

Automatic Code Generation from Coloured Petri Nets for an Access Control System

Kjeld H. Mortensen
University of Aarhus, Department of Computer Science,
Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark
k.h.mortensen@daimi.au.dk

Abstract

We describe in this paper a general method for automatic code generation from Coloured Petri Nets (CP-nets or CPN). The method is supported by the Design/CPN tool which has been extended during the past few years, such that it also can be used to generate code automatically from a CPN model. We do not describe the algorithms for code generation but rather the context such a tool is used in.

The rough outline of the method is as follows. One models the system of interest with CP-nets and Design/CPN. The modelled system behaviour is debugged and analysed, and when one has significant confidence in the model then the automatic code generation tool is applied, giving the final executable implementation as a result. Thus the behaviour of the model and executable are identical, and the traditional implementation phase has been eliminated.

In this paper we demonstrate that the method is usable in practice for an industrial example, namely an access control system developed by the Danish security company Dalcotech A/S.

This CPN model is a first version of the next generation of access control systems to be developed by Dalcotech. We describe the model and how they apply the automatic code generation method in order to obtain a system implementation quickly and safely. In this way Dalcotech now has the capability to reduce the time spent in the implementation phase dramatically. Another benefit is that they also dramatically reduce the amount of time spent on debugging the implementation.

1 Introduction

Coloured Petri Nets (CP-nets or CPN) [6, 9] is a language with an unambiguous formal definition and can therefore be subject to compilation and execution in a computing environment. The Design/CPN tool [30] has so far supported this in the form of a simulation engine, which is the traditional approach to execution of CPN models.

In an earlier project with Dalcotech [1] we made a proof-of-concept experiment with automatic code generation for a complex alarm system [20]. Standard ML code was extracted from a CPN model and burned into two PROMs that were mounted in a prototype of the final hardware. (See [15, 19] for more information on the functional computer language ML.) The experiment showed that automatic code generation from CPN models can be used in practice, but the experiment also showed a need to produce more efficient code and to develop additional tool support, e.g., for debugging, testing, and interfacing libraries.

However, automatic code generation was not the focus of that project. In fact the implementation of the system was done by hand in C++ based on the CPN model that was built. This was very time-consuming. Therefore it was decided to follow up on the challenges in a new project dedicated to automatic code generation. The project is called AC/DC (Automatic Code Generation from Design/CPN) [24] which is the main topic of this paper. The AC/DC project was supported by the Danish National Centre for IT-Research (CIT) [26].

The AC/DC project was initiated in April 1997 and ended successfully in April 1999. The project was supported by a consultancy group of 4 people (2 from the University of Aarhus [27], 1 from Dalcotech [28], and 1 from DELTA [29]) and a work group of 2 people from Dalcotech and 2 people from University of

Aarhus. About 2 man-years was spent on the project. The total budget was 1 million Danish kroner of which CIT contributed with half of the funding.

The purpose of the AC/DC project was to develop techniques and tools for automatic generation of code from CPN models. In this way it is possible to obtain an automatic implementation of systems that are designed by means of CP-nets and the Design/CPN tool. This eliminates a time-consuming and error-prone manual implementation phase. It also implies that the final system is behaviourally equivalent to the system design that was validated by means of simulation. Testing of the system can be accomplished already in the modelling phase. We elaborate on the technique and tools for automatic code generation in Sect. 2.

In the AC/DC project we applied the techniques and tools in practice. This was accomplished by means of a CPN model of a new embedded system for controlling buildings (access control, alarm system, energy control, energy measurement, etc.). The access control system was designed and specified by a CPN model, and it was implemented by means of automatic code generation directly from the model alone. We elaborate on the access control model in Sect. 3.

In Sect. 4 we compare our work with others, and finally we conclude the paper with a summary of main results of the AC/DC project and a report on ongoing and future research.

2 Automatic Code Generation Techniques and Tools

The method for automatic code generation constitutes a number of techniques and tools which we present in this section. Firstly the general approach is presented and then we explain how specialised libraries can be added in order to support the environment used by Dalcotech. The section after this (Sect. 3) describes the application of the method on an industrial case.

2.1 General Approach

The method for automatic code generation is summarised in Fig. 1. The figure indicates a number of stages and intermediate products. The central idea is that the system in mind is modelled by means of CP-nets in the Design/CPN tool. The Design/CPN tool supports several analysis techniques, such as state spaces and simulation, in order to debug the model. Once sufficiently confident that the model is accurate and complete we are ready for the code generation stage. The code, in the form of ML source, is extracted directly from the simulator because the simulator already implements a super-set of the model. (Details on the simulator can be found in a thesis [4].) Graphics simulation feedback is removed because it has no influence on the semantics of the implementation and is not needed in the final system. We only extract the kernel of the simulation engine and the implementation of the CPN elements of the specific model (cf. Fig. 2).

The ML source can now be compiled by whichever ML compiler is suitable for the target hardware platform in mind. In the figure there are two more stages which are specific for the AC/DC project, and we therefore postpone these technical stages to Sect. 2.2 below.

There are two major advantages of taking the implementation directly from the simulator. Firstly we get an implementation which corresponds exactly to what is included in the model. This immediately gives a number of benefits:

- Always consistent documentation of the system. As the model is considered as documentation it is only the model that is ever modified. Never the implementation.
- The implementation is of standard quality since the ML compiler always generates the same style of code.
- Analysis results found in the model can also be applied on the running system, assuming that the environment has been emulated accurately in the model.
- Errors are discovered early in the development process. The turn-around time is short should we find an error in the running system.

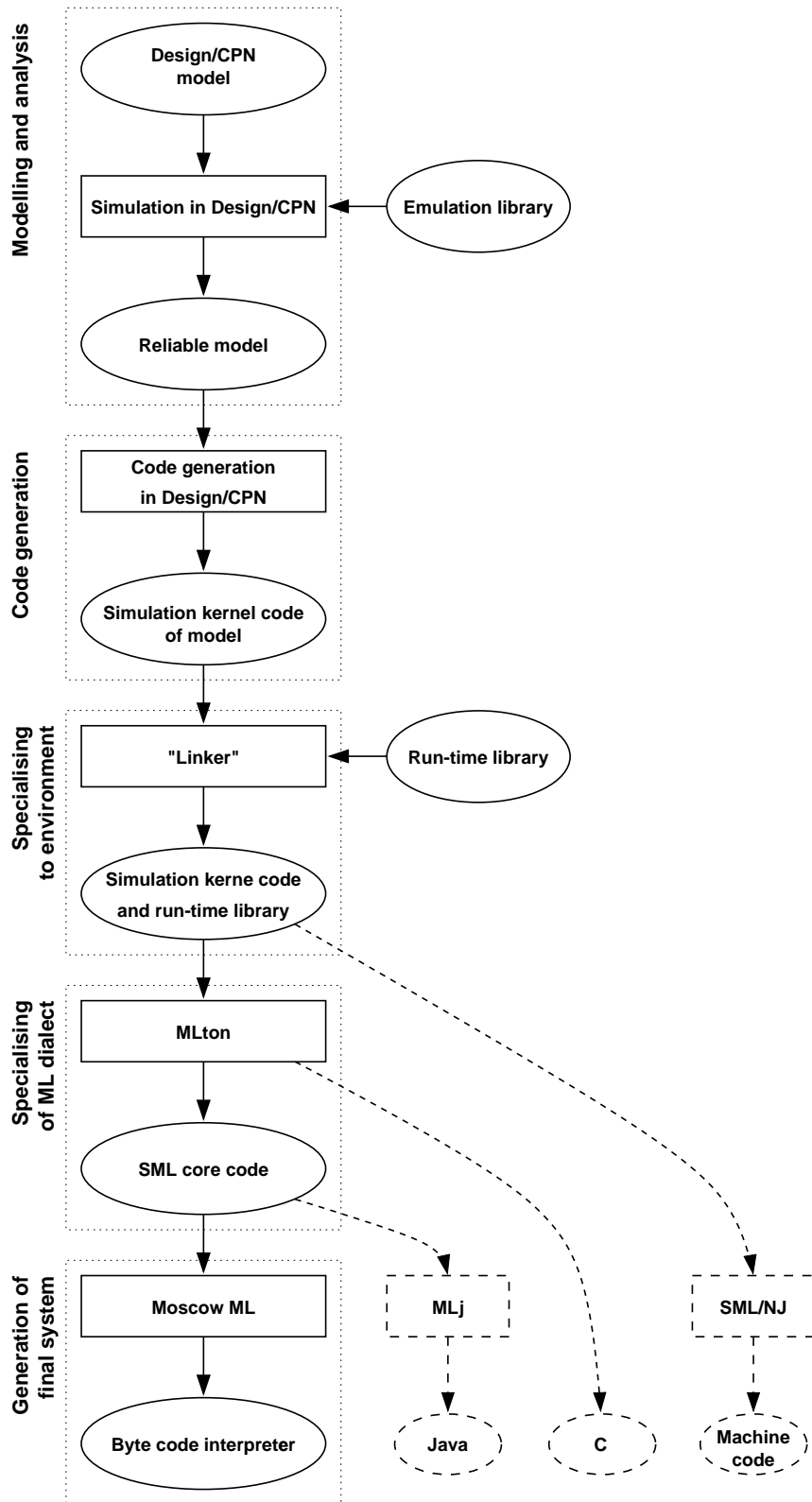


Figure 1: Overview of method as developed in the AC/DC project.

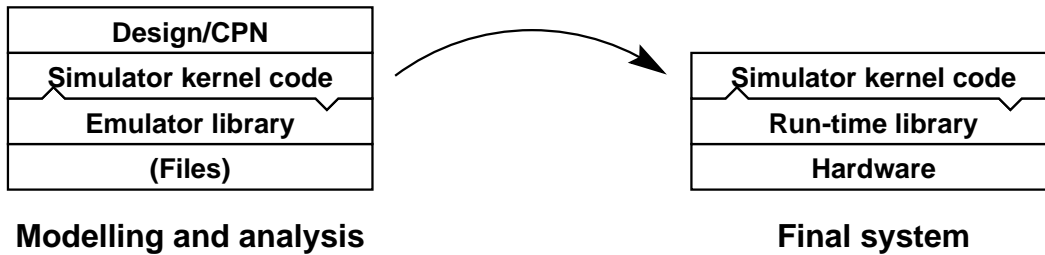


Figure 2: Same code used in simulator and final system implementation.

Secondly since the code generation method is automatized, we eliminate a time consuming and error-prone implementation phase.

- Errors do not originate from implementation since it is made automatically from the ML compiler.
- We save time because the implementation phase has been reduced to “pushing a button”.

Figure 1 also suggests the possibility for using a number of different ML compilers to generate the final system. Each compiler has each own advantages and limitations which we elaborate on in the following. We have chosen to focus on four compilers which generate very different kinds of code, namely SML/NJ, Moscow ML, MLton, and MLj. There are other ML compilers but their features are covered by these four. SML/NJ [14] generates native machine code and produces very fast executions. However the size of the code generated is very large. Moscow ML [21] generates byte-code which needs an interpreter. The execution is typically 50 times slower than SML/NJ but the size of the generated code is typically 75 times smaller than SML/NJ. MLton [23] generates C source code for which there exists numerous optimised compilers. Execution time is typically 3 times slower than SML/NJ but the size is about half of what SML/NJ generates. MLj [5] generates Java which has the advantage of being portable and supported by many platforms. We do not have experience with this compiler, but we expect that it has slower execution time than SML/NJ in the case of a Java interpreter. There are therefore a comfortable range of compilers for ML, and it is up to the user of the code generation method to judge the trade-offs in a given application.

Since we have no control over the implementation we may encounter some problems which depend on the target platform chosen. We lose control over performance of the automatically generated code. The simulation engine may not handle all kinds of models efficiently and could therefore be useless for some real-time systems. We also lose control over memory management which may be inappropriate for some kinds of embedded systems. We see later that these problems are not significant in the AC/DC project.

2.2 Code Generation for Embedded Systems at Dalcotech

Above we have described the general techniques and tools for automatic code generation. In the case of CPN models made by Dalcotech we need to take account of other factors. There are a number of requirements of embedded systems and limitations of specific environments described below. To solve these issues a special add-on library was made to handle the specifics of the environment used by Dalcotech. The post-processing supports add-on libraries (see Fig. 1).

Requirements and Limitations

Most of the security systems designed by Dalcotech, such as Seculon [20], contains an embedded controller which is the core of our concern. The controller has, as other controllers in embedded systems, limited memory and is required, to some extent, to exhibit real-time behaviour. However the limitations are not as severe as other embedded systems such as those found in pumps, wrist watches, and weights.

With the systems produced by Dalcotech we also need to take into consideration the fact that they are required to run in special standard hardware and software configurations. Otherwise the systems will not

be approved by national and international quality organisations. The controlling unit must run under MS DOS and with a special network called LonWorks. The network uses the protocol called LonTalk.¹

The reader may wonder why we consider an access control system to be a kind of embedded system, since the system is distributed over a network and there is user interaction. We generate only code for the controlling unit which is part of a larger LonWorks network configuration. The controlling unit senses hardware generated messages sent via the network and may send messages to the network. We therefore consider the controller unit to be a self-contained embedded system.

LonTalk Library

In order to support the LonTalk protocol in CPN models we have developed a library written in ML such that it is directly usable as inscriptions in Design/CPN. The LonTalk library is documented in a manual [18].

LonWorks is a networking architecture typically used for process control in building automation and manufacturing environments. The network speed is 40 kbit/s, and the protocol used, LonTalk, is OSI based. The protocol is reliable in the sense that if a send-message call terminates successfully it is guaranteed that the message was also received successfully and intact. (Authentication is also supported but not used in this project.) Devices, such as actuators and sensors, are attached to the network via a *Neuron chip*. A Neuron supports the LonTalk protocol, and input/output to the device in question. For instance, a device could be a lamp (actuator) or an infrared detector (sensor). A Neuron is responsible for maintaining and communicating values of network variables to other Neurons on the LonWorks network, and are therefore in some sense analogous to a neuron in a brain.

On the application level one does not know about the details of devices directly, but rather on an indirect level by means of *network variables*. A network variable is an abstraction of the state of devices on the network. For instance, the application programmer updates the value of a network variable which represents the state of a lamp or reads another network variable which represents the state of an infrared detector. A network variable may even be related with another network variable such that if one is updated then the other is also updated automatically. On system startup the network variables are configured to fit the setup of devices in question.

Apart from initialisation and configuration functions the interface of the library has two important functions, namely those to be used for input/output of messages to/from the LonWorks network. Below we have listed the signature of the functions:

```
val LONin  : LON_Millisecond -> LON_TIMED_EVENT
val LONout : NV_IDX * NV_VAL -> LON_RESULT
```

The function `LONin` attempts to read a message from the network within the specified time-out (`LON_Millisecond`). If a message arrives before time-out then it is returned immediately to the caller, otherwise at time-out the caller is informed that no messages arrived (`NO_EVENT`). The `LONout` function sends a message to the network and returns a diagnostic value. The `LONin` function should be called regularly in order to prevent input buffer overflow.

While modelling with CP-nets one often wishes to make a simulation in order to analyse the system. In a model where LonTalk is being used it is usually not possible to make a simulation because the network is not present. In order to remedy this problem a special version of the LonTalk library can be used which is able to emulate messages sent and received on the network. All one has to provide is a file which specifies which messages are to be received by the model and at which points in time. In this way one can debug the model with a simple list of messages and then run the model on the input and investigate how the model responds either at the end of a simulation run or in a stepwise fashion. This emulation version of the library is also documented in [18].

Generic Post-processing Tool

When we build models which are based on LonWorks we need to use the LonTalk library somehow in the model. In the model we wish to specify that we send and receive messages on the LonWorks network.

¹LonWorks and LonTalk are trademarks of Echelon Corporation [31].

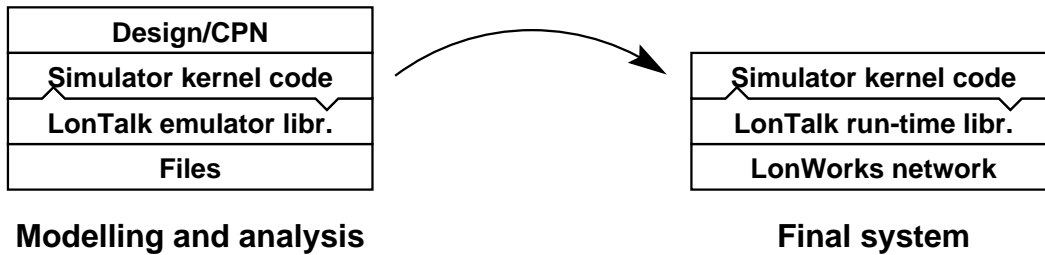


Figure 3: Dalcotech's use of libraries and automatically generated code.

While simulating with Design/CPN we cannot rely on the network being present. On the other hand the code generated from the model must run in a LonWorks environment. Figure 3 depicts the situation for Dalcotech and is therefore a specialisation of Fig 2.

To support this we have built a post-processing tool which takes the generated simulation code as input. The tool recognises platform dependent code, such as the LonTalk library, via special include statements specified by the user in the global declaration node: `use "_<library-name>.sml"`. It is the leading underscore which is recognised by the post-processing tool. The tool produces one self-contained source file, with the platform dependent code linked in, which can be compiled by whatever compiler is needed in relation with the target hardware in question. In the AC/DC project we run on the DOS platform and the compiler we use is Moscow ML [21]. This compiler generates compact byte-code for a DOS-based interpreter, and is therefore useful in the AC/DC project. The post-processing tool is documented in a manual [17]. In Fig. 4 the code generation method as used by Dalcotech is summarised. It is a specialisation of the general method as depicted in Fig. 1.

The disadvantage of Moscow ML is that it does not support the full module language of SML/NJ [14] which is what the code generator of the simulator produces. We therefore apply a, so called, de-functoriser [23] which unfolds the source code to the Standard ML core language [15]. The unfolded source can be compiled by Moscow ML.

The post-processing tool does not depend on the LonTalk library as used in the AC/DC project. The tool is in fact generic in the sense that it can be parameterised with whatever platform dependent libraries are needed. A library which is used with the tool must conform to the procedure as described in the manual [17]. In short one needs to make two versions of the library: A simple version which should consist of at least an interface, and a full version with complete functionality. The simple version is used in the model and the full version is linked with the simulator generated code by means of the post-processing tool. In the case of the LonTalk library the simple version is a simple emulation of the LonTalk protocol as described earlier in this section. The advantage is that we can use a platform dependent library both in the simulator and the final system, but the disadvantage is that it is the user's responsibility to ensure that the two versions of the library behave similarly.

3 Access Control Model

In this section we present the access control model which was made by Dalcotech in the AC/DC project. We demonstrate how the code generation method, described in the previous section, was successfully applied, and we show that the model architecture is flexible in case a customer requests new features.

The model was built mainly by two people from Dalcotech which took less than two man-months including specification, design/modelling, analysis, and code generation. People from the consultancy group was only involved in two model reviews. The reviews were very useful and effective means for improving the modelling strategy and architecture. The first review resulted in guidelines for continuing the modelling work while the second review was used for minor adjustments in order to finish the model.

The access control system works roughly as follows (see Fig. 5): A person outside the room may open the door by punching his personal code on the code entry unit (1). The system recognises the person,

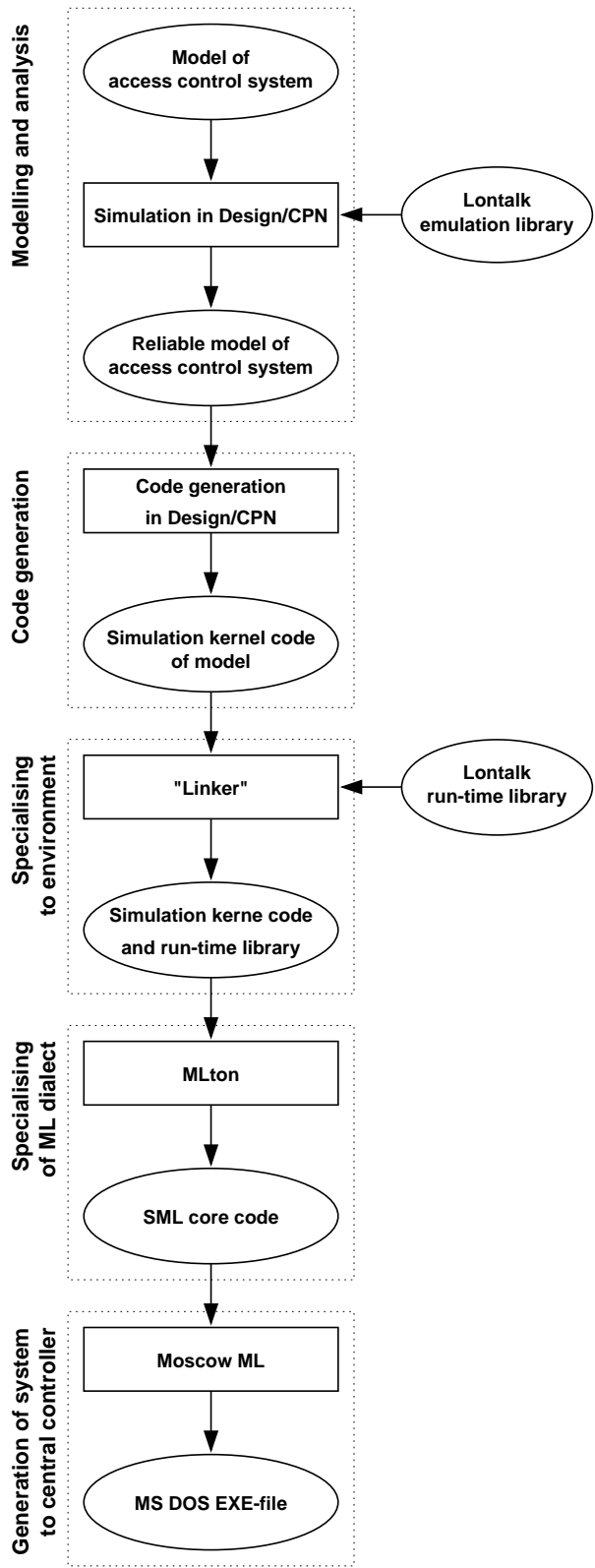


Figure 4: Overview of method as applied by Dalcotech.

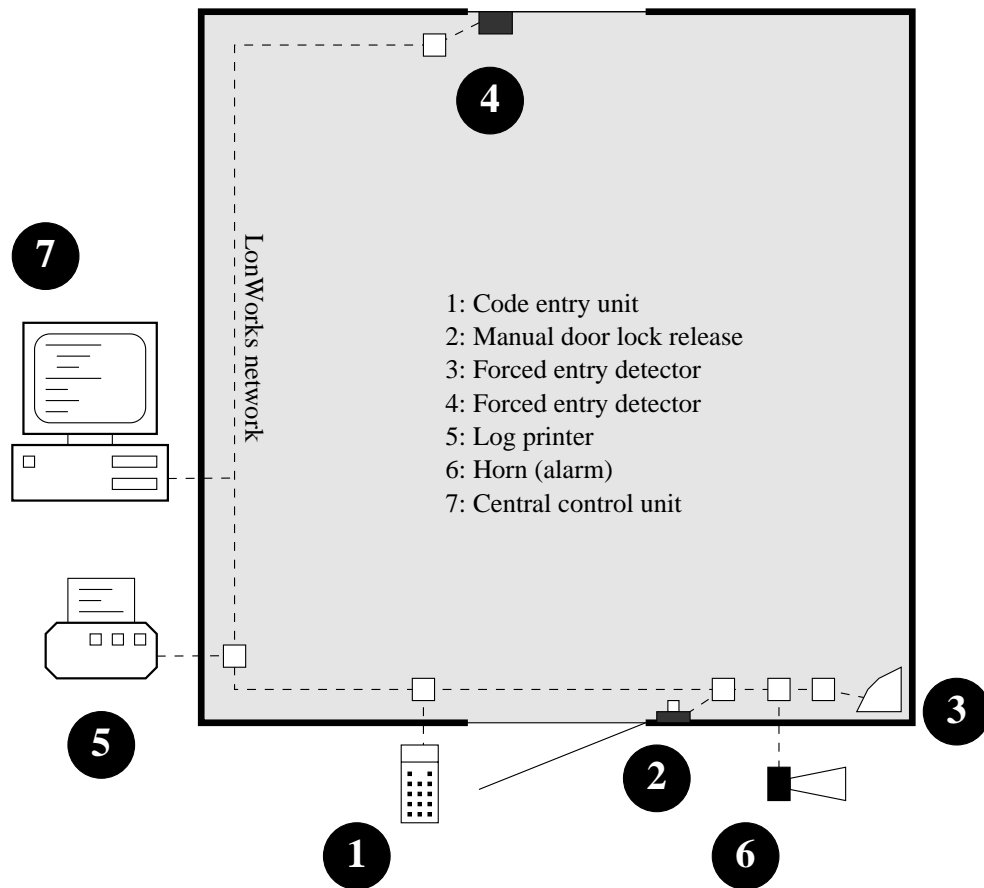


Figure 5: Access control scenario. White squares on the network represents the standard interface (Neurons) to various devices.

checks that the person is allowed in the area at this time, disables the alarm, and unlocks the door for a limited period of time. The system log is updated by printing status information on the log printer (5). More people may enter, each punching a personal code, and the system updates the log. In order to leave the room a person must push the open door button (2). The last person leaving the room must punch in the personal code in order to enable the alarm again. In alarm mode the horn (6) can be triggered in case one of the entry detectors are activated (3 and 4). There is other interaction possibilities and combinations of actions, but they are left out of this paper.

3.1 Model Structure

The structure of the CPN model of the access control system is depicted in Fig. 6. The model is of moderate size, and consist of 21 pages and 70 transitions. There are three important parts in the model, namely initialisation (*Initialize*), Neuron (*Neuron*), and event handler (*Main*). They are described in the following sections.

Initialize

The model is initialised in several steps in a sequential fashion, see also Fig. 6.

1. All network variables are initialised in order to related each network variable with a devices in the network. (See also Fig. 7.) It is worth noting here that this configuration covers three code entry units

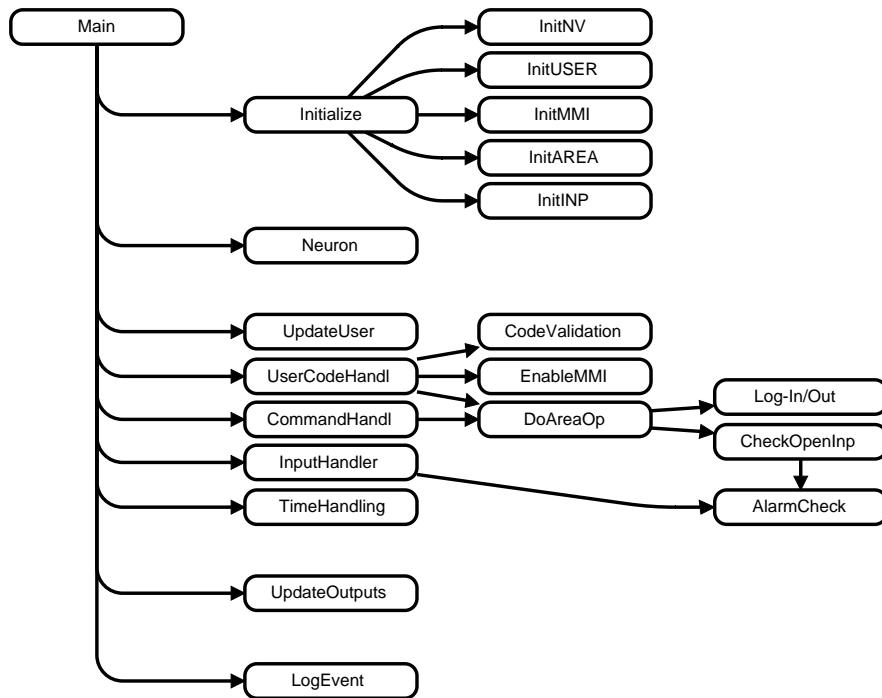


Figure 6: Hierarchy page of CPN model.

which corresponds to covering three rooms (or doors). Furthermore there are 10 user configurable network variables.

2. The user database is initialised with names related with entry codes and where people are allowed to be and when.
3. The database containing information about man-machine interfaces (MMI), such as the code entry unit, are initialised with logic for managing enabling and disabling of alarms in case the correct code is punched.
4. The expected initial states of inputs are initialised. For instance we assume that all input devices are off.
5. Finally we initialised the status of the areas. All areas a initially armed.

After this sequence the configuration is extracted from the databases in the model and sent to the network. If the configuration fails then error information is passed on to the main loop of the model.

Neuron

The Neuron is a standard component in LonWorks based systems. Care should be taken to model such a component as it is expected to be reused in other models of LonWorks systems. The model made by Dalcotech is depicted in Fig. 8. Relevant colour sets are shown below:

```

colorset NV_UPDATE = product
  NV_IDX *      (* LonWorks Network variable index *)
  NV_VAL;      (* LonWorks Network variable value *)
colorset LON_EVENT = union
  ...
  NO_EVENT +  (* No events from LonWorks *)
  
```

Initialize Network Variable Setup

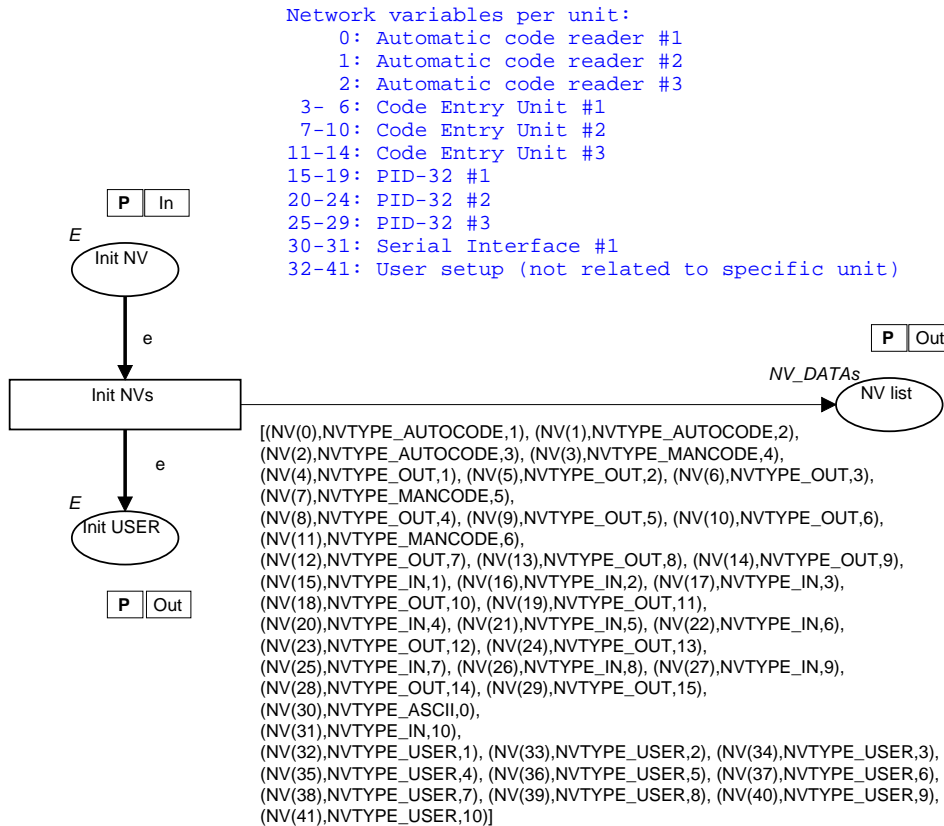


Figure 7: CPN page modelling the initialisation of network variables.

```

UPDATE: NV_UPDATE; (* Update of network variable *)
colorset LON_TIMED_EVENT = product
LON_EVENT * (* LonWorks event *)
INT;      (* Ticks passed since last time reporting ticks *)

```

As with the physical Neuron, the model is divided into two parts: A device driver part (left) and a part providing an interface to the application (right). The device driver part just makes basic input/output calls to the LonTalk library which is the basic interface to the network. The LonTalk protocol is rather complicated and is therefore not modelled as a CP-net. The application part converts network messages to higher level messages (events) appropriate for the application (upper half) and also converts higher level messages to network messages (lower half). In case no message (`NO_EVENT`) is received from the network then timer information is propagated. This information can be used elsewhere in the access control model for managing user access times and alarm timeouts. Notice that the model does not use timed CP-nets but instead stores information on the (real) time passed since last event was read (`LON_TIMED_EVENT`).

The Neuron is modelled such that either input or output is treated exclusively, never both at the same time. This is to avoid multi-threaded behaviour in the access control model. Why we have made this choice will become more evident below where we describe the model part for event handling.

We could also imagine that the model would benefit from using timed CP-nets. Timeouts and delays could then be expressed directly as modelling primitives and then potentially make the model simpler. The timing logic of the transition scheduler of the simulation engine would then need to be related with the system clock. Otherwise the timeouts and delays would not be real-time. Unfortunately this simulation architecture is not very well studied, and there are problems to be solved. For instance, what if the model

Neuron I/O Handling

Converts messages between the low level network variables and the message types used in the remainder of the model.

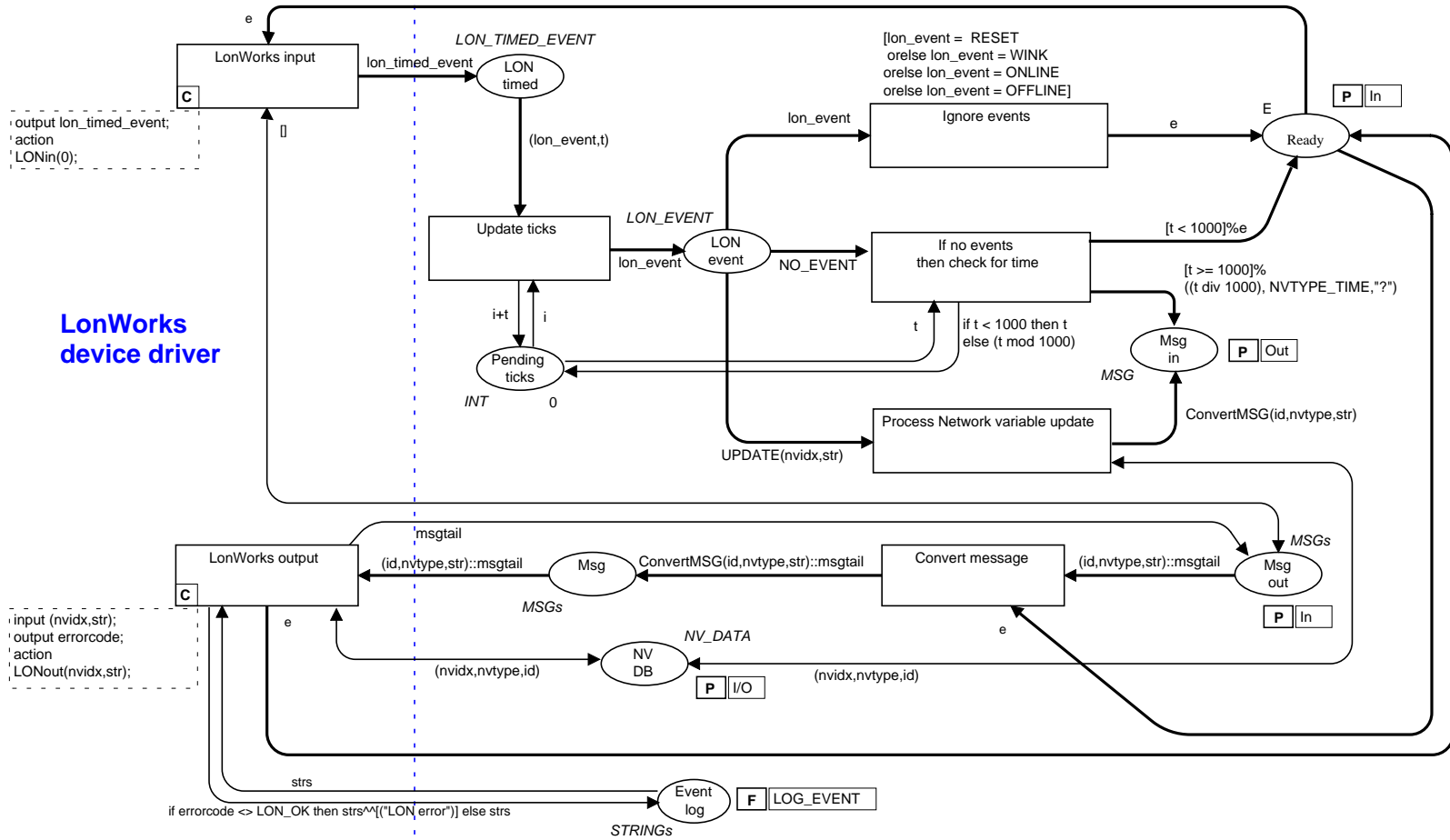


Figure 8: CPN page modelling the Neuron.

gets significantly behind the real-time clock? This could happen if a calculation in a transition takes up too much CPU time. We leave this as a research topic for the future.

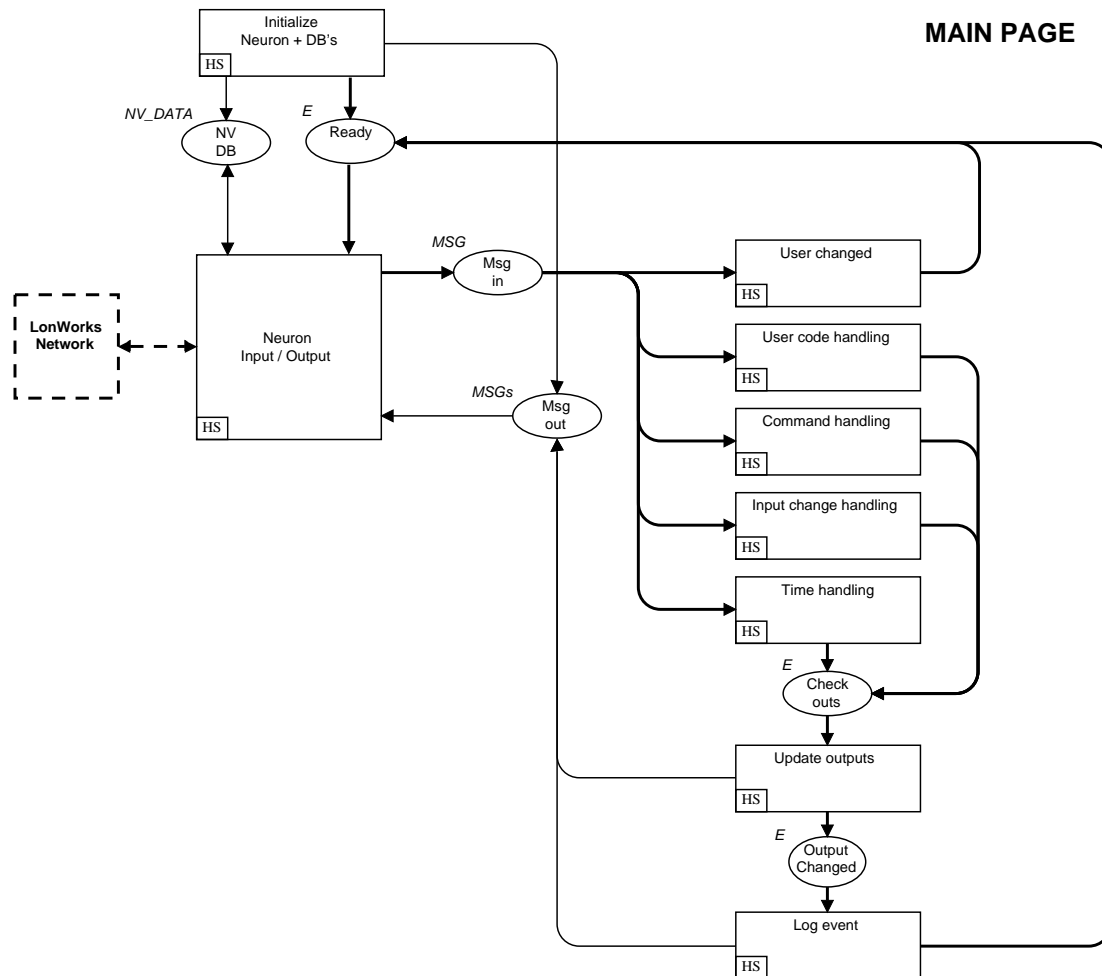


Figure 9: CPN page modelling the main event handling cycle.

Event Handling

Event handling is a central cycle in many systems, in particular reactive systems where user input is involved. The access control model also has an event cycle which is depicted in Fig. 9. The relevant colour sets are listed below:

```

colorset MSG = product
  INT *      (* Id *)
  NV_TYPE * (* Network variable type *)
  NV_VAL;   (* Data *)
colorset MSGs = list MSG;

```

Once the system is initialised (top left substitution transition) the Neuron is activated such that new messages from the network can be read. The Neuron, *Neuron Input / Output*, converts a network message to an event which is put on the place *Msg in*. This is subsequently dispatched to the appropriate event handler which can be one of *User changed*, *User code handling*, *Command handling*, *Input change handling*, or *Time handling*. Once an event is fully treated then the event cycle

checks if we need to generate any messages to the network (Update outputs), and then generates messages for the log printer on the network. For instance, if a user has punched in an access code, then the system registers this and then sends an unlock door message, a turn on green light indicator message, and a log entry message to the printer. Finally the Neuron is activated again for processing all assembled messages on `Msg out`, and then the Neuron is ready to process the next input message from the network.

Note that the description just provided is purely sequential. As noted before where we described the Neuron page, concurrency is something we have chosen to avoid at this level. Handling of multiple events concurrently would make the model much more complicated because there are several data bases accessed throughout the model, and ensuring mutual data base consistency would be difficult. Also, we know that the target hardware platform only has a single CPU, thus there is not an obvious performance benefit of having concurrent activities in the model. However, the idea was to start with a simple solution and then make it more advanced with respect to concurrency as more experience was obtained.

3.2 Code Generation Method Applied

In the following we describe in detail how the code generation tools are applied for the access control model. We assume that we have the access control model and that we are sufficiently confident that it works. In the global declaration node we have made a reference to the platform dependent LonTalk library by including the statement `use "Lontalk.sml"`. The steps described below corresponds to the high-level description in Sect. 2.

We load the access control model into the Design/CPN tool and invoke the ML simulation engine (called the switch to simulator). As part of the syntax check and simulator setup, the ML source code is extracted and then saved to a file, say `/tmp/ml.out`. Then we call the post-processor on a Linux machine to make platform dependent source:

```
linux% cpnsim2mosml.pl /tmp/ml.out /tmp/accsctrl.sml
```

The post-processor recognises the special use-statement and includes the full version of the lontalk library. Finally we move `accsctrl.sml` to the DOS platform and create the final executable:

```
dos% mosmlc accsctrl.sml
```

This command creates a DOS executable called `mosmlout.exe` which now can be started, assuming the LonWorks network is attached to the machine. The whole procedure takes 10–20 minutes on a 200MHz Pentium PC with 64Mb RAM for the access control model.

Once the automatically generated implementation is executing on the DOS platform we can investigate the behaviour and responsiveness of the system. We have a simplified demonstration hardware scenario similar to Fig. 5. Instead of having an alarm horn there is a lamp, and instead of specific sensors such as forced entry detectors there is a switch which can be controlled manually. This is a cheap but still effective scenario for testing the running system.

We are now able to investigate the running system. Although we are able to run the same system in the simulator of Design/CPN, we may discover anomalies which were not found while simulating. The state space tool in Design/CPN could have helped us to find more problems as a means to make an exhaustive simulation, but this tool was never applied in the AC/DC project, mainly because the model has an infinite state space (messages on the network is unbounded). The model can however be modified such that we obtain a finite state space, but this was not pursued further. Another interesting property of the running system is how it performs. How quick can it react to user inputs and is it sufficiently fast to generated alarms? On the 33MHz DOS machine with 32Mb of RAM the system was indeed running fast enough. The system reaction time to user input is less than half a second, which is within the margins of this kind of access control systems. We could in principle have carried out more systematic performance analysis in Design/CPN as it is supported in the tool, but this was out of scope of the project.

3.3 Benefits of Model Architecture

The model we have describe above in Sect. 3.1 has a number of useful engineering properties. It has reusable components, is extensible, and adaptable. It is of great interest for Dalcotech to learn from mod-

elling experiences and use this model in future projects. Usually a customer will need an access control system with special features and it is therefore important to avoid building the system from scratch every time. Reusing existing modules is faster and safer. In the following we show that the model has such properties.

Extensible

Consider the CPN module in Fig. 9 which depicts the event handler of the access control model.

The structure of the event handler is divided into cases for each kind of possible event. A new customer would typically request support for new devices in the system which consequently would induce new kinds of events from the network. Extending the case structure is relatively easy as one just needs to add a new substitution transition to handle the new kind of event. An event may require updating some of the access control data bases in the system, but these are easily accessible as they are located on fusion places.

It is also relatively safe to add support for a new event with respect to data base integrity, because only one event at a time can be processed by the system.

Reusable

Consider the CPN module in Fig. 8 which depicts the Neuron of the access control model. A Neuron is a standard component to interface between devices and the LonWorks network. We therefore only need to model a Neuron once.

If we model the Neuron as one page, as in our case, we can easily use the save/load sub-page feature in Design/CPN in order to imitate a simple module facility. Should we need the Neuron module in another model, all we have to do is to make a substitution transition and load the Neuron module via load sub-page and then assign the ports to the sockets around the substitution transition.

As the Neuron is a standard component it will be used frequently and therefore also tested in many different contexts. We therefore expect such a module to become increasingly robust over time. As the module matures it will become safer to use in new contexts.

Another example of re-usability in the access control model is the underlying data bases which are used to manage the access control logic. This manager is not described further, but we would like to note that the functions operating on the data bases are written in ML and therefore easy to include as libraries in the global declaration node. ML libraries are in general frequently used as reusable components in Design/CPN models.

Adaptable

Consider the initialisation phase which is described in Sect 3.1. One step in the initialisation consists of sending configuration information to the LonWorks network (Fig. 7) such that network variables are configured for the attached devices. The configuration is in fact adaptable by the end-user within a limited range. For instance, a specific code generated system targeted for small buildings will typically be pre-configured to be able to handle about three rooms with 20 sensors and actuators as is the case in Fig. 7. In this way Dalcotech does not need to code generate a new system for every new customer, but can instead have a limited number of products each covering a range of building sizes.

4 Related Work

There are more than 50 tools for Petri nets which are actively being developed [32]. Many of these tools have some sort of simulation support for Petri nets. Typically a given Petri net model is translated into a different language (C/C++, Java, SML, etc.) which subsequently is interpreted or compiled by already established and well-known tools. In this section we compare our work with other tools and approaches, however limited to tools supporting high-level Petri nets or other high-level languages such as object-oriented languages.

LOOPN++

LOOPN++ supports a kind of object-oriented high-level Petri nets [10]. It contains a translator which can generate C++ source code which is then subsequently compiled to native machine code and executed. The tool has also been extended to generate Java source code [11]. LOOPN++ is only a compiler and does not contain a simulator, and is therefore very different from Design/CPN in the respect that LOOPN++ does not have a direct behaviour relation between model and generated code (cf. Fig. 2).

CPN-AMI

CPN-AMI is a CASE environment based on some kind of high-level Petri nets [13]. Among other components CPN-AMI contains a simulator (CPN/DESIR) and a code generator (H-Tagada) which generates ADA code. The code executed by the simulator and the generated ADA code are different. Thus there is not as strong relationship between simulation and generated code as with the code generation method of Design/CPN (cf. discussion of Fig. 2). However, the H-Tagada code generator is able to separate a Petri net model into processes (G-objects), thus making potential for true concurrent executing in multi-processor environments. This is not supported in the Design/CPN code generator.

Another tool called Artifex [25] is similar to CPN-AMI in that it has both a simulator and code generator, however Artifex generates C/C++ code. There are a number of other tools which has the same code generation architecture as these.

PEP

PEP [3] has many different kinds of translators which can translate back and forth many different kinds of languages such as P/T nets (Petri box), high-level nets (M-net), and textual language ($B(PN)$)². A modification in one language representation can automatically be updated in another, in a consistent fashion. This is analogous to what sometimes can be found in UML notation based environments. Updating a textual representation of a class will consistently update a diagrammatic representation of a class and vice versa. An example of such system is the Mjølner System [8].

The code generation method of Design/CPN does not support what can be called reverse engineering, i.e., translation from implementation to a CPN model. However, this is not the intention either. Once the code has been generated it should neither be investigated nor modified manually. One would then ask if we could support reverse engineering, but this would be difficult in the case of SML and Petri nets because they are radically different languages. This is not the case for the UML based environments where the diagrams and textual languages are very similar in structure. Also in the case of PEP the various languages supported are mutually similar, and could thus be worth investigating closer in case we wish to support translation in both directions in Design/CPN.

ML compared with C/C++

We are often asked whether it would be easier to generate C/C++ source code instead of ML. There are two main reasons that C/C++ would be harder to generate in the current architecture. Firstly the inscription language of CPN models is ML. Most of the inscriptions are directly implementable in ML. Generation to C/C++ would require a translation, in particular for the cases of pattern matching. Pattern matching is a powerful construct and is frequently used in CPN models. ML directly supports pattern matching but C/C++ does not.

Secondly the ML compiler is interactive which is in contrast to most compilers for C/C++ which are batch compilers. It is easy to change a CPN model given an interactive compiler because we just send new source code to directly overwrite existing representations of the model. With C/C++ batch compiling the turn-around time is longer because a new application needs to be generated.

In general the CPN language is more in the nature of functional languages where side effects are not allowed. C/C++ is imperative and side effects are more naturally used.

5 Conclusion

Dalcotech is a small company and it is therefore limited what the company can invest of resources on a project like AC/DC. However, given that Dalcotech was already familiar with CP-nets and previous positive experience with CPN development projects they were quite certain that the AC/DC project would be feasible [7].

Although Dalcotech has used CP-nets in practice for a number of years, it is in AC/DC the first time they use CP-nets directly for automatically generating the implementation. In this way Dalcotech now has the capability to dramatically reduce the time spent in the implementation phase. Additionally they also dramatically reduce the amount of time spent on debugging because errors in the implementation in principle originate from the model only.

In the AC/DC project Dalcotech has successfully conducted a code generation case study where a CPN model has been built for the next generation of access control systems the company has planned to produce. By means of the code generation method and their model, Dalcotech has obtained an automatic implementation of the embedded controlling unit of the access control system. This non-trivial case study proves that the method can be applied in practice by engineers who are not completely familiar with the advanced technology behind automatic code generation. Additionally, Dalcotech has during the modelling process gained better insight into access control systems in general.

The AC/DC project has for our part provided an excellent opportunity to develop and apply the techniques and tools in the context of an interesting industrial case study. Furthermore, we have been able to extend the use of CP-nets into the new area of automatic code generation for embedded systems and developed a method to support this [16]. It has been a useful challenge to match the needs of Dalcotech, and we have thus become more experienced with the techniques and tools and more confident that our method works in practice [7].

A significant part of Dalcotech's revenue is from development contracts and it is therefore expected that automatic code generation from CPN models will have a significant role in the future [7].

Another ongoing activity is the effort to conduct technology transfer of the method to other companies. A full day seminar was conducted for the Danish industry. The seminar included introduction to CP-nets, the method for automatic code generation, and its application. Such seminars are held under the auspices of DELTA. In this way we help to make advanced computer science technology more easily available for a wider audience. At the seminar the attendants were motivated and asked a lot of relevant questions. One of the main concerns, however, was how they could convince their companies to use the CPN methodology. We proposed a number of arguments that could be used: The CPN language is simple to learn and yet powerful to use. It is a visual approach where diagrams can be explained for non-experts. The CPN methodology can be introduced in a company for enriching existing methods, such as UML — not as a replacement. Typically CP-nets can be used in situations where the existing methods are limited, e.g., in analysis and verification phases. We have also seen in a previous CPN project with Dalcotech that the CPN methodology was used as part of a valid argument for approving a security system product based on CPN.

We have also made contact with research groups in Copenhagen who are working with optimising ML compilers such that garbage collection can be avoided [22]. Most ML systems have a garbage collector of some kind. They are interested in AC/DC because they are looking for an industrial case study. We are interested because for embedded systems an automatic garbage collector may cause unpredictable and unwanted long waiting times, and getting rid of garbage collection is expected to be a benefit in practice. In the access control system the garbage collector has never been an issue, but this may be different for other kinds of embedded systems, e.g., where real-time behaviour is more critical.

Finally we need to generalise and optimise code generation in several directions. There is ongoing work of defining parametrised CP-nets [2, 12]. Thus we need to consider code generation for parametrised CPN modules which is expected to be realistic because SML also supports parametrised modules. It is also interesting to minimise the amount of generated code. With a minimal CPN model and Moscow ML we need at least about 160kb memory which is the best we can hope for at the moment because Moscow ML is the only ML compiler we know of which generates the least amount of code. Can we find alternative strategies for code generation? What if we generate code, not for a ML interpreter or Java virtual machine, but instead a CPN virtual machine? Can we make a compact CPN interpreter for which we can generate small sized applications? According to the description of the Petri net tool called PACE [32], it can translate

models into code for a virtual Petri net machine. This should be investigated further.

Acknowledgements

The AC/DC project was supported by the Danish National Centre for IT-Research (CIT) [26]. The first version of the CPN code generator was developed by T.B. Haagh and T.R. Hansen and is documented in their thesis [4].

Seculon is a trademark of Dalcotech A/S [28]. LonWorks and LonTalk are trademarks of Echelon Corporation [31].

Finally we acknowledge the four referees of this paper who have provided many useful suggestions for improvement.

References

- [1] T. Andersen. SECU-DES, Improved Methodology for the Design of Communication Protocols in Security Systems. Final Report, ESSI Project 10937, Dalcotech A/S, May 1996. Online version: www.esi.es/ESSI/Reports/All/10937/.
- [2] S. Christensen and K.H. Mortensen. Parametrisation of Coloured Petri Nets. Technical Report DAIMI PB – 521, University of Aarhus, Computer Science Department, April 1997. ISSN 0105-8517.
- [3] B. Grahlmann. The PEP Tool. In *Tool Presentations of ATPN'97 (Application and Theory of Petri Nets)*, June 1997.
- [4] T.B. Haagh and T.R. Hansen. Optimising a coloured petri net simulator. Master's thesis, University of Aarhus, Department of Computer Science, Denmark, 1994.
- [5] Persimmon IT. MLj a SML to Java Bytecode Compiler. Web site. <http://www.dcs.ed.ac.uk/home/mlj/>.
- [6] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1992.
- [7] B. Jørgensen. Målte dybden før springet. *CIT NYT*, 2:8–9, June 1999. Article in newsletter, in Danish.
- [8] J.L. Knudsen and B. Magnusson M.Löfgren, O.L. Madsen, editors. *Object-Oriented Environments: The Mjølner Approach*. Prentice-Hall, 1993.
- [9] M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, December 1998.
- [10] C. Lakos and C. Keen. LOOPN++: A new language for object-oriented petri nets. In *Proceedings of Modelling and Simulation (European Simulation Multiconference)*, pages 369–374, Barcelona, 1994. Society for Computer Simulation.
- [11] G.A. Lewis. Producing network applications using object-oriented petri nets. Master's thesis, University of Tasmania, November 1996.
- [12] T. Mailund. Parameterised Coloured Petri Nets. In K. Jensen, editor, *Proceedings of the Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, Department of Computer Science, University of Aarhus, 1999. To appear.
- [13] MARS-Team. The CPN-AMI environment. Technical report, MASI lab, Institut Blaise Pascal, Université Pierre & Marie Curie, Paris, France, October 1993.
- [14] D. McQueen. Standard ML of New Jersey. Web site. cm.bell-labs.com/cm/cs/what/smlnj/.
- [15] R. Milner, R. Harper, and M. Tofte. *The Definition of Standard ML*. MIT Press, 1990.

- [16] University of Aarhus, Dalcotech A/S, DELTA, and CIT. Automatisk kodegenerering fra farvede petri net. Final report for CIT-project number 106. In Danish, May 1999.
- [17] J.-H. Paulsen. *Design/CPN Automatic Code Generation User's Guide*. University of Aarhus, Department of Computer Science, Denmark, version 0.6.6 edition, June 1999.
- [18] J.-H. Paulsen. *Design/CPN LonTalk Library User's Guide*. University of Aarhus, Department of Computer Science, Denmark, version 0.4.5 edition, June 1999.
- [19] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2 edition edition, 1996.
- [20] J.L. Rasmussen and M. Singh. Designing a security system by means of coloured petri nets. In J. Billington and W. Reisig, editors, *17th International Conference on Application and Theory of Petri Nets 1996*, volume LNCS 1091 of *Lecture Notes in Computer Science*, pages 400–419, Osaka, Japan, June 1996. Springer-Verlag.
- [21] S. Romanenko and P. Sestoft. *Moscow ML owner's manual*, version 1.43 edition, April 1998. Web site: www.dina.kvl.dk/~sestoft/mosml.html.
- [22] M. Tofte, L. Birkedal, M. Elsmann, N. Hallenberg, T.H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ml kit. Technical Report 25, University of Copenhagen, Department of Computer Science, 1998.
- [23] S. Weeks. MLton User's Guide. Available online: www.neci.nj.nec.com/PLS/MLton/.
- [24] AC/DC Project. CIT project number 106. Web Site. www.daimi.au.dk/CPnets/ACDC.
- [25] Artifex by ARTIS. Web Site. www.artis-software.com.
- [26] The Danish National Centre for IT Research. Web Site. www.cit.dk.
- [27] Coloured Petri Nets at the University of Aarhus. Web Site. www.daimi.au.dk/CPnets.
- [28] Dalcotech A/S. Web Site. www.dalcotech.dk.
- [29] DELTA Software Engineering. Web Site. www.delta.dk.
- [30] Design/CPN Online. Web Site. www.daimi.au.dk/designCPN.
- [31] Echelon Corporation. Web Site. www.echelon.com.
- [32] World of Petri Nets. Web Site. www.daimi.au.dk/PetriNets.