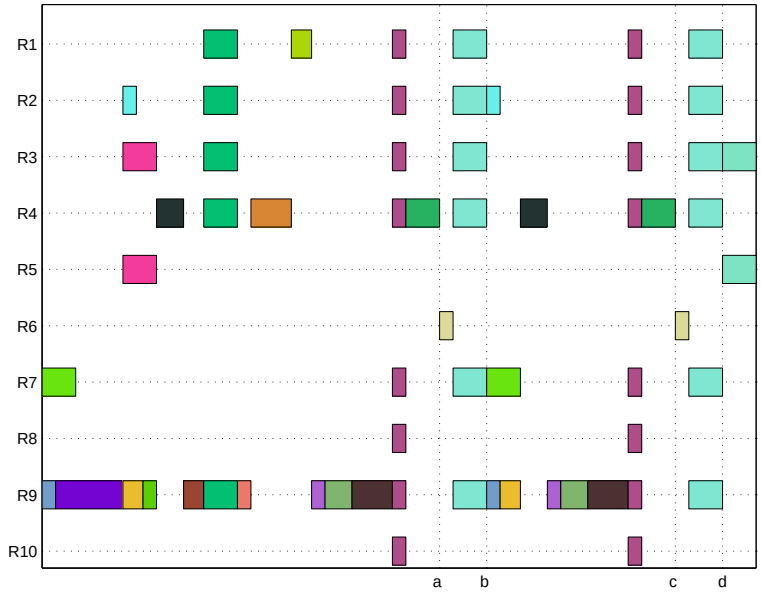

Figure 9: An optimal schedule for the condensed activity graph.

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 8 and Lemma 9). 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 9 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 11 we need to construct the schedule for the original problem from the inside outwards. 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 interval [c, d] 42 times. Next, 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 9 will succeed the task that ends at d, and the precedences admit that it is executed in parallel with the task starting at d on the resources 3 and 5. We believe that the schedule is still optimal after extension in this case.

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 cases 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 absence of race conditions of the activity graph, which can be proven statically by construction of a *relevancy mapping*.

The expressivity of the activity graphs is sufficient for a subset of practical cases from industry. It is possible to model parallelism 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. Finding this subset 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] 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`.).

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

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

[6] 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.

[7] 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.

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

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

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

[11] K. Miyashita and K. Sycara. CABINS: A framework of knowledge acquisition and iterative revision for schedule improvement and reactive repair. *Artificial Intelligence Journal*, 76(1-2):377–426, 1995.

[12] N.J.M. van den Nieuwelaar, J.M. van de Mortel-Fronczak, and J.E. Rooda. Design of supervisory machine control. Accepted by European Control Conference 2003.

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

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

[15] S. Roels. Applicability of model-checking methods to scheduling in machines. Master's thesis, Nijmeegs Instituut voor Informatica en Informatiekunde, University of Nijmegen, The Netherlands, October 2002. Confidential.

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

[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] J.I. van Zante-de Fokkert and T.G. de Kok. 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.

Appendix: Proofs

DEFINITION 1 (WITNESS) *A* witness *for an instance (v, γ) is a finite set of instances W such that either*

- *$W = \emptyset$ and $(v, \gamma) = (n^0, \gamma^0)$, or*
- *there exists an (instance of an) inductive clause $\frac{H}{H'}$ in Definition 3(ii) such that "$(v, \gamma) \in V$" occurs in H', and the set of instances appearing in H equals W.*

DEFINITION 2 (PROOF) *A* proof *of an instance (v, γ) is a finite labeled tree such that the root is labeled with (v, γ), and if a node is labeled with (v', γ'), then the labels of its children together are a witness for (v', γ'). The* size *of a proof P is the number of nodes in the tree, denoted by $|P|$.*

Note that every instance of an unfolding has a proof.

LEMMA 3 *If X is a relevancy mapping and (v, γ) is an instance of the unfolding that has been created using X, then the domain of γ equals $X(v)$.*

PROOF. The lemma can be proven by induction on the number of applications of the clauses of Definition 3 that are needed to show that (v, γ) is part of the unfolding.

- Base: 1 application. Then (v, γ) is the initial instance, and clearly it holds that the domain of γ^0 equals $X(n^0)$.
- Induction: assume that it holds for an instance whose existence can be "proven" with n applications. We can easily show, using the extremal clauses of Definition 3, that the proposition must also hold for instances whose existence can be proven with $n + 1$ applications.

□

LEMMA 4 *Let (v, γ) be an instance in an unfolding $(V, \mapsto)$ such that $q \in X(v)$. Then a* return 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 applications 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 the path exists.

Now let us assume that the desired path exists 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 distinguish two cases:

- The edge $(v, \gamma) \mapsto (v', \gamma')$ is added by inductive clause 1,2,4, or 5. Therefore, v' is not a join, and we can conclude 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 $(v_1, \gamma_1) \mapsto (v', \gamma')$, ..., $(v_m, \gamma_m) \mapsto (v', \gamma')$ such that $(v, \gamma) = (v_i, \gamma_i)$ for some $1 \leq i \leq m$. Moreover, we can show with n applications of the base and inductive clauses that (v_i, γ_i) belongs to V for all $1 \leq i \leq m$. 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 is they key lemma that ensures that race conditions do not appear in activity graphs which have a relevancy mapping.

LEMMA 5 *If (v, γ) and (v', γ') are two different instances such that $q \in X(v) \cap X(v')$, then a path from (v, γ) to (v', γ') (or the reverse path) exists of which all nodes are labeled with q by X.*

PROOF. Consider two instances (v, γ) and (v', γ') such that $q \in X(v) \cap X(v')$. Let P be the proof of (v, γ) and let P' be the proof of (v', γ'). We prove the lemma by induction on $|P| + |P'|$. The base case it that $|P| + |P'| = 2$, in which case $(v, \gamma) = (v', \gamma') = (n^0, \gamma^0)$. We assumed, however, that the instances are different.

From this contradiction we conclude that the proposition holds for the base case. Now consider the case where $|P| + |P'| = n$. The induction hypothesis then says:

> If (v, γ) and (v', γ') are two different instances with proofs P and P' respectively, and $q \in X(v) \cap X(v')$, and $|P| + |P'| < n$, then a path from (v, γ) to (v', γ') (or the reverse path) exists of which all nodes are labeled with q by X.

Using Lemma 4 we can say that two paths that are completely labeled with q exist:

$$(u_0, \gamma_0) \mapsto (u_1, \gamma_1) \mapsto (u_2, \gamma_2) \mapsto ... \mapsto (u_m, \gamma_m)$$

$$(u'_0, \gamma'_0) \mapsto (u'_1, \gamma'_1) \mapsto (u'_2, \gamma'_2) \mapsto ... \mapsto (u'_n, \gamma'_n)$$

where $(u_0, \gamma_0) = (u'_0, \gamma'_0) = (n^0, \gamma^0)$, $(u_m, \gamma_m) = (v, \gamma)$, and $(u'_n, \gamma'_n) = (v', \gamma')$. Consider the longest common prefix of these paths; assume that the paths are equal up to index i. Clearly, if $i = m$ or $i = n$, then the proposition holds. The other case is that $i < m$ and $i < n$. Then $(u_i, \gamma_i) = (u'_i, \gamma'_i)$, and $(u_{i+1}, \gamma_{i+1}) \neq (u'_{i+1}, \gamma'_{i+1})$. We show that this results in a contradiction by distinguishing two cases:

- u_i is a fork. Then u_{i+1} and u'_{i+1} are not join nodes by assumption (just above Definition 3). If $u_{i+1} = u'_{i+1}$, then we can conclude by Definition 3 that $\gamma_{i+1} = \gamma'_{i+1}$, which contradicts the assumption that the $i + 1$-th instances are different. Hence, $u_{i+1} \neq u'_{i+1}$. However, both are labeled by the relevancy mapping with q. This violates the disjunction requirement on forks of the relevancy mapping. This contradiction shows that u_i cannot be a fork.
- u_i is not a fork. Therefore, we know that $u_{i+1} = u'_{i+1}$. The only way to achieve that $(u_{i+1}, \gamma_{i+1}) \neq (u'_{i+1}, \gamma'_{i+1})$ is that $u_{i+1} = u'_{i+1}$ is a join node, and that there exists a predecessor of (u_{i+1}, γ_{i+1}), say (w, γ_*), and a predecessor of $(u'_{i+1}, \gamma'_{i+1})$, say (w', γ'_*), such that $w = w'$ and $\gamma_* \neq \gamma'_*$. Note that by Definition 3 we know that $\gamma_*(w = w') = \gamma_i(u_i) = \gamma'_*(w = w')$. Since $w = w'$, we know that $w = w' \in X(w) \cap X(w')$. Moreover, P clearly contains a proof for (w, γ_*) that is smaller than P, and P' clearly contains a proof for (w', γ'_*) that is smaller than P'. Therefore, we can apply the induction hypothesis. Thus a path $(w, \gamma_*) \mapsto^+ (w', \gamma'_*)$ or a path $(w', \gamma'_*) \mapsto^+ (w, \gamma_*)$ exists that is completely labeled with $w = w'$. In the first case, $\gamma'_*(w = w') > \gamma_*(w = w')$, since the value of w is incremented on exit of (w, γ_*) and never reset, because we assumed that it cannot be a conditional node (just above Definition 3). This contradicts the information derived above. The second case is similar.

We can conclude from these contradictions that the case that $i < m$ and $i < n$ cannot occur. This proves the lemma. □

LEMMA 6 *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). By Lemma 3 we know that 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 (see Definition 3), 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 $\gamma(v) = \gamma''_i(v''_i)$. Now we apply Lemma 5 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 a path from (v'_i, γ'_i) to (v''_i, γ''_i) exists that is completely labeled with v'_i (or the other way around, but that case is similar). Moreover, $v'_i = v''_i$ is not a conditional node (since we assumed just above Definition 3 that conditionals cannot directly lead to a join in activity graphs) and its value is therefore never reset. In other words, $\gamma''_i(v''_i) > \gamma'_i(v'_i)$ which is in contradiction with knowledge derived above.

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 is used in the proof of Lemma 8.

LEMMA 7 *Consider an activity graph $\mathcal{A}$ with a conditional node v and relevancy mapping X. If v is reachable from the initial node, then there exists a non-conditional node v' such that $v' \in X(v)$.*

PROOF. Consider the activity graph $\mathcal{A}$ with a conditional node v and relevancy mapping X, and assume that v is reachable from the initial node of $\mathcal{A}$. Let us assume that $X(v)$ does not contain a non-conditional node, and consider the path in $\mathcal{A}$ from the initial node to v, say $v_0, v_1, \ldots, v_n$, where v_0 is the initial node and v_n is v, and such that $v_i \neq v_j$ for all $0 \leq i \neq j \leq n$. Definition 2 tells us that v_{n-1} must be a fork node or a conditional node. If it is a conditional node, then we can repeat this argumentation. Since the initial node cannot be a conditional node, a fork node v_i on the path exists such that v_j is a conditional node for all $i < j \leq n$.

Now consider the two successors of v, say v' and v''. Clearly, $v' \neq v_j$ and $v'' \neq v_j$ for all $i < j \leq n$, since conditional nodes can have only one predecessor. Note that we assumed (just above Definition 3) that v' and v'' cannot be join nodes. Therefore, v' and v'' must be conditional nodes, since the definition of the relevancy mapping gives that $v', v'' \in X(v)$ and we assumed that $X(v)$ only contains conditional nodes. Moreover, $X(v')$ and $X(v'')$ only contain conditional nodes, since $X(v) = X(v') = X(v'')$.

We can repeat the argumentation above and conclude that $\mathcal{A}$ must contain infinitely many conditional nodes, which is a contradiction by Definition 1. Hence we conclude that $X(v)$ does contain a non-conditional node. □

LEMMA 8 *Every instance has a unique witness.*

PROOF. Let us consider an instance (v, γ) in the unfolding. First, observe that (v, γ) has at least one witness, because it is in the unfolding. We prove that this witness is

unique by contradiction. Thus, we assume that W and W' are distinct witnesses of (v, γ), and distinguish three cases:

- v is a merge, activity, fork or exit. Then we know by the inductive clauses that we can write $W = \{(v', \gamma')\}$ and $W' = \{(v'', \gamma'')\}$. Thus, $(v', \gamma') \mapsto (v, \gamma)$, and $(v'', \gamma'') \mapsto (v, \gamma)$. By definition of the relevancy mapping X, we know that $v \in X(v')$ and $v \in X(v'')$. Lemma 5 says that a path $(v', \gamma') \mapsto (v_1, \gamma_1) \mapsto^* (v'', \gamma'')$ (or the other way around, but that case is identical) exists such that $v \in X(v_i)$ for all instances (v_i, γ_i) on that path. We distinguish two cases:
  - $(v_1, \gamma_1) \neq (v, \gamma)$. By Lemma 6 we can say that v' is a fork node and that $v_1 \neq v$. By construction of the path we know that $v \in X(v_1)$. Thus, the relevancy mapping for the successors of the fork node v' does not satisfy the disjunction restriction, which is a contradiction.
  - $(v_1, \gamma_1) = (v, \gamma)$. According to the inductive clauses we can derive that $\gamma''(v) > \gamma(v)$, since the value of v is increased on exit of (v, γ) and never reset because v is not a conditional node. Moreover, since $(v'', \gamma'') \mapsto (v, \gamma)$ and v is never reset we know that $\gamma(v) \geq \gamma''(v)$. Combination of these two equations gives us that $\gamma(v) > \gamma(v)$, which clearly is a contradiction.
- v is a join. Then we can write $W = \{(v_i, \gamma_i) \,|\, 1 \leq i \leq n\}$ and $W' = \{(v'_i, \gamma'_i) \,|\, 1 \leq i \leq n\}$, where $v_i = v'_i$ for some n. Now assume that $\gamma_i \neq \gamma'_i$ for some i (otherwise, $W = W'$). Then $(v_i, \gamma_i) \mapsto (v, \gamma)$ and $(v'_i, \gamma'_i) \mapsto (v, \gamma)$. Note that we assumed (see the text just above Definition 3) that a conditional or a fork cannot be followed by a join node. Therefore, $v_i = v'_i$ is not a conditional or fork. By Lemma 5 we thus know that a path from (v_i, γ_i) to (v'_i, γ'_i) exists (or the other way around, but that is similar) that is completely labeled with $v_i = v'_i$ by the relevancy mapping. Moreover, by Lemma 6 we can conclude that the path passes through (v, γ) (since $v_i = v'_i$ is not a fork). Since $v_i = v'_i$ is not a conditional, its value is never rest. Thus, we can derive that $\gamma'_i(v_i = v'_i) \geq \gamma(v_i = v'_i)$ and $\gamma(v_i = v'_i) > \gamma'_i(v_i = v'_i)$, since the value of $v_i = v'_i$ is incremented on exit of (v'_i, γ'_i). This clearly is a contradiction.
- v is a conditional. Then we know by the inductive clauses that we can write $W = \{(v', \gamma')\}$ and $W' = \{(v'', \gamma'')\}$. Thus, $(v', \gamma') \mapsto (v, \gamma)$, and $(v'', \gamma'') \mapsto (v, \gamma)$. By Lemma 7 we know that $w \in X(v)$ for some non-conditional node w. By definition of the relevancy mapping also $w \in X(v')$ and $w \in X(v'')$. We now can repeat the argumentation of the first item to show a contradiction.

Therefore, (v, γ) has a unique witness. □

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 9 *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 8 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 10 *Let $\mathcal{A} = (N, n^0, \rightarrowtail, c)$ be an activity graph. If for all cycles in $\mathcal{A}$, 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 $\mathcal{A}$ 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 v' 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. □

LEMMA 11 *If $(v, \gamma) \mapsto^+ (v', \gamma')$ and $w \in X(v) \cap X(v')$, then (v, γ) occurs on the return path of (v', γ') for w.*

PROOF. Straightforward using Lemma 6. □

LEMMA 12 *If X and X' are relevancy mappings for an activity graph, then the two resulting unfoldings are isomorphic.*

PROOF. Consider the two resulting unfoldings $(V_1, \mapsto_1)$ and $(V_2, \mapsto_2)$, respectively. Inductively, we construct a mapping $f : V_1 \to V_2$ such that

$$f(v_1, \gamma_1) = (v_2, \gamma_2) \quad \Rightarrow \quad v_1 = v_2 \tag{A.1}$$

$$f(v_1, \gamma_1) = (v_2, \gamma_2) \quad \Rightarrow \quad \gamma_1 \text{ and } \gamma_2 \text{ agree on intersection of domains} \tag{A.2}$$

$$(v, \gamma) \mapsto_1 (v', \gamma') \quad \Leftrightarrow \quad f(v, \gamma) \mapsto_2 f(v', \gamma') \tag{A.3}$$

Suppose that f has been defined for all predecessors of some node (v', γ_1') of V_1. Let W be the unique (by Lemma 8) witness for (v', γ_1'). We consider three cases:

1 $W = \emptyset$. Then (v', γ_1') is the initial instance, i.e., $v' = n^0$ and $\gamma_1'(v) = 0$ if $v \in X(n^0)$ and it is undefined otherwise. We map the initial instance of $(V_1, \mapsto_1)$ to the initial instance of $(V_2, \mapsto_2)$, i.e., we define $f(v', \gamma_1') = (n^0, \gamma_2^0)$, where $\gamma_2^0(v) = 0$ if $v \in X'(n^0)$ and it is undefined otherwise. Since both γ_1' and γ_2' map a subset of nodes to 0, we see that γ_1^0 and γ_2^0 agree on the intersection of their domains.

2 $|W| = 1$. In this case, let (v, γ_1) be the unique element of W. Since a witness for a join node always contains at least two elements, v' is not a join node. Let $f(v, \gamma_1) = (v, \gamma_2)$. Then γ_1 and γ_2 agree on the intersection of their domains. We consider three subcases:

   (a) $v \in J \cup M \cup A$. Then $\gamma_1' = \gamma_1[v := v + 1]$. By Definition 3, $(v, \gamma_2) \mapsto_2 (v', \gamma_2')$, where $\gamma_2' = \gamma_2[v := v + 1]$. By the definition of relevancy mapping, the domain of γ_1 equals that of γ_1', and the domain of γ_2 equals that of γ_2. Thus, also γ_1' and γ_2' agree on the intersection of their domains. Thus, if we extend f by defining $f(v', \gamma_1') = (v', \gamma_2')$ we maintain properties (A.1) and (A.2). Clearly, f also preserves the (unique) incoming transitions of (v', γ_1') and (v', γ_2').

   (b) $v \in F$. Similar to the case $v \in J \cup M \cup A$.

   (c) $v \in C$. Similar to the case $v \in J \cup M \cup A$.

3 $|W| > 1$. In this case v' is a join node. Let $W = \{(v_1, \gamma_1^1), \cdots, (v_n, \gamma_1^n)\}$. Let $f(v_i, \gamma_1^i) = (v_i, \gamma_2^i)$, for $1 \leq i \leq n$. Since v_i is in the domains of γ_1^i and γ_2^i, γ_1^i and γ_2^i agree on the intersection of their domains, and W is a witness, $\gamma_2^1(v_1) = \cdots = \gamma_2^n(v_n)$. Hence $f(W)$ (defined by pointwise extension) is a witness for the node instance (v', γ_2'), where $\gamma_2' = \cup_{i=1}^n \gamma_2^i[v_i := v_i + 1]$. Thus, if we define $f(v', \gamma_1') = (v', \gamma_2')$, we preserve nodes and incoming transitions. The proof (A.2) is left to the reader (use lemma 11).

□