Requirements Engineering for Reactive Systems with Coloured Petri Nets: the Gas Pump Controller Example*

João M. Fernandes[1], Simon Tjell[2], and Jens Bæk Jørgensen[2]

[1] Dept. of Informatics, Universidade do Minho, Braga, Portugal
[2] Dept. of Computer Science, University of Aarhus, Aarhus, Denmark
jmf@di.uminho.pt, tjell@daimi.au.dk, jbj@daimi.au.dk

Abstract. The contribution of this paper is to present a model-based approach to requirements engineering for reactive systems, and more specifically to controllers. The approach suggests the creation of a CPN model based on several diagrams, for validating the functional requirements of the system under development. An automatic gas pump controller is used as case study. We propose a generic structure for the CPN model to address the modelling of the controller, the physical entities which the controller interacts with, and the human users that operate the system. The CPN modules for modelling the behaviour of the human users and the controller are instances of a generic module that is able to interpret scenario descriptions specified in CPN ML.

1 Introduction

A reactive system is “a system that is able to create desired effects in its environment by enabling, enforcing, or preventing events in the environment” [9]. This characterisation implies that in requirements engineering for reactive systems it is useful, and often necessary, to describe not only the system itself, but also the environment in which the system must operate [1].

In this paper, we are particularly interested in controllers, i.e., a type of reactive systems that control, guide or direct their environment. This work assumes that a controller (to be developed) and its surrounding environment are linked by a set of physical entities, as depicted in fig. 1. This structure clearly identifies two interfaces A and B that are relevant to two different groups of stakeholders, users and developers, in the task of requirements analysis.

From the user’s or client’s point of view, the system is composed of the controller and the physical entities. Typically, the users are not aware of this separation; they see a device and they need only to follow the rules imposed by interface B to use it. In fact, they may not even know that there is a computer-based system controlling the system they interact with.

* This research work was conducted while J.M. Fernandes was on a sabbatical leave at DAIMI, University of Aarhus and was partly supported by project SOFTAS (POSC/EIA/60189/2004).

From the developer's point of view, the environment is also divided in two parts with different behavioural properties; the physical entities have predictable behaviour while the human actors may exhibit disobidience with respect to their expected behaviour. Thus, the description of the behaviour of the environment must consider the physical entities (usually called sensors and actuators) which the system interacts with through interface A. In some cases, these physical entities are given, and software engineers cannot change or affect them during the development process, but need to know how they operate. Additionally, some relevant behaviour of the human users that interact with the system through interface B must be taken into consideration and actually reflected in the CPN model.

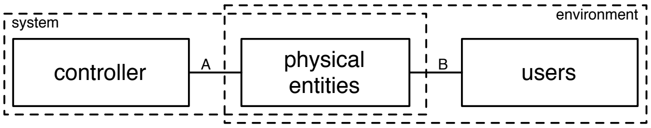

Fig. 1. A controller and its environment.

This paper presents a model-based approach to requirements engineering for reactive systems, and more specifically to controllers. The approach aims at obtaining a CPN model that describes the requirements through scenarios combined with a description of the behaviour of the physical entities which the controller interacts with. We propose a generic structure for the CPN model to hold two important properties: (1) *controller-and-environment-partitioned*, which means that it constitutes a description of both the controller and its environment, and that it distinguishes between these two domains and between desired and assumed behaviour; (2) *scenario-based*, meaning that it was constructed on the basis of the behaviours described in scenario descriptions. Our proposal continues the results presented in [8, 3, 2] and is illustrated in the development of a gas pump controller, which is a well-known example in the literature [4].

The paper is structured as follows. Sect. 2 introduces the Automatic Gas Pump case study that is used in this paper. In sect. 3, we present the main requirement models, in the form of use case diagrams and sequence diagrams, that were created for the case study. The CPN model for the case study, obtained with our approach, is discussed in sect. 4. We make some conclusions in sect. 5.

2 Case Study

As case study, we consider an Automatic Gas Pump, which is a computer-based system that permits customers to buy fuel in a self-served way. There exists one storage tank for each type of fuel (diesel, gasoline 92 octane, and gasoline 95 octane). The pump must be deactivated for a given type of fuel, when the quantity of fuel in the associated tank is less than a given threshold (to be defined). There are also three different nozzles, one for each type of fuel.

To fill a car's tank with fuel, first the customer must insert a credit card and introduce the PIN code. If the card is valid and the introduced PIN code is correct, the customer may start to fill the car's tank with fuel, by picking a nozzle. When a given nozzle is picked by the customer, the price per litre of the respective type of fuel is shown in the display. While the fuel is being pumped, the pump must show in real-time the quantity pumped and the respective price.

After the nozzle has been returned to the holster, the credit card company is contacted and requested to withdraw from the customer's account an amount equal to the price of the fuel that has been tanked and to credit it to the station's account (the credit card company retains a fixed percentage of the transaction that is deduced to the station). The customer may also get a printed receipt, if the same credit card is reinserted in the pump, no later than five minutes after returning the nozzle.

Based on the general structure of a reactive system (fig. 1), our approach suggests the development to be started by creating a so-called entity diagram, that depicts the controller system to be developed and all the entities in its environment.

This entity model, which can be seen like a context diagram as proposed by several software methods, has an important role in the approach, since it defines without ambiguities the scope of the controller and identifies the entities that exist in its environment. This clear separation between controller and environment must be preserved in the subsequent models, since our aim is to obtain a CPN model that is controller-and-environment-partitioned.

Fig. 2 shows the entity diagram for the case study. It clearly identifies the name and direction of each message that flow in interfaces A or B. This diagram serves as a reference for the development process and in the next sections for each diagram proposed we identify which parts of the entity model is being addressed.

3 Use Cases and Scenarios for the Case study

In this section, we show the artefacts (models and diagrams) that we suggest to use before constructing the CPN model. These artefacts allow the developers to formalise the user requirements and serve as a basis for obtaining a CPN model. The artefacts are shown here in a specific (ideal) order, but in an engineering development context it is expected that an iterative process must be followed.

The use case diagram for the automatic gas pump controller is depicted in fig. 3. With respect to fig. 1, the use case diagram covers interface B (between the users and the system), and identifies the functionalities provided by the controller.

The use cases are briefly described below:

- **UC1 buy fuel** permits the customer to fill the car's tank with the chosen type of fuel.

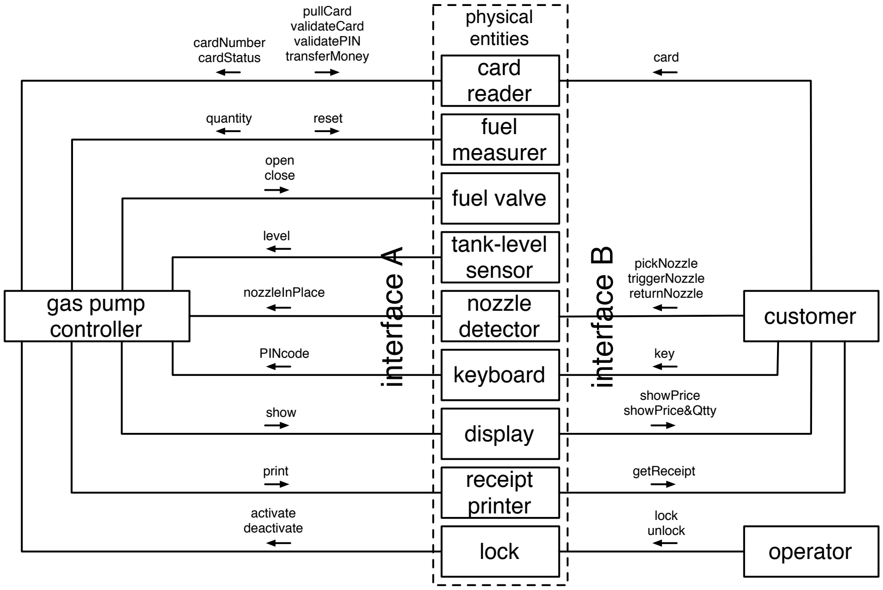

Fig. 2. An entity diagram for the Automatic Gas Pump, with a clear identification of the messages that flow in interfaces A (between the Gas Pump Controller and the Physical Entities) and B (between the Physical Entities and the Users).

- **UC2 initiate payment** validates if the customer has a valid credit card and if its PIN code is correctly entered in the keyboard. If this is the case, the pump is unblocked to allow fuel to be pumped.
- **UC3 get receipt** prints a receipt, if the customer reinserts the credit card no later than five minutes after returning the nozzle to its resting position.
- **UC4 de/activate pump** activates or deactivates the pump. The state of the pump must be easily visible to the customer.

As usual, the use case diagram identifies and names the use cases that the gas pump controller must support, and shows the external actors participating in the use cases. The actors in the use case diagram are the humans, customers and operators, that use the gas pump.

To describe the individual use cases in detail, their textual descriptions can be supplemented with sequence diagrams that specify some behavioural scenarios accommodated by the use cases. The scenarios describe desired behaviour of the gas pump controller in its interaction with the human actors and cover interface B with respect to fig. 1. These scenarios are thus adequate to be discussed with the client and also the final users of the system, since they permit a graphical and easy-to-understand representation of the user requirements, and omit design and implementation issues.

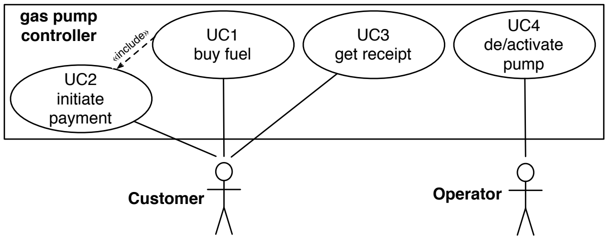

Fig. 3. Use case diagram for the Automatic Gas Pump controller.

As an example, the description of the main scenario for UC1 is presented next, including references to the sequence diagram that is depicted in fig. 4(a):

1. The customer starts the payment by introducing a valid credit card and typing the corresponding PIN code;
2. If the credit card is valid and the PIN code correct, the customer picks the nozzle of the wanted type of fuel;
3. The system shows the information related to the selected type of fuel (price per litre) and “0” as the number of litres pumped;
4. While the nozzle is being used, the customer can pump fuel to the car’s tank and the system updates the display showing the volume of pumped fuel and its respective price;
5. When the customer finishes pumping fuel, he returns the nozzle to its rest position (in the holster);
6. The system withdraws the amount corresponding to the price of the pumped fuel from the customer’s account, retains its commission, and credits the rest to the station’s account.

Alternative scenarios for a use case can be created, namely when it is sufficiently rich and complex. Fig. 4(b) shows an alternative scenario for UC1 that describes a situation where the user initially introduces a valid credit card, types its correct PIN code, picks a nozzle, but cancels the transaction by returning the nozzle to the holster (i.e., without putting fuel in the car’s tank). Therefore, at the end, the system does not transfer money from the customer’s account to the account of the station.

Similar textual descriptions and sequence diagrams exist for the other use cases. There is a dependency relationships between UC1 and UC2, meaning that to complete its execution, UC1 needs the functionalities provided by UC2. This dependency is specified in the use case diagram by an *include* relation and in the sequence diagrams by *ref* operators.

The next step in the modelling process is to refine the scenario descriptions, to introduce more detailed information in order to permit further development tasks to be conducted. In our approach, this entails two different things. Firstly, it is important to refine the user-level sequence diagrams by indicating the particular physical entity with which the users do exchange messages at each point in time.

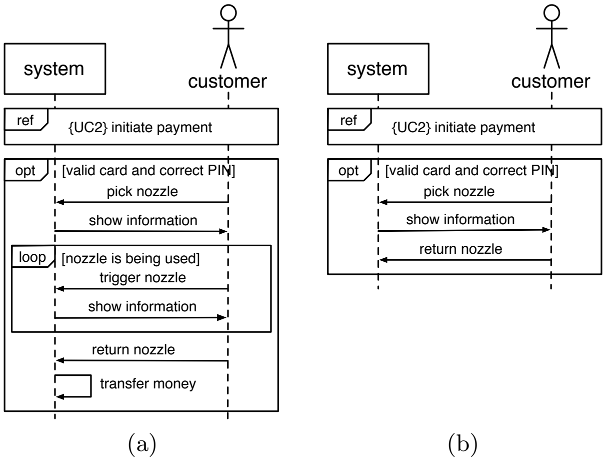

Fig. 4. Sequence diagram at the user level for UC1: (a) the main scenario, and (b) an alternative scenario.

The sequence diagram depicted in fig. 5 is an example of a scenario that details the exchange of messages in interface B. This can be seen in contrast to the diagram in fig. 4, where the user exchanges messages with the whole system, seen as a monolithic structure.

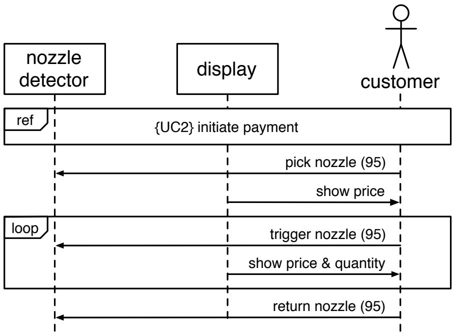

Fig. 5. The behaviour of the customer and the physical entities, during the main scenario of UC1.

Secondly, the messages that flow in interface A need also to be considered in our approach. This permits developers to introduce details about how the controller actually reacts to stimuli from the physical entities, that were supposedly initiated by the user. These refined scenario descriptions can be considered as part of the system requirements. The sequence diagram depicted in fig. 6 is an example of a scenario that details the exchange of messages in interface B. This diagram is used to specify the behaviour of the gas pump controller, and more particularly the interaction of the controller with the physical entities. Therefore, the controller must be considered as the central element of that sequence diagram.

In summary, sequence diagrams as the one shown in fig. 5 describe requirements expressed as scenarios for the use cases, while sequence diagrams like the one in fig. 6 should be considered as specifications for a given scenario of a use case. This distinction assumes that "a requirement is a desired relationship among phenomena of the environment of a system, to be brought about by the hardware/software machine that will be constructed and installed in the environment, while a specification describes machine behaviour sufficient to achieve the requirement" [6].

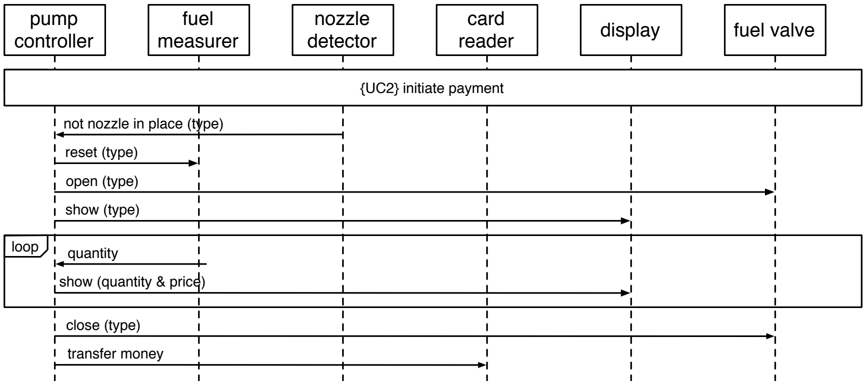

Fig. 6. The behaviour of the gas pump controller when interacting with the physical entities, during the main scenario of UC1.

4 The CPN model for the case study

The next development step is to construct a CPN model that represents all the behaviours described by the collection of considered sequence diagrams. The CPN modelling language was chosen, since CPN models are executable and formal, can provide a good balance between graphical and textual constructs, can address both the behaviour and the data of the system, and handle modelling aspects such as concurrency and locality in a graceful manner [7].

The construction of the CPN model is based on scenarios, which is important to guarantee that the model reflects all the partial behaviours identified and discussed with the clients and users of the system under development. Additionally, the CPN model must be structured in such a way that the separation between the controller and the environment, as expressed in fig. 1, is preserved and easy to identify. Therefore, the approach ensures that the CPN is constructed to be controller-and-environment-partitioned and scenario-based.

4.1 Top-level Module

Fig. 7 shows the topmost module of the hierarchical CPN model for the case study, constructed from the sequence diagrams and following the structuring principles proposed in this paper. The module contains three substitution transitions: Human Actors, Physical Entities, and Controller. These three substitution transitions represent different domains and are used for modelling the functional requirements, the behavioural domain knowledge, and the behaviour of the controller, respectively.

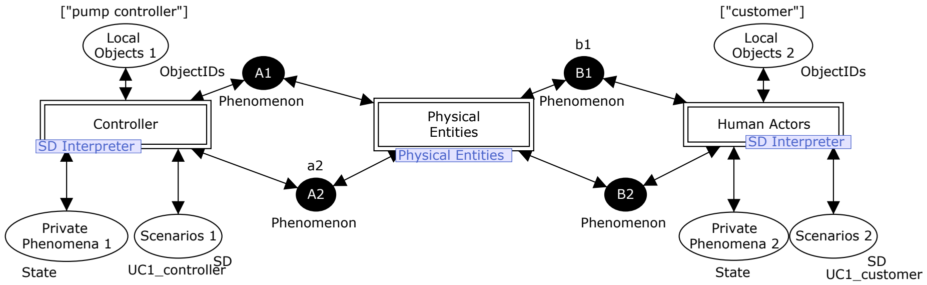

Fig. 7. The topmost module of the CPN model

The structure in fig. 7 is generic to reactive systems with a close interaction with the physical environment and operated by human actors. The structure embodies the guidelines that we are proposing for the modelling of such systems, their requirements, and their environment. The basic idea of the structure is to assist the modeller in maintaining a proper separation between the three modelling domains. The structure allows the description of scenarios for the behaviour of human actors and for the behaviour of the controller at an abstract level, both by means of high-level sequence diagrams, which are translated into a textual form for interpretation and execution by the Human Actors and Controller modules, respectively. Additionally, we use a regular CPN module (Physical Entities) for describing the behavioural properties of the physical entities through which the customer and the controller interact. By "regular", we mean a CPN that directly uses the graphical constructs (places, transitions, arcs, etc.) to describe the behaviour of the considered domain.

The three domains interact through a collection of shared phenomena [5]. A *shared phenomenon* is a state or an event that is observable by both domains while being controlled by only one domain. In contrast, a *private phenomenon* is only observable within the controlling domain (not to be confused with the controller domain). The controlling domain is the domain that is able to affect a shared phenomenon, i.e., to cause changes to a shared state or to generate a shared event. An observing domain is able to react on, but not affect, an observed phenomenon. No external domains are able to observe and thereby react on phenomena that are private to other domains. The shared phenomena perspective helps in the task of identifying the interfaces through which the domains are interacting. This allows us to enforce a strict partitioning of the representations of each of the domains in the CPN model, in order to make it useful for requirements engineering. In the top module of the CPN model (fig. 7), the interfaces of shared phenomena are emphasized as black places, each one denoted by a letter and a number:

A1: Shared phenomena between the controller and the physical entities, and controlled by the controller.

A2: Shared phenomena between the controller and the physical entities, and controlled by the physical entities.

B1: Shared phenomena between the physical entities and the human actors, and controlled by the physical entities.

B2: Shared phenomena between the physical entities and the human actors, and controlled by the human actors.

4.2 Physical Entities Module

The customer does not interact directly with the pump controller. In fact, he might not even be aware of the existence of a pump controller, i.e., a computer-based system controlling the system he interacts with. Instead, the customer does interact with the physical entities. The Physical Entities module is used for describing the behaviour of the actuators and the sensors that connect the controller with its physical environment. This behaviour is also referred to as the *indicative* (or given) properties of the environment; the physical entities have given behaviour patterns, which serve as a framework for the operation of the controller. These patterns of behaviour must be taken into account when the controller itself is designed, because they form part of the resulting behaviour of the environment when the controller is deployed. Furthermore, we consider that the physical entities are not integrated parts of the controller itself and this is the reason why they are explicitly modelled as a separate domain.

Fig. 8 shows the internals of the Physical Entities module (with just a subset of the entities). Each physical entity is represented by a substitution transition. Two internal states (the white places Nozzle Triggered and Fuel Valves) are used for modelling phenomena that are private to the physical entities; i.e., they are hidden from both the customer and the pump controller. The black and grey places are connected to the black places in the top module. A simple colour coding scheme is applied: black places hold locally controlled shared phenomena, while grey places hold remotely controlled shared phenomena.

Each substitution transition in the Physical Entities module encapsulates the behaviour of one particular physical entity; as an example, fig. 9 depicts the

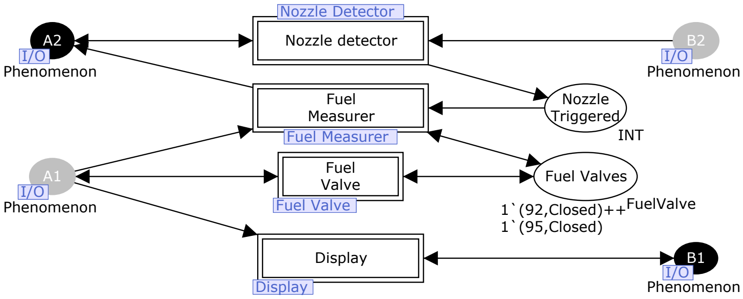

Fig. 8. The Physical Entities module.

module for the Fuel Measurer. The description is restricted by the fact that communication is performed exclusively through the interface of shared phenomena. The result is a collection of descriptions of the indicative behavioural properties of the physical environment. This part of the environment differs significantly from the part of the environment containing human actors (modelled in the Human Actors module) by the lack of free will and by the resulting deterministic nature. The physical entities exhibit strict reactive behaviour and do not generate events or change states in a spontaneous manner. Once the behavioural properties of the physical entities have been described, the descriptions can be maintained for executing various scenarios and for experiments with various possible design specifications for the pump controller.

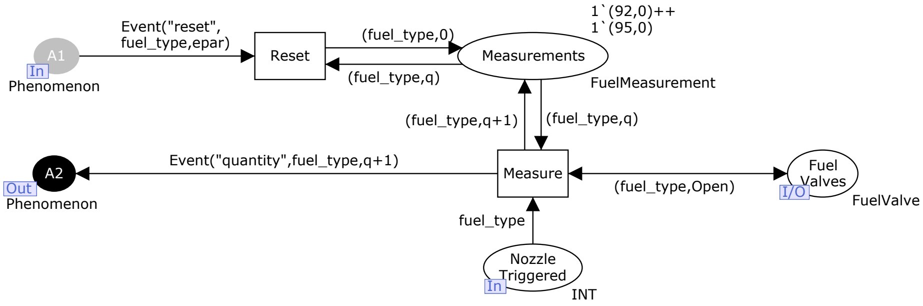

Fig. 9. The Fuel Measurer module.

In the approach to requirements engineering of reactive systems that we suggest in this paper, the modeller is only expected to specify CPN model structure, when the behaviour of the physical entities is being described. Everything else (i.e., controller and human users) is modelled by parameterizing a generic CPN module.

4.3 Controller and Human Actors Modules

Both the controller and the human actors are represented in fig. 7 by substitution transitions that refer to the SD Interpreter module. This generic module acts as an interpreter for textual CPN ML representations of the basic elements of UML 2.0 sequence diagrams. This module is utilized both for executing scenarios in which the user interacts with the system and for representing the behaviour of the controller.

As shown in fig. 7, the instances of the SD Interpreter module are parameterized through three places: one place specifies which objects (as found in the sequence diagram) are local to the instance (Local Objects 1 and 2), another place specifies possible private phenomena to the domain (Private Phenomena 1 and 2), and a third place specifies the behaviour as a scenario in the form of a sequence diagram (Scenarios 1 and 2). Each instance communicates with the physical entities through its set of shared phenomena. The communication consists of messages about the occurrence of events or changes to shared states. Furthermore, shared states can be part of the predicates used in the sequence diagrams.

Fig. 10 shows the internals of the SD Interpreter module. This module is basically the specification of a machine, which is able to execute sequence diagrams specified in CPN ML. The execution may be affected by incoming events and state changes and may itself cause state changes and generate events through the interface of shared phenomena. When a sequence diagram is executed, the modeller needs to specify which objects are local. All communication between local and non-local objects is performed through shared phenomena.

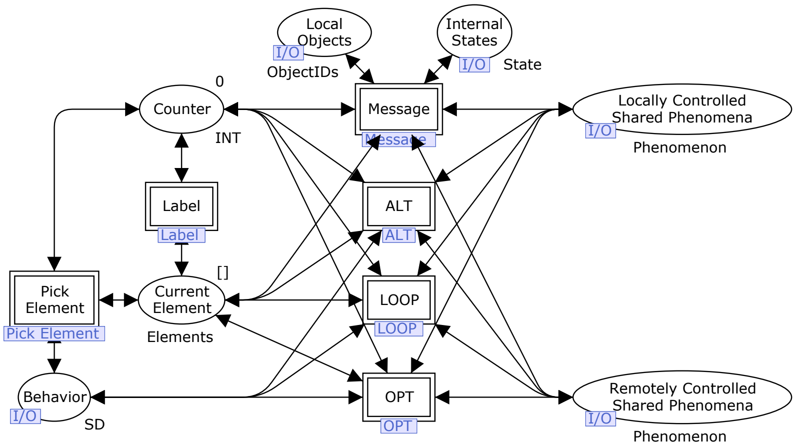

Fig. 10. The SD Interpreter module.

Fig. 11 illustrates the CPN ML representation of the sequence diagram in fig. 5 (a similar representation exists for the sequence diagram in fig. 6). This sequence diagram specifies a scenario of UC1 as seen by the customers through their interaction with the physical entities. Here, it has been translated into a list value, which is placed (as a token) in the place Scenarios 2 and interpreted by the Human Actors substitution transition.

A simple language has been developed to represent the basic features of UML 2.0 sequence diagrams, such as optionals (OPT), alternatives (ALT), loops (LOOP), and messages (arcs). Each of these features is handled by a separate substitution transition in the SD Interpreter module. The interpreter utilises the parameterised knowledge about local objects (and the derived implicit knowledge about remote objects) to determine the direction of messages (events or state changes) during the execution of the list representation of a sequence diagram.

If a message is outgoing (i.e., generated by a local object), this is reflected by the interpreter altering a local phenomenon, either by generating a new event token or by modifying the value of a state token in the place called Locally Controlled Shared Phenomena. Alternatively, if a message is incoming (i.e., generated by a remote object), the interpreter halts until this message is detected in the place called Remotely Controlled Shared Phenomena. This is the basic mechanism for the synchronisation of the instances of sequence diagram interpreters with the physical entities modelled in regular CPN modules. In the example of fig. 5, the customer object is local to the Human Actors instance, while the physical entities are remote. The ALT, LOOP, and OPT operators do not involve any exchange of messages, but rely on the interface of shared places in order to evaluate predicates that may involve shared states. When the interpreter encounters a predicate in one of these operators, the current value of a relevant shared state is investigated to evaluate the predicate.

The basic operation of executing the CPN ML representation of a sequence diagram as performed by the SD Interpreter module can be described as follows: The interpreter traverses the CPN ML list one element at a time. This is controlled by a counter (maintained in the place Counter) that somehow resembles a program counter. The substitution transition Pick Element picks out the next element of the list based on the current state of the counter found in the single token value found in the Counter place. A single element is produced in the Current Element place and from here it is consumed and handled by one of these substitution transitions based on its type: Message, ALT, LOOP, OPT, or LABEL. As an example of how specific elements are handled, Fig 12 shows the contents of the LOOP substitution transitions that handles the elements used for representing loop structures of arbitrary levels of depth found in sequence diagrams. In can be seen how shared phenomena are evaluated through access to the interface places described earlier (Remotely Controlled Shared Phenomena and Locally Controlled Shared Phenoma). This makes it possible for the interpreter to evaluate the predicates that may exist in the definition of a specific loop in order to determine when to enter and leave the loop based on shared phenomena.

Fig. 13 documents the collection of colour sets that are used through out the model.

```
val UC1_customer =
UC2_customer ^^
[
Message((”customer”,”nozzle_detector”),
 EventOccurrence(”pick_nozzle”,95,EventParameter(0))),
Message((”display”, ”customer”),
 StateChange(”display”, 0, AnyStateParameter)),
LOOP_HEAD(1,INT_(5),NoPredicate,”a”,”b”),
 Label(”a”),
 Message((”customer”, ”nozzle_detector”),
  EventOccurrence(”trigger_nozzle”, 95,EventParameter(0))),
 Message((”display”, ”customer”),
  StateChange(”display”,95,AnyStateParameter)),
LOOP_TAIL(),
Label(”b”),
Message((”customer”,”nozzle_detector”),
 EventOccurrence(”return_nozzle”,95,EventParameter(0)))
];
```

Fig. 11. The CPN ML representation of the sequence diagram found in fig. 5

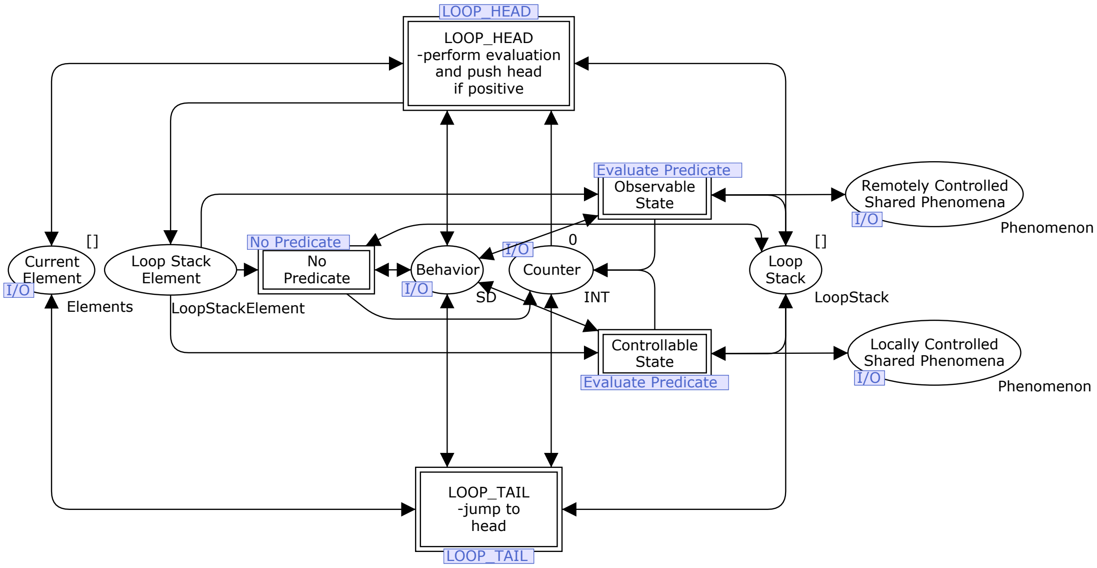

Fig. 12. The LOOP module.

```
colset ObjectID = STRING;
colset ObjectIDs = list ObjectID;
colset EventID = STRING;
colset Direction = product ObjectID * ObjectID;
colset EventParameter = INT;
colset OptionalEventParameter =
         union EventParameter:EventParameter + AnyEventParameter;
colset EventIndex = INT;
colset StateID = STRING;
colset Event = product EventID * EventIndex * EventParameter;
colset StateIndex = INT;
colset StateParameter = INT;
colset OptionalStateParameter =
         union StateParameter:StateParameter + AnyStateParameter;
colset StateChange = product StateID*StateIndex * OptionalStateParameter;
colset EventOccurrence =
         product EventID * EventIndex * OptionalEventParameter;
colset State = product StateID * StateIndex * StateParameter;
colset PredicateType =with NEQ | EQ | GT | LT | GTE | LTE;
colset StateChangeOrEvent =
         union EventOccurrence:EventOccurrence + StateChange:StateChange;
colset Message = product Direction * StateChangeOrEvent;
colset Predicate =
         product StateID * StateIndex * PredicateType * StateParameter;
colset OptionalPredicate =
         union Predicate:Predicate + NoPredicate + NonDeterministic;
colset Label = STRING;
colset Phenomenon = union State:State + Event:Event;
colset OPT_HEAD = product OptionalPredicate * Label * Label;
colset INTorINF = union INT_:INT + INF;
colset LOOP_HEAD =
         product INT * INTorINF * OptionalPredicate * Label * Label;
colset LOOP_TAIL = UNIT;
colset ALT_ELEMENT = product Predicate * Label * Label;
colset ALT_ELEMENT_ELSE = product Label * Label;
colset ALT_HEAD = Label;
colset ALT_TAIL = UNIT;
colset AltStackElement = product ALT_HEAD * BOOL;
colset AltStack = list AltStackElement;
colset Element = union Message:Message + Label:Label +
         OPT_HEAD:OPT_HEAD + LOOP_HEAD:LOOP_HEAD +
         LOOP_TAIL:LOOP_TAIL + ALT_HEAD:ALT_HEAD +
         ALT_ELEMENT:ALT_ELEMENT + ALT_ELEMENT_ELSE:ALT_ELEMENT_ELSE +
         ALT_TAIL:ALT_TAIL;
colset LoopStackElement = product LOOP_HEAD * INT * INT;
colset LoopStack = list LoopStackElement;
colset Elements = list Element;
colset SD = Elements;
colset Lst = list INT;
colset SCState = State;
colset OptionalEvent = union Event_:Event + NoEvent;
colset SCTransition =
         product SCState * OptionalEvent *
         OptionalPredicate * SCState * OptionalEvent;
colset SCStates = list SCState;
colset SCTransitions = list SCTransition;
colset SC = product ObjectID * SCState * SCStates * SCTransitions;
colset FuelValveState = union Open + Closed;
colset FuelValve = product INT * FuelValveState;
colset FuelMeasurement = product INT * INT;
```

Fig. 13. The colour sets used in the model

4.4 Discussion

The reflections behind the work presented in this paper are inspired by the work of Jackson and particularly by his work on Problem Frames [5]. The reactive system we deal with in this paper fits well in the Commanded Behaviour Problem Frame specified in [5] but our approach from the Problem Frames approach in a central point: we explicitly model the human actors observing the states and events of the domain of physical entities. This is necessary in order to synchronise the execution of scenarios between the human actors and the physical entities (and the system as a whole).

The main purpose of the CPN model we present in this paper is to provide the modeller of a reactive system with a generic structure that can be used as a starting point for capturing functional requirements and knowledge about the physical environment in a sensible way.

The requirements are specified as a collection of scenarios describing use cases in which the final system must be able to interact according to the expected behaviour. To validate the scenarios, the CPN model suggests the behaviour of the controller to be specified at a relatively-high abstract level. This permits to base a prototypical design of the controller on sequence diagrams that describe scenarios of use cases. At the same time, different sequence diagrams are used to describe scenarios of the behaviour of the human actors; and thereby required behaviour of the entire system consisting of the physical entities in combination with the controller.

The specification of the behaviour of the controller is relatively abstract, since it does not necessarily include descriptions of any internal components of the controller. At a later point in the development process, such components may be introduced by refining the sequence diagrams used to describe the controller behaviour (as the one found in fig. 6). The abstract description of the controller behaviour is necessary to permit the modeller to execute the scenarios specified for the human actors in a simulated environment with responses from the system. This is as an important property of the modelling approach, since it may be helpful in the complex task of specifying and validating the functional requirements.

5 Conclusions

The contribution of this paper is a model-based approach to requirements engineering for reactive systems. Its application is illustrated in an automatic gas pump controller. The approach suggests the creation of a CPN model based on the requirements expressed as use cases and sequence diagrams, for validating the functional requirements of the system under development.

A generic structure is proposed for the CPN model, so that it is possible to address the modelling of the controller, the physical entities which the controller interacts with, and the human users that operate the system. We suggest the CPN modules for modelling the behaviour of the human users and the controller to be instances of a generic module that is able to interpret scenario descriptions specified in CPN ML. This proves to be a good solution, since the size of the CPN module remains the same independently of the number of considered scenarios.

In contrast, for modelling the behaviour of the physical entities (actuators and sensors) we use regular CPN modules, i.e., modules that directly use the graphical constructs of the CPN language (places, transitions, arcs, etc.), to model behaviour.

The CPN language is a good choice for modelling these two types of modules, since it allows the complexity of the model to be split between graphical and textual constructs, and also between the data and the control perspectives.

As future work, we plan to extend the SD Interpreter module to handle all UML 2.0 sequence diagrams constructs, and also to apply our approach to other types of reactive systems, like for example interactive systems, workflow systems, and robotic systems.

References

1. J. Desel, V. Milijic, and C. Neumair. Model Validation in Controller Design. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 467–95. Springer, 2004.
2. J.M. Fernandes, S. Tjell, and J.B. Jørgensen. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. Technical report, DAIMI, University of Aarhus, Denmark, July 2007.
3. J.M. Fernandes, S. Tjell, J.B. Jørgensen, and O. Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *6th Int. Workshop on Scenarios and State Machines (SCESM 2007), at ICSE 2007*. IEEE CS Press, 2007.
4. D. Heimbold and D. Luckham. Debugging Ada Tasking Programs. *IEEE Software*, 2(2):47–57, 1985.
5. M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.
6. Michael Jackson and Pamela Zave. Deriving Specifications from Requirements: an Example. In *17th International Conference on Software Engineering (ICSE '95))*, pages 15–24, New York, NY, USA, 1995. ACM Press.
7. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Software Tools for Technology Transfer*, 2007. In Press. DOI: 10.1007/s10009-007-0038-x.
8. O. R. Ribeiro and J. M. Fernandes. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006)*, pages 237–56, 2006.
9. R.J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003.