UML AND PETRI NETS FOR DESIGN AND ANALYSIS OF DISTRIBUTED SYSTEMS

B. Bordbar, L. Giacomini and D.J. Holding

Department of Electronic Engineering, School of Engineering, Aston University,
Aston Triangle, Birmingham B4 7ET, UK
Tel: Tel: +44 (0)121 359 3611 Fax: +44 (0)121 359 0156
e-mail: {B.Bordbar,L.Giacomini,D.J.Holding}@aston.ac.uk

Abstract: This paper presents a modification to UML, to model and analyse discrete-event dynamic systems (DEDSs) representing Manufacturing systems. It shows how Petri Nets can be used to improve the representation and analysis of the dynamic model of a system specified using UML. Finally the technique is illustrated by its application to a simplified production line.

Keywords: Discrete-event dynamic systems, Petri-nets, object modelling techniques

1. INTRODUCTION

This paper describes the design of a supervisory control system for a distributed manufacturing process which forms part of a wider manufacturing system. The focus of the paper is on the design of a verifiable discrete event controller using a UML based method. The approach adopted involves (i) using Petri net models instead of conventional Statecharts to provide analytic Dynamic Models; and (ii) using compositional Petri net techniques to synthesise the Interconnection Model. The model of the complete controller can be then analysed and verified using Petri net theory. The approach is demonstrated by application to a prototype packaging machine.

Recent advances in computer technology have resulted in a widespread use of Discrete-Event Dynamic Systems or DEDSs in manufacturing, robotics, traffic management, logistics, and computer and communication networks (Cassandras, 1999; Sobh *et al.*, 1994). DEDSs require complex control systems (Ramadge and Wonham, 1987) to ensure correct and optimal operation. To model complex DEDSs, researchers have developed bottom up, top down and hybrid synthesis techniques. However, these approaches concentrate on functional abstraction, and have produced incomplete specifications and designs (Firesmith, 1993). In order to facilitate the design of complex systems, produce more understandable designs and specifications, facilitate the transition between design and implementation and to enable software re-use, several researchers including Booch (1991), Rumbaugh *et al.* (1991), Ellis (1994), Douglass (1999), have advocated a paradigm shift towards object oriented (OO) techniques. The various approaches have converged with the development of the UML.

2. UML BASED DESIGN

The Unified Modelling Language (UML), originally a methodology for software designers, is the most recent product generated by the aggregation of previous generation Object Oriented methodologies (Booch et al., 1999). UML takes the designer through the design life cycle, starting from the description provided by users or experts down to the final software product. UML preserves convergence and clarity in design by prescribing a set of steps that involve the creation of a series of graphs, the generation of an evolving model of the system, and

the rigorous examination of this model. Thus, the application of UML by different people with different skills results in comparable and highly portable final designs.

UML consists in a set of graphs or charts with explanatory comments that can be expressed in a formal way or in plain spoken language. The main diagrams recognized by the standard are nine, falling into two categories: static aspects diagrams and dynamic aspects diagrams. The designer can choose quite freely a subset of them. Also the order in which they are designed is not fixed, apart from the following rule of thumb

1. Use case diagram
2. One diagram in the static diagrams set
3. One statechart for each object in the system
4. One diagram in the dynamic diagrams set
5. Implementations diagrams

2.1 Use cases and class diagram

A UML design procedure (Booch et al., 1999) starts with the study of the *use cases*, which are detailed written descriptions of 'what the objectives are' and 'how the job is carried out'. Studying the use cases enables the designer to recognise different '*key agents*' of the system, known as *Objects* in UML terminology. Considering common features and operations of key agents, objects are extrapolated into collections called *Classes*. Classes can be organised in a graph (or a collection of graphs), to build a '*class diagram*', that describes the *static relationship* between the classes. The classes are represented graphically by rectangular boxes connected together by lines or links that can be either of association type or of generalisation type. An *association* is a structural relationship that specifies the connection between one or more members of the classes. A *generalisation* is a relationship between a general class and a derived class, i.e. one can define a new class from another class, by means of inheritance.
The operations defined in the class diagram include all the services that can be requested from an object to effect the behaviour. The class can request itself one of its services.

2.2 Dynamical Aspects

2.2.1 Statecharts

The UML dynamical model is conventionally elaborated by constructing a Statechart model using the information provided by use case and class diagram. Statecharts, (Harel, 1987) are used to depict the behaviour of each class/object, and show the states (or configurations of its attributes) and the operations / events which modify the states.

2.2.2 Interaction Diagrams

Once the class structure is defined, diagrams affiliated to the dynamic interaction between the instances of the classes can be drawn. An object is an instance of a class, and the object's state is denoted by the value of its attributes. Only executing the operations defined in the class can change the values of the object's attributes. Objects participate in the interaction diagrams as follows:

- *collaboration diagram*: shows the structural organization of the objects and their interconnection or communications (service requests or *messages*),
- *sequence diagram*: shows the time sequence of operations.

Since the sequence diagram and collaboration diagram are isomorphic, the designer may be decided to use only one of the two. For our purposes the sequence diagram is better suited.

2.2.3 A Petri net Dynamical model

In order to promote our capability to analyse and verify the underlying system, it is desirable to substitute the Statechart with an analytic representation such as Process Algebras, Automata or Petri nets. In this paper we harness the capabilities of Petri nets for modelling asynchronous concurrent system, and replace the Statechart dynamical model with an analytic Petri net model. As a mathematical formalism, Petri-net theory can be used to analyse DEDS characteristics such as synchronisation, concurrency, conflicts, resource sharing, precedence relations, event sequences, non-determinism and system deadlocks (David and Alla, 1992); (Desrochers and Al-Jaar, 1995).
For general information regarding Petri nets, we refer to (Peterson, 1981) and (Murata, 1989). For completeness, a short definition of a Petri net is reported below.

A *Petri net* is a triple $\boldsymbol{N} = (S, T, F)$ where S is a finite set of *places* and T is a finite set of *transitions*, in which $S \cap T = \varnothing$, the empty set. $F \subseteq (S \times T) \cup (T \times S)$ represents the directed arcs between places and transitions. A *marking* of *N* is a function $\mathbf{m}: S \rightarrow \{0,1,2,3, \ldots\}$, assigning to each place $s \in S$, the number of tokens in s under the marking **m**. A transition t is *enabled* under **m** (i.e. it may *fire*) if each of its inputs have at least one token. If t *fires*, it changes the marking **m** to a new marking $\mathbf{m}'$ by removing one token from each of the input places and adding one token in each of the output places. A marking $\mathbf{m}'$ is said to be reachable from the marking **m** if it exists a sequence of firing transition from **m** to $\mathbf{m}'$.

3. COORDINATION AND SYNCHRONISATION

The final and crucial stage in the synthesis process is the instantiation of the objects in the system and the design of coordination and synchronisation logic that satisfies the requirements specified in the Collaboration and Sequence diagrams. What is missing in the UML procedure is a methodological way of building and verify the coordination and synchronization aspects. Using Petri net for this task provides a consistent, tangible, and relatively straightforward approach.
In practice, this reduces to the instantiation of the Petri net models of the objects, and the synthesis of appropriate coordination and synchronisation logic. However, the process of compositional synthesis is not an ad-hoc procedure. For example, simply decomposing the collaboration and sequence diagrams into a bag of rules that are imposed on the objects ignores the important sequence information and will over constrain the model.

To maintain the precedence relationships in the compositional synthesis of two Petri nets, we first provide a scenario of the desirable states of the composite Petri net and the order that we expect the desirable states to appear. This is achieved by connecting the two component Petri nets together via the systematic addition of interconnection arcs, places or transitions. The composite net is then analysed to ensure conformance with the use case. The example in Section 4 demonstrates the approach used.

3.1 The Graph of Desirable States

Let us assume that our system is made of m objects. For each object a Petri net is instatiated. Assume that Γ denotes the part of the use case dealing with the synchronisation of n of the above components into an overall system (i.e. the various objects are synchronized two by two, or less frequently, in groups of 3/4 components).
Assume that $(\boldsymbol{N}_1, \mathbf{m}^1{}_0)$, …, $(\boldsymbol{N}_n, \mathbf{m}^n{}_0)$, where $\mathbf{m}^i{}_0$ denotes the initial marking, are safe and live Petri Nets, representing object instances of these n components of the system. A proportion of the information provided by Γ has already been captured in the body of the dynamics of the Petri nets $(\boldsymbol{N}_1, \mathbf{m}^1{}_0)$, …, $(\boldsymbol{N}_n, \mathbf{m}^n{}_0)$.
The reachability graph of the simple juxtaposition of the Petri Nets for the single components must contain the desired reachability graph for the Petri Net synchronized following the use case Γ. As a result we might need to block some undesirable transitions to create the desirable behaviour.

The first step is to construct a directed graph, which we shall refer to as the *Graph of Desirable States* (GDS) and as the name suggest includes the set of all desirable states and their relations together. The word "desirable" reflects the facts that this graph, as we shall see later, embraces all we expect from the system to do. $R_\infty(\boldsymbol{N}_i, \mathbf{m}^i{}_0)$ denotes the set of all reachable markings of the Petri Net $(\boldsymbol{N}_i, \mathbf{m}^i{}_0)$, for each $\mathbf{m}^i \in R_\infty(\boldsymbol{N}_i, \mathbf{m}^i{}_0)$, and let *enabled*$(\mathbf{m}^i)$ denote the set of all enabled transitions of $\boldsymbol{N}_i$ under the marking $\mathbf{m}^i$.
Each node of GDS is labelled by a (n + 1)-tuple of the form $a = (\mathbf{m}^1, \dots, \mathbf{m}^n, U)$ where $\mathbf{m}^1, \dots, \mathbf{m}^n$ are reachable markings of the components $\boldsymbol{N}_1, \dots, \boldsymbol{N}_n$ and U is a (possibly empty) subset of *enabled*$(\mathbf{m}^1) \cup \dots \cup$ *enabled*$(\mathbf{m}^n)$, the set of all enabled transitions under $\mathbf{m}^1, \dots, \mathbf{m}^n$ that indicates the subset of fireable transitions with the current marking that are ' undesirable'. For the node labelled with $a = (\mathbf{m}^1, \dots, \mathbf{m}^k, U)$ we shall write $\boldsymbol{m}(a) = (\mathbf{m}^1, \dots, \mathbf{m}^k)$ and $U(a) = U$.

3.2 Heuristic for construction of GDS

Consider the set E_0 of all transitions enabled under initial marking $\mathbf{m}^1{}_0, \dots, \mathbf{m}^n{}_0$. Relying on the information provided by the use case Γ, it is possible that we need some of the transitions in E_0 to be prevented from firing. Assume that U_0 denotes the set of all such transitions, notice that U_0 can be the empty set. Create the first node, which shall be referred to as the *initial node*, and label it with $a_0 = (\mathbf{m}^1{}_0, \dots, \mathbf{m}^n{}_0, U_0)$. From this node start firing each of the desirable transitions $E_0 \setminus U_0$, to obtain another set of nodes with their marking and set of undesirable cases. Put arcs connecting node a_0 and the newly created ones labelling them with t, where t is the name of the corresponding firing transition. The procedure is repeated for each of the new nodes created.

GDS captures the behaviour expected from the composite net. For example, if there is a node *a* of GDS with no output then our design of the system expects a deadlock, which is anomalous. Starting and ending in the same node of GDS represents a cyclic phases of the system. It is often required to have live systems, i.e. starting from each node and considering an action (a transition) we expect to find a path from that node to another node which has t labelling an outgoing arc.

Now the task ahead of us is to synchronise Petri net $(\boldsymbol{N}_1, \mathbf{m}^1{}_0), \dots, (\boldsymbol{N}_k, \mathbf{m}^n{}_0)$ together and generate a new Petri net $(\boldsymbol{N}, \mathbf{m}_0)$ with the dynamical behaviour pictured in GDS. For GDS where $\forall$ *a, b* nodes of GDS, $\boldsymbol{m}(a) = \boldsymbol{m}(b) \Rightarrow U(a) = U(b)$, standard techniques (Juan et al. , 1998) can be applied to compose the two Petri Nets, as shown in the next section. As an example, an almost general rule applicable when we want to prevent a transition t_k in Petri Net N_j from firing under a certain marking $\mathbf{m}^i{}_k$

in Petri Net N_i, a new place is added and connected as input/output to the transition t_k. The place is also connected to transitions in Petri net N_i in such a way that, when the transitions give rise to the marking $\mathbf{m}^i_k$, the token is removed. The firing of the transitions moving out of the marking $\mathbf{m}^i_k$ will put the token back in the place (see Fig. 1 for an example).

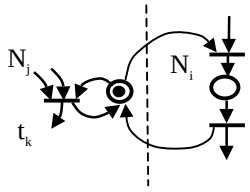

Figure 1

4. EXAMPLE OF A PRODUCTION LINE

The approach is demonstrated by considering the design of a controller for a simplified production line comprising loosely-coupled independently-driven mechanisms such as conveyor belts, wrapping film feeders, film sealers and cutters as shown in Fig. 2.

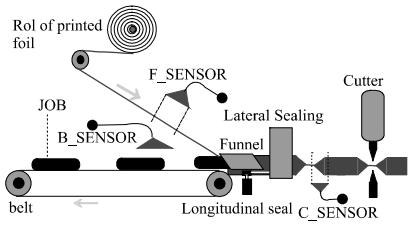

Figure 2 : Production Line

The product (JOB) flow is supplied via a belt and a proximity sensor located at the 'decision point' detects each approaching JOB. When the sensor triggers, the angular position of the motor driving the belt is held so to be able to measure the position of the product with respect to the underlying belt. The packaging film is supplied by unwinding a roll of printed foil. The printed image needs to be positioned centrally on the product; this is done using printed marks (TAG) which are detected by a sensor with the same logic as for the belt. The packaging film is then formed into a tube via a funnel, and a longitudinal sealing roller welds the two edges of the film together. If both JOB and TAG are within production bounds, the product is pushed inside the funnel from where it is carried along by the film. To produce individually packaged products, the tube is sealed between packs by a lateral sealer, and then cut by a cutting machine.

The process is independently driven, in the sense that major modules of the system are controlled individually and independently. A discrete layer, which collects information from relevant modules, synchronises the different parts together. To see this in detail, we consider that the plant consists of four modules Belt, Film, Welder and Cutter. Each of the modules has its own control. For example, the belt is driven by a motor, which is controlled to track a reference model Γ_{BM} and the Film is driven by a second motor controlled to track a reference model Γ_{FM}. To achieve the control objectives related to the synchronisation of Belt and Film the following heuristic is implemented.

Mutual synchronisation of Belt and Film: In order to have each JOB wrapped in Film we need to have both {JOB at decision point} and {TAG at decision point}. As a result, under either of the cases {JOB at decision point and TAG at decision point} or {JOB not at decision point and TAG not at decision point} the controllers of Belt and Film follow their usual reference models Γ_{BM} and Γ_{FM}. However, if {JOB at decision point} but {TAG not at decision point} then the belt must decelerate (to stop if necessary) by switching to another suitable reference model Γ_{BS}. In the meantime, the film feed will continue until the sensor detects TAG; following this the belt is accelerated to ensure that the JOB and TAG are synchronised as the JOB enters the funnel. The case for {TAG at decistion point} but {JOB not at decision point} can be treated similarly.

The Welder and Film are synchronised by applying a heuristic similar to the one between Belt and Film. Similarly, the Cutter is synchronized with the Film.

4.1. The class diagram

The description in Section 3 plays the role of the use case for the production line of Fig. 2. It can be seen that the main objects are the four physical machines: Film, Belt, the Welder, and Cutter. The product to be wrapped, JOB, is identified with the belt.

The class diagram for the production line of Fig. 2 is shown in Fig. 3. The box representing each class is divided into three regions: name, list of attributes, list of operations. The Belt and Film classes are exactly dual. They have 4 states, described by boolean variables:

- dp: in the proximity of the sensor
- Wait: stopped to wait for the other part to arrive
- Out: a part has been wrapped and a new one is awaited for but not yet in the sensor proximity
- Wrap: JOB and TAG are moving synchronously aligned with each other, then the wrapping is taking place

The actions that move the objects from one state to the other are

- New: a new JOB/TAG has been detected from the sensor
- Ab: stop moving and wait
- Start: restart moving after a waiting phase
- Go: no waiting phase, the wrapping take place

- Exit: the product has been wrapped and goes out of scope

Similarly follow the classes Cutter and Welder.

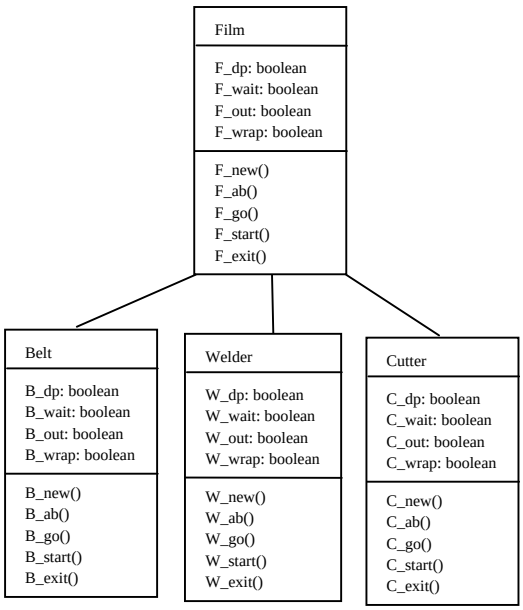

Figure 3: Class diagram

4.2. The sequence diagram

The response of a class to an external solicitation is determined by the internal dynamics of the class and the external interactions or communications as defined in a sequence diagram. The complete sequence diagram for the production line involves one object of each of the classes Belt, Film, Welder, Cutter. For clarity, the sequence diagram can be translated into a set of smaller sequence diagrams by considering subsets of the objects. For example, the temporal sequence of the service calls for the Belt-Film subsystem are shown in the sequence diagram of Fig. 4. The arrows indicate service requests from the sender to the receiver. The labels on the arrows are the operations that are requested. Fig. 4 refers to the scenario where the TAG comes first and the Film has to wait for the JOB to arrive. The sequence diagram for the case when the JOB arrives first can be derived similarly.

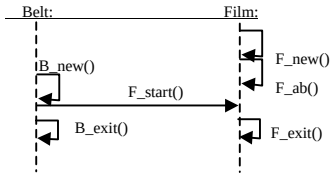

Figure 4: Sequence diagram: tag coming first

4.3 Synthesising Petri net model

To keep the example clearer, we focus on the classes Belt and Film and their interaction.

First we assign Petri nets to each of the classes. Generally, the Petri Net of a class is formed by using a place to represent each Boolean attribute and a transition for each operation that changes the attributes values. As the reader can notice, in this particular example, the classes are all structured in the same way as shown in Fig. 5 (a).

We create an initial marking for each instantiated object by considering the initial state of the corresponding components of the production line. The system starts with B_out and F_out.

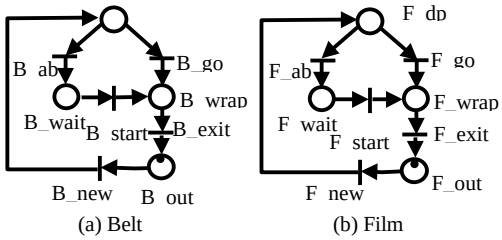

Figure 5: Petri Net for the classes Sensor and Belt

4.3.1 The GDS for the mutual synchronisation of Belt and Film

The next stage in the compositional process is designing the synchronisation logic which enforces the mutual synchronisation heuristic for the Belt and Film, as defined in Section 4. To do this we create the scenario of desirable cases for the composition of the Belt and Film.

To help us we make use of the discursive use case provided in Section 4, plus the further insight of the dynamical behaviour given by the sequence diagrams, and the Petri Nets of the components, to generate the GDS, as described in section 3.1.

For example, starting with (B_out, F_out) we will have a set of transitions enabled. These are B_new and F_new. Following the use case, none of the two transitions is undesired, therefore U = ∅. Let us suppose, that JOB (associated with Belt_class) arrives first.

Then, the marking is (B_dp, F_out) and the transitions enabled are (B_ab, B_go, F_new). Because we want to stop the belt if the JOB is at decision point but the TAG is still out of scope, U = {B_go}. Proceeding in this way the graph in Fig. 6 is built. Now, we have condensed in a Petri Net style all the informations about the dynamics of the two co-operating subsystems. To synchronise the subsystems Film and Welder (or Film and Cutter), the use case relative to their interaction is taken into consideration and a similar procedure is applied. Due to the similarity of the classes and use cases, the resulting GDS will be a copy of the one in Fig. 6, with the relevant names changed.

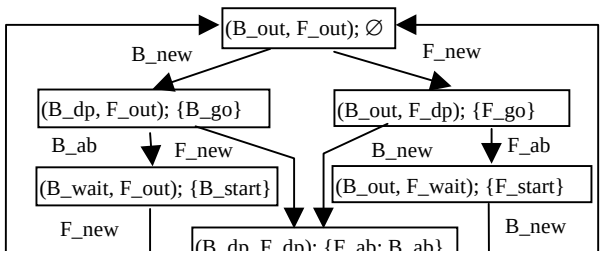

Figure 6: GDS

4.3.2 Coordination and synchronisation

For the GDS in Fig. 6, let us examine the set of undesirable transitions one by one.
B_new is not allowed to fire if and only if the marking is (B_out, F_wrap), i.e. it should be prevented from firing when the place F_wrapping is tokenized. This is achieved adding the place SP1, that is always marked except when F_wrapping is marked (in fact its token is removed from the firing of F_go or F_start and it is replaced by F_exit). The same applies to F_new. These two conditions are required to prevent the synchronizing procedure getting 'confused' when, due to the small space allowance between JOB and TAG, one of the two goes out of scope before than comes a new one before the wrapping procedure is finished.
As far as B_ab is concerned, it is always an undesirable transition except when F_wrap is marked, therefore a double sided arc between F_wrap and B_ab is added (and the same applies to F_ab).
B_exit is not desired before F_wrap gets marked (for the wrapping to take place the places F_wrapping and B_wrapping should be both marked), therefore place SP6 is added with an arc to B_exit; it is marked by the firing of F_go or F_start. Similarly SP4 is added for F_exit.

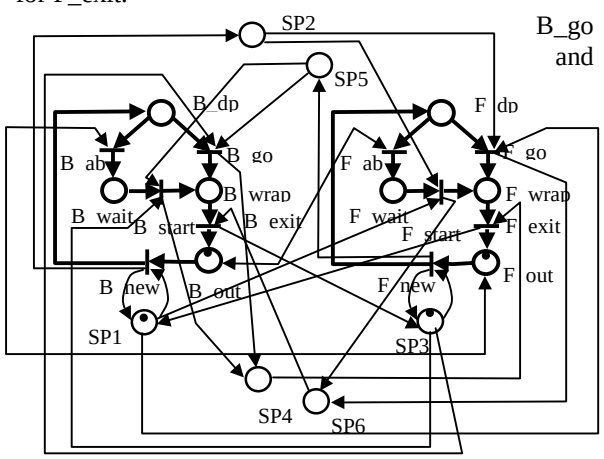

Figure 7: Petri Net for the discrete part of the

B_start should fire as soon as F_dp is marked, therefore SP5 is added. SP2 is added to enable F_go and F_start as soon as B_dp is marked.
All this results in the Petri of Fig. 7, this graph is live and bounded and has the reachability graph in Fig. 8. The reader can notice the strong similarity with the graph of desirable case.

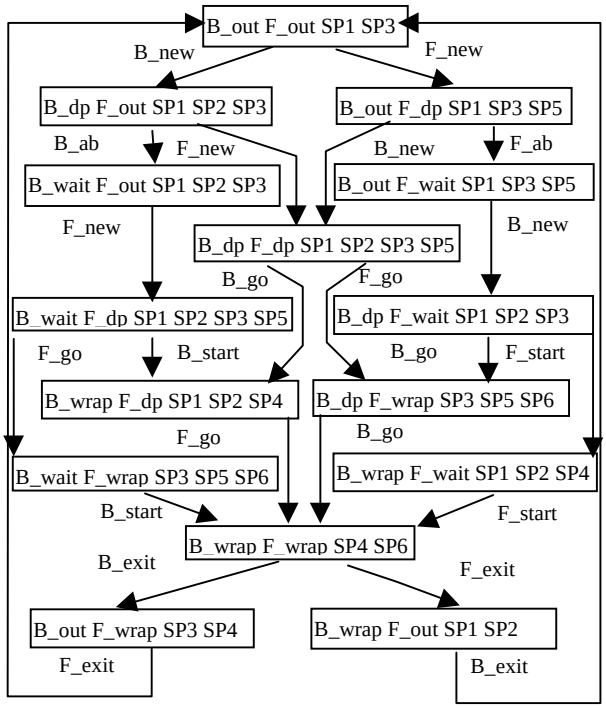

Figure 8: Reachability graph for the petri net in figure

5 IMPLEMENTATION

The subsystem Film-Belt has been implemented in Matlab (vers. 5.3) and the supervisory part, corresponding to the Petri net, is implemented as a StateFlow block. The Simulink model is shown in Fig. 9 can be seen. Each state in the Petri net of the components is translated into a state of the state-chart, and the full state-chart is shown in Fig. 10. The belt and film systems are in two parallel sections (indicated by the dashed smoothed box). The synchronisation places are implemented with boolean variables, updated when a transition takes place. The StateFlow states of the Film and Belt during a typical synchronisation operation are shown in Fig. 11.

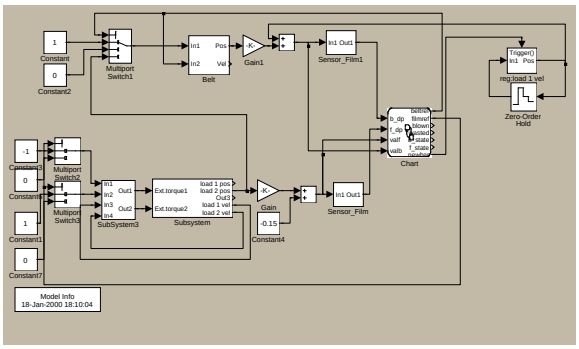

Figure 9: Simulink scheme

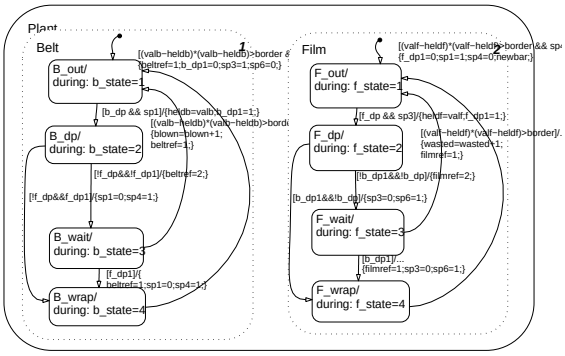

Figure 10: Stateflow chart

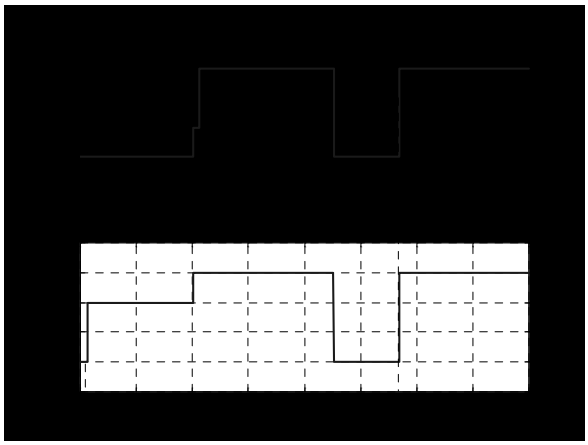

Figure 11: States of Film and Belt in the chart: 1=_out, 2=_dp, 3=_wait, 4=_wrapp.

6 CONCLUSIONS

This paper has presented an integrated approach to UML for modelling and analysing real-time systems. It has shown that Petri-net theory can be used to improve the representation and analysis of the dynamic model of a system that is specified using UML, making the design engineer more confident that the model accurately represents the system. Moreover, the Petri-net dynamic model can be used to implement a controller based on current supervisory control theory. The technique has been illustrated by its application to a wrapping machine that forms part of a larger production line.

ACKNOWLEDGEMENTS

This work was supported by EPSRC (UK) Grant GR/L31234.

REFERENCES

Booch, G., *Object-Oriented Design With Applications*, Benjamin Cummings, 1991.

Booch, G., J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

Cassandras, C.C., Lafortune, S.: *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.

David, R. and H. Alla, *Petri nets and Grafcet: Tools for modelling Discrete-event Systems*, Prentice-Hall, 1992.

Desrochers, A.A. and R.Y. Al-Jaar, *Applications of Petri Nets in Manufacturing Systems*, IEEE Press, 1995.

Douglass, B.P., *Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison Wesley, 1999.

Ellis, J.R., *Objectifying Real-Time Systems*, New York: SIGS Books, 1994.

Firesmith, D.G., *Object Oriented Requirement Analysis and Logical design: A Software Engineering Approach*, Wiley, 1993.

Harel, D.: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, Vol. 8, pp. 231--274, 1987.

Juan, E.Y.T., Tsai, J.J.P., Murata, T.,: Compositional verification of concurrent systems using Petri-net-based condensation rules. *ACM Trans. on Programming Languages and Systems*, Vol. 20, No. 5, pp. 917--979, 1998.

Murata, T., Petri Nets: properties, analysis and applications, *Proceedings of the IEEE*, vol. 77, No.4, pp. 541-580, 1989.

Peterson, J.L., *Petri-net Theory and the Modelling of Systems*, Prentice-Hall, 1981.

Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. *SIAM Journal on Control & Optimization*, vol.25, no.1 (1987) 206-230.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorenson, *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.

Sobh, M., C.J. Owen, K.P. Valvanis. and D. Gracanin, A subject indexed bibliography of discrete-event dynamic systems, *IEEE Robotics and Automation Magazine*, Vol.1, No.2, pp. 14-20, 1994.