Developing Tool Support for Problem Diagrams with CPN and VDM++

Kristian Bisgaard Lassen and Simon Tjell

Department of Computer Science, University of Aarhus,
Aabogade 34, DK-8200 Aarhus N, Denmark,
krell@daimi.au.dk, tjell@daimi.au.dk,
WWW home page: http://daimi.au.dk

Abstract. In this paper, we describe ongoing work on the development of tool support for formal description of domains found in Problem Diagrams (PDs). The PDs are used for representing the structure and parallel decomposition of a software development problem while Coloured Petri Net (CPN) models are used for formal specification of assumed and desired behaviour of the domains found in the PDs. VDM++ is used for specifying algorithms for automatically translating PDs into CPN models, for exporting CPN models into an XML format, and for combining modified CPN models.

1 Introduction

PDs are an essential tool in the Problem Frames approach, which was introduced in [1] and further elaborated on in [2]. They are used for documenting the problem structuring and analysis activities that take place during the requirement engineering phase of software development projects. PDs extend Context Diagrams and contain information about the following aspects: (1) the domains (physical entities, people etc.) that exist in the surroundings of the machine being developed, (2) the interfaces formed by shared phenomena (states and events), and (3) an abstraction of a requirement about the behavior of the machine seen as reflected through its causal effects on the domains that surround it. Collections of PDs are used to document the decomposition of a problem - i.e. the identification of parallel subproblems that are found in the overall problem.

While PDs provide an excellent means to documenting the knowledge about structure in the problem domain, they intentionally leave out any description of behavior. Behavioral descriptions are important for two purposes: (1) to document expected behavior of the physical domains in the environment and (2) to capture requirements about the behavior of the environment as caused by the interaction of the machine and the existing behavioral properties of the domains of the environment.

In this paper, we describe an approach to providing behavioral descriptions to the domains and requirements of a collection of PDs by means of a Coloured Petri Net (CPN) model. We consider CPN as an appropriate formalism for this purpose because of it natural ability to model concurrency, resource sharing, conflicts and locality - all of which are properties that are found in the physical environment. Several important transformation activities in the approach we describe are automated. These transformations are specified as VDM++ operations on VDM++ objects representing CPN models and PDs.

Throughout the paper, we will refer to a simple example considering the development of a controller for a sluice gate. We use this example, because it is well known in the Problem Frames community and because it is sufficiently simple and still contains examples of typical concerns such as concurrency in the environment, required vs. assumed behavior etc. The purpose of the sluice gate is summarized by the following description:

> *"A small sluice, with a rising and falling gate is used in a simple irrigation system. A computer system is needed to raise and lower the gate in response to the commands of an operator. The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it's fully open , at the bottom it's fully shut. The connection to the computer consists of four pulse lines for motor control, two statuslines for the gate sensors, and a status line for each class of operator command."* [2]

For the purpose of the explanations in this paper, we have extended the description with a requirement that the sluice gate must be open for at least 1 hour in total between sunrise and sunset.

Fig. 1 shows three PDs (PD_1, PD_2, and PD_3). The annotation $\ll PD \gg$ specifies that the problems are expressed using the PD syntax. The overall problem is derived from the problem description above and is expressed in PD_1. The problem is to develop a machine (*Sluice Gate Controller*) that interacts with two *domains* in the environment (*Gate* & *Motor* and *Sluice Operator*). The machine is connected to these two domains through interfaces of shared phenomena (a and c). These connections specify where direct interaction is possible. The text in the right gives some detailed information about the interfaces. The two-letter abbreviations denote the controlling domain of a set of phenomena. E.g. the interface a is formed by two shared phenomena controlled by the *Sluice Gate Controller* (*Direction* and *Control*) and one shared phenomenon controlled by the *Gate* & *Motor* domain (*Sensors*). Each shared phenomenon is controlled by exactly one domain. If the phenomenon is a state, the controlling domain is the only domain able to cause changes to the state value. If the phenomenon is an event, the controlling domain is the only domain able to cause an occurrence of the event. It should be noted that phenomena can be abstractions of complex values - as an example, the *Sensors* phenomenon consists of two boolean values: one for the state of the top sensor and one for the bottom sensor.

The dashed ellipse is the requirement (i.e. to control the gate according to the requirements). The requirement is connected to the two domains,indicating

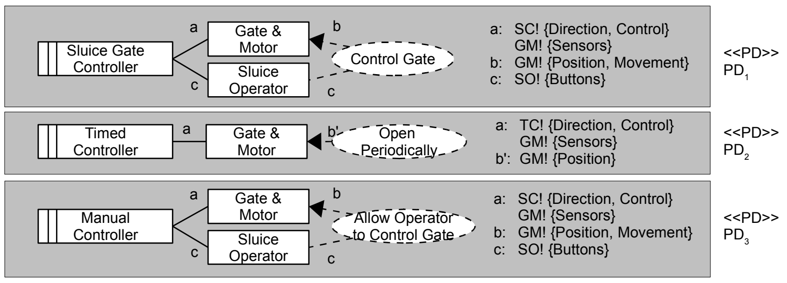

Fig. 1.

that the requirement is expressed in terms of the vocabulary of shared phenomenom provided by these domains. One connector has an arrow head - this means that the requirement *constrains* the phenomena in *b*. The other connector is a simple line - this means that the requirement only *references* the phenomena in *c*. The reason for this distinction is that the is should not be possible to make requirements about the behavior of the *Sluice Operator* domain, since this domain represents a person acting spontaneously and out of free will. The *Gate & Motor* domain, on the other hand is a *given* domain with some level of predictability - i.e. we know something about the casual relations between the phenomena in the interface of the domain. Therefor, it makes sense to express requirements about the desired behavior of this domain when it is interacting with the potential solution to the problem - the *Sluice Gate Controller*.

While PD_1 is the overall problem we want to analyze and solve, PD_2 and PD_3 are subproblems. These subproblems have been identified by *problem decomposition* [2] of the overall problem. By decomposing the problem into subproblems, various aspects of the problem are identified. The result is a collection of PDs. The decomposition is parallel rather than hierarchical meaning that the subproblems can be seen as different views (or projections) of the problem described in the first PD. The decomposition is a manual task which is based on knowledge about the problem domains. PD_2 documents a subproblem in which the sluice gate controller is required to periodically open the sluice gate. The controller domain is represented by a subdomain (*Timed Controller*), which is interacting only with the motor and the sensors - i.e. the behavior of the sluice operator is not relevant for the expression of this subproblem and has been left out. The requirement is expressed in terms of a subset of the *b* set of shared phenomena (b'). This is so, because the requirement in this case only concerns the physical and observable state of the gate. Actually, the requirement also relates to the real time, but this is considered a private phenomenon to the requirement domain. PD_3 documents another subproblem: the problem of developing a controller that allows the sluice operator to manually raise and lower the gate. This subproblem also has a controller and a requirement and is related to both of the domains found in PD_2. For this very simple example, the decomposition has now helped us realize that the problem of developing the sluice gate controller can be seen as (at least) two subproblems. Each subproblem has a solution and to develop the final controller machine these two solutions need to be recomposed. We do not consider this task here.

2 Tool Support for CPN Descriptions of PD Domains

The purpose of the tool support being developed is to make it possible and practically feasible to use CPN as a formal modeling language for describing behavior of the domains found in PDs.

Fig. 2 shows a possible workflow for the work on analyzing the Sluice Gate Controller problem. The ellipses are different representations of PDs while the boxes are transformation activities that will be described in the following subsections. The starting point for each PD is a matching VDM++ object. Such an object is an instance of the *ProblemDiagram* class found in Fig. 3. It simply captures the structure of a PD and refines the information slightly by specifying the types of shared phenomena.

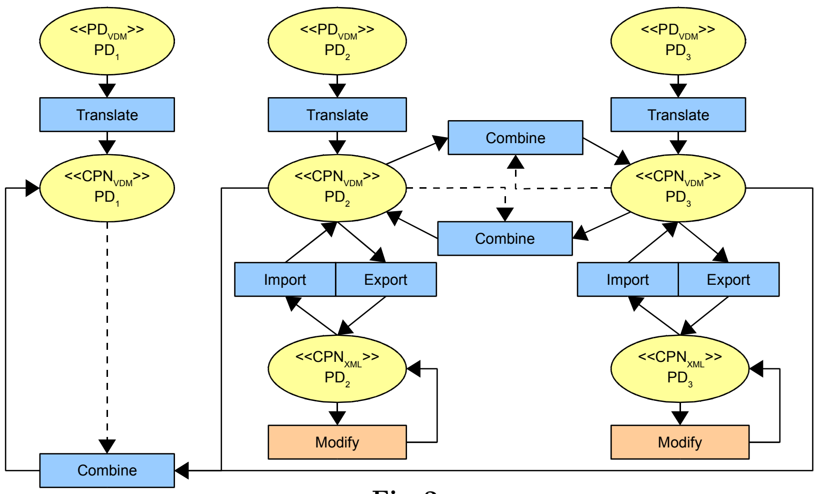

Fig. 2.

2.1 Translating

The purpose of the translation activity is to produce a modular CPN model (composed by a collection of inter-linked CPN modules) based on the structure of a PD. In the workflow in Fig. 2, the three PDs are tranlated from their VDM++ representations into respective VDM++ representations of CPN structure (denoted $\ll CPN_{VDM} \gg$). The transformation is handled by the *translate* operation shown in Fig. 4. Its input is a *ProblemDiagram* object and the output is a *CPNH* object (a *CPNH* object encapsulates, among other things, a collection of *CPN* objects and information about channels connecting these).

```
class ProblemDiagram
  types
    public Name = [seq1 of char];
    public Title = Name;
    public Domain = Name;
    public Domains = set of Domain;
    public Machine = Name;
    public Requirement = Name;
    public Controlled = Domain;
    public Observed = set of Domain;
    public Category = Name;
    public SharedPhenomenon ::
      name : Name
      type : <State> | <Event>
      category : Category
      controlled : Controlled
      observed : Observed
      conref :
      [<Constrained> | <Referenced>];

  instance variables
    public title : Title := nil;
    public machine : Machine := nil;
    public domains : Domains := {};
    public requirement : Requirement := nil;
    public sharedphenomena : SharedPhenomena := {};
  (...)
end ProblemDiagram
```

Fig. 3.

```
public translate : ProblemDiagram ==> CPNH
translate(pd) == (
  dcl cpnh : CPNH := new CPNH();
  cpnh.S :=
    mkDomainModules({pd})
    union {mkRequirementModule(pd)};
  cpnh.S :=
    cpnh.S union
    {mkLinkPage({pd}, cpnh, {|->}, {|->})};
  cpnh.FS :=
    mkFusionSets(
      dunion {cpnm.CPN.P | cpnm in set cpnh.S});
  return cpnh;
);
```

Fig. 4.

The syntax of the CPN modeling language has been almost completely specified in VDM++ based on the definitions of [3]. The specifications in VDM++ are relatively complex, since they include not only the rules for the structure of CPN models but also a representation of the type system used in the inscription language - CPN ML, a flavor of Standard ML. This representation in VDM++ is necessary in order to be able to handle the generation (and import/export) of colour sets used to annotate places in CPN models. A colour set specifies the type of tokens that a given place can hold. In the same manner, the much simpler syntax of PDs has been specified loosely based on the definitions found in [4].

As an example, we will look at parts of the CPN model being created when the PD_3 from Fig. 3 is translated. Refering to the translate algorithm of Fig. 4, the translation is started by the instantiation of an empty $CPNH$ object. Next, two types of modules (or pages) are added to the model: one domain description module for each domain of the PD (including the machine domain) and a single requirement module. We call these the *domain modules*, the *machine module* and the *requirement module* respectively. As mentioned earlier, a CPN module is a non-hierarchical net structure and the CPN model contains several of these. In our case, the modules are linked together through *fusion places* as we shall see later. This technique allows for a set of places in different CPN modules to be tied together in a *fusion set*. All places within the same fusion set will share the same marking (i.e. the same collection of tokens wrt. number of tokens, their data values etc.). This makes it possible for the modules to communicate or interact through fusion places. In the work presented here, the fusion places are used for representing the interfaces of shared phenomena through which the environment domains and the machine interact.

We will have a closer encounter with the domain modules in Subsection 2.3, where the manual modification of domain descriptions is discusssed. An empty domain module contains one place per shared phenomenon in all its interfaces with other modules. Initially, these places are connected to transitions that cause phenomenon modifications freely. This structure represents free and spontaneous behavior - i.e. the total lack of knowledge about the behavioral aspects of the domain. During the manual modification of the domain description, the behavior will gradually be refined (limited).

The requirement module is used for the specification of requirements. We will not discuss the contents of this module in this paper, but simply state that this is where the details of the requirements in a PD are specified. We are currently working on an automated technique for validating the behavior exhibited by the domain descriptions when attached to the solution description. The validation is performed with respect to real-time requirements expressed in a formalism similar to that of UML 2.0 Sequence Diagrams.

The next step of the translation is the generation of a single special CPN module: the *link* module. This module will not be manually edited. The purpose of the module is to provide the interface of shared phenomena between interacting domains while at the same time recording timed traces of shared phenomena activity. The latter is made possible by a proxy-based approach, in which all phenomenon activity caused by a controlling domain is detected and recorded in a synchronous manner by the link module before being asynchronously forwarded to any possible observing phenomena. This dual-type coupling makes it possible to both enforce recording of all phenomena activity and preserve a proper level of independency between domain descriptions.

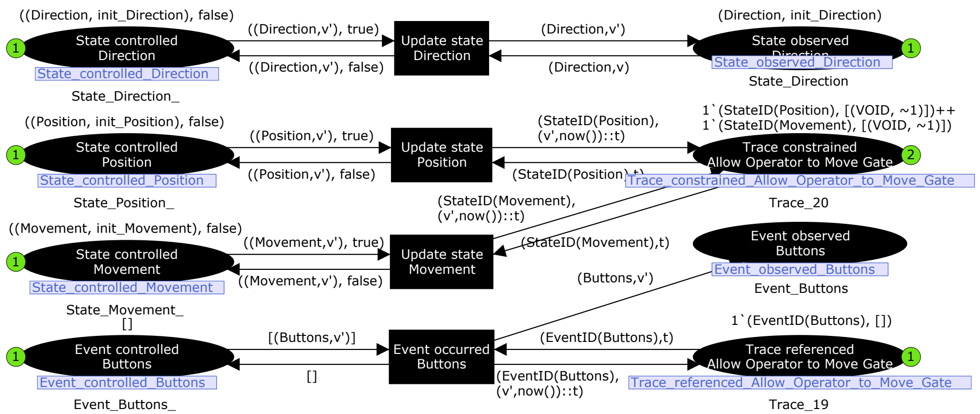
((Direction, init_Direction), false)
State controlled Direction
State_controlled_Direction
State_Direction_
((Direction,v'), true)
((Direction,v'), false)
Update state Direction
(Direction,v')
(Direction,v)
(Direction, init_Direction)
State observed Direction
State_observed_Direction
State_Direction
((Position, init_Position), false)
State controlled Position
State_controlled_Position
State_Position_
((Position,v'), true)
((Position,v'), false)
Update state Position
(StateID(Position), (v',now())::t)
(StateID(Position),t)
1`(StateID(Position), [(VOID, ~1)])++ 1`(StateID(Movement), [(VOID, ~1)])
Trace constrained Allow Operator to Move Gate
Trace_constrained_Allow_Operator_to_Move_Gate
Trace_20
(StateID(Movement), (v',now())::t)
(StateID(Movement),t)
((Movement, init_Movement), false)
State controlled Movement
State_controlled_Movement
State_Movement_
((Movement,v'), true)
((Movement,v'), false)
Update state Movement
Event observed Buttons
Event_observed_Buttons
Event_Buttons
(Buttons,v')
[]
Event controlled Buttons
Event_controlled_Buttons
Event_Buttons_
[(Buttons,v')]
[]
Event occurred Buttons
(EventID(Buttons),t)
(EventID(Buttons), (v',now())::t)
1`(EventID(Buttons), [])
Trace referenced Allow Operator to Move Gate
Trace_referenced_Allow_Operator_to_Move_Gate
Trace_19
1
2

Fig. 5. Excerpt of the automatically generated link module for PD_3

Part of the link module for PD_3 is shown in Fig. 5. We use solid black to emphasize automatically generated elements (i.e. all the elements in this case). The module is graphically structured in 3 columns. The first column contains a set of places that are connected through fusion sets with matching places in domain modules or the machine module. Each of the places carry inscriptions on the top (the initial marking) and in the bottom (the colour set definition). The initial marking specifies the initial collection of tokens held in the place. All these tokens are of a specific type as defined by the colour set definition. Looking closely at the inscriptions, it is seen that each phenomenon of the PD is repsresented by a colour set (e.g. *State_Direction_*). These colour sets are declared automatically as part of the translation. The structure of the colour sets are different depending on whether the colour set is used for a state or an event phenomenon. For states, a 2-tuple containing a 2-tuple and a boolean value is used. For the *Direction* phenomenon, an example of a value of this colour set is found in the initial marking of the topmost place: $((Direction, init\_Direction), false)$. The inner 2-tuple is composed by a constant identifying the phenomenon in question matching the name in the PD (*Direction*) and the current value of the phenomenon initialized by a constant (*init_Direction*). The boolean in the outer 2-tuple is initialized to $false$. Whenever the phenomenon value is changed by the controlling domain, this value will be set to true by the changing transition. This is a way to signal to the monitoring transition of the link module that the state was changed - and at the same time serves to prevent the phenomenon from being changed again before the current change has been recorded to the timed trace and reported to any observing domains. Both the recording and reporting is performed by a single transition per phenomenon - in this case, the *Update state Direction* transition. The *Direction* phenomenon is observed by the *Gate* & *Motor* domain through a fusion set. A place in this fusion set is therefor found in the third column of the link module. The monitoring transition updates the value of the single token found in this place with the new value of the phenomenon. The *Direction* phenomenon is not constrained by the requirement of PD_3, so its changes are not recorded to a trace. The *Position* phenomenon is, on the other hand, and its monitoring transition is connected to a trace place (the second from the top in the third column). This place holds a token per phenomenon referenced or constrained by the requirement - in this case two phenomena (*Position* and *Movement*). Each token is a 2-tuple consisting of a phenomenon identifier and a list. The elements of the list are 2-tuples in which the first field is a phenomenon value and the second field is a timestamp. The timestamp identifies the model time at which the phenomenon value became valid - i.e. the time at which a state phenomenon changed its value or the time at which an event occurred. Whenever the phenomenon is modified by the controlling domain, a time-stamped element is added to the trace list. This trace is used for validation against the requirement expressed in the requirement module. The monitoring of events is synchronized between the controlling domains and the link module in a different manner: a list token is used. If an event has occurred, the list will contain an element in the form of a 2-tuple in which the first field identifies the event an the second type is a value for the event. As long as the list is not empty, the controlling domain will be prevented from generating more events before the occurrence of the current event has been monitored and recorded and/or reported by the link module. This guarantees that all events are observed and their is preserved in the traces.

The last step of the translation algorithm ties together the places representing shared phenomena with those of the link module in fusion sets - i.e. all shared phenomena are represented by two fusion sets. The first fusion set contains two places: one in the controlling domain and one in the link module (e.g. *State Controlled Direction*). The second fusion set contains one place in the link page and one per observing domain (e.g. *State Observed Direction*). Apart from providing the recording and reporting of phenomena activity, the link module and the way fusion sets are used ensure that only one domain controls any given phenomenon and that only observing domains have access to the phenomenon values.

2.2 Importing and Exporting

The translation from PDs results in a VDM++ object representation of a CPN model. The workflow in Fig. 2 shows that it is possible to export such a representation to obtain a XML file for the CPN model. It is also possible to import such an XML file to obtain the corresponding VDM++ object.

Exporting to the XML file format makes it possible to use CPN Tools [5] for editing the CPN models in order to fill out behavioral descriptions. The export feature has been specified in a VDM++ class able to directly write an XML file from a *CPNH* object. Importing from the XML file format makes it possible to handle manually modified CPN models within the VDM++ framework. The import feature has been implemented as a Python script that parses the XML file and generates a VDM++ data file containing a single *CPNH* object.

2.3 Modifying

Fig. 6 shows the behavioral description of the *Gate* & *Motor* domain module. The solid black places were autogenerated by the translation algorithm and the white transitions and places, all arcs and their inscriptions have been added manually to specify knowledge about the assumed behavior of the physical gate, the motor and the sensors. In this way, a causal relation ship between these physical entities is explicitly expressed in the model. Briefly explained, the gate is modeled using an internal state phenomenon representing a discretization of the gate position as a value between 1 and 10. This place is modified by two transitions representing the movement of the gate in discrete time-consuming steps (it takes 15 time units to move one step up and 10 to move one step down). The domain controls three phenomena: *Sensors* is observed by the sluice gate controller while *Position* and *Movement* are constrained by the requirement. The *Update Position* and *Update Sensors* transitions monitor changes to the internal representation and updates the controlled phenomena accordingly. Wrt. the *Position* phenomenon, it is worth noticing, that the values of this phenomenon are more coarse (*Top*, *Bottom*, and *Between*) than those of the internal representation of the position (1...10). The domain module also specifies how the motor is controlled by the machine domain through the *Control* and *Direction* phenomena.

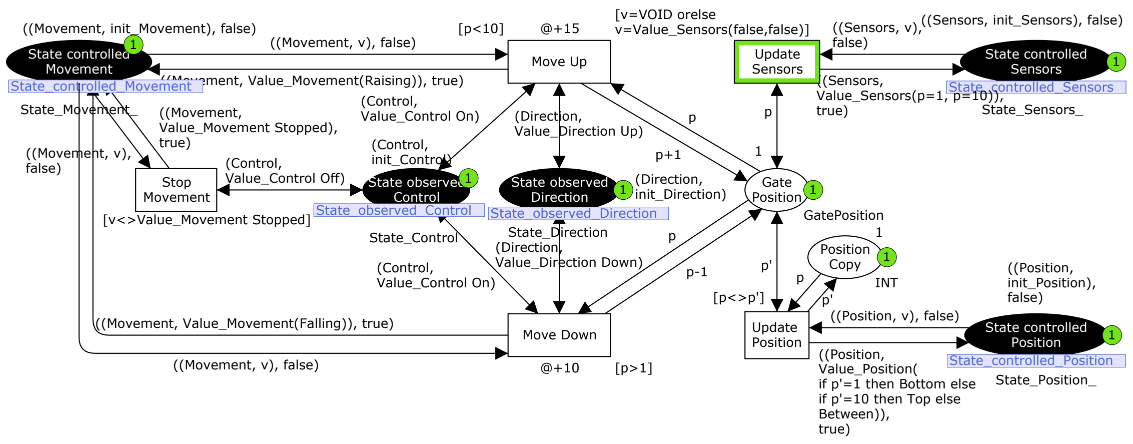
((Movement, init_Movement), false)
State controlled Movement
1
State_controlled_Movement
((Movement, v), false)
((Movement, Value_Movement(Raising)), true)
[p<10]
@+15
Move Up
[v=VOID orelse v=Value_Sensors(false,false)]
Update Sensors
((Sensors, v), false)
((Sensors, init_Sensors), false)
State controlled Sensors
State_controlled_Sensors
((Sensors, Value_Sensors(p=1, p=10)), true)
State_Sensors_
State_Movement_
((Movement, Value_Movement Stopped), true)
((Movement, v), false)
Stop Movement
[v<>Value_Movement Stopped]
(Control, Value_Control On)
(Control, init_Control)
(Control, Value_Control Off)
State observed Control
State_observed_Control
State_Control
(Direction, Value_Direction Up)
State observed Direction
State_observed_Direction
(Direction, init_Direction)
State_Direction
(Direction, Value_Direction Down)
p
p+1
1
Gate Position
GatePosition
p
p-1
p'
Position Copy
1
INT
p
p'
[p<>p']
Update Position
((Position, v), false)
((Position, init_Position), false)
State controlled Position
State_controlled_Position
State_Position_
((Position, Value_Position(if p'=1 then Bottom else if p'=10 then Top else Between)), true)
(Control, Value_Control On)
((Movement, Value_Movement(Falling)), true)
Move Down
((Movement, v), false)
@+10
[p>1]

Fig. 6. The manually modified domain module for the *Gate & Motor* domain

2.4 Combining

In many situations, as illustrated in the workflow of Fig. 2, it is necessary to *combine* descriptions found in CPN models. As an example, the *Gate & Motor* domain discussed in the previous sections is found in all three PDs, but it should only be necessary to specify its assumed behavior once - i.e. in one CPN model. For this reason, we have defined an algorithm for structural combination (merging) of CPN models.

```
public combine : ProblemDiagram * (set of (ProblemDiagram * CPNH)) ==> CPNH
combine(pd, ds) == ( (...) )
pre (forall mk_(pd', cpnh) in set ds &
  structureValid(pd', cpnh) and
    forall d in set pd'.domains union {pd'.machine} &
      d in set pd.domains union {pd.machine} => interfaceSubset(pd', pd, d)) and
        (let empty_cpnh = translate({pd}) in
          forall d in set pd.domains union {pd.machine} &
            let empty_cpnm in set empty_cpnh.S be st empty_cpnm.Name = d in
              exists cpnh in set {cpnh | mk_(-, cpnh) in set ds} &
                exists cpnm in set cpnh.S & cpnm.Name = d =>
                  not exists cpnm' in set cpnh.S &
                    cpnm <> cpnm' and cpnm.Name = cpnm'.Name and
                    not cpnm.CPN.structEqual(empty_cpnm.CPN) and
                    not cpnm.CPN.structEqual(cpnm'.CPN));
```

Fig. 7.

The outline of this algorithm is shown in Fig. 7 in the form of its signature and preconditions. The algorithm will produce a *CPNH* object based on the structure of a given PD and the manually modified descriptions found in a set of 2-tuples. Each 2-tuple consists of a PD and a *CPNH* object. The PD defines the structure from which the *CPNH* object in the tuple was translated. The preconditions ensure that at most one refined description of any domain that exists in the output structure is found in the input descriptions - i.e. to avoid conflicting descriptions. In Fig. 2, a dashed line is used to point out the PD defining the structure of the result of the combination while solid lines go from the input CPN models and to the resulting output CPN model. The output CPN model contains all domain descriptions found in the input CPN models - provided that no domain has conflicting descriptions. One example of the application of the combine algoritm is seen in the lower left corner of Fig. 2, where a CPN model for PD_1 is generated as a combination of the descriptions of domains already given during any possible manual modification of the CPN models for PD_2 and PD_3. Two other uses of the combine algorithm are seen between PD_2 and PD_3. In this case, if a domain has been described in one CPN model, the description module can be adopted into the other CPN model meaning that the two models can be refined in a parallel manner.

The precondition specifies two additional requirements to the input: (1) the $structureValid$ operation is applied to require that all $CPNH$ objects in the set of input tuples do still match the structure defined by the respective PDs from which they were translated and (2) the $interfaceSubset$ operation is applied to require that for a given domain, any existing instances of the same domain in the input PDs should not have access to any other shared phenomena than those accessible to the domain in the resulting PD. The second requirement means that the combine algorithm can only extend (and not limit) the interface of domains.

3 Conclusions

In this paper, we have described the initial results from work on tool-support for automated handling of CPN models used for describing behavior of domains in PDs. Our experiences so far tell us that - if assisted by the automated translate and combine algorithms - the use of CPN models is indeed a feasible and practical approach to the problem. The main area for future work is the work on automating the process of checking traces of phenomena activity generated by the execution of CPN models against real-time requirements. We are currently working on tool-support for this. The checking can be performed either offline or online. In the offline approach, a trace is generated and then checked. In the online approach, the trace is checked while it is being generated - i.e. possibly during the interactive execution of the CPN model.

References

1. M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.
2. M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems.* Addison-Wesley, 2001.
3. K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.
4. J. G. Hall, L. Rapanotti, and M. Jackson. Problem Frame Semantics for Software Development. *Software and System Modeling*, 4(2):189–198, 2005.
5. CPN Tools. http://wiki.daimi.au.dk/cpntools/cpntools.wiki.