Executable Design Models for a Pervasive Healthcare Middleware System

Jens Bæk Jørgensen and Søren Christensen

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

Abstract. UML is applied in the design of a pervasive healthcare middleware system for the hospitals in Aarhus County, Denmark. It works well for the modelling of static aspects of the system, but with respect to describing the behaviour, UML is not sufficient. This paper explains why and, as a remedy, suggests to supplement the UML models with behaviour descriptions in the modelling language Coloured Petri Nets, CPN. CPN models are executable and fine-grained, and a combined use of UML and CPN thus supports design-time investigation of the detailed behaviour of system components. In this way, the behavioural consequences of alternative design proposals may be evaluated and compared, based on models and prior to implementation.

Keywords: Executable models, detailed behaviour, Petri nets, CPN, system design, middleware, pervasive and mobile computing, supplementing UML.

1 Introduction

This paper considers design of a system to support pervasive and mobile computing [3, 6] at hospitals, the *pervasive healthcare middleware system, PHM* [1]. The system is being developed in a joint project by the hospitals in Aarhus County, Denmark, the software company Systematic Software Engineering [22], and the Centre for Pervasive Computing [17] at the University of Aarhus.

PHM will be part of Aarhus County's new electronic patient record, EPR [16], a comprehensive, general-purpose hospital IT system with a budget of approximately 15 million US dollars. EPR 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 design of PHM, *UML* [12, 14] is applied, primarily to model static aspects. However, the PHM design must also address a number of behavioural and

dynamic aspects applicable to distributed systems in general, and to pervasive and mobile systems in particular. UML, as it is currently standardised, has some shortcomings that prevent us from expressing key behavioural aspects in terms of UML models only. To overcome these shortcomings, *Coloured Petri Nets, CPN* [9, 10, 18], is used as a supplement to UML. CPN is a mature and well-proven modelling language suitable to describe the behaviour of systems with characteristics like concurrency, resource sharing, and synchronisation. The perspectives offered by UML state machines and activity diagrams are combined in CPN, in the sense that focus is equally balanced on states and activities, and their interrelations, in the same model. The main contribution of this paper is to describe and justify a combined use of UML and CPN for the design of PHM.

The paper is structured as follows: Section 2 introduces the *session manager* component of PHM, whose design will be the focus of discussion in this paper. Section 3 pinpoints a number of shortcomings in UML behavioural modelling. A CPN model of the behaviour of the session manager, together with an introduction to the CPN modelling language itself, is presented in Sect. 4. Section 5 justifies the combined use of UML and CPN, and the conclusions are drawn in Sect. 6.

2 The Session Manager

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. The scope of this paper is restricted to discussing design of the *session manager* component, which we present now. For a description of the other main components of PHM, please refer to [1].

Sessions In PHM, 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 [4]. 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.

Session Management Sessions are managed by a session manager, which is one of the most central and most complex components of PHM. The main classes and relationships of concern for session management are shown in Fig. 1.

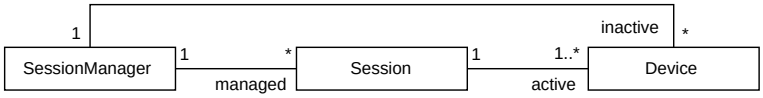

Fig. 1. Session management class diagram.

A session manager manages zero to any number of sessions, and a session comprises one or more devices. Seen from the point of view of a 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 a 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 a session manager by invoking its operations. One interaction scenario is shown in Fig. 2, which illustrates

the communication between a `SessionManager` object and two `Device` objects, `d1` and `d2`. 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 Shortcomings in UML Behavioural Modelling

Figure 2 is an example illustrating one single possible sequence of interactions between a session manager and some devices. In the session manager design, of

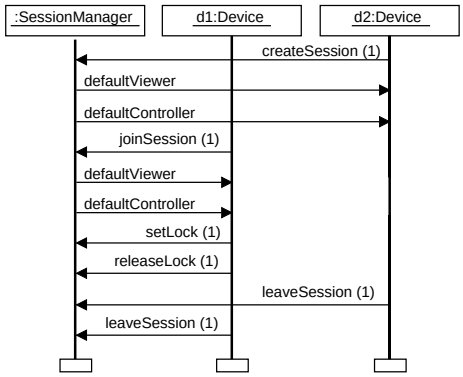

Fig. 2. Session manager / device communication sequence diagram.

course much more is needed. It is crucial to be able to specify and investigate 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. 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.

Execution The first shortcoming 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 – no kind of UML diagram as defined in the current standard [12] is executable.

We are well aware that if this problem was the only one, it might be alleviated by using UML tools from, e.g., I-Logix's Rhapsody suite [20] or Rational's Rose RealTime [21] that do offer execution of UML behavioural diagrams – with execution algorithms which are, by necessity, based on proprietary semantic decisions. We are also aware that 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 [12], and many proposals to define a formal semantics for state machines have been published, e.g., [2, 11].

Dependencies The state machines for the `SessionManager` and `Device` classes are closely interrelated. For both classes, all state changes of concern are con-

sequences of devices invoking operations in a 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 a number of dependencies between the three main functional areas of the session manager, 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 explicitly released before permission to leave can be granted.

It is difficult to capture the 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 – replacements are not atomic operations; a device first detaches the old and then attaches a new viewer/controller, and is temporarily suspended in between. We have tried to use both concurrent and-states and the history mechanism of state machines (the latter is controversial [15]), 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.

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 state machines does not allow us to express this in a feasible way.

4 Session Manager Behaviour in CPN

Because of shortcomings such as the ones discussed above, instead of restricting ourselves to UML diagrams only, we also use the modelling language of *Coloured Petri Nets, CPN* [9, 10, 18] for the design of PHM. In this section, we present a CPN model describing the session manager behaviour and at the same time give an informal primer to CPN.

CPN background CPN is one dialect of a broader category of graphical modelling languages known as *Petri nets* [13]. Harel's original paper on statecharts [7] recognises Petri nets as a powerful means to describe behaviour, but notes as a main problem that Petri nets cannot be hierarchically decomposed. Since the publication of [7] in 1987, this problem (and many others) has been solved [9].

A CPN model resembles a board game, with strict rules that define the possible executions of the model. The CPN modeller's task is to specify an appropriate board, tokens, etc. to reflect the domain being modelled. A CPN model is a graphical structure, supplemented with inscriptions and declarations of data types, variables, and functions. The tokens may carry complex data values ("colours"), which is one of the main virtues of CPN. Use of functions and expressions to manipulate data values allows the complexity of a model to be appropriately split between graphics, and declarations and inscriptions.

CPN gives a modelling convenience corresponding to a high-level programming language with support for data types, modules, and, indeed, hierarchical decomposition.

Model overview A CPN model is structured as a set of modules that have well-defined relations between them. The CPN model of the session manager consists of four modules, a top-level module `SessionManager`, shown in Fig. 3, and a module for each main functional area, `ConfigurationManager`, `LockManager`, and `ViewCtrManager`.

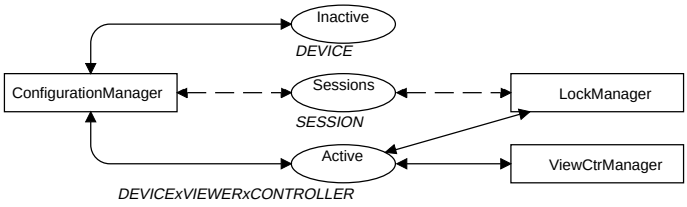

Fig. 3. `SessionManager` module.

The boxes in the figure are *substitution transitions*, corresponding to other modules of the model. Substitution transitions are the CPN mechanism to describe hierarchical decomposition, and a CPN counterpart of composite states of UML state machines.

The CPN model describes the behaviour of the session manager by tracking the states of devices and sessions. Thus, the CPN model represents the combined behaviour of interacting objects instantiated from the class diagram of Fig. 1 (and additional objects like viewers, controllers, and locks). Space does not allow us to present the entire model. Instead, we will describe one selected, representative module, the `ConfigurationManager`, shown in Fig. 4, and introduce the basic CPN concepts as we proceed.

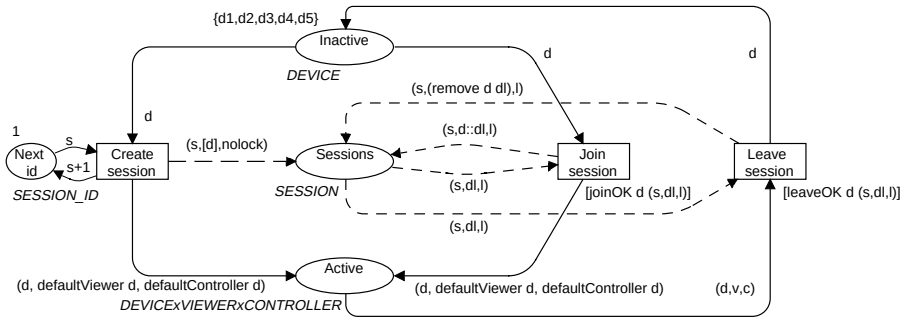

Fig. 4. `ConfigurationManager` module.

Modelling of states A CPN model describes both the states and the actions of a system. The *state* of a CPN model is a distribution of *tokens* on the *places*. Each place is drawn as an ellipse and has a *data type*, written in italic capital letters, which determines the kinds of tokens the place is allowed to contain.

In Fig. 4, the session manager's tracking of devices as being inactive or active is modelled by the places `Inactive` and `Active`. `Inactive` has data type `DEVICE`, whose elements are interpreted as devices, i.e., a `DEVICE` token on `Inactive` corresponds to an inactive device. Similarly, a token on `Active` models an active device, and its association with a viewer and a controller as catered for by the data type `DEVICExVIEWERxCONTROLLER`. The `Sessions` place is used to keep track of the ongoing sessions by containing tokens comprising lists of devices. The elements of data type `SESSION` are triples `(s,dl,l)`, where `s` is a session identifier, `dl` is a list of devices, and `l` is a lock indicator. The `Next id` place contains an integer, used to assign a unique identifier to each session – the integer is incremented each time a new session is created.

The figure shows the *initial state* of the model, where `Inactive` contains five device tokens, `Active` and `Sessions` are both empty, and `Next id` is marked with the integer `1`.

Modelling of actions The *actions* of a CPN model are represented using *transitions*, drawn as rectangles. In Fig. 4, the transitions correspond to session manager operations. A transition and a place may be connected by an *arc*. Solid arcs show the flow of `DEVICE` tokens, and dashed arcs the flow of `SESSION` tokens (different graphical appearances are used only to enhance readability, and have no formal meaning). The actions of a CPN model consist of transitions removing tokens from input places and adding tokens to output places, often referred to as the *token game*. Input/output relationship between a place and a transition is determined by the direction of the connecting arc. The tokens removed and added are determined by *arc expressions*, e.g., the expression on the arc from the `Inactive` place to the `Join session` transition is `d`, where `d` is a *variable* that can be assigned data values.

Execution semantics A transition which is ready to remove and add tokens is said to be *enabled*. One condition for enabling is that appropriate tokens are present on the input places. More precisely, it must be possible to assign data values to the variables appearing on input arcs such that the arc expressions evaluate to tokens available on the input places. In Fig. 4, enabling of the transition `Join session` requires that the place `Inactive` contains some `DEVICE` token that can be bound to the variable `d` appearing in the expression of the arc going from `Inactive` to `Join session`. Moreover, the other input place, `Sessions`, must contain a token that can match the arc expression `(s,dl,l)`, which designates a triple consisting of a session identifier `s`, a device list `dl`, and a lock indicator `l`. An additional condition for enabling comes from the *guard*, which is a boolean expression optionally assigned to a transition, and which must evaluate to true for the transition to be enabled. The guard of the transition `Join session` is the expression `[joinOK d (s,dl,l)]`, which tests whether it is allowed for the device `d` to join the session identified by `(s,dl,l)` – `joinOK` is a function enforcing the rules for joining.

An enabled transition may *occur*. The occurrence of the transition `Join session` models that a previously inactive device `d` joins a session. When `d` joins a session with session id `s`, the token `(s,dl,l)` residing on the `Sessions` place is updated as described by the arc expression `(s,d::dl,l)`, which adds `d` to the list `dl`. Moreover, `d` is removed from the `Inactive` place, augmented with a default viewer and controller, and added to the `Active` place.

The individual modules of a CPN model interact when the model is executed. In the modules of Figs. 3 and 4, places with the same name (e.g., the two `Inactive` places) are conceptually glued together, thus allowing exchange of tokens between the modules, when the token game is played.

Model perspective The perspective of the CPN model is the communication between, and thus combined behaviour of, the objects of concern, i.e., a session manager, devices, sessions, etc. Occurrence of any transition corresponds to invocation of an operation of a session manager by some device. The result of the occurrence reflects the corresponding state change for the invoking device and involved session, e.g., a device changes from inactive to active, and a session is extended with an additional device – or, in the modules not shown, a lock is set on session data, or a device changes from active to suspended.

5 Justification of CPN use

Using CPN as a supplement to UML for modelling the session manager behaviour is a deviation from an established standard, and therefore should be well justified. We do this now by revisiting the UML shortcomings of Sect. 3, and arguing that they are all alleviated by the use of CPN.

Execution CPN offers executable models, ensured by a well-defined, formal execution semantics in terms of the enabling and occurrence rules. In the PHM

project, the behavioural consequences of alternative design alternatives are investigated via execution of CPN models. Changing a decision, e.g., the rules governing joining and leaving of sessions, can be captured and investigated quickly by modification and execution of the session manager CPN model.

In the PHM project, the implementation of the executable CPN design models involves 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 statecharts and state machines. Therefore, when use of CPN is considered in a 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.

The formal semantics of CPN is the cornerstone for executable models. An additional gain of the formal semantics is that CPN models may be formally verified [9]. There exists industrial-strength tools supporting both execution and verification of CPN models. The model of this paper is made with Design/CPN [19], which is used by many companies and research institutions world-wide.

Dependencies Functional area dependencies can be properly described in CPN because of the fine-grained nature of CPN models, in particular the support for tokens carrying data values. 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 Fig. 3, replacement of viewers and controllers (modelled by the substitution transition `ViewCtrManager`) does not involve the `Sessions` place at all.

CPN models scale well, e.g., a more fine-grained locking scheme, which as noted in Sect. 3 results in many dependencies, can be modelled based on letting `SESSION` tokens comprise lists of locks, instead of just one single lock.

Bookkeeping CPN allows use of container data types. In the session manager model, the place `Sessions` has a data type defined and used with the purpose to do the desired bookkeeping, i.e., tracking which devices are in sessions together.

6 Conclusion

The proposal to combine UML with CPN is not new, e.g., in [8] the ambitious aim is formal verification via automatic generation of CPN models from UML models. In the PHM project, the immediate advantages of CPN are used to obtain executable and fine-grained design models, focusing on the communication between, and thus combined behaviour of, the objects of concern. In this way, CPN is a means to gain valuable insight early in the development project.

The scope of this paper has been a specific project, and even the design of a specific component, but the encountered UML shortcomings and proposed

alleviation are of a general nature. Thus, in a number of projects, the early design decisions may be improved by supplementing UML models with behaviour models in CPN, e.g., when designing pervasive and mobile systems. However, more work is required in order to assess the feasibility of a theoretically well-founded and more general integration of the UML and CPN modelling languages.

References

1. J. Bardram and H.B. Christensen. Middleware for Pervasive Healthcare, A White Paper. In *Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, 2001.
2. M.v.d. Beeck. Formalization of UML-Statecharts. In Gogolla and Kobryn [5].
3. J. Burkhardt, H. Henn, S. Hepper, K. Rintdorff, and T. Schäck. *Pervasive Computing – Technology and Architecture of Mobile Internet Applications*. Addison-Wesley, 2002.
4. F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad. *Pattern-Oriented Software Architecture*. John Wiley and Sons, 1996.
5. 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.
6. U. Hansmann, L. Merk, M.S. Nicklous, and T. Stober. *Pervasive Computing Handbook*. Springer Verlag, 2001.
7. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.
8. R.G. Pettit IV and H. Gomaa. Validation of Dynamic Behaviour in UML Using Colored Petri Nets. In *Workshop on Dynamic Behaviour in UML Models: Semantic Questions*, ≪UML≫ 2000, York, England, 2000.
9. K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1-3*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1992-97.
10. 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.
11. S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformations. In Gogolla and Kobryn [5].
12. OMG Unified Modeling Language Specification, Version 1.4. Object Management Group (OMG); UML Revision Taskforce, 2001.
13. W. Reisig. *Petri Nets, an Introduction*. Springer-Verlag, 1985.
14. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
15. 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.
16. Aarhus Amt Electronic Patient Record. www.epj.aaa.dk.
17. Centre for Pervasive Computing. www.pervasive.dk.
18. Coloured Petri Nets at the University of Aarhus. www.daimi.au.dk/CPnets.
19. Design/CPN. www.daimi.au.dk/designCPN.
20. I-Logix. www.ilogix.com.
21. Rational Software Corporation. www.rational.com.
22. Systematic Software Engineering A/S. www.systematic.dk.