Distinguishing Environment and System in Coloured Petri Net Models of Reactive Systems

Simon Tjell
Department of Computer Science
University of Aarhus, Denmark
Email: tjell@daimi.au.dk

***Abstract*— This paper introduces and formally defines the *environment-and-system-partitioned* property for behavioral models of reactive systems expressed in the formal modeling language Coloured Petri Net. The purpose of the formalization is to make it possible to automatically validate any CPN model with respect to this property based on structural analysis. A model has the *environment-and-system-partitioned* property if it is based on a clear division between environment and system. This division is important in many model-driven approaches to software development such as model-based testing and automated code-generation from models. A prototypical tool has been implemented for performing the structural analysis of Coloured Petri Net models and the principles of this tool is described. The aim of the paper is to make the guidelines and their formalized definitions along with a proof-of-concept for the automatic validation of the structure of models based on structural analysis.**

I. Introduction

This paper introduces a technique for checking for compliance with certain restrictions of the structure of models of reactive systems. These restrictions are defined by a reference model for requirements and specification. We can see this as our abstract model. The abstract model is used for checking concrete models that are expressed by means of the Coloured Petri Net modeling language. The main purpose of the restrictions (as they are applied here) is to ensure a clear division between the parts of the model that represent the system under development and the parts that represent the environment in which this system is to operate. This distinction is particularly important for reactive systems since this class of systems is characterized by a high level of interaction between the system and its environment. In [1], Wieringa defines a reactive system in the following way:

> A reactive system is a system that, when switched on, is able to create desired effects in its environment by enabling, enforcing, or preventing events in the environment.

When a model of a reactive system is to be used during the development of software for controlling a reactive system, the model is typically used for specification purposes. If the model of the controller being developed is combined with a (partial) model of its environment, it is possible to execute scenarios at very early stages in the development process using the model. When the system has been specified, the next step is to go from the model to the actual implementation of what is specified by the model. In this situation, it is clearly of great importance to be able to determine which parts of the model should be implemented by the system and which parts belong to the requirements to and assumption about the behavior of actors in the environment. In the case of reactive systems, these actors are typically both human users of the system and physical entities such as sensors and actuators.

When the system has been implemented, it is necessary to test it in order to assure that it complies with the specification. One way of doing this is by model-based testing, where the model is used for automatic generation of a large collection of test cases. This approach to testing is strictly black-box oriented and therefore, a clear identification of the interface between environment and controller is needed.

The contribution of this paper will hopefully be relevant to areas such as model-based development and model-based testing, where both environment and system are part of a composite model.

The paper will present a collection of formalized guidelines for models of reactive systems in the formal modeling language Coloured Petri Nets. The guidelines derive directly from an existing reference model for the specification of (reactive) systems. The main contribution of the work presented in this paper is the adoption and implementation of these guidelines into the context of Coloured Petri Nets. The adoption is documented and discussed and a prototypical implementation of tool-based support is presented.

The paper is structured as follows: The reference model is described in Section II and Section III gives an introduction to Coloured Petri Nets. These introductions are necessary for the understanding of how the reference model is adopted in the modeling language as described in Section IV. In Section V we describe different types of communication between the domains and define the rules for identifying legal and illegal communication. This identification is performed by the prototype application described in Section VI. In Section VII we discuss some related work before concluding in Section VIII where future work is also discussed.

II. The Reference Model for Requirements and Specifications

In [2], Gunther et al. formally define a reference model for specifying requirements to software system. This section will provide a brief introduction to the reference model. The idea behind defining the model resembles the idea behind

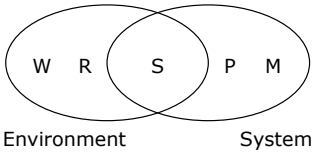

Fig. 1. The five software artifacts

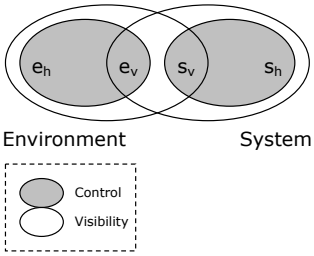

Fig. 2. Designated terms

the ISO seven-layer reference model for network protocol stacks [3]. This model has become widely accepted as the de-facto standard for discussing and describing types of network protocols.

The model is composed by five artifacts:

- **W** is the collection of all relevant domain knowledge (the world). This knowledge captures what is assumed about the behavior of the environment. An assumed behavior could be that when a motor is started, this will make something move.
- **R** represents all requirements that describe the desired behavior of the system. Desired behavior is described in terms of the system's expected impact on the environment.
- **S** denotes the specifications of the system. The system is implemented based on these specifications.
- **P** is the program that implements the specifications.
- **M** is the programming platform (the machine) on which the implemented program executes.

The Venn diagram in Figure 1 shows how two of the artifacts (*W* and *R*) mostly belong to the environment, two (*P* and *M*) belong mostly to the system and the fifth (*S*) exists in the interface between the two domains.

The relationship between the artifacts can be described by these two points:

1) The purpose of the program (*P*) is to implement the specifications (*S*) and this implementation is restricted by the programing platform (*M*).
2) If the program implements the specifications (*S*), then it will satisfy the requirements (*R*) if the environment is described by domain knowledge about the world (*W*).

The specifications provide a description of the interface between the environment and the system by means of a common terminology based on a collection of *shared phenomena*. A phenomena is an event or a state, which is controlled by one of the domains and which is either visible or hidden for the other domain. Figure 2 illustrates the different categories of phenomena.

A phenomena is *controlled* by a domain if the domain causes changes to the phenomena (in the case of states) or generates the phenomena (in the case of events). As seen in Figure 2, the phenomena in e_v and e_h are controlled by the environment while the phenomena in s_v and s_h are controlled by the system.

If a phenomena is *visible* in a domain, this means that reactive behavior in that domain could be initiated by the phenomena. On the other hand, a domain is not able to react based on *hidden* phenomena. To pin this out, the system is only able to react on phenomena that exist in e_v, s_v and s_h while the environment is only able to react on phenomena in e_h, e_v and s_v. This distinction is important in order to be able to identify the interface between the system and the environment.

When comparing Figure 1 and Figure 2 we see that the specifications (*S*) forming the interface between the environment and the system is formed by two collections of phenomena: e_v and s_v. These two collections are called the *shared phenomena*. In an *environment-and-system-partitioned* model, all interaction between the two domains is performed by means of shared phenomena. This paper focussed on the identification of shared phenomena in CPN models as we shall later see.

III. Coloured Petri Nets

This section will present an informal introduction to the syntax and semantics of Coloured Petri Nets, which will be followed by a brief formal definition of the syntax. A formal definition of the semantics can be found in [4].

Coloured Petri Nets (or CPN) is a graphical modeling language with formally defined syntax and semantics. A CPN model expresses both the states and the events of a system. Events are represented by *transitions* (drawn as rectangles) and the states are represented by *places* (drawn as ellipses). These two types of nodes are connected by *arcs* that lead either from a transition to a place or from a place to a transition. The graphical structure is supplemented by declarations of functions, constants etc. in a functional programming language. This language is called CPN ML [5] and is an extension of the more commonly known Standard ML [6]. Expressions in this language are also used for annotations with different purposes in the graphical nodes of the model.

We will now look into how state is represented and how it is changed. Each place is able to hold a (possibly empty) collection of values of a given data type. We call these values *tokens*. The current state of a given place is determined by the collection of tokens at a given point in (model) time. We can see this as a local state while the global state of a model is the composition of local states. The local states and thereby the global change when transition *fires*. A transition must become *enabled* before it is ready to fire. There are several restrictions involved in the determination of which transitions

are enabled. The main restriction is a quantitative one: the transition must be able to consume a number of tokens from all its *input place*. An input place to a transition is a place from which an arc leads to the given transition. A transition also have *output places*. Those are the places to which an arc leads from the given transition. The quantitative restriction is defined by the expressions in the arcs connecting the input places to the transition. A restriction is purely quantitative if it only expresses a required number of tokens to be *consumed* from each of the input places. A more precise restriction is a qualitative one, where requirements to the values of the tokens being consumed are expressed. Finally, the enabling of a transition could be further restricted by use of a *guard*. This is an expression that evaluates to a boolean value over the values of all tokens being consumed. A transition is only enabled if its guard evaluates to true. When a transition fires, the result of the state change is manifested by the production of new tokens in its output places. The values of these tokens may or may not depend on the values of the consumed tokens and depend on constants or function that are expressed in the outgoing arcs.

Here follows the formal definition of the syntax. This definition is important here, because the following restrictions will be based on it. As described in [4], the syntax of a non-hierarchical CPN is given by a tuple:

$CPN = (\Sigma, P, T, A, N, C, G, E, I)$

satisfying the following properties:

1) Σ is a finite set of non-empty *types* (colour sets)
2) P is the finite set of all *places*.
3) T is the finite set of all *transitions*.
4) A is the finite set of all *arcs* s.t. $P \cap T = P \cap A = T \cap A = \emptyset$
5) N is the *node* function s.t. $N : A \rightarrow P \times T \cup T \times P$
6) C is the *colour* function s.t. $C : P \rightarrow \Sigma$
7) G is the *guard* expression function s.t. $G : T \rightarrow Expr$, $\forall t \in T : [Type(E(a)) = \mathbb{B} \wedge Type(Var(G(t))) \subseteq \Sigma]$
8) E is the *arc expression* function s.t. $E : A \rightarrow Expr$, $\forall a \in A' : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$ where $p(a)$ is the place of $N(a)$
9) I is the *initialization* function s.t. $I : P \rightarrow Expr$, $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$

We define the A' relation that relate all pairs of nodes that are connected by arcs s.t.: $A' \subseteq (P \cup T) \times (P \cup T)$, $A' = \{(n_1, n_2) | \exists a \in A : N(a) = (n_1, n_2)\}$ The first component of an element in the relation is the source node and the second element is the destination node.

IV. Expressing the Reference Model in CPN

This section describes how the reference model is to be expressed by means of subsets of the nodes in a CPN model. We start out with a brief introduction of an example model that will be used to illustrate the principles being presented here. Figure 3 shows an example of a very simple CPN model. In this case, a vending machine has been modeled at a rather abstract level. The left-most white nodes model the behavior of the environment, which consists of a person who wants to buy a product from the vending machine. On the right-hand side, the white nodes model the behavior of the vending machine. In between the two domains, we see four dark-colored places (the colors have no associated semantics). These places form the interface between the system and its environment. One important thing to notice here is that places are not only used for representing states but also as abstract communication channels. This is for example the case when the information about the occurrence of an event is passed from the environment where it occurred to the system where it is observed. The event is passed through the *Environment Events* place, which is part of the interface. In general the places emphasized by the black fill in the figure are the elements of the interface *S*, which was described as an important part of the reference model. We will next look into how the phenomena in *S* are identified and categorized.

We start out by identifying how phenomena are represented in a CPN model. In [2], the term *phenomena* covers both states and events. In the CPN model we would like to enforce that all communication between the two domains (environment and system) be performed through places only - i.e. the interface between the two domains consists of a collection of places that are used for holding either states or information about the occurrence of events. In case of states, the approach is to make sure that a place representing a state always contains a token of which the value corresponds to the current state. This value is then changed by one of the domains - the controlling domain - when the state changes and can be read by the other domain if the state is part of the group of visible phenomena. By use of an ID component in the state tokens, it is possible to represent a group of states in one single state place. The modeling of events is done with an different approach. In this case, we want the controlling domain to produce a token to the event place when an event occurs. This token can then be consumed by the observing domain if the event is one of the visible phenomena. The approach is different from the modeling of states because a token will only exist in the event place when an event occurs and will be removed when it has been observed. Internally in the domains, both places and transitions are used to describe the hidden behavior and phenomena.

We start out by linking the two groups of hidden phenomena to the nodes of the CPN model:

$e_h \subseteq P \cup T$ and $s_h \subseteq P \cup T$

We do the same for the groups of visible phenomena:

$e_v \subseteq P$ and $s_v \subseteq P$

Notice that the set of visible phenomena is only composed by places (and not transitions). This difference represents and important property of the interface between the environment and the system. If transitions had been allowed as part of the interface, the occurrence of a transition representing the occurrence of an event in one domain would be able to

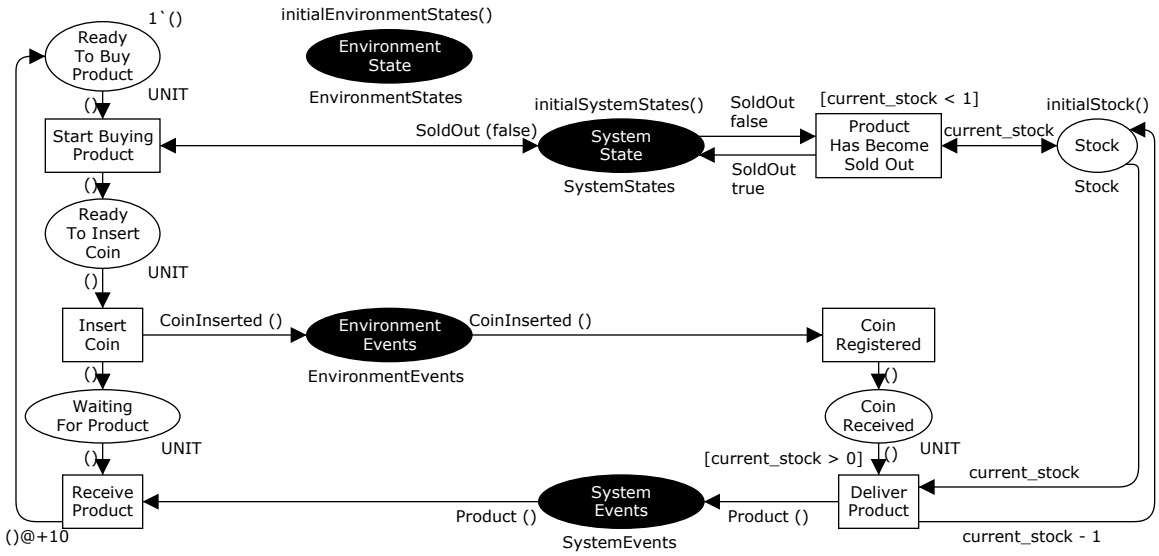

Fig. 3. A CPN model of a vending machine

directly trigger a reaction in the other domain. This would limit the level of modeling power in the model since the choice of reaction should be made internally in the reacting domain - and this would not be the case, if the reaction was triggered directly by a transition in the interface. Another advantage of limiting the interface to consist of places only is that every event that occurs in the composite model will be represented by a token. A disadvantage of the approach is that is caused by the standard semantics of (Coloured) Petri Nets; a transition is not forced to occur (fire) if it is enabled. A problematic consequence of this property could be that the system fails to react on events or state changes occurring in the environment in a timely manner - i.e. events may accumulate in the interface place and state changes may occur without the system observing it. In the specific example, this problem has been overcome by the introduction of a simple notion of time allowing us to specify a duration of the *Receive Product* transition (seen as the $()@+10$ inscription in its outgoing arc). Since all (observing) system transitions are specified with a zero duration, this simple solution enforces these transitions to occur instantaneously (when enabled) while the *Ready To Buy Product* transition will be disabled until the expiration of the duration of the *Receive Product* transition (10 time units). In this case, it has only been necessary to specify a duration of this one transition. In other cases, it might be necessary to specify durations for more environment transitions. It may also be done for some system transitions of interest in case it is wanted to make estimates about performance properties.

For convenience in the following, we define four subsets that define all hidden, all visible, all environment and all system nodes respectively:

$H = e_h \cup s_h$, $V = e_v \cup s_v$, $E = e_h \cup e_v$, and $S = s_h \cup s_v$

By environment and system nodes we mean nodes that belong in (and are controlled) by the environment and the system respectively.

In the vending machine example, *H* is the set of all non-filled nodes in the left and right-hand side of the figure while *V* is the set of all visible nodes; the places filled with black. *E* is the set of all nodes belonging to the environment; in this case all white nodes in the leftmost side of the figure along with the two black interface places *Environment State* and *Environment Events*. In the same way, *S* is the set of all system nodes; the white nodes in the rightmost side along with the two black interface places *System State* and *System Events*.

We define a collection of very trivial rules that restrict the categorizations of nodes in a model:

- No nodes belong to both the system and the environment: $S \cap E = \emptyset$
- No nodes are both hidden and visible: $V \cap H = \emptyset$
- All nodes can be categorized into two of these subsets: $S \cup E = V \cup H = P \cup T$

These rules are examples of the rules that are checked automatically by a tool of which a prototype has been implemented. This is described in Section VI.

Based on the system and environment categorization, we define the relation D relating pairs of nodes in which one node belongs to the system domain and the other belongs to the environment domain:

D is the cross-domain relation s.t.
$D \subseteq (T \cup P) \times (T \cup P)$,
$D = \{(n_1, n_2) | (n_1 \in S \wedge n_2 \in E) \vee (n_1 \in E \wedge n_2 \in S)\}$
We write $D(n_1, n_2)$ as shorthand for $(n_1, n_2) \in D$

V. Communication

We identify two more sets of places which are the places being used for communication between the two domains through shared events and shared states. A shared event is an event that is generated by one domain and observed by the other. A shared state is a state that is controlled by one

domain and observed by the other. All interaction between the two domains is performed through the collection of shared events and states. Figure 4 and 5 show examples of the two types of communication. It can be noted that the reading of a shared state (the arc with the inscription $e3$) is performed by a two-headed arc. This is syntactic shorthand for two arcs with the same inscription and is a semantically a replacement of two arcs with opposite directions and the same inscription between the place and the reading transition. The fact that the inscription is the same for both the input and the output arc to and from the transition ensures that the value of the state token is not modified by the reading transition.

We start out by identifying two sets of places; one for representing shared states and one for shared events:

- P_{SS} is the set of places representing shared states: $P_{SS} \subseteq V \cap P$.
- P_{SE} is the set of places used for exchanging shared events: $P_{SE} \subseteq V \cap P$.

These two sets of places hold all places through which the communication between the environment and the system is performed. We now move on to defining some restrictions to this communication. These restrictions are defined by identifying a collection of arc categories. All arcs to and from P_{SE} and P_{SS} should belong to exactly one of these categories. We also require that no place is used for both events and states: $P_{SS} \cap P_{SE} = \emptyset$.

A. Communication Through Shared States

Shared states are represented by tokens on places. The value of a token represents the current state of a given property. The state is changed by the controlling domain by changing the value of the token. The state is readable by the other domain. Since the state is reflected by the value of a token, this token should always be available at the place. Firstly, we define a group of categories of arcs that are involved in reading and writing the shared state:

- A_{SSW_1} is the set of arcs used for writing new values of shared states s.t.
  $A_{SSW_1} = \{(t,p) \in A' | t \in T, p \in P_{SS}, \neg D(p,t)\}$
  An example of such an arc is seen in Figure 4 (annotated with the expression $e1$).
- A_{SSW_2} is the set of arcs used for removing the token representing the current value of a state before writing the new value by placing a new token s.t.
  $A_{SSW_2} = \{(p,t) \in A' | t \in T, p \in P_{SS}, \neg D(p,t)\}$
  An example of such an arc is seen in Figure 4 (annotated with the expression $e2$).
- A_{SSR_1} is the set of arcs used for reading the current values of a shared states s.t.
  $A_{SSR_1} = \{(p,t) \in A' | t \in T, p \in P_{SS}, D(p,t))\}$
  An example of an arc in this set is seen in Figure 4 as part of the double arc with the expression $e3$. The part in question points from the shared state place to the reading transition. The semantics of a double arrow makes it possible for a transition to read a token without changing the state of the place holding the token - i.e. the token will be immediately returned to the place when the transition is fired.
- A_{SSR_2} is the set of arcs used for reproducing tokens of shared states after reading the current values s.t.
  $A_{SSR_2} = \{(t,p) \in A' | t \in T, p \in P_{SS}, D(p,t))\}$
  An example of an arc in this set is seen in Figure 4 as part of the double arc with the expression $e3$. The part in question points from the reading transition to shared state place.
- A_{SS} is the set of all arcs being used for communication through shared states s.t.
  $A_{SS} = A_{SSW_1} \cup A_{SSW_2} \cup A_{SSR_1} \cup A_{SSR_2}$

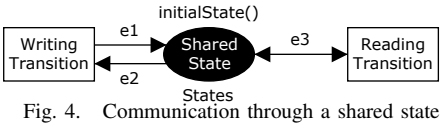

Fig. 4. Communication through a shared state

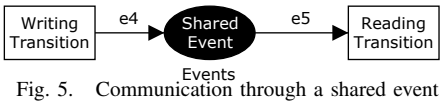

Fig. 5. Communication through a shared event

A shared state is controlled by one domain and read by the other domain. This means that only the controlling domain is able to change the current value of the state. The controlling domain is the domain in which the place holding the token representing the shared value belongs. If such a place belongs to the environment, it means that the shared state is controlled by the environment while it might be observed by the system. To ensure the visibility (and availability) of such states, the places holding the state values must be part of V. In Figure 4, the writing transition is in the controlling domain and the reading transition is in the observing domain. The double arc with the $e3$ annotation is shorthand for two arcs with the same expression. Since the expression ($e3$) is the same in both the arc removing a token from the shared state place and the arc placing a token back on the place, the value of the token will not be changed by the firing of the reading transition.

It should be ensured manually or automatically that places holding shared states will always (in any marking) contain a *multiset* (a bag) of constant cardinality (equal to the cardinality of the multiset created by the initial marking of the place. The reason why it is important that the tokens representing states are always present at the places is that they should always be available for reading by the observing domain. This is important in order to ensure the intended level of separation between the controlling and the observing domain. Provided that cardinality of the collection of tokens that are produced by output arcs can be predicted statically, the following rule should hold in order to ensure the availability of shared state tokens:

$|A_{SSR_1}| = |A_{SSR_2}| = |\{((p,t),(t,p)) \in A_{SSR_1} \times A_{SSR_2} | E(p,t) = E(t,p)\}|$

This property could be evaluated by use of place invariants or by bounding analysis of full state space. Alternatively, the property could be dynamically observed by monitoring the number of tokens on the relevant places runtime - i.e. during tool-based execution of the CPN model. We will not discuss this further here, but it remains as an open topic for future work.

B. Communication Through Shared Events

Like shared states, shared events are also represented by tokens on places - but in the case of shared events, these tokens are consumed when transitions representing the reading of events fire. The values of the tokens specify the type and possible data the events. Figure 5 shows an example of a place holding tokens representing shared events. The occurrence of a visible event in one domain is modeled by the firing of the writing transition. This results in the production of a token identifying the event in the shared event place. In the other domain, this can cause a reaction as modeled by the reading transition.

- A_{SEW} is the set of all arcs used for writing (generating) shared events s.t.
  $A_{SEW} = \{(t,p) \in A' | t \in T, p \in P_{SE}, \neg D(t,p)\}$
  An example of an arc in this set is seen in Figure 5 with the expression $e4$. This arc produces a token in the place holding shared events. One token represents one occurrence of a specific event.
- A_{SER} is the set of all arcs used for reading (consuming) shared events s.t.
  $A_{SER} = \{(p,t) \in A' | t \in T, p \in P_{SE}, D(t,p)\}$
  An example of an arc in this set is seen in Figure 5 with the expression $e5$. The reading transition becomes enabled when events have occurred in the controlling domain.
- A_{SE} is the set of all arcs being used for communication through shared events s.t.
  $A_{SE} = A_{SEW} \cup A_{SER}$

C. Internal Communication

We have now identified the communication that crosses the interface between the two domains and thereby conducts all interaction between the system and the environment. In both domains, the internal house keeping is performed by what we call internal communication. In this context, the term *communication* should be interpreted with a bit more abstraction since it covers both communication between internal components and the update and reading of internal (hidden) states. The common denominator is that this kind of communication is hidden and only visible from within the domain where it takes place. This is one the most important restriction that the work in this paper aims at enforcing by automation. We define A_{Int} to be the set of all arcs used for internal communication s.t.

$A_{Int} = \{(n_1,n_2) \in A' | \neg D(n_1,n_2)\}$

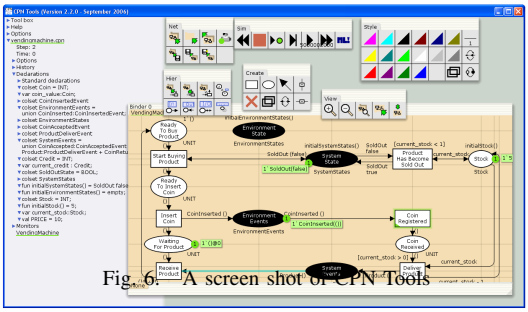

Fig. 6. A screen shot of CPN Tools

D. Legal Communication

After having identified three different types of communication (internal, through shared events and through shared states) it is now possible to sum up by identifying a basic rule over the set of arcs:

$A' \backslash (A_{Int} \cup A_{Ext}) = \emptyset$

where A_{Ext} is the set of all arcs used for external communication s.t. $A_{Ext} = A_{SS} \cup A_{SE}$

This rule is satisfied when all arcs in the model can be categorized by the three categories of legal communication. If this is the case, the model is said to be *environment-and-system-partitioned*.

VI. Implementing a Prototype

This section describes the development of an application that validates a CPN model with respect to the *environment-and-system-partitioned* property based on structural analysis. We will call this application the *validator*. One of the most commonly used tools for developing and analyzing CPN models is a graphical tool called CPN Tools [7]. Figure 6 shows a screen shot of CPN Tools. We will call this tool the *editor*. This tool uses a XML file format for storing and reading the structure of models (nodes, arcs, declarations, graphical layout, inscriptions etc.). The overall purpose of the validator is to analyze such a file with respect to the guidelines defined in this paper and provide the user with information about the level of compliance with the guidelines for the given model. This process is a performed as a combination of manual and automatic operations. The model is defined in the editor as the user is used to doing it. This means that the user will not experience any changes to the way the editor is used. At any point during the specification of a model, it should be possible to execute the validator.

Figure 7 shows a screen shot of the validator. The leftmost column contains the names of nodes that were encountered in the XML file. The following columns contain check-boxes that are used for specifying the memberships in the previously described categories. When this categorization has been performed by the user, the button *Check* is pressed. This will start the automated part of the process in which the validator applies all the formalized guidelines and responds with the result in the text field found in the bottom of the window. The screen shot shows a situation where the specification of the vending machine we discussed earlier has been loaded by the validator. The idea is that the validator is to operate seamlessly with the editor. This means that when the model is changed

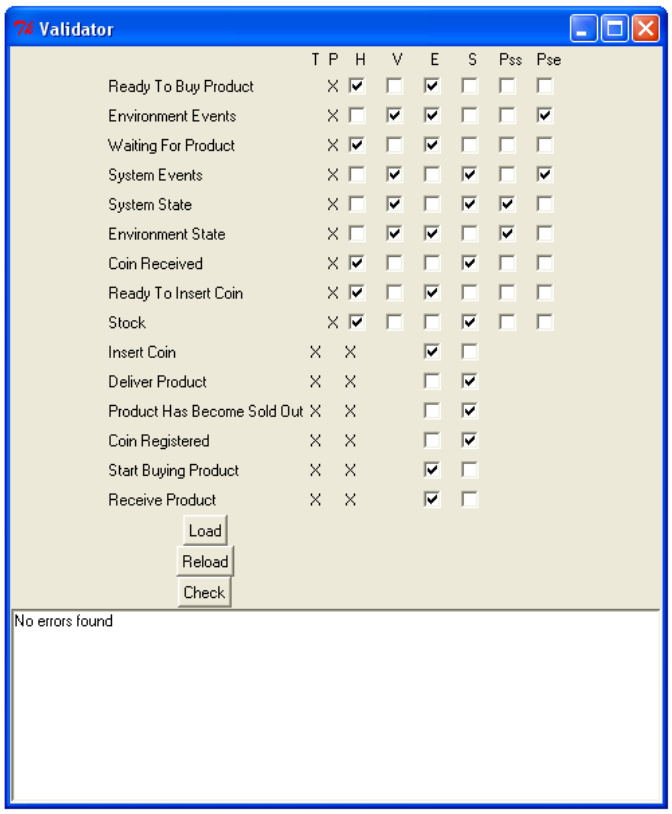

Fig. 7. A screen shot of the prototype implementation

by the editor, this should be reflected in the validator. For this purpose, the validator has the *Reload* button. When this button is pressed, the validator will parse the XML file again while - importantly - maintaining all categorization choices that were already made by the user. This means that if new nodes are added to the model, the user will only have to categorize these new nodes while already existing categorizations are preserved in the tool. This is also the case if a node is deleted from the model specification.

The response when an automated check of a model is performed is composed by a collection of error messages. In many situations, multiple error messages will occur in the response. The following list presents and describes the meaning of the error messages:

At least one node exists in both V and H
: A node has been found that is categorized as being both visible and hidden.

At least one node exists in both S and E
: A node has been found that is categorized as belonging to both the system and the environment.

At least one node exists in both Pss and Pse
: At least one node has been categorized as both a shared event place and a shared state place.

Not all nodes in Pss are in V
: One or more nodes that have been categorized as holding shared states is at the same time declared to be hidden. This makes it invisible to the non-controlling domain.

Not all nodes in Pse are in V
: A place holding shared events is not declared as visible.

All nodes must exist in either E or S
: Not all nodes have been categorized with respect to the environment and system subsets of nodes.

All nodes must exist in either V or H
: Not all nodes have been categorized with respect to the subsets of visible and hidden nodes. This error could also be part of the reason for the error messages telling that not all nodes used for shared phenomena have been declared as visible.

At least one arc is illegal
: This error message tells that something is fundamentally wrong about the structure of the model. It is based on the automatic categorization of arcs and is generated when one or more arcs could not be fitted into the two legal categories: internal and external arcs. On reason for the occurrence of this error would be if two hidden nodes in each their domain were connected by an arc.

No errors found
: None of the errors were detected.

When the checking of a model results in one or more error messages, this is a symptom of the existence of one or both of two possible problems:

1) the nodes of the model have been incorrectly categorized in the validator.
2) the structure defined in the editor contains arcs that represent illegal communication.

After having received the error messages from the validator, the person working with the two tools is now expected to attempt to eliminate their causes. The first possible problem is handled in the validator by manually assuring that all categorizations of nodes are consistent with the intended structure of the model. The second possible problem is more profoundly connected to the model itself and should be taken care of in the validator, where the structure should be manually evaluated in order to make sure that the system and the controller domains are clearly distinguishable and that all communication is performed through the specified interface.

The following listing informally describes the main flow of the validator:

1) Open XML file
2) Initialize DOM parser
3) Read and store all places ($\rightarrow$ P)
4) Read and store all transitions ($\rightarrow$ T)
5) Read and store all arcs ($\rightarrow$ A')
6) Repeat until the model is changed
   a) Let user define subsets of nodes (V, H, E, S, P_{SS} and P_{SE}) by toggling membership for each node
   b) Check compliance with rules for subsets of nodes (e.g. no node belongs to more than one domain

etc.)

c) Identify subsets of arcs based on the subsets of nodes (A_{Int} and A_{Ext})

d) Check compliance with the guidelines by assuring that $A' \backslash (A_{Int} \cup A_{Ext}) = \emptyset$

7) If the model is changed, restart from point 1 while preserving already registered memberships in the subsets of nodes.

In this description, the events of buttons being pushed are not explicitly present.

VII. Related Work

The work that has been presented in this paper is closest related to other work where the distinction between environment and system in models is explicitly handled and where the interface between the two domains is identified. It is also more generally related to work on modeling methodologies for reactive systems and structural analysis of (Coloured) Petri Nets.

A new class of low-level Petri Nets - Reactive Nets - is introduced in [8]. This is done as a step toward another new class - Reactive Workflow Nets - that is used for analysis of workflow systems with explicit distinction between environment and system. The authors argue that the standard semantics for Petri Nets is not suitable for modeling a workflow system by a composite model. Therefore, the standard semantics (called the *token-game* semantics) is supplemented by *reactive* semantics for the system. In the token-game semantics, a transition *may* fire when it is enabled. In the reactive semantics, a transition *must* fire when it is enabled. The authors suggest to use both semantics in combination - the token-game semantics for the environment and the reactive semantics for the system. This makes it important to be able to clearly distinguish the two domains in the model. These *may* and *must* terms are further explained in [9]. It would be very interesting to try to transfer this idea to the world of Coloured Petri Nets.

A comparison of three formalisms for modeling user-driven interfaces can be found in [10]. The authors recognize the importance of identification of the environment and the system and provide an excellent discussion about why the interface should be formed only by places (and not transitions) when a reactive system is modeled.

A good example of a relevant case study where Coloured Petri Nets are applied is found in [11]. In this paper, the authors describe the work with modeling a complex conveyor belt system. The model represents both a controller and a controlled process as part of the environment in which the controller operates. The authors describe how the interface between environment and system is identified. This is important in their work, because the goal is to (semi-)automatically create controller software based on Petri Net specifications. The approach is significantly different from the work that has been presented in this paper, since the interface is based on a collection of arcs rather than a collection of places.

VIII. Conclusion and Future Work

In this paper, an approach to systematized distinction between environment and system in composite models of reactive systems by use of Coloured Petri Nets has been presented. A formal definition has been provided along with a description of how this could be used for the implementation of a tool that automates part of the work necessary for specifying models in which the *environment-and-system-partitioned* property is present. There are still several important steps to take in the direction of automating the process. First of all, the approach should be applied to a number of real-world case studies in order to evaluate its feasibility and in order to have a more solid argument that the approach does actually add enhanced value to the models. In order to strengthen this argument, one of the future plans is to look into how the environment-and-system-partitioned property hopefully facilitates the use of (Coloured) Petri Nets in the context of model-based testing - especially focused at testing of nondeterministic systems where a clear identification of internal and external nondeterminism is necessary.

On a more general level, the future work involves the extension of the formalized guidelines to the field of hierarchical Coloured Petri Nets. This class is already indirectly covered because of the fact that any hierarchical CPN model can always be unfolded to a behaviorally equivalent non-hierarchical model [12]. The goal would be to identify an appropriate hierarchical structure for reactive systems and adopt the guidelines for use with hierarchical models. This approach has already been taken in a case study of an elevator controller (not yet published) but needs to be generalized and formalized.

References

[1] R. J. Wieringa, *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann Publishers, 2003.

[2] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A Reference Model for Requirements and Specifications," *IEEE Software*, vol. 17, no. 3, pp. 37–43, 2000.

[3] U. D. Black, *OSI: A Model for Computer Communications Standards*. Prentice Hall, October 1990.

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

[5] "Cpn ml," http://wiki.daimi.au.dk/cpntoolshelp/cpn_ml.wiki.

[6] R. Milner, R. Harper, and M. Tofte, *The Definition of Standard ML*. MIT Press, 1990.

[7] "Cpn tools," http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[8] R. Eshuis and J. Dehnert, "Reactive petri nets for workflow modeling," in *ICATPN 2003, 24th Int. Conf. on Application and Theory of Petri Nets*, ser. LNCS, vol. 2679. Springer-Verlag, 2003, pp. 296–315.

[9] D. Wikarski, "An introduction to modular process nets." *TR-96-019, International Computer Science Institute, Berkeley, CA*, pp. 1–51, Apr. 1996.

[10] P. A. Palanque, R. Bastide, L. Dourte, and C. Sibertin-Blanc, "Design of user-driven interfaces using petri nets and objects," in *CAiSE '93: Proceedings of Advanced Information Systems Engineering*. London, UK: Springer-Verlag, 1993, pp. 569–585.

[11] B. Däne, A. Mölders, A. Melber, and W. Fengler, "Modeling an industrial transportation facility with coloured petri nets," in *Manufacturing and Petri Nets (at ATPN 97)*.

[12] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, ser. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.