Coloured Petri Nets in UML-Based Software Development – Designing Middleware for Pervasive Healthcare

Jens Bæk Jørgensen
Centre for Pervasive Computing
Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
email: jbj@daimi.au.dk

Abstract

Nowadays, the Unified Modeling Language, UML, is almost universally accepted by the software industry as *the* modelling language. However, the language has severe shortcomings. While UML is well suited to model the static aspects of software systems, the language as it is currently standardised strongly needs improvements with respect to modelling behaviour. Thus, for development of software components with complex behaviours, UML often cannot stand alone. The main contribution of this paper is to position and discuss promotion of Coloured Petri Nets, or more generally high-level Petri nets, as a supplement in UML-based software development. We make the case on a specific example, development of middleware to support what is termed pervasive healthcare, but the observations hold in general for many systems with complex behaviours.

Topics: Relationship between CPN and UML, promotion of CPN in the software industry, system design, application of CPN to pervasive and mobile systems.

1 Introduction

This paper considers design of a system to support pervasive and mobile computing [7, 17, 21] at hospitals, the *pervasive healthcare middleware system, PHM* [2]. The system is being developed in a joint project by the hospitals in Aarhus County, Denmark, the software company Systematic Software Engineering [50], and the Centre for Pervasive Computing [42] at the University of Aarhus.

In the development of PHM, the *Unified Modeling Language, UML* [32, 35, 13, 12], is applied. By many of its proponents, UML is touted as a complete modelling language in the sense that it supports all modelling needs in all software development phases, from requirements specification through analysis, design, and implementation to the final system test. Indeed, UML has many advantages, e.g., it is general-purpose and extensible, it is standardised, a large selection of tools and text books is available, industrial acceptance is overwhelming, and success stories can be counted in hundreds or more [51]. However, the language is certainly not perfect [37], and use of UML as the one and only modelling language in the software industry can of course, from an academic point of view, be challenged. On the other hand, seen from the point of view of an industrial software developer, or his manager, an advocate of an alternative to UML has the burden of proof. The language is a de facto industrial standard, and with this status, it is the default choice of a modelling language in industrial software projects, much in the same way as Java is today's default choice of a programming language within many organisations.

In this paper, we challenge the exclusive use of UML in the design phase of the development of PHM. Here, we will suggest applying *Coloured Petri Nets, CPN* [23, 25, 43], in conjunction with UML. The proposal to combine Petri nets and UML in software development is not new, see, e.g., [39, 40]. However, the combination is often discussed in general terms only. The main contribution of this paper is to demonstrate in detail that CPN is a powerful supplement to UML, by pinpointing a number of specific, important PHM design issues that can be addressed properly in CPN, but not at all or not as easily in UML. Another contribution is considerations on the general positioning of UML and CPN in the software industry, and a discussion of how to better promote CPN. With this paper, we hope to help answering the frequently asked question: Why don't you just use UML?

A deliberate restriction of scope is that this paper does not consider formal verification. In many industrial software development projects, formal verification is not seen as an option at all, for reasons like needs, traditions, time schedules, shortcomings in verification methods and tools, and insufficiently trained software developers. Ensuring software quality is often done by systematic and thorough testing alone (whether this is wise, is a discussion beyond the scope of this paper).

This paper is structured as follows: Section 2 introduces the domain of our case study, healthcare information technology in general, pervasive healthcare in particular, and the PHM system itself. The design of a central component of PHM, the so-called session manager, by means of CPN is presented and discussed in Section 3. Section 4 provides a primer on UML, while Section 5 discusses a UML design of the session manager and compares the CPN and UML design approaches. Section 6 considers the general positioning of UML and CPN, and promotion of the latter, in the software industry. The conclusions, including a discussion of related work, are drawn in Section 7.

2 Healthcare Information Technology

The application area for PHM, the healthcare sector, is a hot topic in the public debate in Denmark, and probably also in many other countries. Everybody agrees that patients should be given the best treatment possible with the available resources. Therefore, rationalisation of hospital work processes is highly desired seen from the perspective of any stakeholder, from the minister of healthcare who wants to keep his budget, down to the nurses who have a strong need for alleviation in their busy work days.

An essential element in the rationalisation is to apply information technology to a much larger extent than has been done previously. A first step in this direction is the introduction of electronic patient records, which is going on in many places right now [5]. One of these places is Aarhus County, whose new electronic patient record *EPR* [41] will soon replace today's paper-based records. EPR is a comprehensive, general-purpose hospital IT system with a budget of approximately 15 million US dollars. It will be initially fielded later this year and put into full operation in 2004. The first version of EPR is made available from desktop PCs placed in hospital offices. However, the EPR users, i.e., the nurses and doctors, are away from their offices and on the move a lot of the time, e.g., seeing patients in the wards. Therefore, stationary desktop PCs in hospital offices are insufficient to support the hospital work processes in the best way, and taking advantage of the emerging possibilities for pervasive and mobile computing is crucial for the next version of EPR.

In the remainder of this section, we describe the notion of pervasive healthcare, and we introduce the pervasive healthcare middleware, PHM, the software system we are considering, and the session manager component, whose design will be the focus for our discussion.

2.1 Pervasive Healthcare

In the ideal situation, the healthcare professionals should bring the patient records with them, wherever they go. Thus, up-to-date information about patients, diagnoses, treatment plans, medication etc. would always be available at any time, in any given situation. The challenge seen from an IT perspective is to develop computer support for such new ways of working, and one potential approach is to exploit pervasive and mobile computing.

The term *pervasive healthcare* [47] covers the healthcare services that can be offered when the hospital staff has access to relevant data and functionality wherever they are. With pervasive healthcare, there may be computers "everywhere", e.g., in the room where the nurses prepare the prescribed medicine for patients, and in the trolleys used to transport trays with medicine. There may be large-screen computers in the wards so it is easy to show pictures like X-rays to the patients. Doctors and nurses may carry small personal digital assistants, PDAs, with which they can, e.g., access EPR and control other devices such as TV sets.

A specific example of a future pervasive healthcare scenario is the following: A nurse is about to prepare a round to the wards to give medicine to patients. The nurse wears a badge that emits a signal enabling various computers to sense her location. While she is approaching the room where the medicine is kept, a computer in this room automatically accesses EPR and fetches the medication plans for the patients that the nurse is responsible for. The computer presents these plans on the screen. The nurse prepares the medicine trays and uses the computer to acknowledge for each patient when the medicine has been poured correctly. Finally, all trays are put on a trolley. The trolley is equipped with a location-sensitive computer, and when the trolley approaches a ward, the medication plans for the patients in that ward are automatically fetched and presented on the screen. In this way, it is fast and easy for the nurse to acknowledge to the system when a patient has taken the medicine. If a patient has a question regarding the medicine, the nurse is able to answer supported by accessing medicine handbooks on-line using her PDA and perhaps using the TV set in the ward to display information.

2.2 The Pervasive Healthcare Middleware – PHM

The *pervasive healthcare middleware, PHM,* is a new computer system designed to support pervasive healthcare. PHM is a distributed system consisting of a number of components running in parallel on various mobile and stationary computing devices, and communicating over a wireless network. Some components run on a central background server, while others are deployed on the mobile devices. An early, overall design for PHM is proposed in [2]. Figure 1 shows selected components of the PHM architecture – the names hint their functionality.

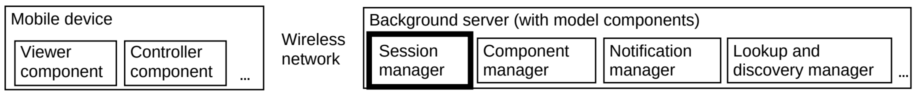


Figure 1: Selected components of PHM.

The scope of this paper is restricted to discussing design of the session manager component, shown with thick border in Figure 1.

A *session* comprises a number of devices that are joined together, sharing data, and communicating in support of some specific work process. A session is appropriate, e.g., if a nurse wants to use her personal digital assistant, PDA, to control a TV set in a ward in order to show an X-ray picture to a patient. In this case, the TV and the PDA must be joined in a session. Another example is a nurse who wishes to discuss some data, e.g., electronic patient record data, or audio and video in a conference setting, with doctors who are in remote locations. Here, the relevant data must be shown simultaneously on a number of devices joined in a session, one device for the nurse and one device for each doctor.

In general, session data is viewed and possibly edited by the users through their devices. The PHM architecture is based on the Model-View-Controller pattern [8]. The model part administers the actual data being shared and manipulated in a session. Each participating device has both a viewer and a controller component which are used as, respectively, interface to and manipulator of the session data. Model, viewer, and controller components communicate over the wireless network.

2.3 The Session Manager

Sessions are managed by the *session manager*, which is one of the most central and most complex components of PHM. The session manager manages zero to any number of sessions, and a session comprises one or more devices. Seen from the point of view of the session manager, a device is either inactive, i.e., not currently participating in any session, or active, i.e., participating in some session. A device participates in at most one session at a time.

The operations that the session manager must provide can be grouped into three main functional areas:

1. *Configuration management*: Initiation, reconfiguration (i.e., supporting devices dynamically joining and leaving), and termination of sessions.

2. *Lock management*: Locking of session data. Session data is shared and must be locked by a device, which wants to edit it.

3. *Viewer/controller management*: Change of viewers and controllers for active devices, e.g., if a nurse enters a room containing a TV, she may wish to view something on the large TV screen instead of on her small PDA display. In this case, viewer and controller replacement on the PDA and the TV is needed.

Devices interact with the session manager by invoking its operations. One interaction scenario is shown in Fig. 2, which illustrates the communication between the session manager and two devices, `d1` and `d2`.

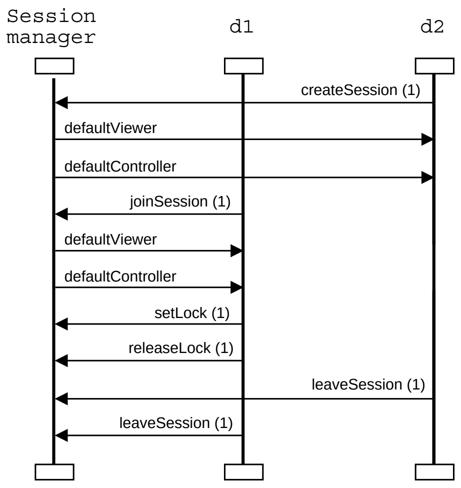


Figure 2: Session manager / device communication.

First, `d2` creates a session, which gets the session identifier 1. The session manager responds to the creation request by providing `d2` with a default viewer and a default controller for the new session. Then, `d1` joins the session, and also gets a default viewer and a default controller from the session manager. At some point, `d1` locks the session, probably does some editing, commits, and later releases the lock. Finally, `d2` and then `d1` leave the session.

3 Session Manager Design in CPN

Figure 2 is an example illustrating one single possible sequence of interactions between the session manager and some devices. In the session manager design, of course much more is needed. It is crucial to be able to specify and investigate the general behavioural properties of session management.

We now present a CPN model, whose purpose is to constitute an initial design proposal for the session manager, by identifying and giving an overall characterisation of the operations to be provided, with focus both on their individual behaviour and their interplay. The static architecture of the session manager in terms of classes and relationships between classes is outside the scope of the use of CPN (here, a fine job can be done with UML). The CPN model is created with Design/CPN [44].

It should be noted that the intention of this paper is not to present a large and complex CPN model. The aim is to create a model, which in the first place serves its purpose in the PHM development by being a design of the session manager, but secondly and equally important, is sufficiently tractable to constitute the foundation for the comparison of CPN and UML to be made in Section 5. In fact, the moderate size of the model to be presented is an advantage for the purpose to promote CPN. As we will argue, even for a modelling task of this size, UML is not sufficient, and the problems encountered in UML are exacerbated for larger models.

3.1 Net Structure and Declarations

The net structure of the CPN model consists of four modules (pages). The top-level of the model is the `SessionManager` module, shown in Figure 3, where the three main functional areas are represented by means of substitution transitions. In this way, each main functional area is modelled by an appropriately named sub-module, `ConfigurationManager`, `LockManager`, and `ViewCtrManager`, respectively.

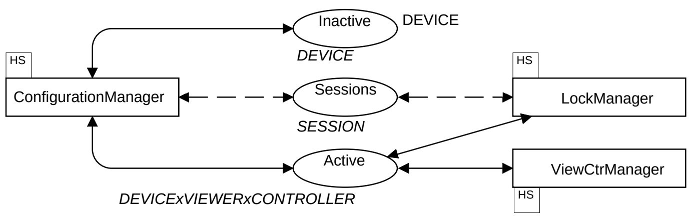


Figure 3: `SessionManager` module.

Figure 4 shows the declaration of constants, colour sets, and variables.

It can be seen that the model contains declarations of simple index colour sets for devices (`DEVICE`), viewers (`VIEWER`), and controllers (`CONTROLLER`). In addition, there are Cartesian product colour sets used to model when devices are associated with viewers and controllers. Sessions are modelled using the `SESSION` colour set, whose elements are triples `(s,dl,l)`, where `s` is a session identifier, `dl` is a list of devices, and `l` is a lock indicator.

The operations of the session manager correspond to the transitions of the CPN model. The detailed behaviour of the operations may immediately be derived from the arc expressions and guards. The latter corresponds to checks that the session manager must carry out before allowing the corresponding operation to be executed. We have chosen to use function calls consistently as arc expressions and guards, e.g., a function call like "`createSession s d`" instead of the expression "`(s,[d],NO_LOCK)`". In this way, the sub-routines of the session manager operations are explicitly identified.

In the following, we describe the model modules corresponding to the three main functional areas of the session manager. For each module, the corresponding session manager operation sub-routines are listed in terms of signatures for the functions used in inscriptions on that module.

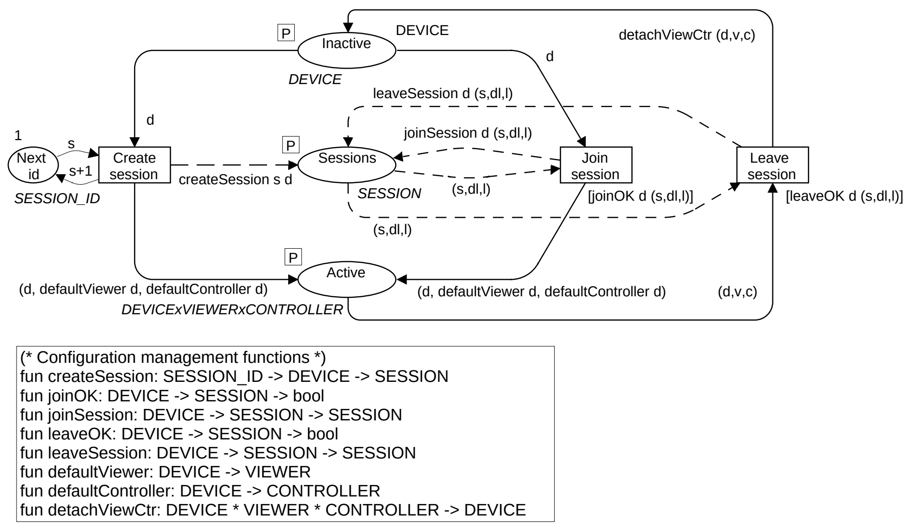
```
-------------------------------------------------------------
(* Constant declarations *)
val NO_OF_DEVICES = ...;
val NO_OF_VIEWERS = ...;
val NO_OF_CONTROLLERS = ...;

(* Colour set declarations *)
color DEVICE = index de with 1..NO_OF_DEVICES declare ms;
color INDEX = int with 1..NO_OF_DEVICES;
color VIEWER = index vi with 1..NO_OF_VIEWERS;
color CONTROLLER = index co with 1..NO_OF_CONTROLLERS;
color DEVICExVIEWERxCONTROLLER =
           product DEVICE * VIEWER * CONTROLLER;
color DEVICExVIEWER = product DEVICE * VIEWER;
color DEVICExCONTROLLER = product DEVICE * CONTROLLER;
color SESSION_ID = int;
color DEVICELIST = list DEVICE;
color LOCK = union L: DEVICE + NO_LOCK;
color SESSION = product SESSION_ID * DEVICELIST * LOCK;

(* Variable declarations *)
var d,d1,d2:  DEVICE;
var i:  INDEX;
var v:  VIEWER;
var c:  CONTROLLER;
var s:  SESSION_ID;
var dl:  DEVICELIST;
var l:  LOCK;
-------------------------------------------------------------
```

Figure 4: Declaration of constants, colour sets, and variables.

3.2 Configuration Management

The `ConfigurationManager` module is shown in Figure 5. It has four places, `Inactive`, `Active`, `Sessions`, and `Next id`. Initially, all devices are inactive, corresponding to all the `DEVICE` tokens initially being at the `Inactive` place. The only transition which is enabled in the initial marking is `Create session`. When it occurs, triggered by an initiating device `d`, a new session is created. The session gets a fresh session id from the `Next id` place, starts in unlocked mode, and a token corresponding to the new session is put on the `Sessions` place. The token corresponding to the initiating device is augmented with a default viewer and controller, and that triple is put on the `Active` place.

The transition `Join session` adds devices to sessions. Any join must be preceeded by a permission check, modelled by the guard function `joinOK`. When a device `d` joins a session `(s,dl,l)`, the appropriate token residing on the `Sessions` place is updated accordingly, by use of the `joinSession` function. Moreover, the `d` token is removed from the `Inactive` place, augmented with a default viewer and controller, and put on the `Active` place. The transition `Leave session` works similarly, but reversely, to `joinSession`. Upon leaving,



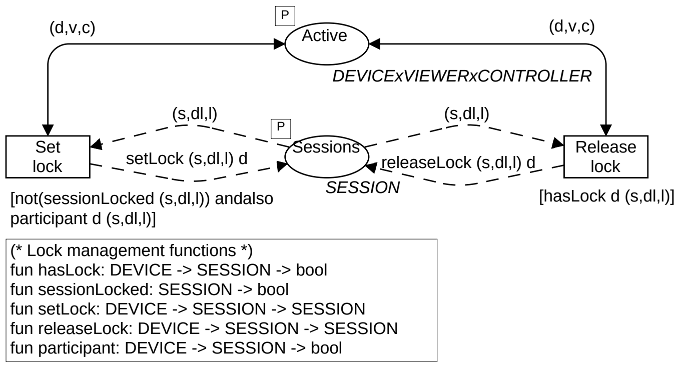
```
(* Configuration management functions *)
fun createSession: SESSION_ID -> DEVICE -> SESSION
fun joinOK: DEVICE -> SESSION -> bool
fun joinSession: DEVICE -> SESSION -> SESSION
fun leaveOK: DEVICE -> SESSION -> bool
fun leaveSession: DEVICE -> SESSION -> SESSION
fun defaultViewer: DEVICE -> VIEWER
fun defaultController: DEVICE -> CONTROLLER
fun detachViewCtr: DEVICE * VIEWER * CONTROLLER -> DEVICE
```

Figure 5: `ConfigurationManager` module.

the applicable viewer and controller for the device are detached.

3.3 Lock Management

The `LockManager` module is shown in Figure 6.



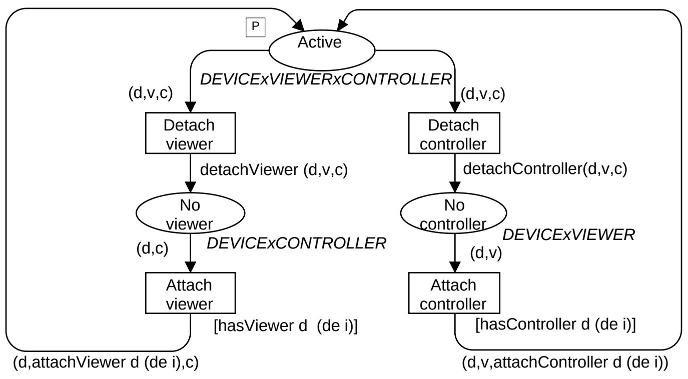
```
(* Lock management functions *)
fun hasLock: DEVICE -> SESSION -> bool
fun sessionLocked: SESSION -> bool
fun setLock: DEVICE -> SESSION -> SESSION
fun releaseLock: DEVICE -> SESSION -> SESSION
fun participant: DEVICE -> SESSION -> bool
```

Figure 6: `LockManager` module.

`LockManager` contains the two places `Active` and `Sessions`, which are already described. In addition, the module contains the two transitions `Set lock` and `Release lock`. When `Set lock` occurs, the selected session `(s,dl,l)` is locked by the requesting device, which is identified by the `d` part of the `(d,v,c)` token. The session can only be locked if it is not locked already and if the requesting device is currently a participant in that session. These two conditions are checked by the guard of `Set lock`. The effect of the transition `Release lock` is to release the lock of the current session. This is only possible if the requesting device is the actual lock holder, modelled by the `hasLock` function of the guard.

3.4 Viewer/Controller Management

The `ViewCtrManager` module is shown in Figure 7.



```
(* Viewer/controller management functions *)
fun hasViewer: DEVICE -> DEVICE -> bool
fun hasController: DEVICE -> DEVICE -> bool
fun detachViewer: DEVICE * VIEWER * CONTROLLER -> DEVICE * CONTROLLER
fun detachController: DEVICE * VIEWER * CONTROLLER -> DEVICE * VIEWER
fun attachViewer: DEVICE -> DEVICE -> VIEWER
fun attachController: DEVICE -> DEVICE -> CONTROLLER
```

Figure 7: `ViewCtrManager` module.

In addition to the place `Active` already described, the `ViewCtrManager` module contains the two places `No viewer` and `No controller`. A token on either `No viewer` or `No controller` models that the corresponding device is suspended – even though the device is participating in a session. This means that the device is temporarily not able to read and write data (only those devices whose tokens are currently on the `Active` place are able to read and, if an appropriate lock is set, write data).

Viewers and controllers are not replaced in one atomic action. The replacement mechanism is modelled by four transitions. The two `Detach` transitions do, as the names indicate, detach the viewer or controller, respectively. There is no precondition for allowing this. The `Attach viewer` transition checks that the viewer that some device requests actually can be provided. The request is checked by the guard "`hasviewer d (de i)`", which evaluates to true if it is possible to equip device `d` with a viewer for device number i, `de i` (as can be seen from Figure 4, the devices are indexed and `i` is a free variable over the index range). The guard function `hasViewer` enforces that some viewers fit with some devices and not with others. The `Attach controller` transition works in a similar fashion.

3.5 Module Dependencies

The interplay between the three main functional areas of the session manager is reflected in dependencies between the three corresponding modules of the CPN model, e.g., there is a dependency between lock management and configuration management, because a device in the process of editing session data is not allowed to abruptly leave the session. We require that the lock is released before permission to leave can be granted. Similarly, there is a dependency between viewer/controller management and configuration management. A temporarily suspended device in the process of replacing its viewer or controller is not allowed to leave the session, because the device might have the session lock set.

3.6 Modelling Decisions

The CPN model is an abstract view of the session manager. Some modelling decisions have been made in order to keep the model relatively simple, but still serving its purpose as constituting an initial design of the session manager. In particular, we have assumed reliable communication and that devices do not crash, i.e., errors in the communication between devices and the session manager are not modelled, and a question like what should happen if the device that is currently having a lock on session data crashes cannot be answered from the CPN model.

In a hectic and busy work day with the PHM system at the hospitals, communication errors will happen on the wireless network, and devices will crash, e.g., be turned off accidentally or run out of battery. To investigate such problems and strategies for coping with them, the CPN model could be extended. However, for the purpose of keeping a clear focus when we make comparative discussions of CPN and UML, we do not discuss how to model these issues in this paper.

4 The Unified Modeling Language – UML

We want to compare the proposed CPN design of the session manager with a design made exclusively in UML. For that purpose, and in order to make this paper relatively self-contained, in this section, we provide a primer on essential elements of the *Unified Modeling Language, UML*. The section may be skipped by readers already familiar with UML.

4.1 UML Background

UML supports object-oriented software development and offers a number of diagram types to model various views or perspectives of systems. Some views capture static aspects and other views describe behavioural aspects. UML as a whole is very big with an abundance of concepts and features, and the interested reader is referred to, e.g., [35, 13] for a more thorough description. The authoritative language documentation is the standard [32], but this massive and complex document cannot serve as an introduction.

The first version of UML appeared in 1997 as a result of an effort to unify a number of different, older object-oriented modelling methods. The current standard is UML 1.4 [32] dated May 2001. A major revision, UML 2.0, is expected to be approved late this year (2002). The main organisational body for the development and standardisation of UML is the Object Management Group, OMG [46]. UML is supported by many commercial tools, e.g., the market-dominant Rational Rose suite from Rational Software Corporation [48] and the Rhapsody suite from I-Logix [45].

4.2 Diagram Types

UML offers the following diagram types:

- Class diagrams
- Object diagrams
- Use case diagrams
- Component diagrams
- Deployment diagrams
- State machines
- Activity diagrams
- Sequence diagrams
- Collaboration diagrams

A *class diagram* is used to describe the architectural, structural composition of a system by identifying classes and their interrelations, and an *object diagram* is a structural description on the object level. A *use case diagram* is applied to capture and describe the future users' requirements for a system to be built. A *component diagram* reflects the actual implementation of a system, and a *deployment diagram* covers the physical architecture in terms of the hardware and software that make up a system.

The behaviour of a system is modelled using state machines, activity diagrams, sequence diagrams, and collaboration diagrams. A *state machine* is used to describe the behaviour of objects instantiated from a certain class, and captures the states and the events that may happen in each state. State machines may communicate with each other, thus allowing modelling of the combined behaviour of a number of interacting objects. State machines are derived from statecharts [18, 20]. In general, statecharts augment conventional state-transition diagrams with notions of hierarchy, concurrency, and a special kind of broadcast communication. In particular, a state of a state machine may be a compound state comprising other states and events. In this way, a state may to some extent correspond to a substitution transition of a CPN model. An *activity diagram* is a special kind of state machine, where there are slightly different rules for trigging and execution of events than for a pure state machine.

Both sequence diagrams and collaboration diagrams show selected examples of communication between the objects of a distributed system. A *sequence diagram* resembles a message sequence chart [22] and focus is on time (an example of a sequence diagram is shown in Figure 2). A *collaboration diagram* is a kind of annotated object diagram, and focus is on objects and their relations, together with their communication. Sequence numbers are attached to arrows between objects to describe a certain chain of communications.

5 Comparative Session Manager Design in UML

In this section, we consider design of the session manager using UML exclusively. As with CPN, we focus on the behaviour of the session manager rather than its static architecture. However, in UML, the two aspects are not separated, and both must be dealt with. Application of UML's behavioural diagrams assumes the existence of well-defined classes, e.g., a state machine always specifies the behaviour of a certain class, and a sequence diagram always shows the communication between objects instantiated from certain classes. Therefore, before we model behaviour, we must define classes. For this reason, Figure 8 shows the main classes and relationships of concern for session management.

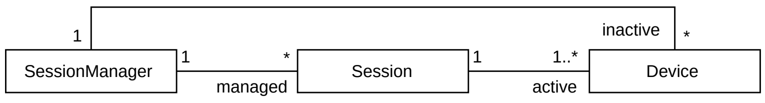


Figure 8: Session management class diagram.

Based on this class diagram, it was attempted to model the general behavioural properties of session management. This requires specification of both the individual behaviour of device and session manager objects, and the combined behaviour when these objects interact. The only UML behavioural diagram types which are candidates to be used in the general design are state machines and activity diagrams. Sequence and collaboration diagrams can only be used to show specific scenarios, i.e., they do not describe the general behaviour.

Therefore, it was attempted to create communicating state machines for the `SessionManager` and `Device` classes, and subsequently investigate their individual and combined behaviour. In this process, a number of severe problems were encountered. The problems that will be discussed in this paper are described below, and are all instances of more general shortcomings in UML behavioural modelling. The problems may be alleviated by using CPN as a supplement to UML, as we will argue below.

5.1 Executable Models

The first shortcoming in UML is lack of executable models. Without executable models, it is in practice impossible to investigate behavioural consequences of various design proposals for session management, prior to implementation. Executable models presume a well-defined formal execution semantics, which UML is currently lacking. None of the diagrams, in particular none of the behavioural diagrams, of the current UML standard [32] have a well-defined formal semantics, whereas the original statecharts of [18] do, defined in later papers, e.g., [19]. However, because of alterations made from statecharts to UML state machines, this well-defined semantics has been obscured – or, some would say, deliberately relaxed to become more "user-friendly".

If the lack of executable models was the only problem encountered in UML, it might be alleviated by using UML tools from, e.g., I-Logix's Rhapsody suite [45] or Rational's Rose RealTime [48] that do offer execution of UML behavioural diagrams – with execution algorithms which are, by necessity, based on proprietary semantic decisions. A formal execution semantics will most likely sooner or later be part of the UML standard (but we need it now for the PHM project). State machines already have an informal, textually described semantics in the current standard [32].

As we know very well, CPN has formal execution semantics in terms of the enabling and occurrence rules, and consequently, CPN does indeed offer executable models. Therefore, CPN models allow us to investigate the behavioural consequences of alternative design choices, and, thus, there is a sound and convenient foundation for pursuing improvements. In the PHM project, the CPN model of Section 3 reflects a number of design decisions for the session manager, many of which may be argued, e.g., would it be better to allow a device to have more than one viewer and one controller at a time instead of just one of each; or should a session always be explicitly terminated by the initiating device instead of being implicitly terminated when the last participating device leaves. Alternatives can be modelled and investigated by making modifications and simulations of the CPN model.

5.2 Modelling of Dependencies

The UML state machines for the `SessionManager` and `Device` classes are closely interrelated. For both classes, all state changes of concern are consequences of devices invoking operations in the session manager. We have had difficulties in describing individual state machines for the two classes, while at the same time properly capturing their communicating behaviour.

The difficulties are caused by the interplay between the three main functional areas of the session manager. The interplay materialises as dependencies between various model entities, as discussed in Section 3.5. It is difficult to capture such dependencies in a proper way with state machines. Undesired interferences between the three main functional areas must be precluded, e.g., that a device loses its lock on session data during viewer/controller replacement. We have tried to use various advanced features of state machines, e.g., concurrent and-states and the history mechanism (the latter is controversial [37]), but have not been able to describe the dependencies between the three main functional areas in a satisfactory way.

In theory, it is possible to create state machines that capture all dependencies, simply by introducing a sufficient number of states, e.g., instead of two individual states like `Has lock` and `Is replacing controller`, introduce states with more complex interpretations like `Has lock and is replacing controller`. However, the approach does not scale well – the size of the state machines grows quickly with the number of dependencies to be modelled. As an example, state machines are not feasible to describe a more fine-grained locking scheme than the current coarse-grained one. Allowing locking of subsets of session data requires simultaneous management of several locks and introduces many dependencies.

Functional area dependencies can be properly described in CPN because of the fine-grained nature of CPN models. In the CPN model of the session manager, e.g., lock management and viewer/controller management cannot interfere with each other in an undesired way. Whether a session is locked or not is captured by a value in the `SESSION` token on the `Sessions` place. As can be seen from Figure 3, replacement of viewers and controllers (modelled by the substitution transition `ViewCtrManager`) does not involve the `Sessions` place at all.

In contrast to state machines, CPN models scale well, e.g., a more fine-grained locking scheme, can be modelled based on letting `SESSION` tokens comprise lists of locks, instead of just one single lock. Moreover, it would be straightforward to extend the CPN model to do deal with unreliable communication and device crashes, as discussed in Section 3.6. It would be much harder to model these issues in UML, because they introduce more dependencies to be handled.

5.3 Modelling of Bookkeeping

A key task of session management is bookkeeping by tracking which devices are currently joined in sessions. Bookkeeping records must be updated each time a device creates, joins, or leaves a session. Proper investigation of session bookkeeping requires container-like data structure such as sets or lists to be supported in the session management behavioural models, e.g., to describe that in the current state, there are two sessions, one with devices {`d1,d2,d3`} and one with devices {`d4,d5`}. The state notion offered by UML state machines does not allow us to express this in a feasible way.

In the CPN model of the session manager, the place `Sessions` has a structured colour set defined and used with the purpose to do the desired bookkeeping, i.e., tracking which devices are in sessions together.

5.4 UML State Machines Versus Petri Nets

UML state machines have some resemblance with low-level Petri nets. The modelling of dependencies and of bookkeeping would also have caused problems if we had chosen low-level Petri nets as our modelling language to describe the behaviour of the session manager. These issues are dealt with smoothly in CPN, exactly because of the increased modelling convenience that characterises high-level Petri nets in general compared to low-level Petri nets.

As an aside, the original statecharts paper [18] sketches a rough idea about parameterised states, which has some similarity with high-level Petri nets. Also, in [18] Petri nets are recognised as a powerful means to describe behaviour, but it is noted as a main problem that Petri nets cannot be hierarchically decomposed. Since the publication of [18] in 1987, as we know, this problem has indeed been solved [23].

6 UML and CPN in the Software Industry

As demonstrated above, CPN may be used to alleviate general and severe problems encountered in UML. Therefore, one might argue that CPN often should be an obvious choice for industrial software developers engaged in modelling behaviour. Why this is not the case anyway is discussed in this section, where we state general observations regarding use of UML and CPN in the software industry, and where we also consider how to better promote CPN.

First of all, UML had a head start in comparison with CPN in the software industry, because the two modelling languages had very different starting points. UML came out of the object-oriented programming community, which set the dominating trend for industrial software development over more than the last decade. Therefore, from its advent, UML has enjoyed much attention in the software industry. In contrast, Petri nets and CPN emerged from mathematics and theoretical computer science as a model for concurrency, and have gained attention mostly within academia. Moreover, from the very beginning, UML became the subject for an attractive commercial market for tool vendors, consultants, etc., which all contributed to the boost of the language. We have not seen anything similar for CPN, or for Petri nets in general.

6.1 UML in the Software Industry

Many software companies are continuously applying UML, typically for the description of static, architectural properties of systems, by creation of primarily class diagrams. A software developer's incentive to use class diagrams is high, both because they are a very usable and convenient abstraction, but also because it is common functionality of UML tools to be able to generate source code level class skeletons automatically from class diagrams. In this way, the artifacts produced during design, i.e., the class diagrams, save the developer for some amounts of work during implementation.

Often, class diagrams suffices, e.g., for traditional administrative systems with the major part of the functionality being database access, such as the first version of EPR, Aarhus County's electronic patient record. For such systems, complex behavioural issues, e.g., involving communication, synchronisation and resource sharing are handled within standard off-the-shelf software components like database management systems and various middleware components such as application servers. Consequently, application-level software developers do not have to deal directly with the complexities themselves. However, when focus moves from database access to developing systems with complex behaviour, UML sometimes needs a supplement.

There are many speculations, not the least in the academic UML community itself, see, e.g., [34], that in many cases, the UML behavioural diagram types are not sufficient, and, as a consequence, not that widely used. In particular, UML state machines and activity diagrams are the only options to describe general behaviour, and they are not always feasible. There are a number of reasons for this. A main reason is the lack of executable models, which, together with the accompanying issue of formal semantics, are hot research topics within the academic UML community, see, e.g., [10]. Many proposals to define formal semantics for various kinds of UML diagrams are published, e.g. [4, 14, 26], and some implementations exist, but none are currently widely accepted and certainly none are standardised. Other reasons which may render UML behavioural diagrams insufficient include the dependencies and bookkeeping issues discussed in Section 5, which both are of a general, often encountered, and severe nature.

6.2 CPN in the Software Industry

A number of industrial CPN projects have taken place, see, e.g., [43] or [24]. Typically, the initiators of a project is a group of CPN experts from a university or another research institution, who establishes a cooperation with a small number of software engineers from some company. Often, such a project happens in isolation, in the sense that it is carried out, some experiences are gained, and a report or paper is written. However, when the project ends and the CPN experts walk away, the use of CPN within the company is in many cases discontinued. CPN seems to have a problem with manifesting itself broadly in the software industry.

If we want to advocate wide-spread and long-term use of CPN in some company, we have to convince not only the software developers, but also some higher-level decision makers like business managers or project managers. In conversations with the latter, we should stress the key business question: How does my company save time and money by using CPN? Sometimes, we should perhaps talk about reducing time to market, increasing return of investment, and limitation of risks, instead of about, e.g., nice theoretical properties like formal semantics. Moreover, in many cases, the arguments should stress CPN as a powerful vehicle to investigate behaviour and dynamic properties by simulation, much more than CPN as a means for formal verification. The latter is used and appreciated in a number of specialised application areas, but is far beyond interest and reach in very many software companies.

One approach to ensure continuous use of CPN in a company could be to define a step in the company's software development process, where complex components are identified and afterwards modelled and simulated using CPN. A written software development process is demanded on higher maturity levels of the Capability Maturity Model, CMM [33], which is gaining momentum in the industry right now. However, a proposal to add a step to a software development process has a high risk of being received with hesitation and eventually be rejected. Real-life software development projects are often characterised with extremely tight budgets and very hard deadlines, and decision makers and company management may think that introducing an additional step in the development process will do exactly the opposite of saving time and money.

A potential, genuine disadvantage of CPN in otherwise UML-based software engineering is that it indeed is a deviation from an established standard. However, UML itself allows various kinds of tailoring (in terms of what is known as constraints, tagged values, and stereotypes) to accommodate the language to situations where the standardised version is not applicable. In many cases, it may well be that the use of CPN is not a more dramatic deviation than the tailoring approved by the UML standard.

A severe drawback of using CPN is that the step from software design, i.e., CPN models, to implementation in an object-oriented programming language typically must be done manually. In the PHM project, the implementation of the CPN design models will involve manual coding. This is a general drawback of CPN, whose elaborated data type concept often is an advantage when creating models, but on the other hand makes automatic code generation from CPN models more complicated than, e.g., code generation from various versions of state machines and statecharts, for which there are many code generators available, e.g., [38]. Therefore, when use of CPN is considered in a software development project, a trade-off must be made between the desire to have strong, executable design models on one side, and ease of implementation on the other. With this remark, it is not said that automatic code generation from CPN models is impossible. CPN and Design/CPN have been used for automatic code generation to other target languages than object-oriented ones, even in industrial settings [29].

7 Conclusions

The use of UML in conjunction with Petri nets has been studied intensively in recent years. Much of the work done is concerned with automatic translation from certain types of UML diagrams into Petri nets, often aimed at formal verification, e.g., [36]. Also, Petri nets have been used to give precise execution semantics to different classes of UML diagrams, e.g., [3]. A small number of examples of combined use of UML and CPN in software development have appeared in the literature, e.g., design of user interfaces is described in [11]. High-level Petri nets in general in conjunction with mobile computing has been investigated in [31], where, again, the focus is on formal verification. Much research has addressed development of concepts and theories that combine the ideas of object-orientation in general (not just UML) and Petri nets [1]. Specifically in [27], Object Petri Nets are defined, which extend CPN with object-oriented features like inheritance and polymorphism. Other examples on work in this area are [6, 15, 28].

In this paper, we have taken the practitioner's pragmatic attitude, by suggesting to apply both UML and CPN in the design of the PHM system, by taking immediate advantage of the best of both worlds. More specifically, we propose using UML to describe static aspects and using CPN as a supplement to model behavioural aspects of complex components. We believe that this proposal carries over to projects, where there really is a need for executable models and a fine-grained investigation of behaviour. In [12], it is stated that UML lacks facilities to model the interaction between system components in a sufficiently fine-grained way. This general-term observations is sustained in this paper, where we have pointed out issues that are addressed better with CPN than with UML. The problems with modelling of dependencies and book-keeping in UML arise exactly because the UML behavioural diagrams are too coarse-grained. Moreover, in this paper, we have discussed how to position CPN in the UML-dominated world of software engineering. In particular, we have addressed a question posed in [39], where it is recognised that one of the key challenges in promoting Petri nets is to find the right projects and the right development phases to apply Petri nets in otherwise UML-dominated software engineering. One type of projects where Petri nets may be used successfully could be design of systems to support pervasive and mobile computing. Such systems are characterised by classical and well known complications that apply to many distributed systems [9], plus a number of new behavioural problems to be tackled, e.g., regarding mobility. Moreover, off-the-shelf standard middleware components supporting pervasive and mobile computing architectures are not well-established on the market yet.

With the success of UML, the software industry has in large scale adopted modelling as such as a valuable discipline, and today, modelling is generally accepted as a natural ingredient in everyday software development. Many software developers appreciate UML class diagrams as a productive asset to help them in their work. Those who have tried also to model behavioural aspects in UML might have encountered problems such as the ones discussed in this paper. Therefore, for many developers, the motivation to use a supplementary modelling language together with UML may be quite high. In this way, the success of UML can be seen as a good chance to establish CPN more broadly in the software industry.

References

[1] G. Agha, F.D. Cindio, and G. Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[2] J. Bardram and H.B. Christensen. Middleware for Pervasive Healthcare, A White Paper. In *Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, 2001.

[3] L. Baresi and M. Pezzé. On Formalizing UML with High-Level Petri Nets. In Agha et al. [1].

[4] M.v.d. Beeck. Formalization of UML-Statecharts. In Gogolla and Kobryn [16].

[5] M. Berg. Accumulating and Cordinating: Occasions for Information Technologies in Medical Work. In *Computer Supported Cooperative Work*, volume 8, 1999.

[6] O. Biberstein, D. Buchs, and N. Guelfi. Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism. In Agha et al. [1].

[7] J. Burkhardt, H. Henn, S. Hepper, K. Rintdorff, and T. Schäck. *Pervasive Computing – Technology and Architecture of Mobile Internet Applications*. Addison-Wesley, 2002.

[8] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad. *Pattern-Oriented Software Architecture*. John Wiley and Sons, 1996.

[9] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems – Concepts and Design*. Addison-Wesley, 2001.

[10] W. Damm. Understanding UML, Pains and Rewards. In Gogolla and Kobryn [16].

[11] M. Elkoutbi and R.K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In Nielsen and Simpson [30].

[12] G. Engels, R. Heckel, and S. Sauer. UML – A Universal Modeling Language? In Nielsen and Simpson [30].

[13] H.E. Eriksson and M. Penker. *UML Toolkit*. John Wiley and Sons, 1998.

[14] R. Eshuis and R. Wieringa. An Execution Algorithm for UML Activity Graphs. In Gogolla and Kobryn [16].

[15] H. Giese, J. Graf, and G. Wirtz. Closing the Gap Between Object-Oriented Modeling of Structure and Behaviour. In R. France and B. Rumpe, editors, *≪UML≫ 1999 - The Unified Modeling Language, 2th International Conference*, volume 1723 of *Lecture Notes in Computer Science*, Fort Collins, Colorado, 1999. Springer-Verlag.

[16] M. Gogolla and C. Kobryn, editors. *≪UML≫ 2001 - The Unified Modeling Language, 4th International Conference*, volume 2185 of *Lecture Notes in Computer Science*, Toronto, Canada, 2001. Springer-Verlag.

[17] U. Hansmann, L. Merk, M.S. Nicklous, and T. Stober. *Pervasive Computing Handbook*. Springer Verlag, 2001.

[18] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.

[19] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering*, 5(4), 1996.

[20] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts:The STATEMATE Approach*. McGraw-Hill, 1998.

[21] A. Helal, B. Haskell, J.L. Carter, R. Brice, D. Woelk, and M. Rusinkiewicz. *Any Time, Anywhere Computing – Mobile Computing Concepts and Technology*. Kluwer Academic Publishers, 1999.

[22] ITU-T Recommendation Z.120: Message Sequence Chart. International Telecommunication Union; Telecommunication Standardization Sector (ITU-T), 1999.

[23] K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1992.

[24] K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1997.

[25] L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2), 1998.

[26] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformations. In Gogolla and Kobryn [16].

[27] C.A. Lakos. Object-Oriented Modelling with Object Petri Nets. In Agha et al. [1].

[28] C. Maier and D. Moldt. Object Coloured Petri Nets – A Formal Technique for Object Oriented Modelling. In Agha et al. [1].

[29] K.H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In Nielsen and Simpson [30].

[30] M. Nielsen and D. Simpson, editors. *21st International Conference on Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, Aarhus, Denmark, 2000. Springer-Verlag.

[31] L. Ojala, N. Husberg, and T. Tynjälä. Modelling and Analysing a Distributed Dynamic Channel Allocation Algorithm for Mobile Computing Using High-Level Net Methods. In K. Jensen, editor, *Workshop on Practical Use of High-level Petri Nets*, Aarhus, Denmark, 2000.

[32] OMG Unified Modeling Language Specification, Version 1.4. Object Management Group (OMG); UML Revision Taskforce, 2001.

[33] M.C. Paulk, C.V. Weber, and B. Curtis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison Wesley, 1995.

[34] J. Rumbaugh. The Preacher at Arrakeen. In Gogolla and Kobryn [16].

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

[36] J. Saldhana and S.M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In *International Conference on Software Engineering and Knowledge Engineering*, Chicago, Illinois, 2000.

[37] A.J.H. Simons and I. Graham. 30 Things That Go Wrong in Object Modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems.* Kluwer Academic Publishers, 1999.

[38] J. Staunstrup. *IAR visualSTATE Concept Guide, version 4.* IAR Systems, 1999. www.iar.com.

[39] G. Wirtz. Application of Petri Nets in Modelling Distributed Software Systems. In D. Moldt, editor, *Workshop on Modelling of Objects, Components, and Agents*, Aarhus, Denmark, 2001.

[40] J. Xu and J. Kuusela. Analyzing the Execution Architecture of Mobile Phone Software with Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2), 1998.

[41] Aarhus Amt Electronic Patient Record. www.epj.aaa.dk.

[42] Centre for Pervasive Computing. www.pervasive.dk.

[43] Coloured Petri Nets at the University of Aarhus. www.daimi.au.dk/CPnets.

[44] Design/CPN. www.daimi.au.dk/designCPN.

[45] I-Logix. www.ilogix.com.

[46] Object Management Group. www.omg.org.

[47] Pervasive Healthcare. www.healthcare.pervasive.dk.

[48] Rational Software Corporation. www.rational.com.

[49] Systematic Software Engineering A/S. www.systematic.dk.

[50] Unified Modeling Language. www.uml.org.