Protocol Modelling Semantics for Embedded Systems

Ashley McNeile
Metamaxim Ltd
48 Brunswick Gardens,
London W8 4AN, UK
ashley.mcneile@metamaxim.com

Ella Roubtsova
Open University
Postbus 2960, 6401 DL
Heerlen, the Netherlands
ella.roubtsova@ieee.org

Abstract

The properties of a domain oriented modelling approach or language are determined by the dominant semantics of the domain. A significant subclass that needs particular attention, because of its prevalence, is that of deterministic interactive embedded systems. Embedded systems contain both hardware and software components interacting with each other and with the users. The components should be modelled separately, and behaviour should be explicitly defined in order to ensure correct interaction between the components.

In this paper we argue that a semantic framework known as Protocol Modelling provides a good basis for modelling interactive deterministic embedded systems. Firstly, we explain how Protocol Modelling represents interaction, and how it supports Hoare's CSP composition operator, thus allowing components of the solution to be modelled separately. Secondly, we show how Protocol Modelling can employ different modelling notations, focusing particularly on Coloured Petri Nets and State Charts. Finally, we describe how it guarantees local reasoning about the trace behaviour of a composite based on consideration of the components. We illustrate these explanations using a simple mobile phone case study.

1 Introduction

Behavioural models use semantic abstractions to represent intended behaviour of a software system. Behaviour, however, is a portfolio concept and we distinguish many different types of behaviour (deterministic and non-deterministic, reactive and interactive, statistical, etc.) appropriate to different types of domain and class of system. Whether a behavioural modelling approach is well suited to a particular problem depends on whether the underlying semantics of the approach is properly aligned to the domain and type of system being designed. We focus on a large class of embedded systems, in which the behaviour is interactive and deterministic, and the system as a whole is composed from components. In this class of system, which includes computer peripherals, multi-processor systems-on-chips (MPSoC) and other embedded systems [10], we need a modelling approach that is suited to describing interaction and supports the assembly of a complete behavioural description by composition of parts, corresponding to the components of the system.

The search for the best way to model interactive behaviour is an ongoing endeavour in the modelling community. Modelling of interactive behaviour in Coloured Petri Nets has been investigated, for example, in work of Ph.A.Palanque et. al [11] and Elkoutbi and Keller [8]. Ph.A.Palanque et. al [11] introduces Petri Nets with Objects (PNO). However, PNO uses conventional Coloured Petri Nets (CPN) semantics and does not provide a semantic basis for composition. The focus of work of Elkoutbi and Keller [8] also uses a hierarchical, rather than compositional, structuring of the behavioural model of a system.

In this paper we show that Protocol Modelling semantics [2] supports both the representation of interactive behaviour and a compositional modelling approach.

The fundamental abstraction of Protocol Modelling is a Protocol Machine (PM). A PM is an interactive, event driven and deterministic machine, and its defining characteristic is its ability to either ignore, accept or refuse any event that is presented to it. This property enables PMs to be composed using the CSP (Communicating Sequential Processes) parallel composition mechanism [4].

There are three properties of PMs that we wish to illustrate in this paper:
1. The semantics of a PM is defined independently of the notation used to describe its behaviour, and this makes it possible to choose the notation most appropriate to a problem and to mix different notations on the specification of a single problem.
2. The result of composing two PMs is another PM, and the CSP composition mechanism guarantees "Obser-

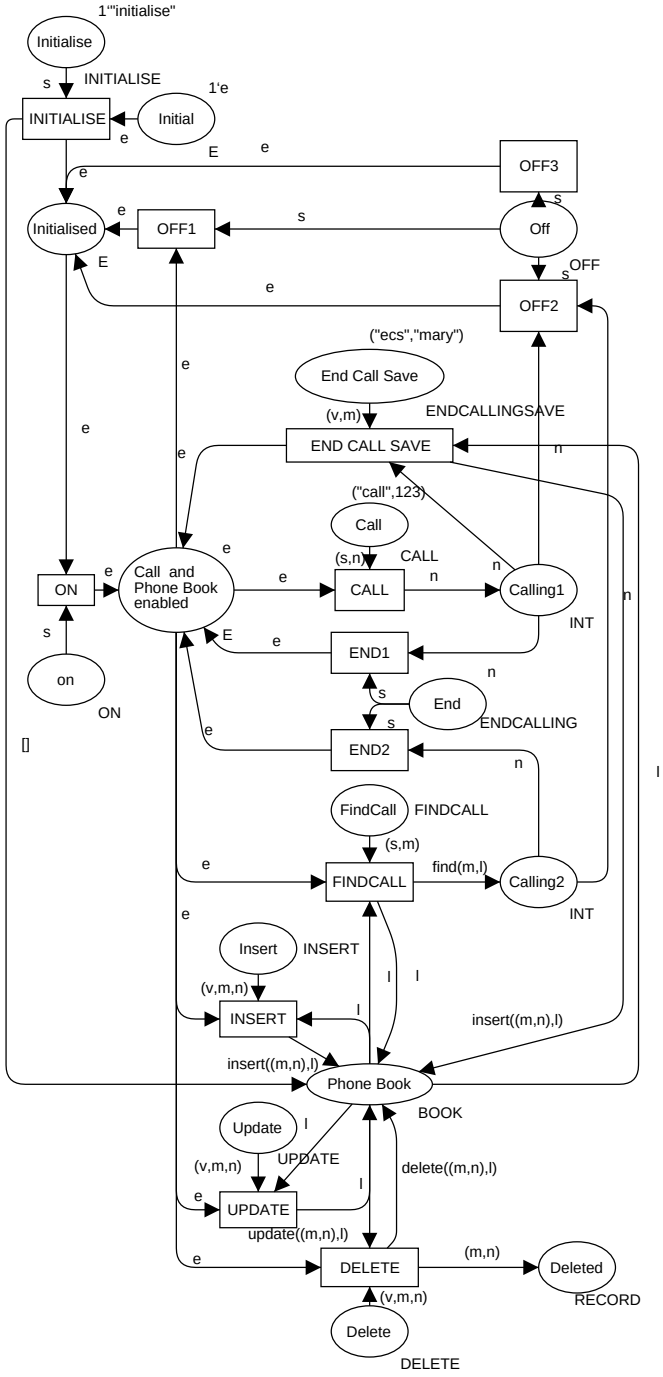

Figure 1. A CPN model of a mobile phone that can save phone numbers

vational Consistency" [5] between the composite PM and the component machines contributing to the composition. Observational Consistency enables "local reasoning": i.e., reasoning about behaviour of the composite based on local understanding of the behaviour of the contributing component machines.

3. Allowing multiple PMs to be composed also allows different levels of abstraction to be used in different PMs used in a specification. In particular, sets of events or states that are treated identically at some level of abstraction can be defined as atomic in a machine that describes that level of abstraction. This feature extends the composition possibilities to address cases where one machine is composed with another at several "join points", defined either in terms of events or states.

The remainder of this paper is organized as follows:

- Section 2 presents a model of a mobile phone with a phone book in CPN [6] using conventional semantics.
- Section 3 discusses the key differences between conventional CPN semantics and the semantics of Protocol Modelling.
- Section 4 describes the key elements of the semantics of Protocol Modelling.
- Section 5 re-castes the mobile phone example using Protocol Modelling semantics, using first CPN and then State Chart notation.
- Section 6 discuses the possibilities of local reasoning and model evolution supported by the PM approach.
- Section 7 provides a summary of the main points of the paper.

2 Case Study

We use a simplified case study of a mobile phone with a phone book that can save names and associated numbers. In this paper, we present models of the system in three different notations: in this section using Coloured Petri Net (CPN) notation with conventional semantics; and in the next section in both Coloured Petri Net and State Chart notation with Protocol Modelling (PM) semantics.

Figure 1 shows the model of the system in CPN [6, 7] with conventional semantics. This model represents the mobile phone software as a single net per phone.

The mobile phone model can be initialised (transition *INITIALISE*) and switched on (transition *ON*). Switching on enables the both the call and phone book functionality (place *Call and Phone Book enabled*). The initialisation means that the token (an empty list) is set into place *Phone Book* of *color BOOK= list RECORD*, where
*color RECORD=product NAME*NUMBER;*

If the functionality is enabled, a call can be made (transition *CALL*) and then ended, either without saving the number (transition *END1*), or ended with saving the number in

the phone book (transition *END CALL SAVE*) as a record *(name, number)*.

If the user wants to call someone but has forgotten the number, an attempt can be made to find the name in the phone book. If the name is found, a call is initiated (transition *FIND CALL*). In this case the number is already in the phone book, so the call is ended without the possibility of a save (transition *END2*).

While no call is active, new records *(name, number)* can be inserted in the 'phone book' and existing records can be updated or deleted (transitions *INSERT, UPDATE, DELETE*):

$fun\ insert(z : RECORD, []) = [z] \mid insert(z : RECORD, h :: t : BOOK) = if\ (z = h)\ then\ (h :: t)\ else\ (h :: insert(z, t));$
$fun\ delete(z : RECORD, []) = [] \mid delete(z : RECORD, h :: t : BOOK) = if\ (z = h)\ then\ (t)\ else\ (h :: delete(z, t));$
$fun\ find(m : STRING, []) = 0 \mid find(m : STRING, (a, b) :: t : BOOK) = if\ (m = a)\ then\ b\ else\ find(m, t);$
$fun\ update((y, z) : RECORD, []) = [(y, z)] \mid update((y, z) : RECORD, (a, b) :: t : BOOK) = if\ (y = a)\ then\ (y, z) :: t\ else\ ((a, b) :: update((y, z) : RECORD, t : BOOK));$

The CPN in Figure 1 contains tokens in places
(Call enabled, 1'e), (Call,("call",123)),
(End Call Save,("ecs","Ann")).
In this marking transition *CALL* can fire, then - transition *END CALL SAVE* can fire and the record *("Ann", 123)* will be saved in the phone book.

3 Comparison of Semantics

The central claim of this paper is that the semantics of Protocol Modelling offers advantages over the conventional CPN approach illustrated above. In this section we motivate this statement by comparing the semantic basis of two approaches along a number of dimensions.

3.1 Interactive Behaviour

Conventional CPN semantics does not support interactive behaviour. Key to modelling interaction is the idea that a system can adopt a number of different states, and that its ability to handle an input event of given type is state dependent: so that in a given state it can accept inputs of some types but not inputs of other types, which are disabled or refused [12]. With conventional CPN semantics, however, reception of an input is represented by placing a token in a place on the net, which remains in place until consumed. There is no concept of disabling a place or refusing a token.

In contrast, the Protocol Modelling (PM) semantics (described in the next section) focuses on interactive behaviour. The idea that a model has well defined quiescent states (when it waiting for an interaction with its environment), and that it can either accept or refuse input events based on state, is central to the semantic model.

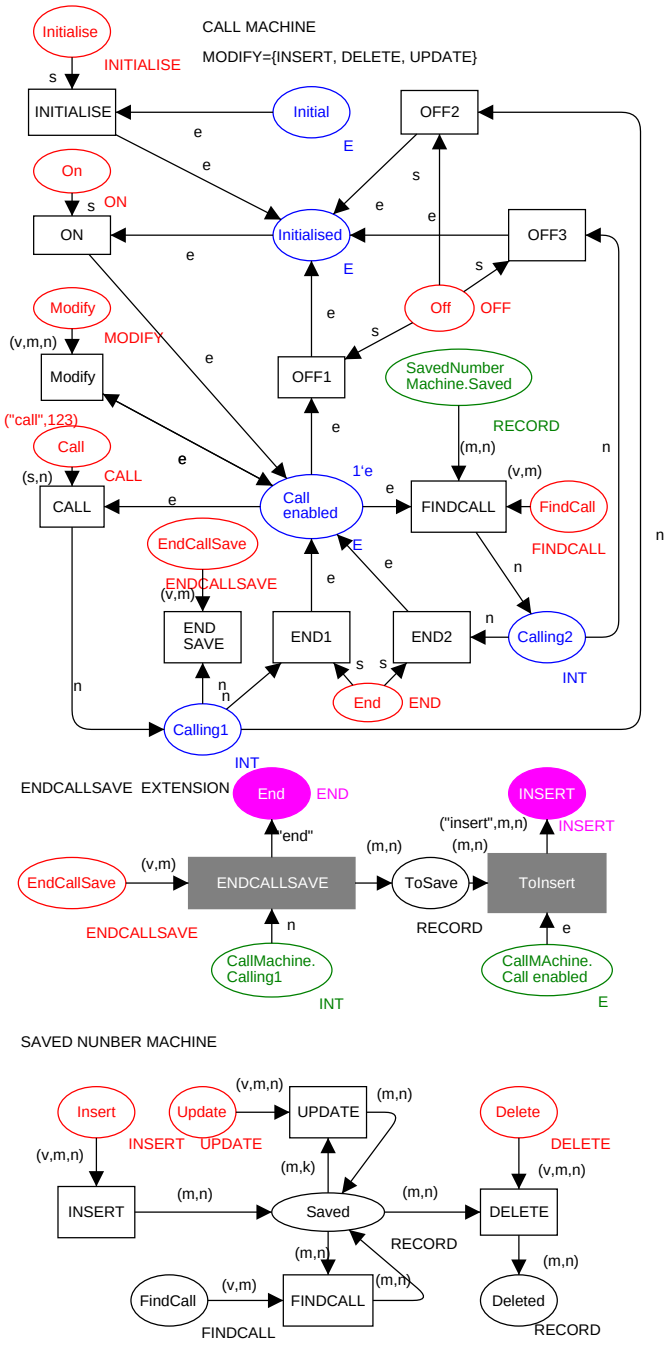

Figure 2. A CPN with the PM semantics

3.2 Composition

In conventional CPN semantics there is no notion of parallel composition of behaviour. This is (at least partly) a consequence of the above, as behavioural composition approaches, such as the parallel composition $(P \parallel Q)$ of Hoare's CSP or the concept of a reaction on Milners' pi-calculus are predicated on the idea that processes can refuse events.

In contrast, models with the PM semantics can refuse events and this allows process algebra style composition, as described below in Section 4

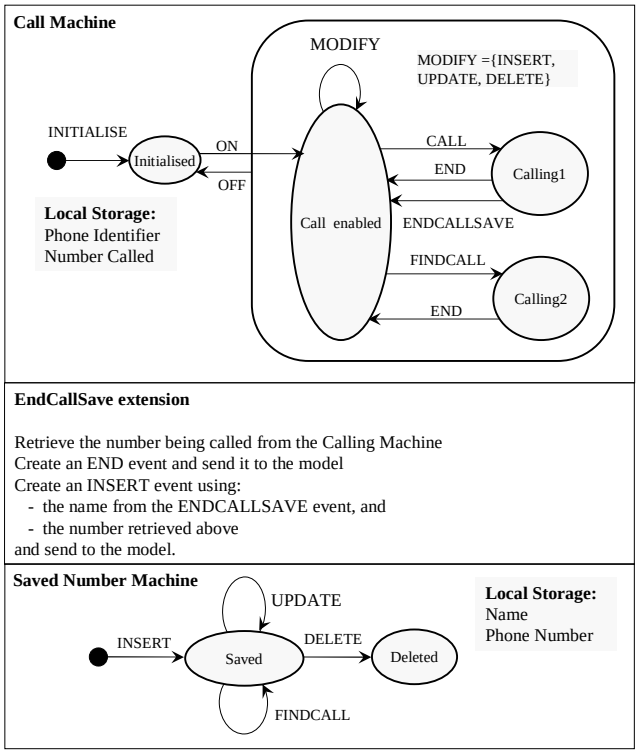

Figure 3. A State chart with PM semantics

3.3 Objects

In conventional CPN semantics there is no notion of "objects": i.e., the idea that different components of the model have different numbers of instances during the execution.

In contrast, PM semantics is object based, and components (machines or composed assemblies of machines) are modelled separately for each "object" (= instantiable entity) identified in the domain. In particular, this means that a "Saved Number" functionality can be modelled as a separate component that has an instance for each number saved in the phone book. This means that the textually defined functions for handling inserting, updating and deleting numbers in the phone book (set out above in Section 2) are not required.

4 Protocol Modelling

In this section we introduce the concepts of Protocol Modelling (PM) [2]. The concepts described in this section are notation independent, and pertain whether the model is described in CPN or in state chart notation. In the section after this we will show how the mobile phone case study can be described as a PM model in both CPN and state charts.

Events. The systems we model using PM are *event driven*, where an *event* (properly an "event instance") is an occurrence or command to the system that (in general) causes the system to change state. In the case of the mobile phone, events are such things as: *"Call phone number 123"*. Every event is presented to the system as a single input comprising, in general, a number of data attributes. Every event is an instance of an event-type, and the type of an event determines the attribute schema for the event, this being the set of data attributes that completely define an instance of the event-type. In a mobile phone, for instance, the attribute schema for the event-type that initiates a call on the phone would be:
(EVENT TYPE, PHONE NUMBER), where *(EVENT-TYPE="CALL").*

This approach to modelling events is identical to that used in other event based modelling approaches [9, 13].

Protocol Machines. In PM, models are built by composing *protocol machines*. A protocol machine (hereafter, just "machine") is a conceptual machine that has a defined *alphabet* of event-types that it understands, and the ability to *accept*, *refuse* or *ignore* any event that is presented to it by its environment [2]. In PM, these machines are the building blocks of the behavioural entities. Complete models are built by composing machines that represent partial descriptions of the whole behaviour being modelled.

A Protocol Machine *behaves* as follows:
- When presented with an event that is not represented in its alphabet, the Protocol Machines *ignores* it, which means the machine does not recognize the event.
- When presented with an event that is represented in its alphabet, it will either *accept* it or *refuse* it.
- Acceptance or refusal of an event by the machine is determined by rules that the machine applies both *before* and *after* the event.

Note that "refusal" of an event means that the machine recognizes the event from its alphabet but is unable to handle the event in its current state, and this normally means that some kind of error message is generated back to the environment. How or where such an error is generated is not of concern for modelling purposes.

Local Storage. Every machine has a *local storage* which it and only it can alter, and only when moving to a new state in response to an event. A machine can update its own local storage, and may read (but not update) the local storage of other, composed, machines.

Determinism. Protocol machines are assumed to be deterministic, meaning that:

- A Protocol Machine is never presented with an event unless and until it is in a quiescent state.
- The new quiescent state that a machine reaches as a result of being presented with an event is completely determined by
  - the last quiescent value of the storage of the whole model (i.e., the union of the local storage of all machines),
  - the event-type and attribute values of the event presented to it.

These mean that whether or not a machine accepts an event is deterministic, and therefore that that executions of a PM model are repeatable.

CSP Composition. Machines in PM can be composed, using the following rules, corresponding to the parallel composition operator $(P \parallel Q)$ of Hoare's process algebra, Communicating Sequential Processes [4]. The alphabet of the composed Protocol Machine is the union of the alphabets of the constituent machines; and the local storage of the composed Protocol Machine is the union of the local storages of the constituent machines. The rules for whether the composed machine accepts, refuses or ignores a presented event are:
- If both constituent machines ignore the event, the composed machine ignores it;
- If either constituent machine refuses the event, the composed machine refuses it;
- Otherwise the composed machine accepts the event.

Objects. A PM model is object based, so that every machine instantiated belongs to an object instance, identified by a unique object identifier (OID). The full description of the behaviour of an object is the parallel composition of all the machines that share the object's OID. Moreover, the composition of machines is across, as well as within, objects thus an event that targets more than one object is refused if any targeted object is unable to accept it.

Model Extension. Model extension is a technique when the handling (determining acceptance and performing updates to local storage) of a new event being introduced into a model can be defined in terms of a number of events already in the model. This idea is essentially similar to Liskov and Wings *extension maps* [3]. Adding a new event-type to a model to handled by model extension requires the addition of an *extension process* to the model that intercepts every instance of the new event-type and generates one or more events of existing event-types. Together, the generated events implement the effect of the new event.

An extension process is not, itself, a protocol machine and its behaviour is defined differently:
- It is used for every instance of its input event-type;
- It generates events and passes them to the model one at a time, waiting for the model to reach quiescence between each;
- It has read-only access to the attributes of the input event, and to the stable states that the model reaches after receiving each generated event.

An event handled by an extension process is accepted by the model iff all the events generated by the extension process for the event are individually accepted.

5 Rendering the Mobile Phone as a PM model

In this section, we show how the Mobile Phone case study can be rendered as a PM model. To emphasize that PM models can be described in different notations, we describe:
- a rendering in CPN, and
- a rendering in state chart notation.

In both renderings, the Mobile Phone model is described using two protocol machines: the Call Machine (which is instantiated once) and the Saved Number Machine (which is instantiated once per number saved in the phone book of the phone). In particular, this allows the event protocol of an individual number (INSERT, then UPDATE any number of times, then DELETE) to be *explicit* in the model, as shown in the bottom panels of Figure 2 (for CPN rendering) and Figure 3 (for state chart rendering).

Also, in both renderings, the ENDCALLSAVE event (which both ends a call and saves the number used for the call in the phone book) is handled by extension, by generating an END event to end the call and an INSERT event to put the number into the phone book.

5.1 CPN PM Rendering

Figure 2 shows the Mobile Phone in CPN PM form. Here the machines are shown in CPN notation, but the notation here has PM semantics which map onto the PM concepts in the previous section 4. In the following paragraphs we propose how such a mapping can be achieved.

Events and Alphabet. A protocol machine rendered in CPN has a distinguished subset of places called *interface places* that receive events from the environment. These places have no input arcs. The color (terminology of CPN) of an interface place corresponds to the event-type of input events that may be placed there. For example, places *On, Off, Insert, Delete, FindCall* in Figure 2 are the *interface places.* $color\ INSERT =$
$product\ EVENTTYPE * NAME * PHONENUMBER,$
where the *EVENTTYPE* is "insert",
the *NAME* is of type *STRING* and
the *PHONENUMBER* is of type *INT*;
An Event instance is visualized by a token of an event type.

The alphabet of a machine is its set of interface places. The alphabet of the Call Machine is:
$A_{CallMachine}$ = { *INITIALISE, ON, OFF, CALL, FINDCALL, END, ENDCALLSAVE, INSERT, DELETE, UPDATE* } Note that the events of types *INSERT, UPDATE, DELETE* are treated identically by the Call Machine and so are represented in this machine by the common alias *MODIFY*. In this way, the difference between these events has been abstracted away in the context of this machine.
The alphabet of the Saved Number Machine is:
$A_{SavedNumberMachine} =$
{ *FINDCALL, INSERT, UPDATE, DELETE* }

Behaviour. A machine rendered in CPN is quiescent when no transition is enabled. An event in the alphabet of the machine placed on an interface place is accepted by the machine iff placing the token causes a transition to become enabled. Otherwise, the event is refused *and the event (token) disappears from the interface place*. Notice that this differs significantly from traditional CPN semantics according to which tokens that do not enable any transition are preserved until they can enable a transition. For example, if the model in Figure 2 is in the local state
{*(Call Machine.Call enabled,* $1`e$)}
then in response to event *("Insert","Ann", 123)* the local storage can be changed to
{*(Call Machine.Call enabled,* $1`e$) ;
(Save Number Machine.Saved, ("Ann", 123))}.

However, the event *"Initialise"* would be refused because the system is in the state *Call enabled* and no transitions are enabled by this event.

Determinism. The CPN with PM semantics are deterministic. Recall that the behaviour of a conventional CPN is a *non-deterministic*. In this sense, PM semantics is more restrictive than conventional CPN semantics. Requiring determinism is the price we pay for being able to argue about behaviour based on analysis of traces, as described below in Section 6.

In order to ensure determinism of behaviour, a CPN with PM semantics should be designed so that, when in a given quiescent state (with no transitions enabled), placement of an event into an interface place can enable at most one transition. If placement of an event were to enable more than one transition, the behaviour would become non-determined because it would depend upon which is chosen to fire. One way of doing this (and the approach adopted here) is to ensure that there is only one interface place for each event type, and that a token placed on one of these can enable only one transition.

Local Storage. The local storage of a CPN with the PM semantics is the current marking of the CPN (excluding markings of the interface places). The local storage of the model in Figure 2 is
{*(Call Machine.Call enabled,* $1`e$)},
where $1`e$ presents an empty token with no attributes [14].

Extension Extension processes can be rendered as a CPN notation, but with *conventional* CPN semantics (as extension processes are not protocol machines). The extension process for *ENDCALLSAVE* is shown in the middle of Figure 2. In this process, access to the local state *Call Machine.Calling1* is used to guarantee the quiescence of the Call machine prior to generation of the *END* event; and similarly access to the local state *Call Machine.Call enabled* is used to guarantee the quiescence of the Call machine at prior to generation of the *INSERT* event. In addition, access to the local state *Call Machine.Calling1* is used to generate the attribute *NUMBER* of the event *INSERT*.

5.2 State Chart PM Rendering

Events and Alphabet. In the State Chart rendering an event of type X is represented by a label X on an arc. The alphabet of a machine is the set of such labels. For instance, in Figure 3 the alphabet of the Call Machine is the set *INITIALISE, ON, OFF, CALL, FINDCALL, END, ENDCALLSAVE, MODIFY*.As with the CPN version, events of types *INSERT, UPDATE, DELETE* are treated identically by the Call Machine and so are represented in this machine by the common alias *MODIFY*.

Behaviour In State Charts a transition is represented by an arc with a label showing the event-type that triggers the

transition. If, when an event in the alphabet of the machine is presented, the current local state of the machine has an output arc labelled with the type of the presented event then this event is accepted and the machine transitions to the new local state to which the arc is directed. If the current local state of the machine has no output arc that the event can fire, then the event is refused.

Determinism. In order to ensure determinism of behaviour, a machine should be designed so that there is at most one transition that can fire for a given current state and event-type.

Local Storage. The local storage is represented as a list of attributes, as shown in the upper and lower panels of Figure 3. For example, the local storage of the Call machine is $\{Phone\ Identifier; Number\ Called\}$.

Extension. Extension processes are represented as pseudocode, as shown for the ENDCALLSAVE event in the central panel of Figure 3.

6 Reasoning about Behaviour and Model Evolution

In this section we discuss the formal relationship between a protocol machine that is created by composition and the component machines used to create it. In particular, we show how it is possible to reason about the behaviour of a composite machine based on the behaviour of its components.

Preliminary Definitions. For the purposes of this section, we define some terms.
1. Two events are *equal* iff they have the same event-type and the same values for corresponding attributes.
2. An *event sequence* is an ordered sequence of events.
3. Two event sequences S_1 and S_2 are equal if they contain the same number k of events and the nth event in S_1 is equal to the n-th event in S_2 for all $1 \leq n \leq k$.
4. A protocol machine has a defined *initial state* for its local storage, in which all attributes of its local storage take initial values according to their type.
5. A *trace* of a machine is an event sequence accepted by the machine starting from the initial state. A trace can be finite or infinite.

Trace Dependent and Trace Independent PMs. In order to reason about the traces of machines, it is necessary to distinguish two types of machine: *Trace Independent* and *Trace Dependent*. A trace independent machine is one whose protocol behaviour (whether it accepts, refuses or ignores an event presented to it) is a function of, and only of, the trace history of the machine, where "trace history" is the sequence of events so far accepted by the machine. A machine that is not trace independent is trace dependent.

In informal terms, a trace independent machine is one that is able to determine whether to accept or refuse an event based only on its own local storage. It does not need to access (read-only) the local storage of any composed machine. In the mobile phone case study, both the Call Machine and the Saved Number Machine are trace independent.

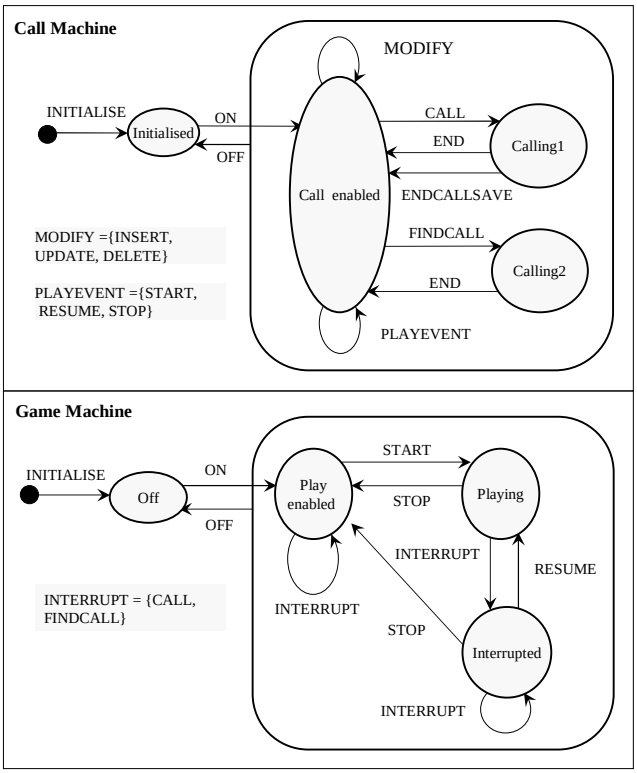

Figure 4. Call Machine and Game Machine

Observational Consistency and Local reasoning. Suppose that PM_1 and PM_2 are protocol machines and that PM_1 and $PM_1 \parallel PM_2$ are both trace independent, then $PM_1 \parallel PM_2$ is *Observationally Consistent* [5] with PM_1. This means that if T is a trace of $PM_1 \parallel PM_2$ and we remove from T the events that are not in the alphabet of PM_1, we will obtain a trace of PM_1 [1].

It has been shown by Hoare that, for deterministic processes, "...the mathematical model (of processes) is based on the relevant directly or indirectly observable properties of the process. These certainly include its alphabet and traces; but for a *nondeterministic* (our emphasis) process there are also its refusals and divergences" [4]. So,

as a set of traces is the mathematical model of a trace independent deterministic process, we use this as the basis for comparison when we study equivalence or relationship of two protocol machines.

The fact that the result of the composition of two protocol machines $PM_1 \parallel PM_2$ is observationally consistent with PM_1 has an important practical consequence: composing another machine with PM_1 cannot "break" its trace behaviour. In other words, any reasoning about behaviour that is predicated on the trace behaviour of PM_1 being observed remains true when considering the behaviour of $PM_1 \parallel PM_2$. This constitutes a form of *local reasoning*, allowing reasoning about behaviour of the whole based only on consideration of the behaviour of its parts. Local reasoning of this kind is an important in facilitating separate modelling of the parts of a software system, and in retaining control over complexity as the model grows.

Model Evolution. Composition provides natural support for model evolution. For example, if we want to extend our system by adding a Game Machine, we can do this without changing the Call Machine. We need to build an independent Game Machine and synchronize it with the Call machine.

A Game Machine is shown in Figure 4. A game can be started when a call is enabled. A game is interrupted by any call event of type *CALL* or *FINDCALL*. The events *START, STOP, RESUME* of the Game Machine are treated identically by the Call Machine and have the alias *PLAYEVENT*. One of the possible traces of the result of the composition is

INITIALISE, ON, START, CALL, END, RESUME, STOP.

Hiding the event *END* that do not belong to the alphabet of the Game Machine and replacing event *CALL* by its alias *INTERRUPT*, we derive the sequence of the Game Machine:

INITIALISE, ON, START, INTERRUPT, RESUME, STOP.

Both machines are trace independent with respect to each other, so the trace behaviour of both machines is preserved in the result of the composition.

7 Conclusion

In this paper we have shown that the Protocol Modelling semantics supports a compositional modelling approach for a large class of embedded system, where behaviour is interactive and deterministic. We have outlined how the compositional approach supports local reasoning about the whole based on consideration of behaviour of the parts, and how composition can be exploited to grow models in an evolutionary style.

We have also shown that Protocol Modelling semantics is notation independent, and how it can be expressed in two widely used behavioural notations: CPN and State Charts. Existing tools for those notations, for example, the CPN tools [14], could be used for modelling, analysis and composition of separate protocol machines, provided that support the semantics of event acceptance and refusal and CSP machine composition (as described in Section 4) were to be added.

In this paper we have illustrated the merits of Protocol Modelling semantics in order to bring this approach to the attention of the embedded systems modelling community. We would like to invite the embedded systems modelling community to explore and develop the ideas further.

References

[1] A. McNeile, N. Simons. State Machines as Mixins. *Journal of Object Technology*, 2(6):85–101, 2003.

[2] A. McNeile, N. Simons. Protocol Modelling. A modelling approach that supports reusable behavioural abstractions. *Software and System Modeling* , 5(1):91–107, 2006.

[3] B. Liskov, J. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[4] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[5] J. Ebert, G. Engels. Dynamic models and Behavioural Views. *LNCS 858*, 1994.

[6] K. Jensen. *Coloured Petri Nets*. Springer, 1997.

[7] L. Kristensen, J. Jørgensen, K. Jensen. Application of Coloured Petri Nets in System Development. *J.Desel, W.Reisig and G.Rosenberg (Eds) ACPN 2003, LNCS 3098*, pages 626–685, 2003.

[8] M. Elkoutbi, R. Keller. Modeling Interactive Systems with Hierarchical Colored Petri Nets. *Proc. of the Conference on High Performance Computing, Boston.*, 1998.

[9] M. Jackson. *System Development*. Prentice Hall, 1983.

[10] P. Koopman, H. Choset, R. Gandhi, B. Krogh, D. Marculescu, P. Marasimhan, J. Paul, R. Rajkumar, D. Siewiorek, A. Smailagic, P. Steenkiste, D. Tomas, C. Wang. Undergraduate Embedded Systems Education at Carnegie Mellon. *ACM transactions on Embedded Computing Systems*, 4(3):500–528, 2005.

[11] P. Palanque, R. Bastide, L. Dourte, C. Sibertin-Blanc. Design of User-Driven Interfaces Using Petri Nets and Objects. *LNCS 685*, 1993.

[12] R. Milner. *Communicating and Mobile Systems - the Pi-Calculus*. Cambridge University Press, 1999.

[13] S. Cook, J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.

[14] University of Aarhus. CPN group. CPN tools. *http://wiki.daimi.au.dk/cpntools/cpntools.wiki*.