A PRACTICAL APPROACH TO INCREMENTAL SPECIFICATION

Charles Lakos
Department of Computer Science
University of Adelaide
Charles.Lakos@adelaide.edu.au

Glenn Lewis
School of Computing
University of Tasmania
Glenn.Lewis@utas.edu.au

Abstract The object-oriented specification of concurrent and distributed systems has tended to emphasise the aspect of substitutability at the expense of code reuse. A variety of constraints has been imposed in order to guarantee substitutability in one form or another. This paper argues that the incremental development of software specifications needs to consider substitutability in the context of code reuse. Further, the common approach of starting with an abstract specification and then progressively refining it (in some general way) means that many existing substitutability constraints are too strong. In the context of Coloured Petri Nets, we advocate the use of three specific forms of refinement — *type refinement*, *subnet refinement*, and *node refinement*. These have weaker demands for substitutability, namely that every (complete) refined behaviour has a corresponding abstract behaviour, but not necessarily vice versa. An examination of case studies in the literature suggests that this approach is applicable in practice.

Keywords: Incremental development, behavioural substitutability, subtyping, Petri Nets, refinement, concurrency.

1. INTRODUCTION

The development of formal methods for object-oriented concurrent and distributed systems has emphasised the distinction between subclassing and subtyping. According to Lalonde and Pugh [16] subclassing is concerned with code reuse, while subtyping is concerned with substitutability, as formulated by Wegner and Zdonik [22]. Note that by code reuse we mean the support for incremental change which is made possible by object-orientation, and not just the use of some code in another context without modification. Many proposals are primarily concerned with subtyping, while subclassing is a secondary consideration [1, 2, 4, 20]. No doubt this preference for subtyping is motivated by the desire to provide suitable properties which can be used in the construction and analysis of systems, while unconstrained subclassing (as in Smalltalk for example [9]) is considered to be of minimal use.

We argue that the above dichotomy is exaggerated. We observe that a primary motivation for object-orientation has always been code reuse, which does not necessarily imply a complete lack of substitutability properties, as is demonstrated by the use of assertions in Eiffel [18]. Our particular interest is in the support for incremental development of system specifications. Here, the designer tends to start with an abstract view of the system, and progressively refines that specification to add more detail. The very use of abstraction means that the designer has captured some aspects of the system (which will persist henceforth) while consciously deciding to ignore others (at least for the time being). In producing a refined version, existing behaviour is maintained (in some sense), while new behaviour is added. (The maintenance of existing behaviour is an important aid to understanding the system, and can also be used for improving analysis algorithms.) It is appropriate to use inheritance in this context, so that the refined version of a system component is derived from the abstract version. The designer does not face a stark choice between subclassing and subtyping. Rather, code is being reused and some level of behavioural compatibility is being maintained. This is also the case when an existing specification needs to be modified, such as with the evolution of protocol specifications [14]. Wegner and Zdoniks comments are particularly pertinent here:

> ... the requirements for substitutability and the associated notion of subtype and behavioural compatibility is too strong in many practical situations. ... template modification (which is at the heart of subclassing) is more powerful than subtyping as an incremental modification mechanism but also less tractable [22].

Our proposals for incremental development are presented in the formalism of Coloured Petri Nets (CPNs) which capture both state and behaviour. In contrast to elementary Petri Nets, CPNs use colour sets to achieve more succinct specifications, which is particularly important for larger systems. We identify three forms of refinement which we call *type refinement*, *subnet refinement* and *node refinement*. The constraints we impose on these forms of refinement guarantee behavioural compatibility between refined and abstract systems. Specifically, every (complete) action sequence of the refined system corresponds to an action sequence of the abstract system, and every reachable state of the refined system (following a complete action sequence) corresponds to a state of the abstract system. The correspondence is achieved by either ignoring the refined action or state components, or projecting them onto abstract components.

In this work, our primary motivation is to support the incremental development of system specifications. We are particularly concerned that our proposals should be applicable in practice. This paper informally presents the three forms of refinement in §2 by means of an example. It considers related approaches from the literature in §3, and assesses the applicability of our proposals by examining some case studies in §4. The formal definitions are found in an abbreviated form in §5 (cf. the more extended presentation with proofs of relevant properties in [13]). The conclusions and suggestions for further work are found in §6.

2. BEHAVIOUR REFINEMENT FOR PETRI NETS

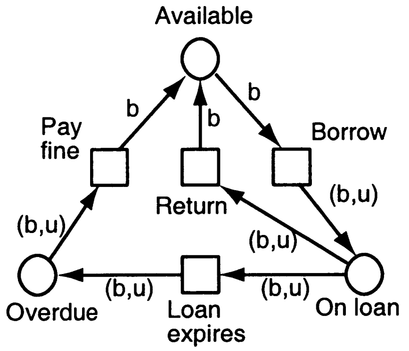
This section presents proposals for the refinement of specifications in the Petri Net formalism, and specifically Coloured Petri Nets (CPNs) [10]. The proposals are illustrated with a simple example of a library loans system which is adapted from [6].

In a CPN, state components are called *places* and are drawn as circles or ovals. Places may contain *tokens*, which consist of values of a particular type (the *colour set* of the place). State changes are called *transitions* and are drawn as squares, rectangles or bars. The arcs connecting places to transitions and vice versa indicate how transitions affect the state, their direction indicating the direction of token flow. Thus, input arcs indicate the consumption of tokens, while output arcs indicate the generation of tokens. The arcs may be annotated with the particular tokens which need to be transferred. The transition *firing modes* will determine the particular values of these annotations.

In our library example, a book will have some associated information, such as the author(s), title, publisher, etc. In a CPN, each book can be represented by a token, and this information can be captured in the token type called *Book*. Each book will be in one of several states such as available, on loan, and overdue. These states can be represented by places in the Petri Net, with the presence of a book token in a place indicating that the book is currently in that state. Thus a CPN for the library books could be as in fig. 1.

Not all the details of the net have been shown in the diagram. Thus, the colour for the token type is not indicated in full, nor is the relation between the transition firing modes and the variables inscribing the arcs. Clearly, however, the firing mode for transition *Borrow* will include sufficient information to identify the book *b* and the user *u*.

A borrower or user of the library will have some associated information, such as their name, contact details, classification of membership, etc. Again, each user can be represented by a token type called *User*, and the various states of the user can be indicated by places. In this case, it is convenient to have only one place to indicate the state of the user with transitions indicating the possible

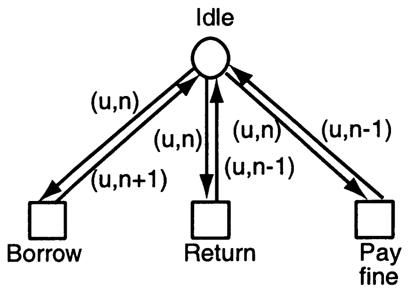


Colours:

C(Available) = Book
C(On loan) = Book × User
C(Overdue) = Book × User

C(Borrow) = Book × User
C(Return) = Book × User
C(Loan expires) = Book × User
C(Pay fine) = Book × User

Figure 1 CPN Books — the lifecycle of library books



Colours:

C(Idle) = User

C(Borrow) = User × Limit
C(Return) = User × Limit
C(Pay fine) = User × Limit

Figure 2 CPN Books — the lifecycle of library users

actions such as borrowing or returning a book and paying a fine. Thus, a CPN for the library users could be as shown in fig. 2.

The above two CPNs could be combined into a composite system by fusing the similarly-named transitions in the two nets. Clearly, more details could be specified, but the example above will be adequate for our purposes of illustrating the forms of refinement which we wish to support.

The first and simplest form of refinement, which we call *type refinement*, is to incorporate additional information in the net in the tokens and firing modes. However, each value of the refined type can be projected onto a value of the abstract type. For example, it may be desirable to introduce a further classification of books to vary the loan period. As far as the subnet for the books is concerned, this will simply involve extending the token type for the places, and extending the corresponding type(s) for the transition firing modes. The changes will affect the firing of the *Loan expires* transition, especially in the composite system. It will certainly be the case, however, that if there is a be-

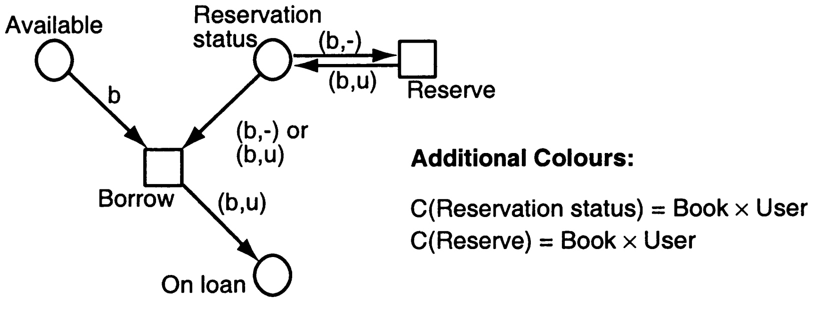


Figure 3 CPN Books — the library books lifecycle refined to include reservations

haviour of the refined system (i.e. a sequence of transition occurrences), then there will be a corresponding behaviour of the abstract system (obtained in this case by projection). This is the generic behavioural constraint which we require for acceptable refinements.

The second form of refinement, which we call *subnet refinement*, is to augment a subnet with additional places, transitions and arcs. (We also classify as subnet refinement the extension of a token type or mode type to include extra values which are independent of previous processing. Here, these values of the extended type are not projected onto values of the abstract type but are ignored in the abstraction.) For example, it may be appropriate to cater for the reservation of books. In this case, we would add a place to hold the reservation status for each book, and the transition *Borrow* would only fire if there were a compatible reservation status on the book for the given user. (We use the value *(b,-)* to indicate that no-one has a reservation for book *b*, and the value *(b,u)* to indicate that user *u* has a reservation on book *b*.) The modified part of the *Books* subnet is given in fig. 3. Again we satisfy the constraint that if there is a refined behaviour, then there is a corresponding abstract behaviour (obtained in this case by restriction), but not necessarily vice versa.

The third form of refinement, which we call *node refinement*, is to replace a place (or transition) by a place (or transition) bordered subnet. In this paper, we advocate the use of canonical forms of such refinements. The basis for a canonical place refinement is given in fig. 4.

It has separate input and output border places — in this case there are two of each. Each input (output) border place may have more than one incident input (output) arc from (to) the environment. Each input border place has an associated *accept* transition which will transfer tokens from the border place to an internal place, here called *buf*. Similarly, each output border place has an associated *offer* transition which will transfer tokens from place *buf* to the output border place. All the border places and the place *buf* have the same token

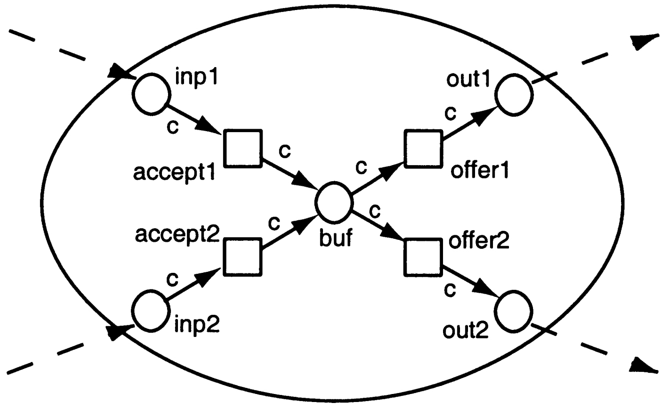


Figure 4 Canonical place refinement

type, which is also the mode type shared by the *accept* and *offer* transitions. None of these transitions has a guard to constrain the flow of tokens. Clearly, the abstract marking of such a canonical place refinement is given by the sum of tokens in the border places and the internal place *buf.*

An arbitrary place refinement will be of the form of the basis of a canonical refinement (as above) augmented by subnet refinement which extends the *accept* and *offer* transitions.

In our running example, such an incremental change might be the identification of the details of processing a book once it has been returned. In other words, the place *Available* might be replaced by a subnet which takes into account the delay in reshelving a book, the possibility of repairs, etc. The node refinement for this place is shown in fig. 5.

Note that this has one input border place called *Returned* and one output border place called *On shelves.* The *Accept* and *Offer* transitions, together with the internal place *Buf* constitute the basis of the canonical place refinement. Further activity is achieved by the subnet refinement which extends transitions *Accept* and *Offer.* (Note that all component places and transitions have the same token/mode type, i.e. *Book*, which is that of the original place *Available.*) The subnet refinement retains the identity of the books, and hence this information determines the abstract marking of the subnet. Thus, the place *Buf* is redundant, since its marking is equivalent to the sum of markings of places *For checking*, *Under repair*, *Irredeemable*, *For shelving.* (In our experience, the place *Buf* is commonly redundant in such place refinements.) Further, this abstract marking is not modified by the various actions internal to the subnet. Clearly, a refined behaviour of the net will have a corresponding abstract behaviour, though the reverse will not necessarily be the case. For example, it may be that a book is

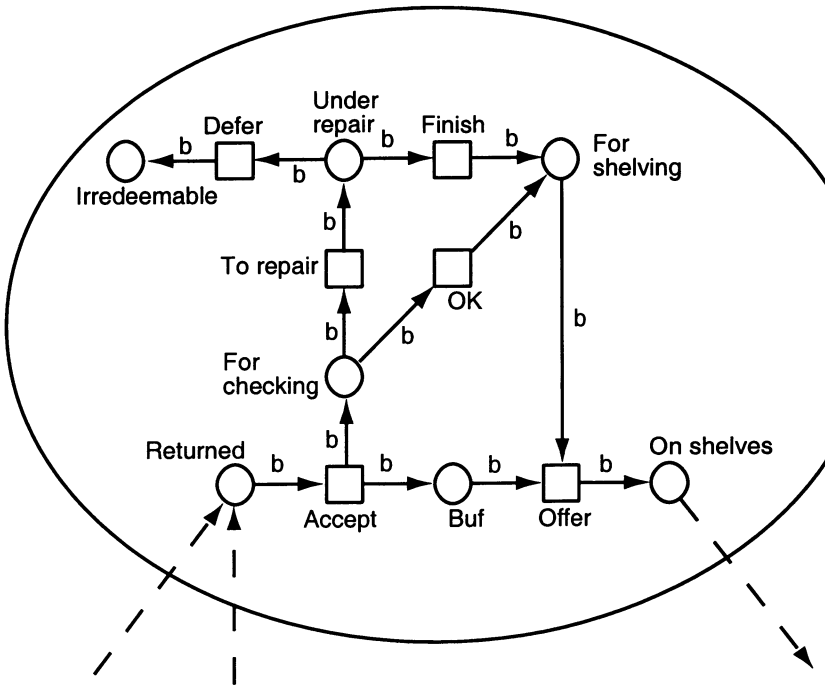


Figure 5 Subnet indicating the processing of returned books

damaged to such an extent that its return to the shelves would be indefinitely delayed, in which case further borrowing of that book would be disallowed.

For transition refinement, the canonical basis is given in fig. 6. It has separate input and output border transitions — in this case there are two of each. Each input (output) border transition may have more than one incident input (output) arc from (to) the environment. Each input border transition has an associated place *recd*, which receives a token equal to the abstract firing mode, when the input border transition has fired with that mode. The transition *switch* can fire when all the input border transitions have fired (with the matching abstract firing mode), thereby completing the input phase. It removes the matching tokens from the *recd* places and puts corresponding tokens into all the *send* places. There is one such *send* place associated with each output border transition. Once such a token is available, the output border transition can fire (with the same abstract firing mode). Initially, all the *recd* and *send* places are empty. The abstract firing of the transition refinement commences with the firing of any of the input border transitions and is *completed* when all the matching output border transitions have fired and the *recd* and *send* places are again empty. Only such complete firing sequences will have corresponding abstract

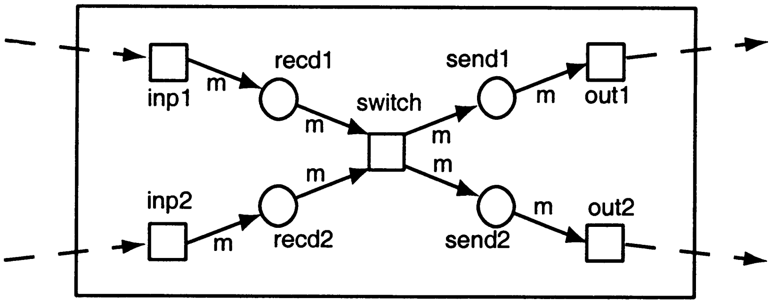


Figure 6 Canonical transition refinement

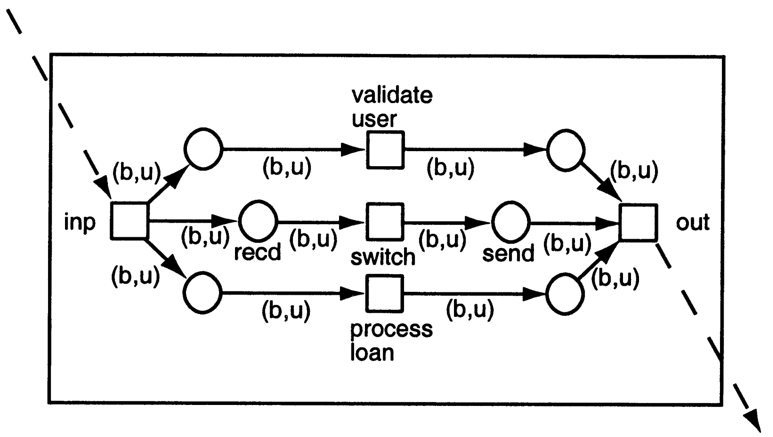


Figure 7 Refined Borrow transition

firing sequences with matching abstract states. The canonical construction ensures that input border transitions fire before the corresponding output border transitions, ensuring the enabling of the corresponding abstract transition.

An arbitrary transition refinement will be of the form of a canonical refinement (as above) together with a subnet refinement which augments the border transitions. In our running example, we may wish to refine the *Borrow* transition to reflect the component activities of checking the student identity and processing the book. This might be achieved in the subnet of fig. 7.

All the component places and transitions have the same token/mode type, i.e. *Book* × *User*, which is the mode type of the original *Borrow* transition. (While this happens to be the case in this particular example, it is not necessarily so.) Note that the transition *validate user* may fail to fire (because the user is not acceptable). In this case, the abstract firing of this subnet will never complete, and hence such an incomplete refined activity will have no corresponding abstract activity with matching abstract state.

Even though the above three forms of refinement can be clearly identified and analysed in isolation, they will commonly be used in combination in practical applications.

The above proposals differ from those typically found in the context of High-Level Petri Nets (such as CPNs [10]). There, abstraction is used to aid the process of developing a Petri Net model, with the abstraction subsequently being discarded when the model is simulated or analysed [21]. Thus CPNs, as formalised by Jensen [10] and implemented in the Design/CPN tool [11], provide *substitution transitions* for building Hierarchical Coloured Petri Nets (HCPNs). Substitution transitions are like macros or textual substitution — they maintain structural compatibility, but there is no concept of abstract behaviour. The semantics of the construct are defined in terms of textual (or graphical) substitution. This means that even if the designer has in mind a notion of abstract firing, there is no way to capture this formally in the model, either as an aid to understanding or as a hint to improve analysis.

Our proposals are therefore more constrained than substitution transitions, since refinements must satisfy certain properties. This has the advantage of ensuring that refinements are behaviourally consistent with the abstractions. This is valuable for understanding a hierarchical model and also has benefits for analysis. Current indications are that these forms of refinement are applicable in practice for the incremental development of system specifications (see §4).

3. RELATED APPROACHES

The proposals of §2 require behavioural compatibility between abstraction and refinement, but not the strong substitutability of other approaches. Some of these approaches are considered in this section and compared with our own proposals, with particular emphasis on their relevance for incremental specification.

Nierstrasz proposes a form of substitutability called *request substitutability preorder* [20]. This is motivated by the notion of active objects providing a service to their environment. A service is determined by the service requests which can be satisfied and hence the name of the preorder. One object is substitutable for another if it supports the same service. Nierstrasz observes that the possible traces indicate the acceptable sequences of requests, but not the sequences that are *necessarily* acceptable, and hence the notion of failures is adopted. Thus the traces of the refined system must include the traces of the original, and if a trace of the original system is performed by the refined system, then its possible failures must be a subset of the corresponding failures of the original.

The motivation for this approach is clearly that of substitutability — a server is replaced by another offering the same service. It is not concerned with reuse of specifications or the development of a server specification, where an abstract view may support a sequence of requests, but where a refined view may only accept that sequence if additional conditions are satisfied, perhaps with additional service parameters. It is worth noting that since failures equivalence is undecidable, Nierstrasz proposes that services be restricted to those that can be specified by a finite state protocol. While our proposals do allow the introduction of a deadlock in a refinement, nevertheless the conditions for behavioural compatibility can be statically checked (see §5).

Bowman et al [4] consider behavioural subtyping in LOTOS. They are dissatisfied with the above proposal of Nierstrasz because they wish to maintain compatible behaviour even when the environment attempts to make a request which is not part of the agreed service. In such a situation, the original server would deadlock, and this behaviour would be expected of a compatible substitute. Such behavioural compatibility is satisfied by reduction — one system *reduces* another if it has a subset of traces and if, after a matching trace, it has a subset of refusals. The problem with this is that it does not allow the refined system to include additional methods, as is common in subclassing in object-oriented languages. To counter this, they introduce the notion of undefined behaviour which corresponds to calling an undefined method in a sequential system. In the context of LOTOS, undefined behaviour is a version of chaos which can accept or refuse any request. Now, the addition of a method in a refinement will result in defined rather than undefined behaviour (as in the original). Thus, the refinement represents a reduction of the original, thus maintaining the desired properties (identified above).

This approach has the strange result of saying that the process *i; a; stop [] b; c; stop* can be refined by *a; stop [] c; stop* and by *a; stop [] b; a; stop*. Clearly, the internal action *i* can be taken, thus committing the system to perform *a; stop*. In our proposal, the refined system could insist on taking this option, but then the other alternative *b; c; stop* would not be available, and certainly would not be available as *c; stop* or *b; a; stop*.

One of the motivations for the above approach is claimed to be incremental system development, but it is unclear how this is achieved since the only concern seems to be a strongly constrained form of substitutability. The specific example cited is that of a trader in the context of CORBA, TINA or ODP. The trader identifies a server which will respond to the desired requests, and as long as this functionality is supported, additional functionality may be present in the chosen server. In this context, it is not clear why Nierstrasz' proposal is inadequate, i.e. why the refinement of the server should also reject requests which cannot be handled by the original service specification.

Another approach, which is applied in the context of workflow modelling is that of van der Aalst and Basten [1]. Here, the refined system can add methods and is required to be bisimilar with the original, provided that the added methods are either blocked or hidden. The implication of this approach is that the occurrence of additional methods voids any requirements on behaviour. Thus, there is a strong requirement of consistency provided the new methods are hidden in some way, but no requirement of consistency if the new methods are visible. The latter seems to be too unconstrained for incremental development. Note that this approach corresponds to the provision of subnet refinement (in our terminology) but does not even cater for type refinement, where additional methods may be refined versions of existing ones. This would require relabelling, as in the following proposal.

Balzarotti et al [2] present a range of choices for the semantics of inheritance. They generalise earlier proposals with the possibility of renaming methods. This allows a refinement to support the methods of the original but under a different name. This is a useful feature, but is not general enough to support our notion of type refinement, since the relabelling is given by a bijective function (and not simply surjective). The strongest preorder is called *strong substitutability* which is the protocol inheritance of van der Aalst and Basten (where additional methods are blocked and bisimilarity is required of the result). Two weaker forms of preorder are *strong substitutability with renaming* (i.e. the above approach with the possibility of renaming methods) and *weak substitutability* (which is the general form of lifecycle inheritance proposed by van der Aalst and Basten, where added methods may be blocked or hidden). Weaker than both of these is the form of preorder called *weak substitutability with renaming*, which is derived in the obvious way. The above preorders are all based on action observability and are considered appropriate for subtyping. By contrast, they propose an *ST-preorder* based on the observability of states, which is considered appropriate for reuse of specifications. This has a more flexible renaming of states (than for actions) since there is an injective map from the OLST algebra of the abstraction into the refinement. However, this is still not as general as required by our proposal for type refinement where multiple refined states may map onto a single abstract state.

In other words, they acknowledge the difference between substitutability and code reuse, but provide different criteria for each. As usual, substitutability is not associated with code reuse nor with inheritance.

In summary, the above approaches support our contention that the emphasis of the study of relations between concurrent systems has been on substitutability rather than on code reuse, and therefore they have not been particularly suited to incremental development.

4. CASE STUDIES OF REFINEMENT

In this section we examine a number of case studies to assess the applicability our proposals. The case studies do not necessarily use an object oriented formalism, but the methodology applied does involve abstraction and incremental development. Note that some are formulated using the Design/CPN tool [11] which supports substitution transitions as a means of abstraction but without behavioural constraints.

4.1. DESIGNING AND VERIFYING A COMMUNICATIONS GATEWAY USING CPNS

Floreani et al have used the Design/CPN tool [11] to specify and verify a gateway between a narrowband packet radio network and a Broadband-ISDN network [7, 8]. The design of this gateway is complex, partly due its distributed nature, and therefore, although the CPN formalism does not directly support incremental development, the methodology employed in the case study involves first developing and verifying an abstract specification of the system, then refining this specification.

In this abstract model, Floreani defines a colour set containing the information required for setting up a call, and a colour set with this information plus addressing information. It is clear that this latter colour set is a type refinement of the former.

As noted above, the substitution transitions supported by Design/CPN are like macros — they maintain structural compatibility, but not behavioural compatibility. In both the abstract and refined models of this case study, substitution transitions are extensively used. For example, substitution transitions are used in the abstract model for sending (receiving) call information to (from) the gateway. These substitution transitions satisfy our requirements for transition refinement, and hence the explicit use of transition refinement would increase the clarity of the model and guarantee behavioural compatibility between refinement and abstraction.

The abstract model also defines the interfaces to the gateway and its internal structure — the gateway is represented by a single place. This was done to ensure correct behaviour of the interfaces before considering the behaviour of the gateway. (This is exactly the situation we see as typical of incremental development.) In the refined model, the place representing the gateway is replaced with a subnet. This subnet handles sending, receiving, and gateway call control. In the abstract model, when a connect indication is made or setup requested the appropriate token will always be accessible from the gateway place. This is not the case in the refined model, where a connect indication may be requested and fail because resources are not available, or if the destination address cannot be found. Similarly, a connect setup request may fail because not enough resources are available. Hence the refined net fits our proposals for place refinement (see §2), but does not satisfy the strong substitutability requirements of the proposals in §3.

4.2. INCREMENTAL MODELLING OF THE Z39.50 PROTOCOL

Another case study has explicitly considered the incremental modelling of the Z39.50 Protocol for Information Interchange [14]. The motivation for the incremental modelling was to capture the structure more clearly and to investigate whether such an approach could be used to capture the protocol evolution (from the 1992 to the 1995 version). Thus, the protocol entities were specified using a minimal message format which could then be refined at a later stage. This corresponds to the use of type refinement in our proposals.

The provision of access control was also captured by the incremental modification of the basic request service. Access control introduces an access rights challenge into the middle of the normal request-response interaction. Inadequate access rights will lead to a different response from the server. Again, this possibility is compatible with our proposals for refinement but not with the more demanding proposals of §3.

4.3. FIELDBUS PROTOCOL

To meet new challenges in factory automation and process control, new local area network architectures, called Fieldbus Networks, have been proposed. The proposed architectures are hierarchical, where floor level devices (sensors and actuators), at low speed, are linked to their controlling devices and controlling devices can be linked among themselves at higher levels and higher speeds. The ISA/DLL protocol has been proposed to handle such distributed control, and has been modelled using Design/CPN [19]. The top-level view of the system is shown in fig. 8.

All the transitions in that view are substitution transitions and capture the logic of the various devices. All the places are simple places which are used to convey delegation of commands and associated time slices to the devices. The *LAS-Model* captures the logic of the *Link Access Scheduler* with regard to arbitrating the synchronous and asynchronous delegation of commands and time slices to the various devices. The *LAS-DLE* is the *Data Link Element* for the *LAS* which therefore captures its own requirements for time slices. Three instances of sensors (which return information about device status) are indicated, as are three instances of actuators (which receive data to modify the device status).

Time slices are allocated regularly to the devices according to their fixed requirements. This is known as the synchronous phase. Time is also allocated

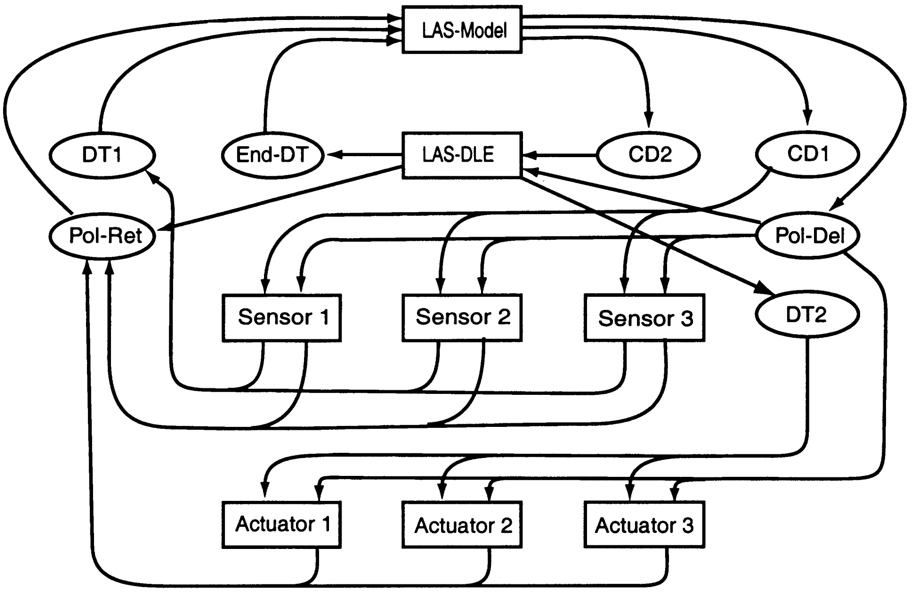


Figure 8 Abstract view of the fieldbus protocol

as available and as required for miscellaneous tasks such as data collection and error reporting. This is known as the asynchronous phase. When the *LAS-Model* wishes to perform the synchronous token delegation for a sensor, it deposits a token in the place *CD1* (for *Compel Data 1*) and then waits till the sensor relinquishes control and returns the relevant status information by depositing a token in the place *DT1* (for *Data 1*). The synchronous token delegation for an actuator only requires data to be sent (i.e. no response is expected) and hence is achieved by the *LAS-Model* depositing a token in the place *CD2* (for *Compel Data 2*), whereupon the *LAS-DLE* deposits a token in the place *DT2* (for *Data 2*) which is consumed by the relevant actuator. The *LAS-DLE* also deposits a token in the place *End-DT* to indicate the time when the actuator will be ready.

When the *LAS-Model* wishes to perform the asynchronous token delegation for sensors, actuators, or even itself, it deposits a token into the place *Pol-Del* and waits till a token is returned into the place *Pol-Ret*. Each device has a rather complex subnet for computing the time consumed out of the asynchronous time delegation and the possible further requests for time.

For our purposes, the key point is to recognise that the above abstraction captures a lot of information about the behaviour of the system. Sensors and actuators have the notion of abstract firing in two different sets of modes — one set of modes relates to the synchronous token delegation, and one set of modes relates to the asynchronous token delegation. Each one of these consumes a token and produces a token after some internal activity. (Actuators are slightly different in only consuming a token during the synchronous token delegation.) Thus, the substitution transitions for sensors and actuators basically satisfy our criteria for transition refinement. The abstract view will probably allow an arbitrary order of token delegation, while the refinement will constrain that order by the internal logic of the various components.

However, as presented in the original paper, these substitution transitions exhibit further complexities. Firstly, they share information via place fusion. This is not supported by our definitions, but the same effect could be achieved by passing this information via additional interface places. Secondly, the system incorporates the notion of simulated time so that simulations can collect performance results. The incorporation of time into our proposals is a matter for further study.

In summary, the above examples (as well as others we have examined [15]) illustrate the prevalence of incremental development of formal specifications. Our assessment is that our proposals are widely applicable, and their use would help to clarify the models and guarantee the behavioural compatibility which is often in the mind of the developer.

5. FORMAL DEFINITIONS OF CPN REFINEMENT

In this section we present the formal definitions of refinement as considered above. These definitions are abbreviated due to the constraints of space but an extended version can be found elsewhere [13]. CPNs are defined in §5.1, and CPN morphisms in §5.2. We identify the notion of *system morphisms* to capture the notion of refinement with behavioural compatibility, in contrast to the more traditional *net morphisms* which (only) specify structural constraints. The definition of our proposed refinements is given in §5.3.

5.1. FORMAL DEFINITIONS OF COLOURED PETRI NETS

We define Coloured Petri Nets in the context of a *universe of non-empty colour sets* Σ with an associated *partial order* $<:\ \subseteq \Sigma \times \Sigma$ which is derived from type compatibility in the theory of object-oriented languages [5]. $X <: Y$ means that the values of X can be used in contexts expecting values of Y, and typically it means that X has extra data components over Y. In this case we assume the existence of a (polymorphic) projection function Π_Y from the values of X to those of Y (which do not appear in any proper subtype). For our purposes, we only make use of the fact that values of X are also values of Y.

Given a universe of colour sets Σ, we define the functions over Σ as $\Phi\Sigma = \{X \to Y \mid X, Y \in \Sigma\}$, the multisets over X as $\mu X = \{X \to \mathbb{N}\}$, and the sequences over X as $\sigma X = \{x_1 x_2 \ldots x_n \mid x_i \in X\}$. (We usually name sequences with an asterix suffix.)

Definition 5.1. A *Coloured Petri Net* N is a tuple
$N = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ where:

a. P = set of places

b. T = set of transitions, with $P \cap T = \emptyset$

c. A = set of arcs, with $A \subseteq (P \times T) \cup (T \times P)$

d. C = colours of places and (modes) of transitions, i.e. $C : P \cup T \rightarrow \Sigma$

e. E = arc inscriptions with $E : A \rightarrow \Phi\Sigma$ where $E(p,t), E(t,p) : C(t) \rightarrow \mu C(p)$

f. $\mathbb{M}$ = set of markings, i.e. $\mathbb{M} = \mu\{\langle p, c\rangle \mid p \in P, c \in C(p)\}$

g. $\mathbb{Y}$ = set of steps, i.e. $\mathbb{Y} = \mu\{\langle t, c\rangle \mid t \in T, c \in C(t)\}$

h. M_0 = initial marking, $M_0 \in \mathbb{M}$

Note that there is at most one arc in each direction for any $\langle$place,transition$\rangle$ pair and that the effect of an arc is given by the arc inscription in conjunction with a particular transition firing mode. The markings of N are multisets of $\langle$place,colour$\rangle$ pairs, while the steps are multisets of $\langle$transition,colour$\rangle$ pairs. While these are derivative quantities, they are included in the tuple so that later it will be clear that morphisms map markings and steps to markings and steps respectively.

The above definition is, to all intents and purposes, equivalent to the common definition [10]. It does not include a guard function defined on transitions, but the same effect is achieved by limiting the colour set associated with the transition.

Definition 5.2. For a CPN N, $x \in P \cup T$, $X \subseteq P \cup T$ we define :

a. *the inputs of* x, ${}^\bullet x = \{y \in P \cup T \mid \langle y, x\rangle \in A\}$

b. *the outputs of* x, $x^\bullet = \{y \in P \cup T \mid \langle x, y\rangle \in A\}$

c. *the border of* X, $bd(X) = \{x \in X \mid \exists y \in P \cup T \setminus X : y \in {}^\bullet x \cup x^\bullet\}$

d. *the environment of* X, $env(X) = \{y \in P \cup T \setminus X \mid \exists x \in X : y \in {}^\bullet x \cup x^\bullet\}$

Thus, the border of a set of nodes X, are those nodes connected to other nodes not in X, and those other nodes constitute the environment of X.

Definition 5.3. The *incremental effects* $E^+, E^- : \mathbb{Y} \to \mathbb{M}$ of the occurrence of a step Y are given by:

a. $E^-(Y) = \sum\limits_{\langle t,m\rangle \in Y} \sum\limits_{\langle p,t\rangle \in A} \{p\} \times E(p,t)(m)$

b. $E^+(Y) = \sum\limits_{\langle t,m\rangle \in Y} \sum\limits_{\langle t,p\rangle \in A} \{p\} \times E(t,p)(m)$

The enabling and firing of steps and sequences is defined in the usual manner:

Definition 5.4. For a net CPN, N, a step $Y \in \mathbb{Y}$ is *enabled* in marking $M \in \mathbb{M}$, written $M[Y\rangle$, if $M \geq E^-(Y)$. If a step $Y \in \mathbb{Y}$ of CPN N is enabled in marking $M_1 \in \mathbb{M}$, it may *fire* leading to marking $M_2 \in \mathbb{M}$, written $M_1[Y\rangle M_2$ with $M_2 = M_1 - E^-(Y) + E^+(Y)$. A step sequence $Y^* = Y_1Y_2 \cdots Y_n \in \sigma\mathbb{Y}$ of a CPN N is enabled in marking $M_1 \in \mathbb{M}$ and may occur, leading to marking $M_2 \in \mathbb{M}$, written $M_1[Y^*\rangle M_2$ if there exists intermediate markings $M'_2, M'_3, \ldots, M'_n \in \mathbb{M}$ such that $M_1[Y_1\rangle M'_2$, $M'_i[Y_i\rangle M'_{i+1}$ for $i \in 2, \ldots, n-1$, and $M'_n[Y_n\rangle M_2$. The set of *reachable markings* $\mathbb{M}_R \subseteq \mathbb{M}$ is given by $\mathbb{M}_R = \{M \in \mathbb{M} \mid \exists Y^* \in \sigma\mathbb{Y} : M_0[Y^*\rangle M\}$, and the set of *enabled step sequences* $\mathbb{Y}^*_E \subseteq \sigma\mathbb{Y}$ is given by: $\mathbb{Y}^*_E = \{Y^* \in \sigma\mathbb{Y} \mid \exists M \in \mathbb{M} : M_0[Y^*\rangle M\}$

Definition 5.5. For CPN N step $Y \in \mathbb{Y}$ is *realisable* by $Y^* \in \sigma\mathbb{Y}$ in marking $M_1 \in \mathbb{M}$ leading to marking $M_2 \in \mathbb{M}$ if $M_1[Y^*\rangle M_2$ and $\sum\limits_{y \in Y^*} y = Y$.

Note that if a step Y is enabled in marking M, then it is realisable by Y.

5.2. THE FORMAL DEFINITION OF CPN MORPHISMS

This section defines morphisms between CPNs and distinguishes between net morphisms (which respect structure) and system morphisms (which respect behaviour).

Definition 5.6. A *net morphism* $\phi : N \to N'$ is a mapping from N to N' which is *structure-respecting*, namely:

a. ϕ is surjective with respect to P',T',A'

b. $\forall \langle x,y\rangle \in A \cap (P \times T) : \phi(x) = \phi(y) \;\vee\; \langle \phi(x), \phi(y)\rangle \in A' \cap (P' \times T')$

c. $\forall \langle x,y\rangle \in A \cap (T \times P) : \phi(x) = \phi(y) \;\vee\; \langle \phi(x), \phi(y)\rangle \in A' \cap (T' \times P')$

This is the common definition of net morphism (albeit with various additional constraints) which is primarily concerned with respecting the adjacency properties of the net [3, 21]. It does not constrain behaviour (except indirectly) — in fact, sets of markings and steps are not normally included. It also does not encompass restriction on places and transitions, i.e. where selected places and transitions (and their associated arcs) are ignored.

We only consider morphisms which are surjective with respect to P', T', and A'. (This means that every abstract component is the image of some refined component(s). In other words, a refinement never deletes abstract components.) Consequently, we can define the preimage of every node.

Definition 5.7. Given a morphism $\phi : N \rightarrow N'$ and $x' \in P' \cup T'$ we define the *preimage* by $N_{x'} = \phi^{-1}(x')$ and write
$N_{x'} = \langle P_{x'}, T_{x'}, A_{x'}, C_{x'}, E_{x'}, \mathbb{M}_{x'}, \mathbb{Y}_{x'}, M_{0x'} \rangle$.

In order to define the behaviour respecting properties of a system morphism, it is desirable to consider complete steps, since the firing of multiple transitions in the refinement may correspond to the firing of one transition in the abstraction.

Definition 5.8. Given a morphism $\phi : N \rightarrow N'$, a step Y of N is *complete* if $\forall t' \in T' : \forall t \in bd(N_{t'}) : \phi(\{t\} \times Y(t)) = \{t'\} \times \phi(Y)(t')$

Thus a step is complete if all border transitions occur with matching modes (which also match the mode occurrence of the corresponding abstract transition).

Definition 5.9. A *system morphism* $\phi : N \rightarrow N'$ is a mapping from N to N' which is *behaviour-respecting*, namely:

a. ϕ is surjective with respect to P', T', A'

b. ϕ is linear and total over both $\mathbb{M}$ and $\mathbb{Y}$

c. $\forall M \in \mathbb{M}_R : \forall Y \in \mathbb{Y} :$
Y is complete and realisable as $Y_1 Y_2 \ldots Y_n$ at marking $M \Rightarrow$
$\phi(Y)$ is realisable as $\phi(Y_1)\phi(Y_2)\ldots\phi(Y_n)$ at marking $\phi(M)$.

d. $\forall M \in \mathbb{M}_R : \forall Y \in \mathbb{Y} : Y$ is complete $\Rightarrow$
$\phi(M + E^+(Y) - E^-(Y)) = \phi(M) + \phi(E^+)(\phi(Y)) - \phi(E^-)(\phi(Y))$

Note that if the refined step is complete then its realisation can be used to derive the realisation of the corresponding abstract step, by projecting or restricting each component step. The modified rule of part (d) for the effect of a refined step (cf. [23]) is used since we cannot consider the component steps (of its realisation) in isolation. Thus, part (c) guarantees the enabling of the abstract sequence, while part (d) captures its overall effect.

The above definition clarifies what we mean by behaviour-respecting, also called *behavioural compatibility*. This kind of morphism is called a system morphism in line with the common Petri Net distinction of a net concerning the structure and a system including behaviour. This definition captures the requirement that we identified earlier, namely that a complete refined behaviour has a corresponding abstract behaviour.

Given two system morphisms $\phi_1 : N \dashrightarrow N'$ and $\phi_2 : N' \dashrightarrow N''$ their composition $\phi = \phi_2 \circ \phi_1 : N \dashrightarrow N''$ is a system morphism (see the proof in [13]). This means that we can combine the different forms of refinement (considered in the subsequent section) and we still have a system morphism.

5.3. CPN REFINEMENTS

We now consider each of the proposed kinds of refinement. In each case, the terminology reflects the way the incremental change is used, i.e. from abstract to refined, but the morphism is always a mapping from refined to abstract.

5.3.1 Type refinement.

The first, and perhaps simplest, form of refinement is to retain the structure of the net without modification but to replace some (or all) of the token and mode types by subtypes. Given our formulation of CPNs, where the arc inscriptions are functions (from modes to token multisets), it may not even be necessary to change the arc inscriptions, provided that they are given by polymorphic functions. The distinctive thing about type refinement is that there is a projection function from subtype to supertype so that every refined state or action has a corresponding abstract state or action.

Definition 5.10. A morphism $\phi : N \dashrightarrow N'$ *captures a type refinement* if:

a. ϕ is an identity function on P, T, A, i.e. $\forall p \in P : \phi(p) \in P$, etc.

b. $\forall x \in P \cup T : C(x) <: \phi(C)(x)$

c. $\forall x \in P \cup T : \forall c \in C(x) : \phi(1`\langle x, c\rangle) = 1`\langle x, \Pi_{\phi(C)(x)}(c)\rangle$

d. $\forall \langle p, t\rangle \in A : \forall \langle t, c\rangle \in \mathbb{Y} :$
$\phi(E^-(1`\langle t, c\rangle))(p) = \Pi_{\phi(C)(p)}(E(p,t)(c)) = \phi(E)(p,t)(\Pi_{\phi(C)(t)}(c))$
$\forall \langle p, t\rangle \in A : \forall \langle t, c\rangle \in \mathbb{Y} :$
$\phi(E^+(1`\langle t, c\rangle))(p) = \Pi_{\phi(C)(p)}(E(t,p)(c)) = \phi(E)(t,p)(\Pi_{\phi(C)(t)}(c))$

Note that there is no change to the structure of the net — to places, transitions and arcs, but the colours associated with the places and transitions are consistently subtyped. The use of the appropriate projection functions gives the corresponding abstract marking and step for every refined marking and step, and the projected effect of a mode or step is the same as the effect of the projected mode or step.

It is a simple matter to prove that a morphism which captures a type refinement is a system morphism (see [13]). It is important to note that the conditions required by such a morphism can be statically checked. In §2 we gave an example of type refinement, where the book token type was extended with information about the kind of loan.

5.3.2 Subnet refinement or extension. The second form of refinement is to add net components — places, transitions and arcs, or even additional token or mode values. As a morphism (from refined to abstract nets), this would be called a restriction, since net components are being discarded or ignored. Where token or mode types are extended, then in contrast to §5.3.1, abstraction does *not* project the additional refined values onto abstract values but rather ignores them. (In the equivalent unfolded PTN, this is the same as ignoring places, transitions, and arcs.) This does not satisfy the structure-respecting requirements for a net morphism (def 5.6), but is does qualify as a system morphism (def 5.9).

Definition 5.11. A morphism $\phi : N \nrightarrow N'$ *captures a subnet refinement* if:

a. The net structure is restricted, i.e $\forall p \in P : \phi(p)$ is defined $\Rightarrow \phi(p) \in P$ and similarly for T, A.

b. $\forall x \in \phi(P) \cup \phi(T) : C(x) \supseteq \phi(C)(x)$

c. $\forall x \in P \cup T : \forall c \in C(x) : \phi(1\text{`}\langle x, c\rangle) = 1\text{`}\langle x, c\rangle$ if $x \in \phi(P) \cup \phi(T)$ and $c \in \phi(C(x))$, otherwise $\emptyset$.

d. $\forall Y \in \mathbb{Y} : \phi(E^+(Y)) = \phi(E^+)(\phi(Y))$ and $\phi(E^-(Y)) = \phi(E^-)(\phi(Y))$

Note that the sets of places, transitions, and arcs, may be restricted by ϕ, and that the colours associated with retained places and transitions may be restricted by ϕ. Again, the morphism is defined for all markings and steps and the restricted incremental effect of the step is the same as the incremental effect of the restricted step. This implies that ignored components can refer to, but cannot permanently modify, retained components.

Again, it is not difficult to prove that a morphism which captures a subnet refinement is a system morphism (see [13]). It is important to note that the condition for such a morphism can also be statically checked. In §2 we gave an example of subnet refinement, where the subnet was augmented to cater for book reservations. The abstract behaviour of the net was restricted by the refinement — if a book can be borrowed given the possibility of reservations (the refined case), then it can also be borrowed if reservations are ignored (i.e. the abstract case).

5.3.3 Node refinement. The third form of refinement is the replacement of a place (transition) by a place (transition) bordered subnet. We refer to this as node refinement to distinguish it from the other forms of refinement being considered, even though traditional Petri Net theory simply refers to it as refinement [3]. The desirable properties of node refinement for CPNs have been considered elsewhere [12]. For behavioural consistency, it was argued that the subnet which refines a place should have the notion of an abstract marking, and the subnet which refines a transition should have the notion of abstract firing. These constraints were captured by requiring the morphism to be structure-respecting, colour-respecting, marking-respecting and step-respecting. A *structure-repecting morphism* is a net morphism of def 5.6. A *colour-respecting morphism* is one where the colours of border places (transitions) of a refined component match the colours of the corresponding abstract component, and where the flow of tokens across the refined node boundary matches the flow of tokens for the abstract node. A *marking-respecting morphism* determines the marking of an abstract place from the marking of the corresponding place refinement. Further, this abstract marking is not modified by internal actions of the refined node. Finally, a *step-respecting morphism* is one where a step sequence of a transition refinement maps into a firing of the corresponding abstract transition provided the border transitions occur with matching firing modes.

Here, we have proposed canonical place and transition refinements, which satisfy the above constraints and are quite general (see [13]). In any case, it turns out that the morphism for a node refinement is both a net morphism and a system morphism.

Given the length of the formal definitions for canonical place and transition refinements, we omit them here but refer the reader to the informal presentation in §2 and the formal definitions in [13]. Again, it is not difficult to show that these canonical node refinements are system morphisms and that their conformance to the definitions is statically checkable.

The canonical transition refinement is the only form which does not have the property that every refined step sequence has a corresponding abstract step sequence with matching abstract states. Instead, this property holds if and only if the refined step sequence is complete, i.e. it contains matching occurrences of the border transitions of the refined transition(s) [13].

In §2 we gave examples of node refinement, one where a place was replaced by a subnet to indicate the more detailed processing of a book on its return from loan, and one where a transition was replaced by a subnet to indicate more of the details of borrowing a book. Again, such refined behaviour had corresponding abstract behaviour, but not necessarily vice versa.

It is important to emphasise that in each kind of refinement considered above, i.e. type refinement, subnet refinement, and node refinement, the criteria for acceptable refinements can be statically determined.

6. CONCLUSIONS AND FURTHER WORK

This paper has argued that the object-oriented specification of concurrent and distributed systems has tended to emphasise substitutability at the expense of code reuse. As a result, the various proposals in the literature tend to be better suited to the replacement of one system component by another, rather than supporting the incremental development of specifications. We have attempted to remedy this in the context of the Coloured Petri Net formalism by proposing three forms of refinement — *type refinement*, *subnet refinement* and *node refinement*. Examination of case studies in the literature seem to suggest that these forms of refinement are widely applicable in practice [15]. The behavioural compatibility requirements which they impose seem to match the kinds of abstraction that designers use in the process of incrementally developing formal specifications.

Recently, we have been developing state space analysis algorithms which will take advantage of the information supplied by the refinements proposed in this paper [17]. The basic approach is to use the analysis of the abstract model to restrict the possibilities considered in the refined model. (This approach rests on our basic requirement for behavioural compatibility.) If the net is refined by type or subnet refinement, the state space algorithm uses the abstract net to determine the firing modes that might be enabled for each state of the refined net. This reduces the time taken to search for the enabled firing modes at each state of the refined net. Where node refinement is used, the algorithm reduces the interleaving of states by computing the reachable states of each refined node separately, thus improving both the time and space required to generate the state space.

Preliminary results indicate that the percentage improvement (compared to the standard algorithm) is heavily dependent on the structure of the abstract and refined nets. For each of the forms of refinement, in certain cases we have observed greater than 95% time improvement. For node refinement we have also observed significant space improvement. Clearly, there is scope for further work in determining the efficiency improvements which are possible using the proposed refinements, and also for characterising the situations where such improvements are maximised.

Since this work is driven by the desire to support practical development techniques, it would obviously be helpful to extend our examination of case studies demonstrating incremental development of specifications. It would also be desirable to formalise the relationship between our proposals and others in the literature in an attempt to bridge the gap between them. For example, it would be worthwhile formulating minimum criteria under which every abstract firing sequence has a corresponding refined firing sequence.

References

[1] W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-Net-Based Approach. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81, Tolouse, France, 1997. Springer-Verlag.

[2] C. Balzarotti, F. DeCindio, and L. Pomello. Observation Equivalences for the Semantics of Inheritance. In *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, Florence, Italy, February 15-18, 1999. Chapman and Hall.

[3] B. Baumgarten. *Petri-Netze: Grundlagen und Anwendungen.* Mannheim: BI-Wissenschaftsverlag, 1990.

[4] H. Bowman, C. Briscoe-Smith, J. Derrick, and B Strulo. On Behavioural Subtyping in LOTOS. In *Second IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems.* Chapman and Hall, 1997.

[5] L. Cardelli and J. C. Mitchell. Operations on Records. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Proceedings of the Conference on Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 75–81, Berlin, September 1989. Springer.

[6] A. Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, 2nd edition, 1994.

[7] D. J. Floreani, J. Billington, and A. Dadej. Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. *Lecture Notes in Computer Science*, 1091:153–171, 1996.

[8] D.J. Floreani. *The Interconnection of Tactical Packet Radio Networks and BISDN.* PhD thesis, School of Physics and Electronic Systems Engineering, University of South Australia, 1996.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[10] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 1, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1992.

[11] K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN^{TM}: A Reference Manual.* MetaSoftware Corporation, 1992.

[12] C. Lakos. On the Abstraction of Coloured Petri Nets. In P. Azéma and G. Balbo, editors, *Lecture Notes in Computer Science: 18th International Conference on Application and Theory of Petri Nets, Toulouse, France, June 1997*, volume 1248, pages 42–61. Springer-Verlag, June 1997.

[13] C. Lakos. Composing Abstractions of Coloured Petri Nets. In *Proceedings of the 21st International Conference on Applications and Theory of Petri Nets (to apppear)*, Aarhus, Denmark, June 2000.

[14] C. Lakos and J. Lamp. The Incremental Modelling of the Z39.50 Protocol with Object Petri Nets. *Lecture Notes in Computer Science*, 1605:37–68, 1999.

[15] C. Lakos and G. Lewis. A Catalogue of Incremental Changes for Coloured Petri Nets. Technical report TR99-02, Department of Computer Science, University of Adelaide, 1999.

[16] W. LaLonde and J. Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, pages 57–60, January 1991.

[17] G. Lewis and C. Lakos. Incremental Reachability Algorithms. Technical report TR99-01, Department of Electrical Engineering and Computer Science, University of Tasmania, 1999.

[18] B Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, first edition, 1988.

[19] A. B. Mnaouer, T. Sekiguchi, Y. Fujii, and T. Ito. Coloured Petri Nets Based Modelling and Simulation of the Static and Dynamic Allocation Policies of the Asynchronous Bandwidth in the Fieldbus Protocol. *Lecture Notes in Computer Science*, 1605:93–130, 1999.

[20] O. Nierstrasz. Regular Types for Active Objects. *ACM Sigplan Notices*, 28(10):1–15, October 1993. Proceedings of the 8th. annual conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'93, Washington DC, 1993.

[21] W. Reisig. Petri Nets in Software Engineering. *Lecture Notes in Computer Science*, 255:63–96, 1987.

[22] P. Wegner and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *Lecture Notes in Computer Science*, 322:55–77, 1988.

[23] Glynn Winskel. Petri Nets, Algebras, Morphisms, and Compositionality. *Information and Computation*, 72(3):197–238, March 1987.