ISSN 1744-1986

The Open University

Technical Report N° 2007/01

On the use of Coloured Petri Nets in Problem Oriented Software Engineering: the Package Router Example

Jon G Hall
Jens Baek Jørgensen
Lucia Rapanotti

12th January 2007

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

On the use of Coloured Petri Nets in Problem Oriented Software Engineering: the Package Router Example

Jon G. Hall[1], Jens Bæk Jørgensen[2], and Lucia Rapanotti[1]

[1] Department of Computing, The Open University, UK
{J.G.Hall,L.Rapanotti}@open.ac.uk
[2] Department of Computer Science, University of Aarhus, Denmark
jbj@daimi.au.dk

Abstract. In this paper, we present an approach to specification of IT systems that combines the use of Coloured Petri Nets (CPN) and the Problem Oriented Software Engineering (POSE) framework—an extension and generalisation of Jackson's Problem Frames to the solution of software engineering problems. Through the case study of a package routing system, we demonstrate how a CPN model can be used to make appropriate POSE descriptions and support a POSE argument for the adequacy of a problem's software solution. The suitability of CPN as a description language for POSE is discussed and demonstrated in the study. We found that the ability to execute CPN models offers potential for showing adequacy of solutions, a key aspect of software engineering.

Topics. System design using nets; relationship between net theory and other approaches; experience with using nets; higher-level net models (CPN); application of nets to real-time and embedded systems; requirements engineering; Problem Oriented Software Engineering.

1 Introduction

In 2003, Denaro and Pezzè wrote [5]: "Software engineering and Petri nets met several times in the past [...]. However, the cross fertilization has never stabilized and the two fields are passing a period of scarce communication". Since then, a number of initiatives have been taken to bring software engineering (SE) and Petri nets (PN) together more closely. Examples are the book [9] and the workshop on Petri nets and software engineering held as a satellite event to the present conference.

This paper also contributes to these efforts. Our specific goal is to enable the use of *Coloured Petri Nets (CPN)* [14]—a well-established Petri net formalism—as a description language for the *Problem Oriented Software Engineering (POSE)* [12] framework so that better solutions to software engineering problems can be built.

POSE is an extension and generalisation of the fundamental ideas of problem orientation (as seen in Jackson's Problem Frames, [13]). In addition, POSE defines a transformational problem-centred approach to the whole software engineering life-cycle from early requirements through to implementation. At the centre of POSE is the notion of a problem[1] with software development viewed as *problem solving*, where the *solution* of a problem is a *machine*—that is, a program running in a computer—that satisfies the problem's *requirement*. Requirements typically concern properties and behaviours that are located in the problem *context*, i.e., the near and far environment of the machine. As such they may be stated in terms of phenomena far from the machine—those, for instance, of a system operator—with the problem context providing descriptions that relates machine behaviour to those distant phenomena. As we shall see, POSE works with many different classes of description language, from informal natural language—in which early problem descriptions might be expressed—through to fully formal descriptions—such as those in a CPN model. The relationship between informal and formal may also be captured in POSE through problem transformation and the associated development of a convincing argument of the adequacy of a proposed solution.

Petri nets have many advantages as a real-world description language, and so for use within POSE, in that they allow an accurate representation of the causal dependence and independence between real-world domains that are found there. In addition, Petri nets have a very well-established and understood formal basis. CPN is particularly suited for the representation of complex contexts in that it offers high-level structuring mechanisms, the simple representation of complex behaviours, and very well-developed tool support.

In this paper, CPNs will be used as one language for describing a problem's elements. The resulting models are used to support the formulation of adequacy arguments for a solution, and we show how the execution of a CPN model can contribute to the validation of possible solutions against the problem's requirement. The case study upon which our presentation is based is to specify a controller for a complex package router system. Although the problem is not new (see, e.g., [13,24]), our use of CPNs for this problem is novel.

The structure of this paper is: Section 2 discusses related work. Section 3 gives a brief introduction to POSE. Section 4 discusses the use of CPN as a description language for POSE. Section 5 describes the package router case study. Section 6 illustrates the ways in which CPN can assist in POSE. Section 7 contains a discussion and conclusion, and offers future work on the combination of CPN and POSE.

2 Related Work

POSE is an extension and generalisation of Jackson's Problem Frame approach [13]. Problem Frames attempt to keep the focus of the software engineer on

[1] See Section 3 for a fuller description of problem.

developing their understanding of the problem to be solved, rather than on a (premature) move to solution of a poorly understood problem. Problem Frames make certain fundamental assumptions: primary is the separation of descriptions of what is given—the *indicative* parts of a problem—from what is required—the *optative* parts of a problem. Originally confined to Requirements Engineering, the influence of Problem Frames has spread to the fields of domain modelling, business process modelling, software architectures and early design—see [3,4,10] for collections of recent work.

The use of Petri nets in general, and CPN in particular, in early software engineering and requirements engineering is rather sparse. This does not necessarily point to any deficiency of Petri nets *per se*, and is perhaps only because the focus of many application projects of CPN has been design analysis, in the sense of modelling solutions rather than problems.

Some work of using CPN in requirements engineering has been carried out by the second author of this paper: in [16], a CPN model is used as an engine for a graphical animation that serves as a kind of prototype and which is used to specify, validate, and elicit requirements; [15] offers a preliminary study on bringing together CPN and Problem Frames, although it leaves some important issues unresolved, namely the separation of indicative and optative descriptions, which is properly addressed in this paper. Another example of using CPN in requirements engineering is described in [8], where CPN are used to prototype user interfaces.

The work presented by Desel *et al.* in [6] shares some of our aims. The subject is model construction and validation in controller design while we address general software engineering problems (illustrated by a control problem in this paper). Moreover, they use signal nets, which are essentially a low-level net type, extended with some high-level nets features; we use CPN, which is a full-fledged high-level net type with well-developed tool support.

Combining Problem Frames with other notations has recently received some attention, for example, Lavazza and Del Bianco's [19,20] or Choppy et al.'s [2] work on combining Problem Frames and UML. The aim of both approaches is to fill in the gap between requirements analysis (performed in Problem Frames) and early design (in UML); neither addresses issues of validation and adequacy argumentation, which are, instead, the main concern of our work.

3 Problem Oriented Software Engineering

The Problem Oriented Software Engineering (POSE) framework of [12] recognises that software engineering processes by necessity include the identification and clarification of system requirements, the understanding and structuring of the problem world, the structuring and specification of a hardware/software machine that can ensure satisfaction of the requirements in the problem world, and the construction of arguments, convincing both to developers, customers, users and other stake-holders that the developed system will provide the functionality and qualities that are needed.

In any realistic development, these are non-sequential, with the consequence that software development is a complex, iterative process. It is also made more difficult by the need to relate human and physical domains to the formal world of the machine. In some areas, such as safety-critical systems, paramount importance is placed on the quality of the software, with this driving an iterative development process. An effective approach to system development must therefore deal with the informal, the formal, and the relationships that exist between them. To this end, POSE brings together many non-formal and formal aspects of software development, providing a structure within which the results of different development activities can be related, combined and reconciled. Essentially, the structure is the structure of the progressive solution of a system development problem; it is also the structure of the adequacy argument that must eventually justify the developed system. POSE does not prescribe any particular development process; rather it identifies steps of development which may be accommodated within the development process chosen. Previous work [12,24,11] has illustrated the solution of mission-critical development problems under POSE.

In POSE, software development is viewed as solving a *problem*, the *solution* (S) being a *machine*—that is, a program running in a computer—that will ensure satisfaction of the *requirement* (R) in the given *problem world* (or *context* W). Because the requirement typically concerns properties and behaviours that are located in the problem world at some distance from its interface with the machine, requirements are distinguished from *specifications* which only concern machine phenomena.

The problem world is a collections of *domains* ($W = D_1, ..., D_n$) described in terms of their known, or indicative, properties, which interact through their sharing of *phenomena* (i.e, events, commands, states, *etc.*). More precisely, a *domain* is a set of related phenomena that are usefully treated as a behavioural unit for some purpose. A domain has a *name* (N) and a *description* (E) that indicates the possible values and/or states that the domain's phenomena can occupy, how those values and states change over time, and which phenomena—shared or unshared—are produced and when. Associated with each domain $D = N : E$ there are three alphabets:

- the *controlled* alphabet: the phenomena shared with other domains, and controlled by D;
- the *observed* alphabet: the phenomena shared with other domains, and observed by D;
- the *unshared* alphabet: all phenomena of D that are not shared with another domain.

Descriptions of a problem's elements may be in any relevant description language; indeed, different elements can be described in different languages. Parts of a problem could be expressed in natural language—'The operator smokes', for example—with others in first order logic—$x = 0 \wedge x' = 1$, for instance—or any other language. So that descriptions in different languages can be used together in the same problem, POSE provides a semantic meta-level for the combination

of domain descriptions; formally, this is the 'meaning' of the ',' between domains in a problem's context and whose role is to share the phenomena.

A (software) solution is simply a domain ($S = N : E$) that is intended to solve a problem. As such it may have one of many forms, ranging from a high-level specification through to program code. In this paper we deal only with specifications. As a domain, a solution has controlled, observed and unshared phenomena; the union of the controlled and observed sets is the set of *specification phenomena* for the problem.

A problem's requirement states how a proposed solution description will be assessed as the solution to that problem. Like a domain, a requirement is a named description, $R = N : E$. A requirement description should always be interpreted in the optative mood, i.e., as expressing a wish. For a requirement R, there are two alphabets:

- *refs*: those phenomena of a problem that are *referenced* by a requirement description.
- *cons*: those phenomena of a problem that are *constrained* by a requirement description, i.e., those phenomena that the solution domain's behaviour may influence as a solution to the problem.

If *refs* or *cons* refer to phenomena of the solution domain S, they must be specification phenomena.

In POSE, problems are transformed into other problems that are easier to solve, or that will lead to yet other problems that are easier to solve. Problem transformations capture discrete steps in the solution process. Several classes of transformation are recognised in the framework, including: interpretation (capturing increased knowledge of the real-world and designed artefacts in problem descriptions); expansion (adding structure to a problem, its context or its solution); progression (simplifying a problem); and solution (providing justification that a solution description adequately satisfies the problems requirement). Each defined problem transformation transforms problems in a way that respects solution adequacy, and is accompanied by a justification for the transformation.

POSE is formulated in [12] as a Gentzen-style sequent calculus [17] whose sequents are problems, written $W, S \vdash R$, and whose transformations are characterised formally together with the conditions for their applicability. For this paper, it will suffice for the reader to interpret the $W, S \vdash R$ to indicate that S is a proposed solution to the problem with context W and requirement R—a full description of the meaning of $W, S \vdash R$ is given in [12]. We return to problem structuring in Section 6.

4 Describing Problems using Coloured Petri Nets

In this paper, CPN will be used as the main language for describing a problem's elements. To use CPNs as a description language for real-world domains makes many requirements of them:

- it exercises their ability to represent real-world (and wished for) behaviours;

– it exercises their ability to represent real-world (and wished for) structures;
– it requires them to be related to other languages for description, specifically to drive the problem solving process.

The evidence from the literature is that CPN performs well in each of these areas: the CPN ability to represent behaviours is beyond doubt; there are many ways in which CPNs can be manipulated and structured; there are many techniques that allow them to be validated.

The paper is, essentially, structured to address each of the remaining points: although the modelling of the behaviour of real-world domains with CPNs is straightforward, and may be accomplished through analysis of the observed real-world behaviour with validation of the produced models, the CPN representation of requirements—the behaviours that are wished for—is less exercised; we introduce a simple, novel way described below and applied in Section 5. The sharing of phenomena that defines the structures present in the real world is modelled in this paper in ways reminiscent of many well-known Petri net operators that work through the merging of identically named transitions ([1], as well as others).

Here is a very simple illustration of the approach we take:

Assume we have been given the problem to design a solution that should increment its integer input from the environment by one, leaving the answer as output to the environment. Let us assume that the environment has a place of `Integer` colour set as state, offers its state as *input*, and is willing to accept an answer from the solution *output*. Such an environment is described as the Environment CPN; it is the W part of Figure 1(a).

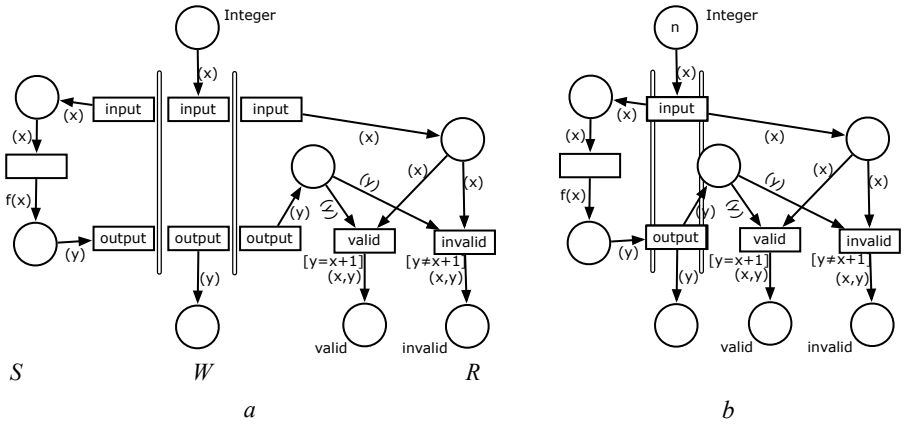

Fig. 1. (a) S is a class of CPNs (that includes a Solution CPN); W is the Environment CPN that supplies *input* and accepts *output*; R the Requirement CPN. The various elements are shown before the transition merging that shares phenomena; (b) the merged CPN, with initial marking $n \in Integer$

Part S of the figure represents the class of CPNs defined by their manipulation of the input variable x through application of the function f. By changing the function f, we can vary the solution specification. The Requirement CPN (part R of the figure) records whether the specification in its context properly matches the optative requirement (essentially, the optative description and its negation associated with the transitions labelled *valid* and *invalid*, respectively): it does this not by preventing behaviours that do not satisfy, but by labelling as valid or invalid those behaviours that are arrived at through execution. When composed into a problem, similarly named transitions in the various CPNs are merged; for this problem, the merging is given in Figure 1 in which is also included an initial marking ($n \in \mathit{Integer}$).

Consider, now, the specification of $f(x) = x + 1$. From the shown initial marking, the reader will easily confirm that all tokens end in the place labelled *valid*, as is wished for in the requirement. For the specification $f(x) = x$, from the initial marking, all tokens end in the place labelled *invalid*.

In general, solving a problem is a complex, usually iterative process, which starts by identifying and describing the problem context and requirement, and culminates in the structuring and specification of a hardware/software machine that can ensure satisfaction of the requirement in the problem context. Under POSE this would require repeated systematic application and justification of a wide range of transformation rules resulting both in a solution description and an auditable design path, something that we do not have the space to describe in detail (but see [24] for a fuller description, albeit not in CPNs). In this paper, rather, our focus is the exploration of CPN as a description language and, to this end, we do not need to synthesise a solution; instead we provide only a CPN intepretation of a — previously synthesised (see [24]) — controller and use it to explore the advantages of such a description.

5 The Package Router Case Study

In this section, we apply the combined POSE/CPN approach to the case study of a package router system. The problem is not new [26] and has been the subject of POSE research [12,24]; however, none of those works make use of CPN as a description language.

5.1 Problem Formulation

A package router is a large machine used by delivery companies to sort packages into bins according to bar-coded destination labels affixed to the packages. Each bin corresponds to a geographical region. The packages slide from a conveyor belt under gravity through a 'tree' of pipes and binary switches, of which the bins are the 'leaves'. The problem is to control the operation of the package router's switches so that packages are routed to their appropriate bins. (There are other aspects of this problem that, for brevity, we do not consider here but which are discussed in [12,24].)

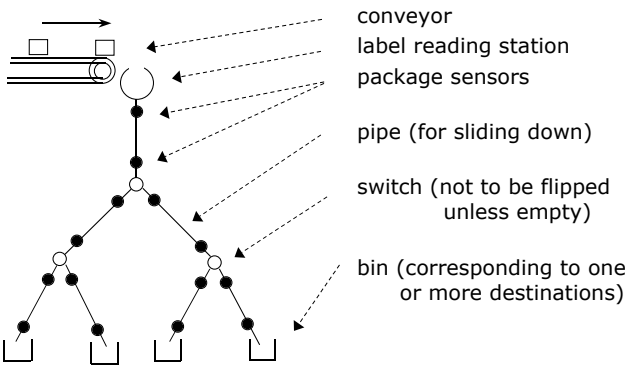

Fig. 2. The Package Router Schematic

We will use the schematic shown in Figure 2 as a starting point for our analysis, leading to the following initial problem formulation under POSE (the merits and drawbacks of this problem representation are discussed in [12]):

$$W, PRSoln \vdash PRReq$$

where

$$W = Bin[1], ..., Bin[\#bins], Switch[1], ..., Switch[\#sw],$$
$$Reading\ Station, Conveyor\ Belt, Package[1], ..., Package[\#pks]$$

Problem-related initial descriptions are listed in Figure 3, first as icons (derived from Figure 2) and then, through interpretation, in natural language. Note that all schematic context domains appear as domains in the problem statement, but that the *Bins*, *Switches* and *Packages* are distinguished by an identifier: although there will be a known fixed number of *Bins* ($\#bins$) and *Switches* ($\#sw$) during operation, the number of *Packages* ($\#pks$) that will flow through the system is unbounded (i.e., *a priori* unknowable).

The sharing of phenomena between problem and solution domains is shown in Table 1. From the table, it is possible to derive the various phenomena alphabets of each domain: for instance, solution domain *PRSol* controls phenomena in $\{left[j], right[j]\}$ and observes phenomena in $\{bin[i], in[j], outL[j], outR[j], read(id, dst)\}$. The requirement refers to $OnBelt[id]$ (i.e, a package with identity id is on belt) and constrains $InBin[i][id]$ (a package with identity id has reached $Bin[i]$, where the bin corresponds to the package's destination).

Our aim is to replace the informal descriptions of Figure 3 with detailed CPN descriptions as explained in Section 4. Their combination will provide an overall executable CPN model which can be used—as we will argue—for constructing an adequacy argument for the problem solution.

5.2 Overview of the CPN Model

Bin[j] $1 \leq j \leq \#bins$		"*Bin*$[j]$ has an incoming pipe with a sensor that fires as a package passes."
Switch[i] $1 \leq i \leq \#sw$		"*Switch*$[i]$ has one incoming and two outgoing pipes, each with a sensor that fires as a package passes. Each sensor activation is communicated to the controller (events *in*$[j]$, *outL*$[j]$, *outR*$[j]$). A switch can be set to either *Left* or *Right*, to determine the exit pipe for a package."
Reading Station		"*Reading Station* has an outgoing pipe with a sensor that fires as a package passes. Each sensor activation is communicated to the controller (event *RS*). *Reading Station* reads a package's details from the package's label and communicates them to the controller (event *read*(id, dst))"
Conveyor Belt		"*Conveyor Belt* supplies packages to *Reading Station*."
Package[id] $1 \leq id \leq \#pks$		"*Package*$[id]$ has a unique identifier *id* and a barcoded destination *dst*. At any time a package can be in state: *OnBelt*, *InRS*, *InBin*$[j]$, for some j, or *InSwitch*$[i]$, for some i."
PRSoln		"The solution to be designed, which can set switch states (events *left*$[j]$, *right*$[j]$)."
PRReq		"To control the operation of the package router so that packages are routed to their appropriate bins. More precisely, each time a package enters the belt, the controller must ensure that the package eventually is delivered to the correct bin, i.e., the bin corresponding to the package's barcoded destination."

Fig. 3. Initial descriptions for the package router problem; see Table 1 for the shared phenomena.

		CONTROLS					
		Bin[i]	*Switch[j]*	*Read. Stat*	*Conv. Belt*	*Package[id]*	*PRSoln*
O	*Bin[i]*					*InBin*$[i][id]$	
B	*Switch[j]*					*InSwitch*$[j][id]$	*left*$[j]$,
S							*right*$[j]$
E	*Read. Stat*					*InRS*$[id]$	
R	*Conv. Belt*					*OnBelt*$[id]$	
V	*Package[id]*						
E			*in*$[j]$,	*RS*,			
S	*PRSoln*	*bin*$[i]$	*outL*$[j]$	*read*(id, dst)			
			outR$[j]$				

Table 1. Shared phenomena for the package router

For ease of presentation we start with an overview of the overall CPN model of the package router problem we arrived at the end of our study. The model was created and executed with the tool *CPN Tools* [27], which includes both a graphical editor and the programming language Standard ML [23]. The model is hierarchically structured and its top level is illustrated in Figure 4. The vertical bars separate the parts of the model which pertain to the problem's solution (to the left), the environment (centre) and the requirement (to the right). Next, we describe and explain each part of the model in turn.

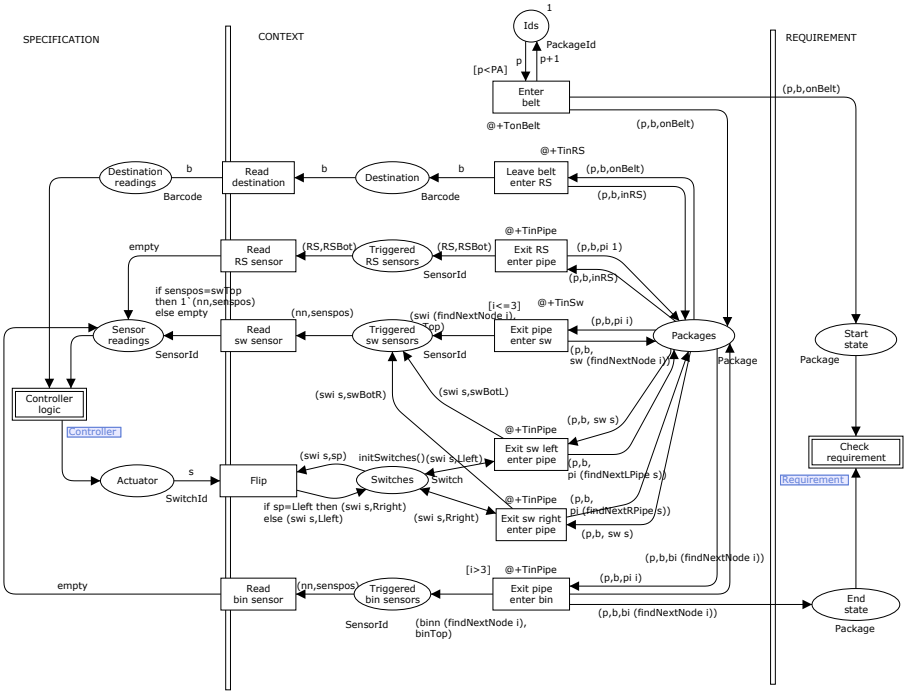

Fig. 4. Top module of the CPN model

5.3 Environment CPN

The CONTEXT part of Figure 4 represents the problem context, i.e., the domains $Bin[1], ..., Bin[\#bins]$, $Switch[1], ..., Switch[\#sw]$, *Reading Station*, *Conveyor Belt*, $Package[1], ..., Package[\#pks]$.

We have declared the colour sets `PackageId`, `SwitchId`, `SensorId`, `PipeId`, and `BinId`, so that the various packages, switches, sensors, pipes, and bins can be identified. Figure 5 shows relevant declarations of the colour sets (as disjoint

sets of integers) for two of the key domains, that is those representing packages and switches.

```
-------------------------------------------------------------------------------
colset PackPos = union onBelt + inRS + sw: SwitchId + pi: PipeId + bi: BinId;

colset SwitchPos = with left | right;

colset Package = product PackageId * Barcode * PackPos timed;

colset Switch = product SwitchId * SwitchPos;
-------------------------------------------------------------------------------
```

Fig. 5. Selected declarations of colour sets to represent problem domains

A package is represented as a 3-tuple `(pid,bc,pp)`, where `pid` is the package id, `bc` is the package's barcode, and `pp` is the package's current position. As can be seen from the declaration of the `PackPos` colour set, a position can either be modelled to be on the belt (`onBelt`), in the reading station (`inRS`), in switch number i (`sw i`), in pipe j (`pi j`), or in bin k (`bi k`), these corresponding to the possible states of a package.

A switch is represented as a pair `(sid, sp)`, where `sid` is the switch id and `sp` is the switch position, i.e., the value `left` or the value `right`.

The CPN model is timed: `Package` tokens carry time stamps. The use of time stamps will allow us a greater ability to reason about the adequacy and qualities of the solution. Note that, in timed CPN models, the time stamp of a token says when the token is 'ready' to be consumed by a transition[2].

Details of each modelled problem domain are given below. These can be related to the overall CPN model of Figure 4 according to the following pattern. Each domain description corresponds to a fragment of the CPN model. Depending on whether the domain has identified states as phenomena, the corresponding CPN fragment stores state information using tokens on places. Each CPN fragment has a 'boundary' which consists of transitions that model the events, i.e., the phenomena, that are shared by the domain with other domains. We do not consider phenomena other than events and states in this paper.

Package domain The *Package* domain is modelled by the place `Packages` and the seven transitions `Enter belt`, `Leave belt enter RS`, `Exit RS enter pipe`, `Exit pipe enter sw`, `Exit sw left enter pipe`, `Exit sw right enter pipe`, and `Exit pipe enter bin`. Each of these transitions models an event which is shared between the Package domain and one of the other domains.

[2] Actually, each CPN model has a global clock whose value is strictly increasing, recalculated after each step to the time that enables at least one transition. Time may not then take contiguous integer values.

For example, the transition `Enter belt` models an event, which is shared between the Package domain and the Conveyor Motor and Belt domain and the transition `Leave belt enter RS` models an event, which is shared between the Package and Reading Station domains. It is modelled that the action represented by each of these transitions takes time by associating a time delay. For example, the transition `Enter belt` has the time delay `TonBelt`, which has two consequences. First, the `Package` token must stay in the `Packages` place for `TonBelt` time units, before it is ready to be consumed by the next transition (which is `Leave belt enter RS`). This models that it takes `TonBelt` time units for from a package enters the belt until it is read by the reading station. Secondly, it takes `TonBelt` time units before `Enter belt` is enabled again, which models that packages arrives on the belt separated by at least `TonBelt` time units.

Conveyor belt domain The *Conveyor Belt* domain is modelled by two transitions, `Enter belt`, which, as we already saw, models an event that is shared with the Package domain, and the transition `Leave belt enter RS`, which models an event that is shared, both with the Package domain and with the Reading Station domain[3].

Reading station domain The *Reading Station* domain is modelled by the transition `Leave belt enter RS` already mentioned, plus the transitions, `Exit RS enter pipe`, `Read destination`, and `Read RS sensor`, and the places `Destination` and `Triggered RS sensors`. The two `Read` transitions model events, which are shared with the *Controller* domain.

Switch domain The *Switch* domain is modelled by two places and five transitions. The `Switches` place is used to model the switch positions, i.e., for each switch, whether it is turned to the left or to the right. The `Triggered sw sensors` place is used to model the sensor triggerings that occur when packages enter and exit switches. The three transitions `Exit pipe enter sw`, `Exit sw left enter pipe`, and `Exit sw right enter pipe` all model events that are shared with the Package domain. The two transitions `Read sw sensor` and `Flip` both model events that are shared with the Controller domain.

Bin domain The *Bin* domain is represented by two transitions, `Exit pipe enter bin` and `Read bin sensor` and by the place `Triggered bin sensors`.

5.4 Requirement CPN

Recall the functional requirement for the controller (Table 1), that:

[3] Note: place `Ids` has no interpretation in the problem domain: it is merely a CPN technical means to assign unique numbers to the `Package` tokens and to ensure appropriate delays between consecutive `Package` tokens.

each time a package enters the belt, the controller must ensure that the package is eventually delivered to the correct bin.

We capture this requirement through an 'optative' CPN which acts to validate behaviour, shown as the right-hand side of the top-level model of Figure 4, and detailed as the CPN model of Figure 6.

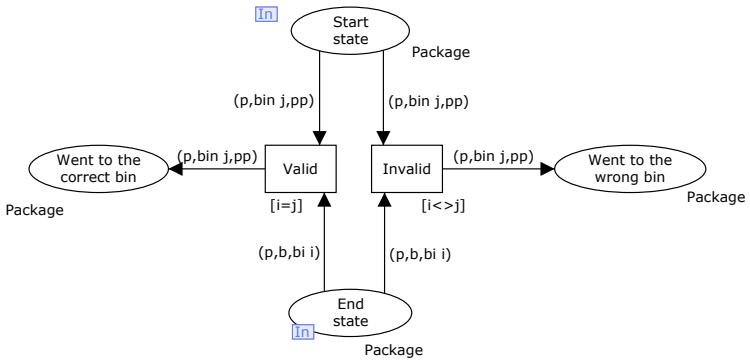

Fig. 6. Requirement CPN: requirement module of the CPN model

The uppermost place, `Start state`, is to contain tokens that correspond to a logging of all packets that have entered the belt. The lowest place, `End state`, is to contain tokens that correspond to a logging of all packets that have been delivered to bins.

For any given `Package` token, whether the correct delivery has occurred can be determined by comparing the barcode of the `Package` token in `Start state` with the bin number of the token with the same package id in the `End state` place. If the two are identical, the model indicates that the package has arrived at the correct bin, which makes the transition `Valid` enabled and, upon such an occurrence, produces a token in the `Went to correct bin` place. Wrongly delivered packages are modelled *mutatis mutandis*.

5.5 Solution CPN

As for all other domains, the controller has a boundary which consists of a number of transitions, each modelling an event that is shared with other domains. From Figure 4, it can be seen that the interface between the controller and the context is modelled by the five transitions:

- `Read destination`, corresponding to a reading at the reading station (phenomenon $read(id, dst)$);
- `Read RS sensor`, corresponding to a package leaving the reading station (phenomenon RS);

– `Read sw sensor`, corresponding to packages entering or exiting a switch (phenomena $in[j], outL[j], outR[j]$);
– `Flip`, corresponding to actuators changing the state of a switch (phenomena $left[j], out[j]$);
– `Read bin sensor`, corresponding to a package entering a bin (phenomenon $bin[i]$).

The controller gets input via the events modelled by the `Read` transitions; the input is represented by tokens on the `Destination readings` and `Sensor readings` place. The controller computes output, modelled by putting tokens on the `Actuator` place and occurrence of the `Flip` transition.

Let us now take a closer look at the internal logics of the controller; the CPN module for this is shown in Figure 7.

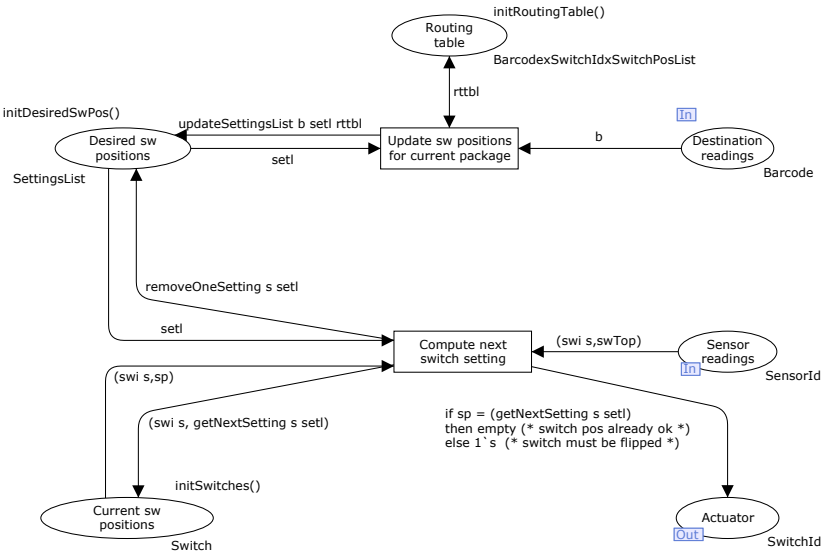

Fig. 7. Controller logic module of the CPN model

To do its job, the controller needs to keep and maintain some state information internally. This information is captured in tokens on the places `Routing table`, `Desired sw positions` and `Current sw positions`.

The marking of `Routing table` is static. It represents a routing table that says, given a barcode and a switch id, whether the switch should be turned left or right, when a package travelling to the bin determined by the barcode passes through. An example of an entry in the table is represented by the value `(bin(1),1,left)`, which determines that a package going to bin 1 must be

switched left in switch 1. The switches are are numbered from top to bottom and let to right, starting from 1. Similarly for pipes and bins.

The marking of `Desired sw positions` at any time holds a list of `left` and `right` values for each switch id. The meaning of, e.g., the value `(3,[right,left])` is that there are two packages in the routing system currently approaching switch 3, and the first of these packages should be switched to the right and the second should be switched to the left. The marking of `Desired sw positions` is updated by two transitions. The `Update sw positions for current package` transition models that the controller gets input from the reading station, which may cause, e.g., the value `(3,[right,left])` to be replaced by the value `(3, [right,left,left])`, indicating that a third package has entered the system, and that this package should be switched left at switch 3.

An occurrence of the transition `Compute next switch setting` is triggered by a token on the place `Sensor readings`, which models a sensor firing. Based on knowledge of the desired switch settings, which is contained in a token on the `Desired switch settings` place, and knowledge of the current position of the switch, which is just to be entered by the package that triggered the sensor, the switch is flipped, if necessary, represented as a token on the `Actuators` place. The list of desired switch settings on the `Desired sw positions` place is updated, e.g., the value `(3,[right,left,left])` is replaced with the value `(3,[left,left])`, when `Compute next switch setting` has occurred (following the triggering of switch 3, in this case).

It should be noted that in the current version of the model, only the top sensor of a given switch is acted upon, the readings from the bottom sensors being ignored, as can be seen by the inscription on arc going from the `Triggered sw sensors` place to the `Read sw sensors` transition in Figure 4. From here, it can also be seen that the readings from the reading station sensor and from the bin sensors are just ignored (see the `empty` inscriptions on the appropriate arcs). Because it is the triggering of the top sensors of the switches that causes the switch to be set correctly, the switches should be really fast.

6 Using CPN within POSE

The transformational nature of POSE provides a framework for arguing the adequacy of software products during their development. In this section we show how the current CPN model might be shown adequate.

Software problem transformations capture and formalise solution preserving relationships between problems: a software problem transformation transforms a single problem—the conclusion—to a set of problems—the premises—and records an argument—the *adequacy argument step*—that justifies the relationship of the premises to the conclusion.

Suppose we have problems $W, S \vdash R$, $W_i, S_i \vdash R_i$, $i = 1, ..., n$, $(n \geq 0)$ and *adequacy argument step* J, then we will write:

$$\frac{W_1, S_1 \vdash R_1 \qquad ... \qquad W_n, S_n \vdash R_n}{W, S \vdash R} \langle\langle J \rangle\rangle \tag{1}$$

to mean that:

> S is a solution of $W, S \vdash R$ with *adequacy argument* $(CA_1 \wedge ... \wedge CA_n) \wedge J$ whenever $S_1, ..., S_n$ are solutions of $W_1, S_1 \vdash R_1, ..., W_n, S_n \vdash R_n$, with adequacy arguments $CA_1, ..., CA_n$, respectively.

The search for a solution extends the tree upwards and, as in other Gentzen-style sequent calculi, problem transformation applications cascade with transformations lower in the tree producing premise problems (those above the line) that transformations at the next level up use as conclusion problems (those below the line). When there are no premise problems ($n = 0$ in the above definition) we have reached a 'leaf' node of the tree, and upwards development is complete for that part of the problem.

As the tree grows, the adequacy argument is constructed. The reader will note that we make no requirement of formality in the adequacy argument step nor, therefore, in the adequacy argument; the informality of the subject matter precludes fully formal treatment of some transformations. Indeed, in the worst case, a software problem transformation may be applied without any justification, whence we have no adequacy argument for the development as those for the premises, i.e., the CA_i, cannot be extended to the conclusion. Of course, that no justification has been given does not mean that it cannot be retrospectively added. However, a problem transformation that happens with no justification incurs the risk that no justification—and therefore no adequacy argument—will be possible. Attitudes to risk may thus influence the application of rules, and thus software development within the framework. The justification of a problem transformation is, however, necessary if we wish to argue the adequacy of a solution resulting from the development, even if the strongest possible (or practical) justification may still be quite weak.

The presented CPN development of the package router has led us to a specification. We also have *some* evidence, in the form of simulation and exploration of the state space, that the specification is very weakly adequate[4]. Here, if *SOLUTION* is the Solution CPN shown in Fig. 7, then we can claim it to be a solution to the problem:

$$\frac{}{W : CONTEXT, PRSoln : SPECIFICATION \vdash PRReq : REQUIREMENT}\ \langle\langle Justification \rangle\rangle \tag{2}$$

To be able to place the *SPECIFICATION* in front of a customer, an important component is the argument—which might be presented in any one of a number of ways—that it is adequate[5].

[4] By which we mean, essentially, that it would not convince many people of the adequacy of the solution!

[5] Of course, we did not begin this paper with the intention of developing an adequate controller, and so the versions of adequacy we mention are for illustration, and should not be taken to imply some deficiency of CPNs or Petri nets.

6.1 Arguing adequacy: validation

In the first case, we might say that we have validated the solution through 'testing,' i.e., simulation in the CPN tool. In this case the *Justification* (together with its rationale) might be something like:

> The package router controller is a (soft) real-time system; switching too late, for instance, may mean that the packages are routed to wrong bins. For this reason, we have used the CPN model to experiment with and investigate timing issues. The constants `TonBelt`, `TinRS`, `TinPipe`, and `TinSw` declared in the CPN model are used to model the number of time units a package spends on the belt, in the reading station, in a pipe, and in a switch, respectively.
> We have assumed that the values of `TinRS`, `TinPipe`, and `TinSw` are given. In contrast, we have used different values for `TonBelt`. This corresponds to that the time a package spends on the belt can be changed (this may be done perhaps by adjusting the speed of the motor that drives the physical belt). In the experiments we report on, we have `TinRS`, `TinPipe`, and `TinSw` all equal to 10, i.e., it is modelled that for a package to go through either the reading station, a pipe or a switch takes 10 time units. Thus, in the CPN model, we model that the physical movement of packages take time. In contrast, we have assumed perfect implementation technology: every action inside the controller takes zero time.

6.2 Arguing adequacy: verification

An alternative approach is to prove correctness of the specification. This leaves us with the same problem transformation, but *Justification* is now verification, with all of its inherent scalability problems:

> The main verification method for CPN is state space analysis [14]. Use of state spaces has the potential to strengthen the adequacy argument. On the other hand, the state space experiments we have carried out have not lead to a very convincing adequacy argumentation. For different values of `TonBelt`, we have generated partial state spaces in analysed them. Specifically we have checked the maximal integer bound on the `Went to the wrong bin` place. If this is zero, the controller specification is correct, for the part of the state space that is covered. If is is greater than zero, then our controller specification is not correct. In one experiment, `TonBelt` was set to 50 and we generated a state space for 16 hours. This state space had 339,359 nodes and covered a time interval on 300 time units; its executions corresponded to four packages being delivered correctly and zero packages delivered wrongly. In another experiment, `TonBelt` was set to 5 and, again, we generated a state space for 16 hours. This state space had 170,758 nodes, but covered only a time interval of 25 time units. In this time interval, no execution corresponded to any packages reaching a bin (either correct or wrong bin).

Whereas being able to simulate and explore the state space very early is a useful tool in software development for, respectively, communication with the customer and for understanding the problem, neither has been widely adopted in—mission-critical—development. One reason is that the *posit and prove* approach tends not to scale well even when it is the full specification the solution is proven against, let alone the requirements, and much work and simplification is usually needed to make the technique tractable, not all of which is benign. Moreover, the necessary informal nature of (early) requirements, of the existence of quality as well as functional requirements, and the vastness of the real-world state space, all argue against any useful formal description being available to the developer for verification. Can the work we have described here offer anything? We return to this question in the discussion and conclusions.

7 Discussion and Conclusion

In this paper, we have presented a case study in which CPN and POSE are used together. The POSE approach demands that we must represent problems, domains, and solutions, and we have described how CPN can be used to do so. CPN has proven to be a very adequate formalism in which to describe problems: it can model causal dependence and independence accurately; there are well-understood operations that correspond to the sharing of phenomena between causally independent domains. Tool support extends the use of CPN from their use in the structuring of problems to the easy use of behaviours in problems: we have shown that there is a role, if limited, for simulation in the description of a problem's solution.

In the previous section, we critiqued the use of verification and validation in the context of mission-critical development without any discussion of the alternatives that might exist under POSE. Here, we describe some further work, still needing to be done, that might provide another approach to the use of CPNs (and Petri nets) for software engineering.

The first and third author, with others, have defined a constructive technique for developing adequate solutions under POSE, even when only informal descriptions exist. The techniques are not necessarily simple: POSE can not remove the complexity of software development and its need for creative thought. Moreover, they may only lead to dead ends—in the case of under-detailed descriptions—but, coupled with other POSE techniques—they can lead to detailed specifications and co-developed adequacy arguments.

The idea, demonstrated in papers such as [25], is to progressively 'simplify' a problem by replacing domains in the problem's context whilst rewriting requirements to compensate for the removed domain's behaviour. This device has been shown to work well in some critical contexts (see, for instance, [25]). A formal approach is given in [21], based on Lai's quotient operator for CSP [18]. Lai's quotient is complex: it must compensate for the removal of an arbitrarily complex domain described as a CSP process term. CPNs (indeed, Petri nets in general) may offer a simpler, much more tractable alternative, the idea being to

consider each transition in the CPN for removal through the transformation in concert with manipulation of the requirement net. Work is required to investigate the nature of these transition transformations.

On other fronts, the first and third author, again with others, have defined a POSE pattern for critical development ([22]) which directs the efforts of the developer in particular directions at different stages of development, and leads to a shown adequate architectural basis for development (should one exist). The current process pattern requires a complex appeal to the behaviours allowed under a candidate architecture; it may be that CPNs, with their highly developed behavioural tools, can help in determining more simply the correctness of this process.

Acknowledgements

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Grants. Thanks also go to our colleagues in, respectively, the Centre for Research in Computing at The Open University, especially Michael Jackson, and in the Department of Computer Science at the University of Aarhus.

References

1. E. Best, R. Devillers, and J. G. Hall. The box calculus: a new causal algebra with multi-label communication. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 609 of *Lecture Notes in Computer Science*, pages 21–69. Springer-Verlag, 1992.
2. C. Choppy and G. Reggio. A UML-Based method for the Commanded Behaviour frame. In K. Cox, J. G. Hall, and L. Rapanotti, editors, *Proceedings of the 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF04)*, pages 27–34, Edinburgh, 2004. IEE. 24 May 2004.
3. K. Cox, J. G. Hall, and L. Rapanotti, editors. *Proc. of 1st International Workshop on Advances and Applications of Problem Frames (at ICSE 2004)*, Edinburgh, Scotland, 2004. IEE.
4. K. Cox, J. G. Hall, and L. Rapanotti, editors. *Journal of Information and Software Technology: Special issue on Problem Frames*, volume 47. Elsevier, November 2005.
5. G. Denaro and M. Pezzè. Petri Nets and Software Engineering. In Desel et al. [7], pages 439–466.
6. J. Desel, V. Milijic, and C. Neumair. Model Validation in Controller Design. In Desel et al. [7], pages 467–495.
7. J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets — Advances in Petri Nets*, volume 3098 of *LNCS*, Eichstätt, Germany, 2004. Springer.
8. M. Elkoutbi and R. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In *Proc. of the 21st Petri Nets Conf.*, volume 1825 of *LNCS*, pages 166–186, Aarhus, Denmark, 2000. Springer.
9. C. Girault and R. Valk, editors. *Petri Nets for Systems Engineering — A Guide to Modeling, Verfication, and Application.* Springer, 2003.

10. J. G. Hall, L. Rapanotti, K. Cox, and Z. Jin, editors. *Proc. of 2nd International Workshop on Advances and Applications of Problem Frames (at ICSE 2006)*, Shanghai, China, 2006. ACM.
11. J. G. Hall, L. Rapanotti, K. Cox, and Z. Jin. Proceedings of the 2nd international workshop on advances and applications of problem frames. In J. G. Hall, L. Rapanotti, K. Cox, and Z. Jin, editors, *International Workshop on Advances and Applications of Problem Frames*. ACM SIGSOFT, 2006.
12. J. G. Hall, L. Rapanotti, and M. Jackson. Problem-oriented software engineering. Technical Report 2006/10, Department of Computing, The Open University, 2006.
13. M. A. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problem*. Addison-Wesley Publishing Company, 1st edition, 2001.
14. K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1-3*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1992-97.
15. J. Jørgensen. Addressing Problem Frame Concerns via Coloured Petri Nets and Graphical Animation. In *Proc. of 2nd International Workshop on Advances and Applications of Problem Frames (at ICSE 2006)*, Shanghai, China, 2006. ACM.
16. J. Jørgensen and C. Bossen. Requirements Engineering for a Pervasive Health Care System. In *Proc. of 11th International Requirements Engineering Conf.*, pages 55–64, Monterey Bay, California, 2003. IEEE.
17. S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, NJ., 1964.
18. L. Lai and J. W. Sanders. A weakest-environment calculus for communicating processes. Research report PRG-TR-12-95, Programming Research Group, Oxford University Computing Laboratory, 1995. 03-1995.
19. L. Lavazza and V. D. Bianco. A UML-based Approach for Representing Problem Frames. In Cox et al. [3], pages 39–48.
20. L. Lavazza and V. D. Bianco. Combining Problem Frames and UML in the Description of Software Requirements. In *Proc. of 9th International Conf. on Fundamental Approaches to Software Engineering (FASE)*, volume 3922 of *LNCS*, pages 199–213, Vienna, Austria, 2006. Springer.
21. Z. Li, J. G. Hall, and L. Rapanotti. From requirements to specification: a formal perspective. In J. G. Hall, L. Rapanotti, K. Cox, and Z. Jin, editors, *Proceedings of the 2nd International Workshop on Advances and Applications of Problem Frames*. ACM, 2006.
22. D. Mannering, J. G. Hall, and L. Rapanotti. A problem-oriented approach to normal design for safety-critical systems. In *Proceedings of FASE 2007*, Lecture Notes in Computer Science, 2007. To appear.
23. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.
24. L. Rapanotti, J. G. Hall, and M. Jackson. Problem-oriented software engineering: solving the package router control problem. Technical Report TR2006/07, Centre for Research in Computing, The Open University, 2006.
25. L. Rapanotti, J. G. Hall, and Z. Li. Problem reduction: a systematic technique for deriving specifications from requirements. *IEE Proceedings – Software*, 2006. In press.
26. W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Commun. ACM*, 25(7):438–440, 1982.
27. CPN Tools. www.daimi.au.dk/CPNTools.