Expressing Environment Assumptions and Real-time Requirements for a Distributed Embedded System with Shared Variables

Simon Tjell and João M. Fernandes

Abstract In a distributed embedded system, it is often necessary to share variables among its computing nodes to allow the distribution of control algorithms. It is therefore necessary to include a component in each node that provides the service of variable sharing. For that type of component, this paper discusses how to create a Colored Petri Nets (CPN) model that formally expresses the following elements in a clearly separated structure: (1) assumptions about the behavior of the environment of the component, (2) real-time requirements for the component, and (3) a possible solution in terms of an algorithm for the component. The CPN model can be used to validate the environment assumptions and the requirements. The validation is performed by execution of the model during which traces of events and states are automatically generated and evaluated against the requirements.

1 Introduction

In this paper, we describe an approach to requirements engineering using Colored Petri Nets that has been devised during an industrial case study concerning an automated hospital bed. A control system allows the bed to be adjusted into different positions by moving the sections on which the mattress rests. The control system depends on transparent sharing of variables among a group of embedded nodes. For this purpose, a communication component has been developed and, in this paper, we focus on the documentation and validation of the requirements for this component.

Fig. 1 shows a simplified overview of the control system. The system consists of a collection of autonomous embedded nodes, connected by a communication bus. Each node is physically connected to a collection of actuators ($A_{1...3}$) and sensors

Simon Tjell
Department of Computer Science, University of Aarhus, Denmark

João M. Fernandes
Departamento de Informática, Universidade do Minho, Portugal

Please use the following format when citing this chapter:

Tjell, S. and Fernandes, J.M., 2008, in IFIP International Federation for Information Processing, Volume 271; *Distributed Embedded Systems: Design, Middleware and Resources*; Bernd Kleinjohann, Lisa Kleinjohann, Wayne Wolf; (Boston: Springer), pp. 79–88.

($S_{1...2}$) that are controlled and monitored by local applications ($App_{1...5}$). The control system is distributed, because an application running in one node is able to monitor sensors and control actuators connected to any node with the limitation that each actuator is controlled by exactly one application. Remote monitoring and control is made possible by a collection of shared variables. A variable value can be used for (1) the current reading of a sensor (e.g. an angle or a push button), (2) the current output for an actuator (e.g. the displacement of a linear actuator), or (3) a link between two applications. Two kinds of variable instances exist: originals and copies. A variable original exists in the node where new values to the variable is written by a locally running application. One example could be that App_2 in $Node_1$ periodically reads a current from the S_1 sensor and writes the measurement to the Var_2 variable original. In $Node_2$, the application App_3 relies on the readings of S_1 for its task of controlling the A_2 actuator. For this reason, $Node_2$ houses a variable copy instance of Var_2 that is kept updated to have the same value as the variable original and thus providing the input from S_1 to App_3.

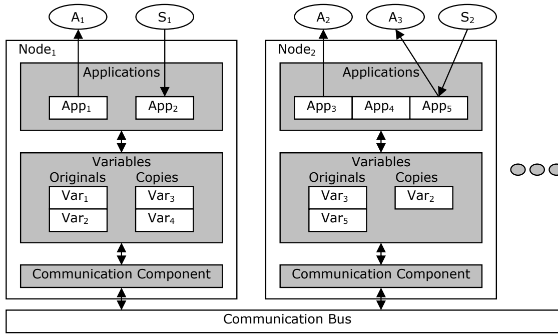

Fig. 1 Informal system overview

The task of keeping variable copies updated with the values of their matching variable originals is handled by the communication component that exists in each node. Messages about variable values are exchanged among the communication component instances through the communication bus. The work presented in this paper focuses on the requirements for the communication component. The approach we present allows a collection of requirements to be represented formally in combination with formal representation of the behavioral assumptions about the environment of the system. We will use the following requirement (Req. 1) as an example: *For any variable copy v_C related to a variable original v_O, the following must be satisfied if a warm-up period of 10000 ms. since system start has passed: if the value of v_O has been stable (i.e. the same) for a coherent period of least 50 ms., v_C must have the same value. The maximum delay allowed before the value of v_C reflects that of v_O is 500 ms.*

2 The CPN Model

This section presents an approach to developing a CPN model that contains a formal representation of (1) the assumed behavior of the environment of the communication component, (2) the real-time requirements, and (3) a possible solution satisfying the requirements (i.e. an initial design for the communication component). The approach extends previous work [3, 10, 11] by the introduction of explicitly expressed real-time requirements and validation of these.

CPN is a formal modeling language with a graphical representation. It is a high-level extension of standard Petri Nets to which it adds: complex data types for token values (allowing distinction of tokens and modeling of data manipulation), a programming language for guard expression and data manipulation, hierarchical constructs, and real-time modeling through timestamps on tokens. The CPN language is supported by CPN Tools [1], which provides a graphical modeling environment where models are developed and experimented with through simulation and state space analysis. The main concepts of the modeling language will be introduced in the description of the CPN model in this section. Further details are found in [7].

The CPN model presented here is hierarchical and has the structure of a tree in which the nodes are modules and the edges are connections between modules. The root node is called the top module and is shown in Fig. 2. The module contains four *substitution transitions* (double-edged rectangles) connected by arcs to four places (ellipses) through which interaction and observation is possible. Both the arcs and the places carry inscriptions. A substitution transition represents an instance of a module which may be found in multiple instances throughout the model. A module is itself a CPN structure that may contain further levels of substitution transitions which allows the model to be structured using many hierarchical levels. In the top module, the substitution transitions represent domains. The places represent collections of shared phenomena about which the domains communicate. A shared phenomenon is either a state or an event, which is controlled by exactly one domain but may be observed by multiple domains. A domain controls a phenomenon if it causes it to happen in the case of an event or cause it to change in the case of a state.

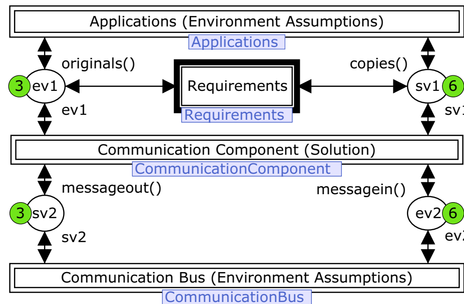

Fig. 2 The top module

The shared phenomena places carry two kinds of inscriptions: color set definitions (`ev1`, `sv1`, `ev2`, and `ev2`) and initial markings (`originals()`, `copies()`, etc.). The color set definition of a place specifies the data type of the tokens that are allowed to exist in the place. The names of the color sets match those defined in the reference model introduced in [4]: the color sets `ev1` and `ev2` contain visible phenomena controlled by the environment (variable originals and incoming messages from the bus respectively) while the color sets `sv1` and `sv2` contain those controlled by the communication component (variable copies and outgoing messages from the nodes respectively). In addition to the visible/shared phenomena, all domains contain collections of hidden and locally controlled phenomena. The initial marking definition of a place tells which tokens will exist in a place in its initial state - for state phenomena, this initialization value will be the initial value of the state and for event phenomena it will typically be a token value representing the information that no events of a given type have been generated before the initial state of the model. The actual contents of these phenomena places will be introduced in Section 2.1.

The top module captures a description of how we have chosen to structure our assumptions about the environment in which our problem and its possible solution is found. The top level module is structured in the same way as a Jackson Problem Diagram [6] and can be described in the following way: *the problem is to develop a communication component that, given the environment assumptions about the behavior of the applications and the communication bus, performs the task of keeping the variable copies updated to match the changing values of the variable originals in a way that satisfies the requirements.*

The top module contains a representation of the requirements connected to two shared phenomena places (`ev1` and `sv1`). This connection implies that the requirements be expressed in terms of the phenomena found in `ev1` and `sv1` - the variable originals and copies. Implicitly, this tells us that requirements cannot be expressed in terms of messages being exchanged through the communication bus since the substitution transition has no connection to these phenomena places. The top module represents the structure of the domains seen from one node and the modeling language allows us to use this structure to represent the behavior of a parameterized number of concurrently operating nodes.

2.1 Modeling Shared Phenomena and Environment Assumptions

The shared phenomena allow interaction among domains. The trivial approach to modeling the shared phenomena would be to define data types for events and states of different kinds and then represent each instance of a state or an event as an individual token in a shared phenomena place. In fact, we have done so in previous works [3, 10]. In the work presented here, we express requirements over timed traces of phenomena observations (changes to states or occurrences of events) and we therefore need a slightly richer representation of shared phenomena in the places, namely one that preserves the history of phenomena observations rather than just

Listing 1 Definition of the `Trace` color set

```
1  colset NodeID = int with 1..NumberOfNodes;
2  colset State =
3   union VariableOriginalValue:VariableValue+VariableCopyValue:VariableCopyValue;
4  colset Event =
5   union MessageInValue:VariableValue + MessageOutValue:VariableValue;
6  colset PhenomenonID = union
7   VariableOriginalID:VariableID + VariableCopyID:VariableID +
8   MessageOutID:VariableID + MessageInID:VariableID;
9  colset Phenomenon = union State:State + NewEvent:Event + OldEvent:Event;
10 colset TimedPhenomenon = product Phenomenon *  Timestamp;
11 colset Phenomena = list TimedPhenomenon;
12 colset Trace = product NodeID * PhenomenonID * Phenomena timed;
```

snapshots. For this purpose, the `Trace` color set has been defined (Listing 1). This listing is an example of declarations in the CPN model expressed in terms of the built-in functional language CPN ML - a variant of Standard ML. The `Trace` color set is a superset of all the color sets found in Fig. 2 (`ev1`, `sv1`, etc.).

The listing contains the definitions of the two kinds of phenomena (states and events), introduces timestamped phenomena, and defines a trace to be a triple containing (1) the identity of a node, (2) the identity of a phenomenon, and (3) a list of timestamped phenomena occurrences/changes. The `PhenomenonID` color set is a union type of a collection of different identifiers. This is practical since some phenomena may need to be identified using an index and a name, while others may be more appropriately identified using an enumerated value or a string. The union type approach is also used for the phenomenon values: this is again practical because it makes it possible to use different data types of varying complexity to represent different phenomena. An event is either old or new: a `NewEvent` element is used to represent an event that has occurred but has not yet been observed while an `OldEvent` represents an event that has been observed. This makes it possible to ensure that the occurrence of an event is only detected once in each observing node. When an observable phenomenon occurs in a domain, this is recorded by adding an element to one or more trace tokens (one token exists per observing node).

The addition of elements to traces is performed using two functions `state` and `event`. In the case of the occurrence of an event, a `NewEvent` element is added. In the case of a state change, a new element is only added if the state indeed changed to a new value. In both cases, the new element is associated with a timestamp indicating the model time at which the phenomenon occurred.

Following the structure found in Fig. 2, the environment of the communication component consists of the applications and the communication bus. The communication component interfaces with these two domains through phenomena related to the variables (originals and copies) and messages (outgoing and incoming) respectively. A description of the structure of the environment is captured in the top module (Fig. 2), while description of the assumed behavior of the domains environment is found in the `Applications` and `CommunicationBus` modules.

The `Applications` module (not shown) describes assumptions about how the applications in the nodes write new values to local variable originals and about the timing of these write operations. The `CommunicationBus` module (Fig. 3) describes

assumptions about how messages are exchanged between nodes with potential loss of messages in the case of physical connection problems. The module is connected to the top module through the places marked by I/O labels. These places are connected to the places with matching names in the top module. In this way, it is possible to describe the details of the communication bus inside the module while avoiding too many possibly confusing details in the top module. Whenever a message is sent by a node (modeled in the `Communication Component` module), a `NewEvent` element is added to a trace token found in the `sv2` place. The `NodeID` in this trace token matches that of the sending node. When the element is added to the trace token, it becomes visible to the `CommunicationBus` module through the `sv2` place.

The semantics of CPN is based on the notions of *enabling* and *occurrence*. For example, the `Detect Message` transition (Fig. 3) is said to be enabled whenever its input place (`sv2`) contains a token that satisfies the constraints of the transition: (1) the pattern expression in the input arc, and (2) the guard expression. The guard expression is seen in brackets in the left-hand side of the transition and it specifies that the transition can only become enabled if the trace contains a new (not yet observed) event. When the transition is enabled it may occur (or *fire*) causing the consumption of one token from the input place (`sv2`) and the production of a collection of tokens in the output places (`sv2` and `Outgoing`). The trace token is updated and placed back in the `sv2` place. The update consists in using the `oldevent` function to change the type of the observed message event from a `NewEvent` to an `OldEvent` element (preserving the parameter values). The consumption and production of the token to the `sv2` place can be seen as a data manipulation operation. In the `Outgoing` place, a collection of tokens is produced by the `broadcast` function: one token representing a message for each receiving node - all with the same variable value information. From the `Outgoing` place, tokens can be consumed individually by either the `Loose Message` transition (modeling a physical connection being unavailable) or the `Transmit Message` transition (modeling a message being successfully delivered to a receiving node). In the later case, the `event` function is used to add a `NewEvent` element containing the variable information to the trace related to the node in which the message is received.

The `CommunicationBus` module also shows an example of how assumptions about the timing properties of the environment are modeled: the `Transmit Message` transition carries a delay inscription (`@+MessageDelay`) indicating that each successful transmission of a message takes `MessageDelay` time units. In this case, the

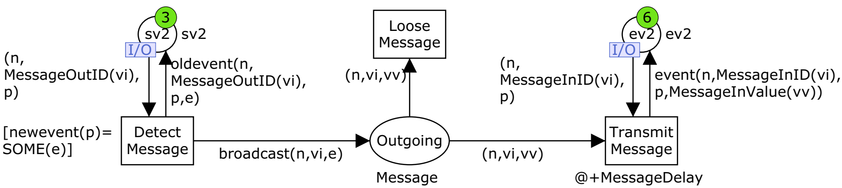

Fig. 3 Environment assumptions expressed in the `CommunicationBus` module

delay is a constant (the average assumed delay), but for a more detailed modeling of this assumption, a stochastic distribution function could have been applied.

The modules described here explicitly represent our assumptions about the behavior and structure of the environment in which the communication component must provide a solution to the problem of maintaining the consistency between the variables.

2.2 Expressing Requirements

In the introduction, Req. 1 was informally presented. Here, we will give an example of how this requirements has been formally expressed in the CPN model using the concepts of Duration Calculus [12]. As seen in Fig. 2, the model contains a `Requirements` module (shown in Fig. 4). This module contains the expression of all requirements about the communication component. The requirement transitions are connected to the phenomena places using double-headed arcs. Informally, this means that the transitions are observing but not modifying the contents of the phenomena places - i.e. the requirements are expressed using transitions that monitor the traces of the shared phenomena (in this case the variables).

The `Requirements` module contains the `Requirements Satisfied` place that initially holds three tokens identifying three requirements (`Req1`, `Req2`, and `Req3`) of which the first was described informally in the introduction. The module also has three transitions with guards containing negations of the three requirements. If a requirement is *not* satisfied, the respective transition will become enabled. If a requirement transition occurs during the execution of the CPN model, the token identifying the requirement is removed from the `Requirements Satisfied` place. By monitoring the contents of this place after or during execution, we are able to detect situations in which the solution fails to satisfy the requirements. We briefly discuss how execution of the CPN model is used for experimenting with a possible solution in Section 2.3.

As described in Section 2.1, the phenomena are recorded in traces in which the elements carry timestamps that record when the phenomena happened - i.e. when a state changed or when an event occurred. The requirements are expressed about the

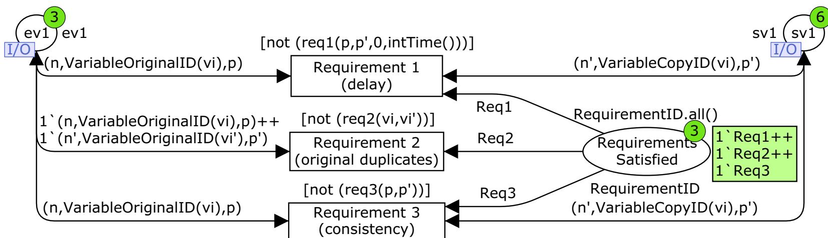

Fig. 4 The `Requirements` module

Listing 2 The `requirement1` function

```
1  fun req1(trace1,trace2,t1,t2) =
2  let val original_trace = intervals(t1,t2,trace1)
3  val copy_trace = intervals(t1,t2,trace2)
4  val max_delay = 500   val min_length = 50   val warmup_delay = 10000
5  in forall original_trace (fn(t1,t2,v) =>
6    implies(t2>warmup_delay andalso t2-t1>=min_length
7          andalso upper_bound(copy_trace)>t2+max_delay,
8      exists copy_trace (fn(t1',t2',v')=>
9       (t1'>t1 andalso v'=v andalso t1'-t1<max_delay))
10 )) end;
```

variable copies and originals. For this reason, the transitions in the `Requirements` module are connected to the phenomena places holding the traces about variable copies and originals through interface places.

We now focus on the definition of Req. 1 seen in Listing 2. The `req1` function is used as a guard expression for the matching requirement transition in Fig. 4. The function returns *true* if and only if Req. 1 is satisfied for a trace of a specific variable original (`trace1`) and a trace of a matching variable copy (`trace2`) within a timespan (`t1`...`t2`). Fig. 4 shows how pattern matching in the input arcs to the `Requirement 1` transition is applied to specify that the variable ID of the traces for the variable original and the variable copy should match (bound to the variable `vi`) while the node IDs (bound to `n` and `n'`) and the traces (bound to `p` and `p'`) may (and will) differ. The satisfaction of Req. 1 is evaluated by comparing the trace of a variable original to one trace representing a matching variable copy. For a given variable ID, there is only one trace for a variable original while multiple traces representing variable copies may exist (one per reading node). The evaluation is performed one copy trace at a time and each evaluation is performed using the `req1` function (Listing 2). Line 1 gives the signature of the function. In lines 2 and 3, the `intervals` function is used to generate lists of intervals based on the traces for easier traversal. Line 4 defines constants used in the representation of the requirement found in lines 5-10. The `forall` function is used for universal quantification over the `original_trace` (line 5). This value is a list of intervals defined by triples: start time of an interval (`t1`), end time of the interval (`t2`), and the state value within the interval (`v`). For all intervals (elements), the `implies` function is used to require that if the interval ends after a `warmup_delay` and is longer that `min_length` and the last interval of the copy trace ends after `t2+max_delay` (lines 6-7), then there should exist an interval in the copy trace (`(t1',t2',v')`) that starts after `t1` and has the value `v` and starts within `max_delay` time units after `t1` (lines 8-9).

Informally, `warmup_delay` is included to allow some update messages to be exchanged before the requirement to the maximum delay is required to be satisfied. When this point is reached, any change to a variable original should be reflected in all its variable copies within the period defined by `max_delay`. The requirement is softened slightly by specifying that the update is only required if the original value remains stable for a period of at least `min_length` time units. Hence, transient values of an original are not required to be reflected in all matching copies.

2.3 Experimenting with Possible Solutions

Until now, following the suggestions of [5], we have deliberately avoided discussing any concrete approaches to solving the problem of maintaining consistency throughout the variables. Instead, we have focused on the environment assumptions and the real-time requirements - i.e. constraints that apply to any possible solution we can think of. Now, we will briefly discuss how the fact that the CPN model is executable makes it possible to experiment with explicitly expressed solutions while monitoring the satisfaction of the requirements through the effects on (part of) the environment. A possible solution is expressed in the `CommunicationComponent` module of the CPN model (Fig. 2). The details of this module are not shown.

The overall purpose of the communication component is to handle the task of maintaining consistency between variable originals and their copies by exchanging messages through the communication bus. Basically, two alternative approaches are possible: event- and time-triggered communication. In an event-triggered approach, the communication component transmits a message containing a new value whenever the value of a variable original is changed by a local application. In a time-triggered approach, the communication component periodically transmit messages containing the current values of local variable originals. In both cases, the values of local variable copies are updated with the values found in incoming broadcast messages received by the communication components.

We have experimented with a solution based on the principles of *Soft State Signaling* [9] that combines a time-triggered messaging scheme with special validity tags on the variables copies. A variable copy is tagged *invalid* if the periodic update messages are not received for a predefined period of time.

In CPN Tools, the CPN model can be executed in an interactive manner - i.e. allowing the user to select transitions to occur and their parameters. The model can also be executed more freely in which case the tool will make free and fair choices of transitions to simulate different scenarios. In addition to this, the *monitor* mechanism can be applied to give an alert (stop simulation) if a global state of the model is reached where the `Requirements Satisfied` place (Fig. 4) does not contain all requirement ID tokens. This is useful, because such a situation would indicate that one or more of the requirement transitions have fired, meaning that a requirement was not found to be satisfied in a state reached during the simulation of the model. Whenever a requirement is found not to be satisfied during a simulation, the task is to investigate whether the cause is to be found in the environment assumptions (or their descriptions), too strict requirements, the proposed solution, or a combination.

3 Conclusions and Future Work

Several authors have already proposed the adoption of Petri Nets for modeling the structure and behavior of distributed embedded systems, and in particular for verifying real-time requirements. For example, [2] discusses a Petri Net-based approach

to verification of embedded systems and introduces a systematic procedure to translating the descriptions into linear hybrid automata in order to use existing model checking tools. A complete design flow based on high-level Petri Nets for real-time embedded systems for modeling both discrete and continuous parts of the systems by the event-based Petri net semantics is presented in [8]. The Petri Net model is used for formal verification and hardware/software partitioning purposes. This paper extends (and generalizes) the results presented in [11] and also suggests the use of a unique language based on Petri Nets for modeling the system and its environment, but the focus is on validating the real-time requirements expressed in terms of the required effects on the environment caused by the system.

Future work includes making the structuring approach introduced here more generally applicable by formally defining it as a structural subclass of CPNs with unmodified semantics. Such a definition could also serve as a foundation for the definition of refinement operations that could be used for refining requirements into specifications taking the environment assumptions into account. It would also be interesting, for example, to look at how environment entities could be modeled using CPN representations of differential equations based on the principles of [8].

References

1. CPN Tools. `http://daimi.au.dk/CPNTools`.
2. L. A. Cortés, P. Eles, and Z. Peng. Verification of Embedded Systems Using a Petri Net Based Representation. In *ISSS 2000*.
3. J. M. Fernandes, J. B. Jørgensen, and S. Tjell. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. In *APSEC 2007*.
4. C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3), 2000.
5. I. J. Hayes, M.A. Jackson, and C.B. Jones. Determining the Specification of a Control System from that of its Environment. In *FME 2003*.
6. M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.
7. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *STTT*, 9(3-4), 2007.
8. B. Kleinjohann, J. Tacken, and C. Tahedl. Towards a Complete Design Method for Embedded Systems Using Predicate/Transition-Nets. In *CHDL 1997*.
9. S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-Based Communication. In *SIGCOMM 1999*.
10. S. Tjell. Distinguishing Environment and System in Coloured Petri Net Models of Reactive Systems. In *SIES 2007*.
11. S. Tjell. Model-Based Analysis of a Windmill Communication System. In *DIPES 2006*.
12. C. Zhou, C. A. R. Hoare, and A. P. Ravn. A Calculus of Durations. *Inf. Process. Lett.*, 40(5), 1991.