Formal Modelling in Embedded System Design: a Case Study

Rinat Khoussainov and Ahmed Patel

Computer Networks and Distributed Systems Research Group
Department of Computer Science, University College Dublin
Belfield, Dublin 4, Ireland
{rinat, apatel}@net-cs.ucd.ie

Abstract. The real-time reactive nature of embedded systems and rigid timing and performance requirements imposed by the external environment significantly complicate the design process for such applications. The use of formal methods for specification and design of embedded systems is a promising solution to overcome these difficulties. This paper investigates the problem of mapping between a high-level formal specification and low-level design and implementation. We analyse possible compromises between the strictness of requirements specified and the easiness for implementation. A complete example is provided of design of an embedded modem concentrator on the basis of SDL specifications and modelling. An advantage of the proposed design flow is that is allows for systematic performance optimisation of the system.

1 Introduction

Recent advances in computing technologies resulted in a wide extension of the area where embedded systems are used, in particular, in the world of data communications. Although these systems were broadly used in life-critical applications in industry, yet design methods for such systems were far from well developed. Historically, embedded systems were designed using ad hoc techniques that were heavily based on previous experience with similar products and on manual design [1]. A study by Schneider [2] highlighted the problem by saying that, “unlike other engineering disciplines, our methods are not founded on science. Real-time systems are built one way or another because that was the way the ‘last one’ was built. And, since the ‘last one’ worked, we hope that the next one will”. Nowadays, most researchers in the area agree that in order to improve efficiency of the embedded systems, design approaches should be based on use of more formal models for description of the behaviour of such systems. The authors in [3] advocate that a design approach should step from a definition of a system at some high level of abstraction down to the decomposition into hardware and software components and implementation of each particular part. If these transformations from the high level of abstraction are done using automatic synthesis, we can obtain implementations that are “correct by construction". Validation of parts uncovered by this approach should also be done as much as possible at the higher levels of abstraction.

In this paper, we discuss problems related with the use of formal models and techniques for design of embedded systems. Particularly, we outline the gap between the formal specification of system properties and description of the system design at a level of details allowing for implementation of the system on a particular hardware and software architecture. We demonstrate one of several possible methods to resolve the problem by providing an example of the use of formal methods for design of an embedded data communication system, namely a modem concentrator. The significance of the paper is that it notifies the difficult points among the design process of such reactive embedded systems that other developers and researchers will most likely experience during development of products with similar functional and performance requirements. The design approach proposed in this paper together with the survey of other formalisms available in the area should allow designers to improve the efficiency of systems they develop and to reduce the number of iterations for obtaining a satisfactory design. An additional advantage of the formal approach used in our sample design is that it provides a strong basis for systematic performance optimisation of the system software.

The rest of the paper is structured as follows:

- Section 2 gives a survey of different formalisms for description of embedded systems.
- Section 3 motivates the use of SDL for description of telecommunication systems using a dependency between the expressive power of different formalisms (i.e., what system characteristics they can specify) and the easiness of implementation.
- Section 4 shows how timing constraints in a real-time system can be specified using SDL.
- Section 5 describes the case study design of an embedded modem concentrator based on SDL specifications.
- Section 6 demonstrates how the use of SDL-based design allows for systematic performance optimisation of the system.
- Section 7 presents overall conclusions and outlines possible directions for future work.

2 Formal Models and Specifications

The set of tasks that a system should implement is often not specified in a strict and unambiguous manner, so the design process requires several iterations to obtain satisfactory results. Moreover, the level of abstraction and details can vary for different parts of the design; different engineers on the project are likely to have different professional skills. Therefore, managing the design is getting more difficult as the complexity of the systems grows. Several formal approaches to the system specification and modelling were proposed by researchers over recent years to overcome this problem.

The design process is often viewed as a sequence of steps that transform a set of specifications described informally into a detailed specification that can be used for manufacturing of the system. Briefly speaking, a designer receives a description of a system in a form like “The device should do this and that in such a given manner”. Eventually, he should produce a description of the same system, which would be detailed enough to allow for its manufacturing. It is easy to see possible sources of serious difficulties for design based on such representation and interpretation:

– No strict way for formulation of system functions and requirements is specified and, hence, there is a potential risk of misinterpretation of given specifications by different designers.
– No formal methods for transformation between specifications are established. Thus, validation of the system functions at each step gets too complicated as well as the reasonable number of such iterations is unclear.

A wide range of potential formalisations of design has been proposed that we consider below. As a rule, these are sets of relations or equations that assume the system they specify allows for decomposition into abstract blocks with properties and interrelations. In this case, we would be able to use some formal notation to impose constraints or to describe relationships between those blocks to achieve desired system functionality and performance parameters. A set of abstract objects with their possible properties and rules for establishing interrelations between the objects is called *formal model* of the system.

Formal models vary in the level of abstraction they provide for representing a system from purely analytical ones to models that are more close to real world implementations, like state-based techniques. The following section gives a survey of the most popular types of formal models used for design of embedded systems. There are two methods for classification of formal models for the embedded systems design:

– The first one uses a classification scheme based on some *similarities in models*, such as use of the same set of abstract objects for description of a system, or employing similar abstract notations to represent interrelations between system components.
– The second approach uses a formalism to describe aspects of embedded system models in a standardised way, i.e., using some universal description scheme.

Lee and Sangiovanni-Vincentelli [4] proposed a formalism describing aspects of models of computation for embedded system specification called *tagged-signal model*. This is a denotational formalism defining a semantic framework of signals and processes, within which models of computation can be studied and compared. However, as it is noticed in [3], the proposed model of computation is very abstract and requires imposing of additional constraints in order to use it for description of particular formal models. Thus, we base our classification on similarities in formal models.

2.1 State-Based Models

State-based models comprise *Finite State Machines* (FSM) [21] and *Finite Automata* (FA) based models and techniques. FSMs are very popular models for embedded systems. The reason for this is that those models are decidable in terms of the memory necessary for modelling and the time required for analysing of these models. However, modelling of complex systems can meet serious practical difficulties. Namely, whereas traditional FSMs are good for modelling of sequential behaviour, they suffer from two major disadvantages, which make them impractical for modelling of concurrent or real-time systems:

– A so called *state explosion* problem [3]: When a group of concurrently functioning machines is substituted by a single machine, the number of states in the substituting machine is equal to the product of the number of states in each machine from the initial group. Although, the number of states alone is not always a good indication for complexity of a system, yet there is a strong correlation.
– *Expressing timing requirements* in the traditional FSM concept: It is easy to see that there are no ways given in the original FSM model to reflect needs of real-time systems to satisfy certain timing constraints or to express interrelations between an embedded system and evolving environment.

In [5], Harel proposed three mechanisms to reduce the size of a Finite State Machine for modelling of practical systems: state hierarchy, simultaneous activity, and non-determinism. The first one uses an FSM model, in which a state can represent an enclosed state machine. The situation of an FSM being in state *s* has the interpretation that the state machine enclosed by state *s* is active or, equivalently, this means that the machine is in one of the states enclosed by *s*. Since the system may be in one of the states *s*, they are also called *or*-states. The second mechanism is concurrency. In the proposed state hierarchy, concurrency is achieved by enabling two or more state machines to be active simultaneously. Therefore, each state of the system is determined by the current states of each simultaneously active machine. Such states are also denoted as *and*-states. The last mechanism, namely non-determinism, is an imprecise specification of the system functionality. While it may simply be a result of erroneous specification, non-determinism can be extremely powerful for reduction of the system complexity by abstraction. There are two positive moments associated with this abstraction. Firstly, it allows us to exclude from any discussion components, whose properties are not considered of interest for particular analysis. Secondly, non-deterministic description of components permits for some optimisation during the implementation phase.

The early attempts to introduce time into state-based models used *timer alarms* for modelling of real-time events in telephony systems [6]. A timer alarm is an artificial input signal that is generated if a specified timing deadline is missed. The best analogy is to compare alarms with interrupts signalling occurrence of an exception.

While it is suitable for specifications of protocols or interfaces, for which sets of input and output signals are strictly defined, these approach seems unrealistic for embedded systems, where no assumption can be made about the behaviour of the external environment. To resolve this problem, Lynch [7] proposed a concept of *Input/Output Automata* (IOA) – a model, in which input actions are distinguished from local actions. A transition is defined for any input action and for every state of the automaton. The IOA model can be used to develop proof techniques for study of discrete event systems. In [8], an extension to the IOA model, called *Input/Output Timed Automata* model (IOTA), was proposed that allowed for the specification of lower and upper bounds on the delay between the enabling of a locally controlled action and its firing. A real-time system is represented in the IOTA model as a set of communicating IOTAs. The same author proposed the CLEOPATRA specification language [1] for specification, verification and simulation of real-time systems. This language is based on the Time-constrained Reactive Automaton (TRA) formalism that, in turn, is derived from the IOTA model. An advantage of this language is that it provides executable specifications so they can be automatically converted into a simulator for the specified system [9].

In the area of telecommunications, SDL is one of the most advanced formal specification languages. SDL (Specification and Description Language) is a standard language for the specification and description of systems. It has been developed and standardised by the ITU (former CCITT) [10]. A system is represented in SDL as a hierarchy of blocks communicating via channels that carry signals. Leaf blocks are partitioned into processes connected inside a block using signal routes. Each process in SDL implements an *Extended Finite State Machine* (EFSM). EFSM is an FSM that can use and manipulate data stored in variables local to the machine, while the traditional FSM can only change its state. The EFSM reacts on signals arriving at the process via an input port, which is a FIFO queue. The time is expressed in SDL by means of *timer* constructs. A timer is a special object that is able to generate signals at specified moments of time and put those signals into the corresponding signal queues to processes. Figure 1 shows an example of an SDL description.

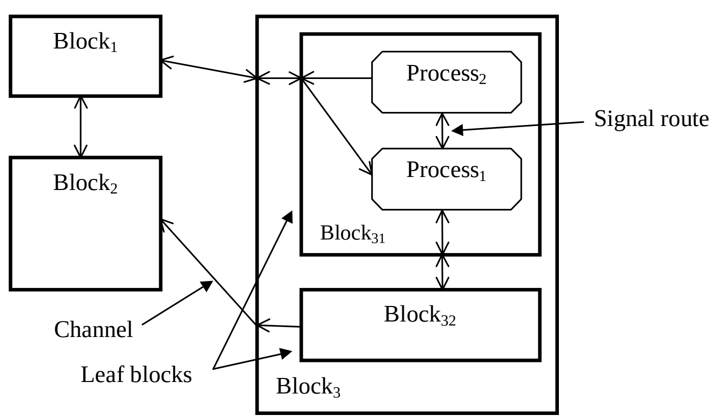

Fig. 1. Example of an SDL description of a system

2.2 Process Algebra Models

Several attempts have been made to extend traditional process-algebra models to represent time issues in the embedded systems [11,12]. Most of these approaches are based on the concept of Communicating Sequential Processes. *Communicating Sequential Processes* (CSP) were defined by Hoare in [11] and the influence of this concept was really important for developing parallel programming languages, such as Occam and Ada [13]. A number of methods were proposed to specify temporal constraints in concurrent systems, such as the *Timed Acceptances Model* [12] by Gerber and Lee. Latter the same authors proposed the *Communicating Shared Resources* (CSR) model [12].

2.3 Logic-Based Models

Temporal logic [13] is an extension of the classical logic. Unlike the classical logic, temporal logic specifications are able to relate properties with state sequences associated with a concurrent program execution. This gives us an ability to assert both the *safety* and the *liveness* properties, the two types of correctness properties used to study concurrent programs [22].

Therefore, temporal logic proves to be fairly appropriate for description of the temporal properties of systems. Pnueli and Manna [14] proposed the use of the temporal logic formalism for the behavioural specification of concurrent systems. They described a time hierarchy of specifications relating the occurrence of time in formulae of a system to the expressive power of that system.

2.4 Petri-Nets Models

A number of subclasses of Petri net models have been derived [13] including *state machines*, *marked graphs*, *free-choice nets*, *inhibitor-arc nets* to name a few. We are particularly interested in *timed* Petri nets, as they allow for introducing of time into the Petri nets. Several models for timed Petri nets were proposed. The early attempts associated delays with transitions [15]. However, since this violated the instantaneous nature of transitions, these approaches were replaced later by associating delays with places rather than with transitions. Razouk [16] proposed the use of both enabling and firing times. The enabling time of a transition is the time during which it must remain enabled before it may fire. The firing time of a transition is the length of time it takes for the tokens to be removed from the input places and to be deposited into the output places. A complete example of a Petri-net based design can be found in [17].

In the next section, we discuss the problem of mapping between high-level specifications and low-level design and implementations on particular hardware and software architectures for different formal models described in this section.

3 From Formal Specification to Design and Implementation

The *problem of correct mapping* from a high level specification, employing an abstract model of the system, to a particular software and hardware architecture is one of the key aspects of the design of embedded systems. This process is controlled at the following two levels (see Figure 2):

- *Formal verification* is the process of checking that the behaviour of a system described by a formal specification satisfies a given property. Formal verification is also used to check a formal specification itself to ensure that it does not describe a system impossible in terms of the underlying formal model.
- *Implementation verification* is the process of checking that the behaviour of a relatively low-level model correctly represents the corresponding high-level formal specification. The low-level model in this definition includes unambiguous descriptions of the system software and hardware components.

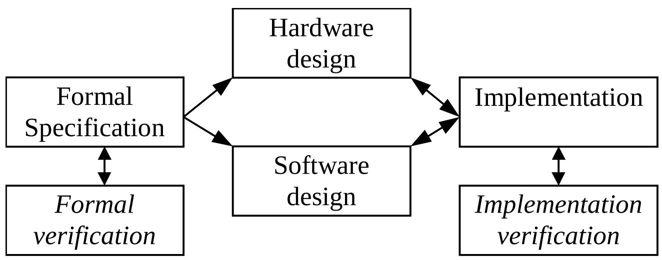

Fig. 2. Development of embedded systems using formal models

While the models described in Section 2 give powerful tools for formal verification of different properties of real-time systems, most of them are weak in providing a technique for translation of the high-level specification into structures suitable for implementation. Moreover, some methods, such as logic-based approaches are just unable to describe the system structure and casual interrelations required at lower levels of details. At the same time, non-strict translation between different levels of abstraction during system design is the potential reason why some of the system properties preserved in the source specification could be violated in the succeeding system design. Hence, further undesired iterations of the implementation verification followed by re-design could be required to obtain the correct design (see Figure 2).

Intuitively, we can note here that among the formal models discussed so far those that operate with more realistic objects seem the most suitable for translation into implementations. Namely, the state-based models are easier to implement just because the state machine or automaton formalisms stay very close to the actual structures and modes of operation employed by real-life computer systems. Whereas the logic-based approaches are significantly more difficult for implementation as the underlying temporal logic formalism does not assume any structured elements in the specified system. Therefore, we concentrate our analysis on implementation possibilities of the state-based models.

Not all specified properties of a system are equally hard to preserve in an implementation. In particular, the *casual relations*, which are expressed in state models by transition and output functions, can be implemented in a relatively strait-forward manner. On the other hand, *responsiveness properties*, which actually distinguish the real-time systems from other applications, require detailed knowledge of the target hardware architecture as well timing characteristics of implemented software components that are unavailable at the specification stage. As mentioned in [3], one way to resolve this issue is to impose restrictions on the target architecture in order to make the mapping problem manageable. In this case specifications operate with limited number of abstract objects whose timing characteristics and implementations for the selected target platform are known in advance. Another method assumes the use of non-strict timing constraints, i.e. the specification itself allows the system to *miss certain deadlines* or to perform a scheduled action with some latency. Obviously, the last technique is suitable only for *soft real-time* systems [3], where missing a deadline just degrades the system performance but does not cause a catastrophic failure. Generally speaking, the more strict timing constraints can be specified for a model, the more informal the way for implementation of this model is. Therefore, *hard real-time* systems, which require all specified deadlines to be met, are the most difficult to implement.

Consider the TRA model [1] as an example. For a channel and an action to fire on that channel, timing constraints are specified by a pair of a condition (a set of automaton states), enabling the action, and a time interval, within which this must happen after the action is enabled. Thus, we can specify the timed behaviour of the system with any desired precision. However, it is not clear from the specification, if it can be implemented on a given hardware platform and how this implementation should be done, if one is possible.

SDL [10] is an example of a formal model that is very suitable for description of casual interrelations and system behaviour. However, SDL is weak in specification of strict timing constraints on system responses. Fortunately, in the field of telecommunications soft real-time systems prevail over hard real-time systems that allows for the broad use of non-strict specification techniques in this area. That is why we have chosen SDL as the most appropriate candidate for specification of an embedded data communication system described in Section 5.

The next section explains how timing constraints of a real-time system can be specified in SDL and why SDL does not provide for strict timing specifications.

4 Timing Constraints in SDL

A general form of a timing constraint for a state-based model can be represented by the following tuple: (S, I, z), where S is a set of enabling internal states, I is the time interval (open, half-open, or closed) within which the response z should be generated if the enabling conditions are still in force. The bounds of time interval I can be specified either in absolute time or in time relative to the moment the response became enabled. As a rule, the second interval bounding method is used.

Let S be a set of states of an SDL process P. The timing constraints for this process can be represented by a set C of the following tuples $c_i = (S_{Ei}, I_{Ei}, z_i)$, where $S_{Ei} \in 2^S$ is the set of enabling states of P, $I_{Ei}=[NOW+t_{i1}, NOW+t_{i2}]$, $t_{i2} \geq t_{i1}$ is the time interval, z_i is an output signal of P, NOW is the moment of time the process enters one of states $s \in S_{Ei}$. To specify the timing constraint, we define a timer T_i for each element of the set C and an internal variable $FIRED_i$: *"DCL FIRED*$_i$ *Boolean := true"*. Figure 3 illustrates how the process diagram for P should be modified to handle the defined timing constraint.

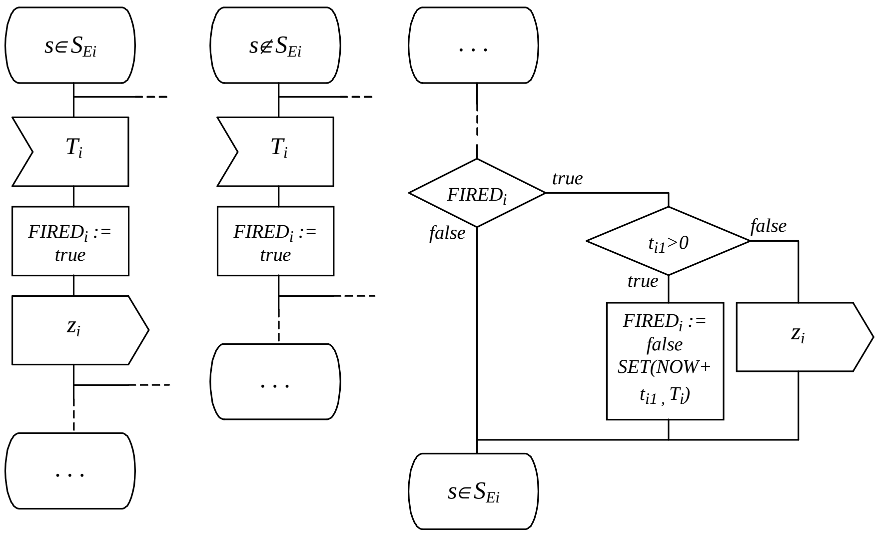

Fig. 3. Adding timing constraints in SDL. An additional branch is added in each state to process the incoming signal from the corresponding timer T_i (Figure 1a and 1b). For each transition entering one of the states $s \in S_{Ei}$ a special construct is added at the end of the transition as it is shown in Figure 1c.

A positive feature of the proposed method for specification of timing constraints in SDL is that it allows for more flexible definition of time intervals in comparison with other state models. In particular, we may specify different firing intervals for the same pair of the output action and the set of enabling states depending upon the conditions under which the system enters an enabling state. For instance, this may be done in conjunction with the input signal that stimulated the enabling transition or the contents of internal variables of the EFSM. Such a dependency is impossible for, say, the TRA formalism. The disadvantage of SDL specification is that it does not guarantee even formally that the defined intervals will be obeyed by the system, i.e. that the required response will indeed be generated within the specified interval. This is due to the nature of SDL itself, where the following non-deterministic components are present in any specification:

– Signals to one EFSM can be processed only sequentially, so a potentially unlimited delay is possible between the arrival of a signal and the system reaction on it, while the corresponding EFSM consumes all the preceding signals in the input queue to that process.
– Unspecified delays are possible between expiration of the timer and placing of the corresponding input signal into the process queue as well as between consumption of the signal by the process and generation of the output response.

These particularities explain why we do not use the t_{i2} parameter in our specifications: we just can not guarantee the upper interval bounds anyway. In [18], a method was proposed to overcome this problem by complementing SDL specifications with Temporal Logic formulae. However, although this approach allows for improvement of the timing strictness, it renders the mapping to an implementation almost as difficult as that is for logic-based models.

5 SDL-Based Implementation

We consider details of the implementation process for a leaf SDL block (see Figure 1) assuming that it is executed on a single CPU. In fact, this represents a uniprocessor multi-task software system. It receives input signals from the environment, i.e., from peripheral devices and responds with corresponding outputs via the CPU output ports. This is a very typical task arising during design of complex embedded systems where each embedded component may be viewed as a separate computing system with specified timing and performance requirements interconnected with others via analogues of SDL inter-block channels. Our case study is based on the design and implementation of an embedded modem concentrator [20].

The system consists of a high-performance RISC CPU and an extendable DSP cluster. In the current version, the Motorola MPC860 CPU is used as the central processing device, while the DSP cluster consists of four Motorola DSP56303 chips. Each DSP implements up to two analogue V.34 modems. The software in the main CPU provides an interface to an Ethernet 10BASE-T link and routing between modems and the Ethernet channel. Besides, it can implement the error correction and data compression procedures according to the V.42/V.42bis standards. Serial ports of DSPs transmit samples that encode analogue telephone line signals. Outputs of all DSP are connected via a common time-divided T1 bus to a Codec. The Codec provides the interface between the T1 bus and 2-wire telephone lines. The diagram in Figure 4 depicts the system structure.

Logically, the system is a TCP gateway. Incoming TCP connections from the Ethernet channel are mapped into V.42/V.42bis connections on the modem ports. The mapping is one to one between a TCP port and a modem port on a DSP. A connection is established on either a TCP connection request or a modem connection request, i.e., carrier detection on the telephone line. All data coming through a TCP connection is fed into a V.42/V.42bis byte stream. By analogy, all data coming from a modem port is forwarded through the corresponding TCP connection. The TCP retransmission and error correction mechanisms are not used over the modem connection, where only a pure V.42/V.42bis byte stream is transmitted. The retransmissions exist either between the TCP port and the peer user, or between the modem port and the peer modem, but not between the TCP port and the peer modem. A TCP connection shutdown causes release of the corresponding modem connection and vice versa. Each pair of a TCP port and a modem forms a *logical channel* of the concentrator.

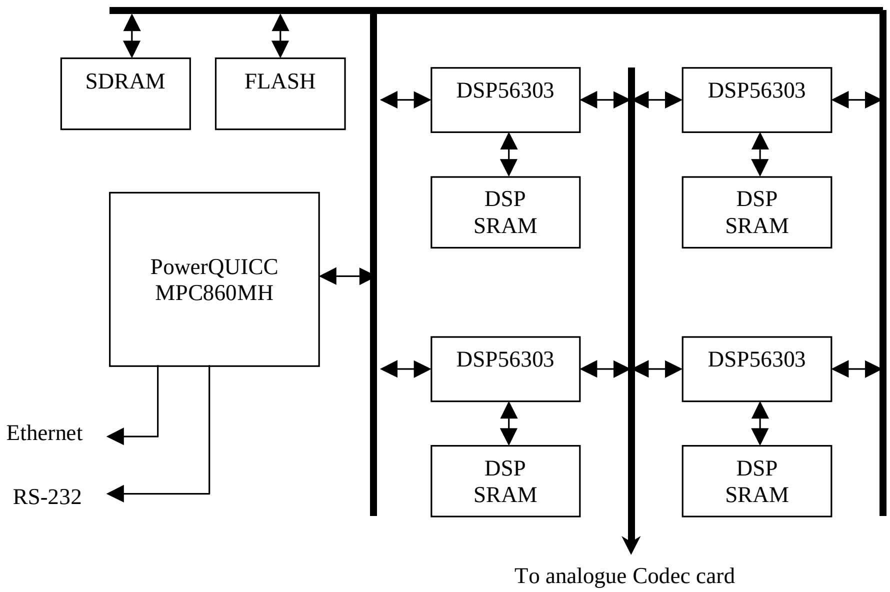

Fig. 4. Structure of the modem concentrator.

We concentrate on the software design for the MPC860. The external environment, i.e. possible sources of requests to the system, consists of the Ethernet controller and the embedded DSP-based modems. Each of these channels produces requests to the software of the following two types: a data block incoming from the channel (*data request*) or an indication that the channel is ready to accept further outgoing data (*indication request*). The data block in this subdivision is not necessarily a block of network data, e.g., a network packet. It may also be a block with control information or a device status report, etc. The real-time restrictions for the system are given by the following two constraints:

– Data requests from channels should be accepted with a delay not exceeding the one allowed by the channel hardware to avoid data loss.
– The software should generate appropriate responses on high-level protocol requests (e.g., V.42 or TCP/IP) within the timeout periods specified by those protocols to keep normal mode of data exchange.

Low-level hardware services as well as the task and memory management in the software are provided by a real-time operating system (RTOS). In our case, this was the WindRiver VxWorks 5.3 [23] RTOS. This RTOS employs common address space model and preemptive fixed priority scheduling.

5.1 SDL Model

We dedicate a separate SDL process for each channel controller in the system. As several same modem channels exist in the system and the only Ethernet channel, we distinguish two types of processes. Modem processes (type *"MdmFSM"*) handle requests from the corresponding modems. In fact, these processes implement the V.42/V.42bis processing and control connection establishment and disconnection on the logical channel of the concentrator. The Ethernet process (type *"EthFSM"*) manages TCP/IP connections via the Ethernet link. Besides, the Ethernet process and the modem ones communicate to each other to exchange data and to provide synchronous handling of the connection status on both sides of a logical channel. This structure is depicted in Figure 5.

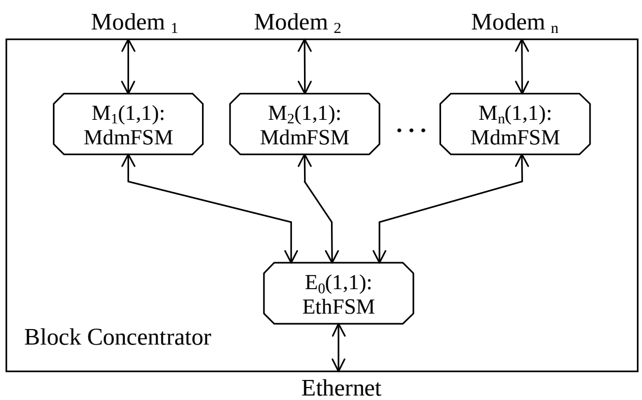

Fig. 5. SDL block diagram.

5.2 Mapping to the RTOS Objects

The following four questions should be answered in order to step from the formal definition to an implementation:

- The mapping between the OS tasks and the SDL processes.
- The order in which transitions in different processes occur, i.e. the CPU scheduling policy for different SDL processes.

– The way for intercommunication with the external environment (i.e., with the channel controllers in our case).
– The method for implementation of the SDL timers mechanism.

All possible requirements to the design of the concentrator can be subdivided into the following three groups: *functional* requirements, *performance* requirements, and *exploitation* requirements. Functional requirements determine how close the design proposed reflects the set of functions the system should perform, i.e. how well the system obeys its specification including *timing constraints*. Performance determines how well a system does what it is intended to do [19]. Consequently, performance requirements establish bounds to the performance indices for the system. As shown in [20], in the case of an IP concentrator this is equivalent to the maximisation of the number of modem channels that the main CPU (MPC860) can support. Exploitation characteristics determine how useful the system is. The point is that even for a system with outstanding performance, it can be completely useless, if, for instance, it crashes every five minutes or if it costs a fortune. However, as we are mainly concentrated on the software part of the system, we may assume that exploitation requirements will be satisfied by correct implementation of the specification.

One may see that, among these requirements, specified timing constraints and performance are the most difficult ones to satisfy. From the design point of view, these problems can be transformed into the following two criteria:

– Reduction of the extra delay between the arrival of a signal at the system and its processing with generation of corresponding responses.
– Increasing of the number of signals (events) the system can handle within the same time interval with respect to the previous criterion.

Therefore, we should reduce the CPU overhead (i.e., the time the CPU spends for anything but event processing) and propose the event handling scheduling minimising the extra delay in processing. We do not consider here reduction of the processing time for single events, as this is a very individual requirement for a particular system.

The architecture proposed is a single-task event processing system with a common OS queue for all existing SDL processes (see Figure 6). Events for all EFSM are put into the main queue and a single RTOS task is pending on this queue, fetches the events, changes states of the corresponding EFSMs and generates appropriate output signals. Each EFSM in the system is represented by a data structure accessible from the control task and containing the state of the EFSM and current values of internal variables. The only exception from this structure is the Ethernet process that is implemented as a separate OS task. The reason for this is that the Ethernet and TCP/IP support was built-in into the RTOS, so its re-use allowed for reduction of overall development efforts. There are two major *advantages* in this architecture. Firstly, it is guaranteed that events are processed at least in exactly the same order they arrive at the system. Secondly, this architecture provides the minimal context-switching overhead. The obvious disadvantage of this structure is that it cannot distinguish importance (priority) of different EFSMs since event for all EFSMs are put into the same

OS queue (see Figure 6). Although it is not a big problem in our homogenous system with a number of same logical channels of equal importance, this can be an issue for other cases (e.g., concentrators with a Quality of Service support). *To overcome* this problem, we propose the use of a priority queue instead of a usual one. In the priority queue, signals for EFSMs with more critical timing requirements can be placed into a sub-queue with a higher priority. The second disadvantage is that it is unclear whether the system provides the minimal delay in processing of incoming SDL events. It is easy to show that the average extra delay in the signal processing is minimal when among a set of input signals belonging to different SDL processes we always select the one requiring the minimal processing time. However, this would require knowledge of the processing time for every signal in advance that is not always available. In addition, we might need to limit the number of signals belonging to the same EFSM that can be processed one after the other to avoid the CPU monopolising.

Nevertheless, these disadvantages of the proposed architecture are fully compensated by the possibility to apply a systematic approach to performance optimisation of the system as described in Section 5.3.

Consider how the signals are transmitted within the system. Events are passed in a common data structure named *message block*. Such a message contains a header with an identifier of the destination SDL process (in our case, this is just the number of the logical channel), type of the signal, and additional parameters if any. Network data is transferred in the same blocks as special signals, in which the message header is followed by the data. DSPs communicate with the MPC860 using the interrupt mechanism. Each DSP is accompanied by a software interrupt handler that receives requests from the DSP and converts them into corresponding messages to the main processing task. By analogy, output messages (data or connection controls) from the main task are put into an output signals list and are sent to the hardware upon coming of the indication requests. Interrupt handlers can pre-empt other OS tasks, so requests from hardware are accepted by the system as soon as possible. Communication with the RTOS Ethernet and TCP/IP stack is performed via intermediate OS tasks – *TCP readers* and *TCP writers*. The TCP readers convert OS Application Programming Interface (API) responses into input messages for the main task, while the TCP writers receive output signals from the main task and convert them into the corresponding API calls. The use of additional processes for communication with the OS kernel allows us to employ blocking input/output operations and, hence, reduce the CPU overhead. The described task structure is illustrated in Figure 6.

SDL timers are implemented in the system on the basis of the MPC860 *Periodic Interrupt Timer* (*PIT*). PIT can generate special timer interrupts at a given frequency. Consequently, we have a timer interrupt handler processing these requests and generating the timer expiration signals for other SDL processes. As we may need to support more than one timer in the system, a table of timers is created with the following information about each of them:

- The flag, telling if the timer is used.
- The number of hardware ticks before expiration.
- The identifier of the EFSM engaged the timer.

– The pointer onto the timer signal processing procedure.

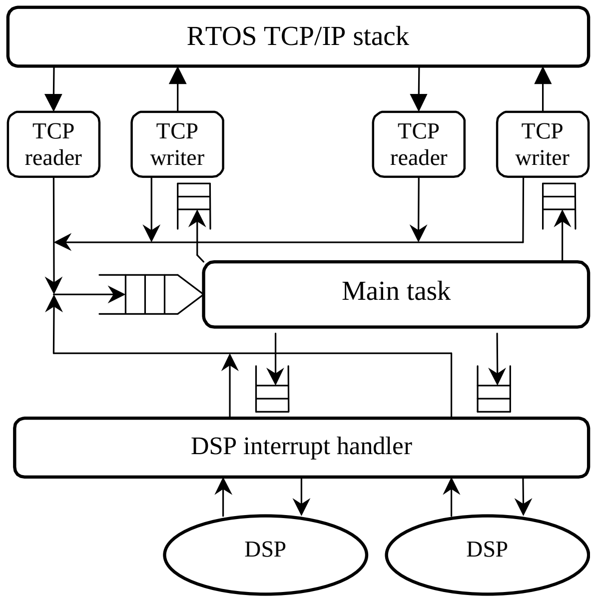

Fig. 6. The task structure.

After each PIT tick, the timer interrupt handler reduces the time before expiration of every engaged timer in the table. For those who expired, the handler generates the timer expiration signals and places them into the main queue. The pointer to the timer signal processing procedure in the timer table allows for avoiding of unwieldy case constructs in the body of the main task that analyse the timer signals. Instead, the main task just calls the corresponding processing subroutine by its pointer and passes the number of the logical channel (EFSM) as a parameter. Besides, as actually the main queue contains pointers to the corresponding message blocks but not the blocks themselves, we can put the same pointer into the queue more than ones. Thus, the only message block is allocated per each timer that eliminates the need in allocation and release of a new signalling message block for each timer's expiration.

For further reduction of the CPU overhead caused by memory allocation for data blocks during transfer, we use *pre-allocated data buffers*. Message blocks are allocated separately for events with transferred data (e.g., "Incoming data from the TCP port", or "Incoming data from the modem port") and for events that do not require the extra memory space to carry data. This also allows us to save memory resources of the system. The allocation is performed during initialisation of the software. Filled and empty blocks circulate in corresponding data buffer cycles – one for each pair of communicating tasks, as it is shown in Figure 7.

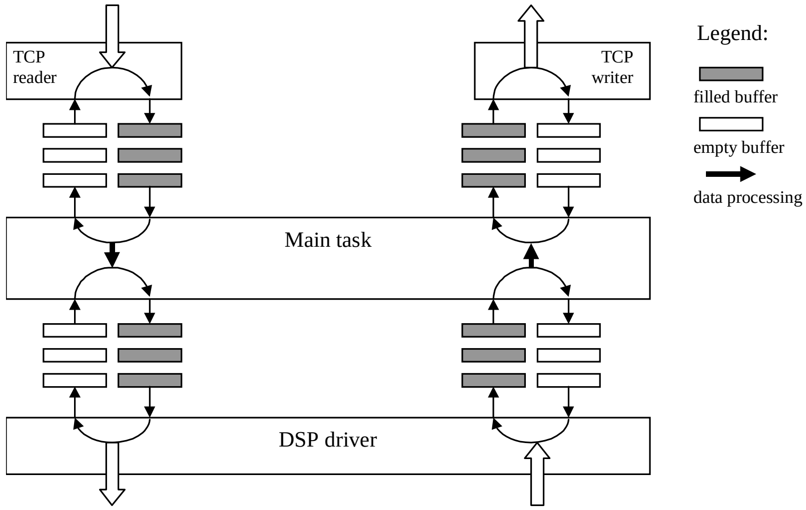

Fig. 7. Data flow through the system.

5.3 Performance Optimisation

Additional advantage of the proposed design is that is allows for systematic performance optimisation of the system. Namely, we can easily emphasise the following mainstream directions for optimisation: reduction of the CPU overhead and increase of the event processing speed. Computations associated with event processing are, usually, individual for every particular system, thus, we mainly concentrate on the overhead minimisation as more widely applicable approach. Considering the overhead as the time the CPU spends for anything but event processing, we can impose the two major criteria for the optimisation: the number of context switches and the number of OS calls per amount of useful work performed by the system. For the modem concentrator, useful work is equivalent to the amount of data transferred.

Let us assume that we have a pair of tasks, communicating via a message queue or a shared memory location guarded by a mutual exclusion semaphore. If one of the tasks is of a higher priority than the other one, the number of possible context switches between these tasks can not be less than the number of requests (e.g., message blocks) transferred to the high priority task. For the case of equal task priorities, this transfer can be performed using only one context switch. Actually, the number of context switches for equal priority tasks depends on the intensity, at which the source task emits requests. The greater the intensity is, the smaller number of context switches occurs. Thus, the equal priority tasks are preferred as they give at least the same context switching frequency for low system workload and they have the better behaviour for a heavily loaded system. With no further time constraints, a task in an equal priority group could occupy the CPU for a prohibitively long duration so that the other tasks in the same group would be unable to react on requests to them in time. We enable the *round-robin* scheduling to overcome the problem. An important question is the size of the time slice. On one hand, it should be large enough to allow a task to process more than one request within a single CPU quantum. On the other hand, the size of the time slice should be small to preserve the system responsiveness.

OS calls are mainly used to operate with RTOS objects, such as queues, mutexes, semaphores, etc. In our case, processing of every event requires a number of communicative acts between tasks that use the mentioned queues or mutexes. Thus, minimisation of the number of OS calls is achieved by reduction of the number of signals (events) generated in the system for the same amount of data transferred. We use the following two optimisations:

- We buffer the data in the message blocks before sending it from DSPs to the main task or from the main task to the TCP writers. This method reduces the number of messages with data passed between the corresponding tasks. As we may still need to transfer small pieces of data, special timers are used in the software to flush contents of the buffers in the DSP driver and in the main task upon expiration of a timeout interval. Thus, a disadvantage of the buffering is that it increases transition delay through the concentrator for separate small data packets.
- We pass the empty message blocks from the DSP driver to the main task and from the TCP writers to the main task not via the main queue but by placing them directly into appropriate blocks lists in the main task. Certainly, this is possible in our system only due to the common address space model employed by the RTOS.

6 Evaluation

In this section we evaluate effects of the optimisations proposed in our design. Accordingly to considerations in the previous section, the major advantage of the scheduling policy used is that it allows the system to keep the number of context switches low as the load grows. The system load can be assessed as the amount of useful work performed by the system within a time unit, where the useful work for the concentrator is the amount of data transferred though it. Therefore, the evaluation plan proposed comprises analysis of the dependency between the system load and the average number of context switches occurring per time unit. The system load is regulated by changing the number of logical channels that perform simultaneous duplex data transfers. The dependency is depicted in Figure 8. It can be clearly seen from the figure that the system demonstrates excellent behaviour, especially for heavy loads.

To evaluate the effect of the free blocks passing optimisation, we measure the system for different loads before and after the optimisation is applied. By analogy with the measurements in the previous section, we change the system load by performing different number of simultaneous duplex data transfers. The number of free buffers received by the main task is used as a performance index for comparison.

Results are presented in Figure 9, where we may notice significant reduction of the number of blocks passed after the optimisation is applied.

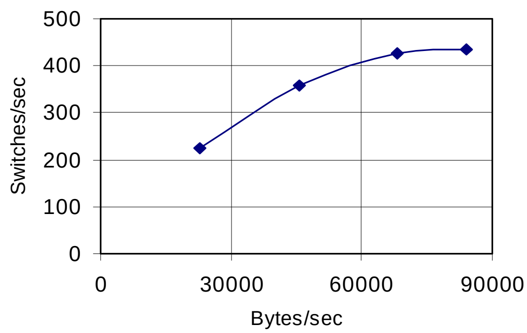

Fig. 8. Context switching in the system.

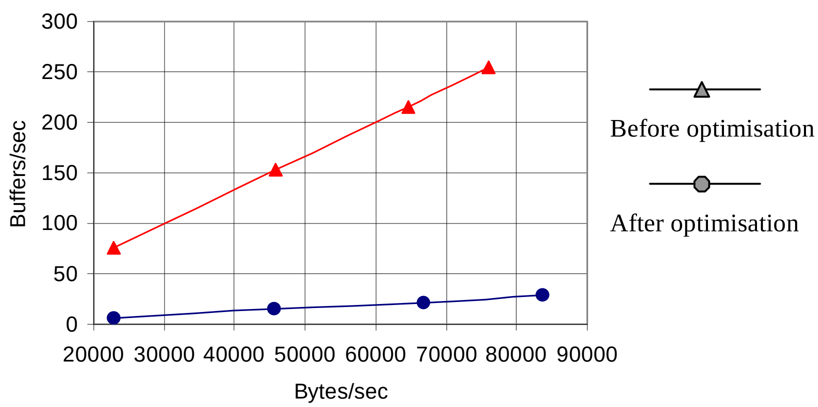

Fig. 9. Passing of free data blocks.

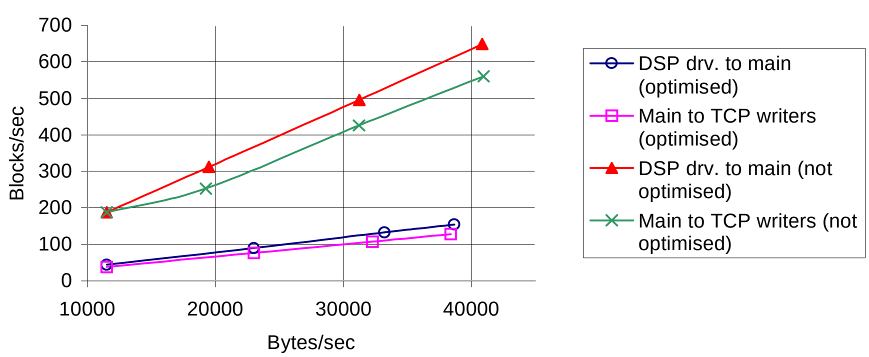

Fig. 10. Buffering effect.

The effect of buffering is assessed as a dependency between the number of data blocks passed to the main task or to the TCP writers per time unit and the load of the system. We expect that after optimisation, for the same system load we will receive a less number of data blocks passed for the same workload. Figure 10 demonstrates this dependency.

7 Conclusions

Embedded systems offer prospective solutions for building of networking products for the dial-up Internet (e.g., modem concentrators). They combine low production and exploitation costs with high performance and flexibility. However, design methods for such systems are often based on ad hoc approaches, which do not guarantee that the implemented device will meet all functional requirements imposed. Moreover, it is hard to assess potential performance capabilities of a final product based on a complex embedded hardware and software development platform. Nowadays, most of researchers advocate use of formal techniques for design and validation of embedded systems.

In this paper, we presented a brief survey of formal models for embedded system design. We also outlined the problem of mapping between a formal specification of system properties and description of the design at a level of details enough for implementation of the system on a particular hardware and software architecture. We demonstrated one of possible design approaches by providing an example of the use of SDL specifications and models for design of an embedded modem concentrator. The presented case study illustrates that for certain classes of applications it is possible to *waive the strictness* of specified constraints, if this allows for *systematic transformation* from a formal model of the system to a particular implementation. Moreover, such a systematic approach provides *additional benefits* for performance optimisation of the system as it was shown in our example.

From the research presented, the authors suppose that the proposed approach may help designers to improve the efficiency of developed systems and to reduce the number of iterations for obtaining a satisfactory design. An interesting and important direction for future work is to formalise and possibly automate the process of transformation from more strict formal specifications than SDL to correct implementations. As a starting point, we may propose the problem of automated checking if a given strict formal specification can be physically implemented on a particular hardware.

References

1. A. Bestavros. “Time-constrained Reactive Automata: Novel Development Methodology for Embedded Real-time Systems”. PhD Thesis, Harvard University, Cambridge, August 1991.

2. F. Shneider. “Critical issues in real-time systems: A position paper”. Technical report 88-914. Department of Computer Science, Cornell University, NY, May 1988.
3. S. Edwards et al. “Design of Embedded Systems: Formal Models, Validation, and Synthesis”. Proceedings of the IEEE. Vol. 85, No. 3, March 1997.
4. E.A. Lee and A. Sangiovanni-Vincetelli. “The tagged signal model – a preliminary version of a denotational framework for comparing models of computation”. Technical Report. Electronic Research Laboratory, University of California, US, May 1996.
5. D. Drinsky and D. Harel. “On the power of bounded concurrency. A Finite automata”. Journal of the ACM. Vol. 41, No. 3, May 1994.
6. B. Dasarathy. “Timing constraints of real-time systems: Control for expressing them, Method for validating them”. IEEE Transactions on Software Engineering. Vol. 11, No. 1, January 1985.
7. N. Lynch and M. Tuttle. “An introduction to Input/Output Automata”. Technical Report MIT/LCS/MT-373. MIT, Cambridge, December 1988.
8. A. Bestavros. “The IOTA: A model for real-time parallel computation”. In Proceedings of TAU’90: The 1990 ACM Workshop on Timing Issues in Specification and Synthesis of Digital Systems. Canada, August 1990.
9. A. Bestavros. “CLEOPATRA. Physically-Correct Specification of Embedded Real-Time Programs”. In Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems. US, June 1994.
10. “CCITT specification and description language (SDL)”. Recommendation Z.100. International Telecommunication Union, Geneva, March 1993.
11. C.A.R. Hoare. “Communicating Sequential Processes”. Prentice-Hall, Inc., US, 1985.
12. R. Gerber and I. Lee. “A proof system for communicating shared resources”. In Proceeding of the 11th IEEE Real-time Systems Symposium. 1990.
13. S.M. Shatz. “Development of distributed software: concepts and tools”. Macmillan Publishing Company, US, 1993.
14. Z. Manna and A. Pnueli. “The temporal logic of Reactive and Concurrent Systems: specification”. Springer-Verlag, 1992.
15. P.M. Merlin and D.J. Faber. “Recoverability of communication protocols: Implications of a theoretical study”. IEEE Transactions on Communication. Vol. COM-24, No. 9, September 1976.
16. R. Razouk. “The derivation of performance expressions for communication protocols from timed Petri net models”. Technical Report 211. Department of Information and Computer Science, University of California, CA, November 1983.
17. R. Esser. “An Object Oriented Petri Net Approach to Embedded Systems Design”. Dissertation. Swiss Federal Institute of Technology, Zurich, 1996.
18. S. Leue. “Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach”. In Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification PSTV'95. Chapmann & Hall, 1995.
19. L. Svobodova. “Computer performance measurement and evaluation methods: analysis and applications”. American Elsevier Publishing Company, Inc., US, 1976.
20. R. Khoussainov. “Design, Optimisation, and Performance Evaluation of an Embedded Network Concentrator”. MSc thesis. University College Dublin, Ireland, February 1999.
21. A. Gill. “Introduction to the Theory of Finite State Machines”. McGraw-Hill Book Company, Inc., US, 1962.
22. G. Andrews. “Concurrent programming: principles and practice”. Benjamin/Cummings Publishing Company, Inc., US, 1991.
23. “VxWorks Reference Manual, 5.3.1”. Edition 1, WindRiver Systems, Inc., 1997.