Asynchronous Signal Paradigm for Soft Real Time Systems

Peter Funk[1], Janet Wennersten[2]
[1] peter.funk@mdh.se Mälardalen University, Department of Computer Engineering,
S-721 23 Västerås, Sweden
[2]janet.wennersten@uab.ericsson.se, Ericsson UAB AB, 721 67 Älvsjö

Abstract

Decomposition into communicating asynchronous entities is often chosen as solution for both telecommunications systems and for real time AI systems [Dodhaiwala et al. 89]. This in contrast to the trend to increase the abstraction level of real time programming by introducing features and paradigms from main stream time sharing systems with an operating system managing garbage collection, process scheduling, etc. For large complex concurrent real time systems which are signal centered, a main stream specification and implementation approach makes it difficult to meet requirements (response time, reliability, etc.). It is also difficult to analyze the overall behavior of the system.

A real time language (called PLEX) combined with a real time operating system and processor, all based on an asynchronous signaling paradigm, has proven to be efficient [Hemdal 1998] for asynchronously communicating real time applications (large telecommunications systems with millions of lines of PLEX code are in operation). PLEX and Petri Nets show similarities and by specifying behavioral parts with Petri Nets [Jensen 1997] powerful analysis tools (liveness, deadlock, etc.) are available. In this paper we analyze the benefits emerging from combining PLEX with Petri Nets, enabling both an efficient implementation and analysis of behavior.

1 Introduction

Communication systems are in essence different to traditional computer systems. In a traditional computer system there are only a few tasks ongoing at the same time. The operating systems with advanced paging and timesharing handle execution and distributes limited resources such as processor time to different processes. Starting and stopping new processes is costly in terms of processor time and overhead for dividing resources amongst processes increases with the number of processes (overhead increases noticeable in most common operating systems which often halt if to many jobs are started). This is acceptable for most applications since the number of tasks/programs/processes are limited. In communication systems where the main task is to start and stop small tasks and who are dimensioned after maximum load, the worse case is thousands of ongoing tasks. If these tasks would be handled as individual processes as in main stream operating systems the increase in overhead makes it difficult to meet requirements on response time. Response time requirements for telecommunication systems are given in milliseconds.

2 Matching Paradigms and Architecture

It is not unusual to tailor processors for their languages and operating systems. Processors for PCs are tailored for traditional imperative programming languages requiring interrupt handling and quick swapping between a limited number of ongoing processes. Sun is developing a processor optimized for direct Java byte code execution. A prototype processor specialized for soft real-time requirements and functional languages [Tjärnström 1998] has been developed for the programming language Erlang [Armstrong, Virding, Wikström, Williams 1996]. Ericsson's APZ processors used in telephone exchanges and the PLEX language belongs to this category where the application, programming language and processor is closely matched and optimized. This gives advantages in terms of capacity and provides a programming paradigm that directly reflects the application domain and architecture. Both PLEX and APZ are based on a signal and component based paradigm, a paradigm that was originally created 20 years ago.

Sometimes concurrent real time programming languages on the market are created by adding real time features to traditional sequential languages. Mixing different paradigms in the same language may be confusing and may reduce quality and increase maintenance costs for systems and are claimed to have caused failure of systems [Hemdal 98]. The price to pay for such often *ad hook* tailoring and adaptation is that it may be very difficult to achieve acceptable or good efficiency in terms of execution time when languages, operating systems and processors are mixed from different paradigms and system architectures. For example running a concurrent asynchronous real time system on a processor optimized for sequential languages and a task oriented operating systems may be unacceptable slow compared with execution on an signal optimized processor (the opposite may also be unacceptable).

3 Efficient Implementation of Asynchronous Systems

Asynchronous hard real time systems is in essence very different from traditional batch oriented sequential systems hence main stream software development tools, methods and main stream programming languages may not always be a suitable choice. Such a choice may for some applications have a to high price in lost efficiency, increased instability and also cause unnecessary confusion by mixing different paradigms (mixing different paradigms is claimed to be one of the main reason for failure in developing some large real time systems [Hemdal 1998]). If signaling is the central concept in an application domain, then translating and implementing a system for this application in an object/process oriented paradigm where signaling is hidden inside object, or restricted to method calls, may not be an obvious choice and may deter the focus of what is important in the functionality, increasing the risk in producing errors (not proven in this research). Figure 1 shows two ways of implementing a concurrent real time system. The first (a) exemplifies a conservative approach where traditional design and coding is used and the result executed on a main stream processor. For some applications with high demands on response time this approach may be to inefficient as discussed in section 2. If the application has higher demands on response time and reliability the second approach is proposed (b). Petri Nets are shortly described in section 5 and the real time programming language PLEX together with the signaling paradigm is described in section 6.

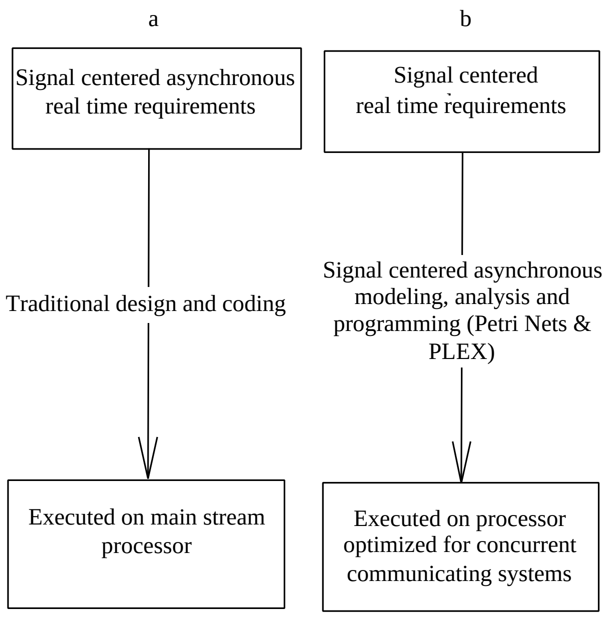
a
b
Signal centered asynchronous real time requirements
Signal centered real time requirements
Traditional design and coding
Signal centered asynchronous modeling, analysis and programming (Petri Nets & PLEX)
Executed on main stream processor
Executed on processor optimized for concurrent communicating systems

Figure 1: Design, implementation and execution of concurrent real time systems.

4 Exact Notations of Behavior

There are two main types of symbolic representations which both use symbolic expressions: sentential representation (natural language descriptions) and diagrammatic/graphical representations. The latter can explicitly capture topological and geometrical relationships which can only be captured indirectly in a textual representation [Larkin and Simon, 1987]. If a language is carefully designed it contains both a textual formalism and graphical formalism that are equivalent (users may choose symbolic or sentential notation and display/manipulate the behavior in either way). For the creative and exploratory phases of forming new knowledge, visualization is often important and the use of diagrams also aids knowledge elicitation and co-operation between those involved [Addis et al., 1993]. Earlier approaches using conventional state machines or state-diagrams encountered difficulties when applied to system design, due to the exponential explosion in the number of states [Harel 87], and were claimed to be hard to read, modify and refine and not suitable for complex specifications [Martin, McClure, 85]. Different approaches to overcome these problems have been explored and graphical languages (often combined with a textual language) are common in system development today; for example:

?? SDL (Specification and Description Language, standardized by the International Telecommunications Union, [ITU-Z100]). The SDL language contains both a graphical and textual part. The graphical part is similar to flow charts. The graphical parts together with the textual part of the language enable the user to describe the functionality in such great detail that executable code can be generated directly. Some formalization efforts have been undertaken, see for example [Leue 1995]. With minor alterations in the semantics, a subset of SDL can be translated to Petri Nets which has been used for protocol verification at Siemens Telecommunication, Germany [Regensburger, Barnard 1998].

?? Statecharts [Harel et. al. 1990]. A graphical notation designed to make it easier to design and implement real time systems. Similar to SDL, it has a graphical part and a textual part and detailed descriptions can be created and used to generate executable code.

?? Process Transition Networks (PTNs) [Malec 1992], [Sandewall 1990]. PTNs can be translated to temporal logic and to a subset of Petri Nets. The notation aids conceptualization and knowledge acquisition and its simplicity makes it easy to use for domains in which the expressiveness is sufficient.

?? Use-Cases [Jacobson et al, 1993]. Not a notation in itself, but which allows different notations or even text documents describing specific examples of how the system to be designed will behave. Formalization and graphical syntax is under development [Regnell et al., 1995].

?? MSC (Message Sequence Charts describing signaling between objects in a distributed system) [ITU-Z120]. A widely used graphical trace language for communicating entities. MSCs may also be used for requirements specifications with a set of suitable tools [Ben-Abdallah and Leue 1996].

?? Petri Net notations [Jensen 1992] are a graphical notation enabling behavioral analysis and model checking. The notations are often used to translate a language tailored for a specific domain or application to/from Petri Nets, thereby giving access to analysis tools available for Petri Nets. For example some parts of SDL (with slightly altered semantics) can be translated to Petri Nets in order to enable model checking [Grahlmann 1998]. Petri Nets are emerging as a common formal notation for representing asynchronous real time tasks.

All of these notations introduce restrictions on what can be expressed in Petri nets.

5 Petri Nets

Petri Nets are used as a powerful formal notation for communicating automata and are expressive enough to capture systems where concurrency is necessary. Petri Nets developed by C. A. Petri in the sixties were the first general theory for discrete parallel systems. Petri Nets have proven to be well suited to describe both synchronous and a synchronous communication. A wide variety of Petri Net notations exist which either extend the expressiveness to new classes of problems or make them easier to use. Examples of extensions are high-level Petri Nets, timed Petri Nets, stochastic Petri Nets and Colored Petri (CPN) nets [Jensen 1997]. Petri Nets have always had a precise formal definition which enables the use of powerful analysis tools (e.g. SPIN [Holzmann, Peled 1994]) that can be used to prove different properties of Petri Nets. Also, there is on-going effort to standardize Petri Nets.

Petri Nets enable a wide variety of verification techniques such as model checking, verification and application of reduction algorithms [Grahlmann 1998]. Both SDL and MSCs have been translated into a subset of Petri Nets in order to use verification tools developed for Petri Nets.

Petri Nets are built with places, input transitions, output transitions, input arcs, output arcs and tokens [Jensen 1992]. Places can hold one or more tokens (in the example, there are two telephone tokens), arcs have the capacity to hold 1 or more tokens (the default being one), transitions have no capacity (cannot hold a token). A transition is enabled if the places with arcs leading to the transition have a number of tokens greater than or equal to the capacity of the arc (default capacity being one). During execution of a Petri net, the tokens will move around in the net and the number of tokens may vary. When using a Petri net, terms such as synchronization, concurrency and merging are difficult to avoid. The Petri net example in Figure 2 contains the primitive constructions: synchronization (e.g. the processes “ring tone a” and “ring signal b” are synchronized by starting the transition “dialing idle b”), concurrency (e.g. “ring tone a” and “ring signal b” are two concurrent processes started by the transition “dialing idle b”). In high level Petri Nets, a token can contain data (the local state of the token). Kurt Jensen states: “Making a CPN model is very similar to the construction of a program” [Jensen 1992]. This may be very useful when specifying and designing complex concurrent real time systems and the similarity with PLEX is summarized in section 7.

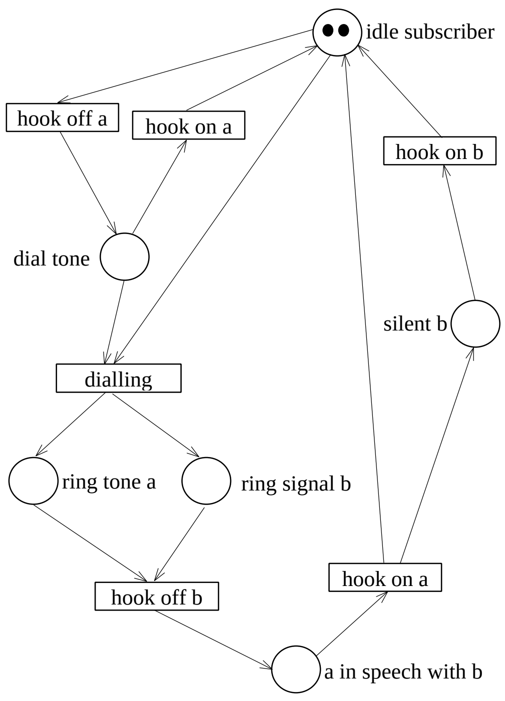

Figure 2: High level Petri net example

We explain below some advantages and drawbacks of Petri Nets compared with traditional state based process oriented approaches.

1. Petri Nets main benefit are their ability to express concurrency and Petri Nets are the first general formal notation for describing discrete parallel systems.
2. In research projects where Petri Nets have been used it is common to simplify or adapt them according to the task and the users' needs. A notation is either defined in terms of Petri Nets or internally translated to Petri Nets. These notations often contain restrictions and simplifications reducing the expressiveness and complexity of the notation compared with a direct use of Petri Nets. Examples where such notations have been used and defined in terms of Petri Nets or internally translated to Petri Nets are PTNs [Malec 1992], SDL and MSC [Grahlman 1998] and structured analysis and design diagrams (SADT diagrams) [Jensen 1997].
3. In Colored Petri Nets each token has a color, where the color may represent a specific individual together with some additional data (a local or global state). A transition can only occur if the tokens with the correct colors are available.

6 PLEX and the Signaling Paradigm

PLEX is a programming language designed for very demanding asynchronous real time applications such as telecommunication systems. Millions of lines of PLEX code implement the functionality of telephone services, features, maintenance, security and billing functionality, proving the languages efficiency both from the point of programming and maintenance (for telecommunication system maintenance is dominating). These systems are allowed to have an average down time on a few minutes a year putting high demands on the code. PLEX is also from the beginning designed to enable functionality change during execution and code may be replaced and upgraded without disturbance in availability. The signaling paradigm on which the language is based puts signals in the center. In contrast to traditional process centered languages PLEX is purely reactive and the only means for starting execution of PLEX code is an external or internal signal (the first signal would be an initialization signal mainly to execute some code initializing data). Every execution is restricted to about 500 lines of code which are executed as an atomic entity[1], eliminating the need for time sharing between programs and reducing demands on the operating system (these *code snippets* triggered by signals carrying data bear similarities to Petri Nets which will be explored in Section 7). It is believed that the limit on *code snippets* contribute to the maintainability since functionality of the system needs to be broken down to small, in it self easy to understand asynchronous communicating pieces. The basic task of the operating system is to buffer signals and once a signal is in front of the queue, the corresponding *code snippet* (we avoid the word *job* since this may give the wrong associations) is executed. This characteristic is one of the reasons that makes systems implemented in PLEX behaving linearly in time under load [Johansson et al. 95], a feature important for security sensitive systems. Parallel execution is performed in central and regional processors (no shared data allowed). The APZ processor family is specially designed and optimized for systems based on the signaling architecture and enable a very fast execution compared with traditional processors. A block structure is used to organize *code snippets* in packages that perform related tasks and share data (and hardware). Figure 3 shows an example of a system in PLEX. There are three code snippets, *c1*, *c2*, *c3*, data carried by or indexed by arguments to the signals, *d1*, *d2*, *d3*, two external signals, *hook off*, *dial tone* and five internal signals, *signal i2*, *signal i4*, *signal i5*, *signal i6* and finally the data after the code has been executed symbolized by d1', d2' (d3 has no prim sign which shows that data has not been altered in this code snippet). The cross at the end of a code

[1] Nothing is allowed to interrupt the execution until the code snippet is completed. There is no need for operating systems with interrupts, process handling mechanisms, garbage collection, etc. These mechanisms are costly in terms of time and number of processor instructions. For Petri Nets and PLEX the lack of necessity of these is one of their strengths when used in real time applications.

snippet line symbolizes the end of execution of this code snippet.

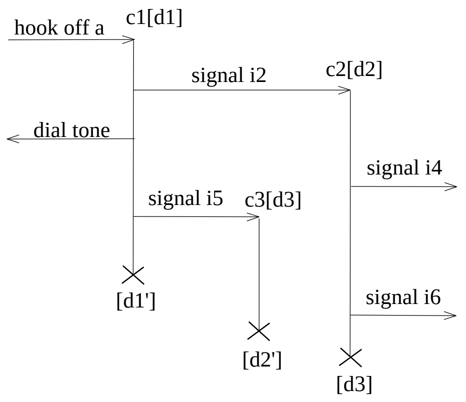

Figure 3: System example in PLEX with both internal and external signal.

It is important to note that the execution in the signal paradigm is purely reactive and in essence different from execution in a process oriented programming paradigm, where processes are spawned, forked, stopped, interrupted and resumed by some operating system. Execution of every code snippet is atomic and code snippets may be executed in parallel if no data is shared (the common way to implement it in the final system is that code divided into different blocks are executed on different processors).

7 Comparison of PLEX and Petri Nets

If all the features of PLEX and Petri Nets could be combined this would result in a very powerful combination. It would enable advanced analysis of systems produced, proving liveness, absent of deadlock, behavior under load, etc. PLEX enables system implementation meeting hard real time requirements, change of functionality without a full stop in service availability. These are all requirements important in global telecommunication systems and are rarely achieved in currently available systems. Also a suitable abstraction level can be provided for the PLEX language within the signal paradigm.

Our initial comparison shows that signals and data in PLEX have a close similarity with signals and tokens in Petri Nets. Signals in Petri Nets trigger a transition if the required tokens (data) are available. In PLEX a signal triggers a code snippet that can be seen as a Petri Net transition (Petri Net transitions may be expanded if a higher level of details is required for more fine grained analysis). Also a layered approach is considered where a code snippet may be represented by a Petri net, and a Petri net may be an abstraction of many code snippets. Since code snippets in PLEX are seen as atomic (within the same level, 4 levels exist) in the sense that nothing can interrupt their execution, there is no need for abstraction on this level and a direct mapping is possible. If time delay is a factor to be included in the analysis, timed Petri Nets may be considered, but this is beyond the scope of this paper. Tokens in Colored Petri Nets contain data describing a current local state.

8 Conclusions and Further Work

The comparison of Petri Nets and PLEX show promising similarities enabling a combination. Petri Nets gain from the fact that PLEX enables very efficient implementation of functionality expressed in Petri Nets. PLEX benefits from the availability of analysis tools developed for Petri Nets. The signaling paradigm and tokens in Petri Nets have the same meaning (a deeper semantic analysis is ongoing). This allows advanced analysis of the structure of systems written in PLEX, analysis which would be very difficult performed directly on code level. The next step in this research will be to define the signaling paradigm and PLEX in Petri Nets (a bi-directional function). This enables us to analyze different aspects and prove properties for code written in PLEX. Furthermore giving parts of PLEX a clear semantic in Petri Nets provides a foundation for automated translation between other notations such as MSCs and parts of SDL (or exactly where the expressiveness of the target formalism is less) which already have been defined in terms of Petri Nets. Also an Case-Based Reasoning approach to reuse as explored in [Funk and Robertson 1995] may be used to identify similarly behaving PLEX code.

References

[Addis, Gooding, Townsend, 1993] Addis T.R., Gooding D.C., Townsend J.J. (1993). K*nowledge Acquisition with Visual Functional Programming*. Knowledge Acquisition for Knowledge Based Systems, 7th European Workshop, EKAW ‘93, Lecture Notes in AI 723, Springer Verlag, pp 379-406.

[Allen 83] Allen J.F. 1983. *Maintaining Knowledge about Temporal Intervals.* Communication of the ACM, November, vol 26, Nr 11, pp 832-843.

[Armstrong, Virding, Wikström, Williams 1996] J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Programming in ERLANG*, Prentice Hall, 1996.

[Ben-Abdallah and Leue 1996] Ben-Abdallah H., Leue S. (1996). *Architecture of a Requirements and Design Tool Based on Message Sequence Charts.* Technical Report 96-13, University of Waterloo, pp 1-19.

[Funk and Robertson, 1995]. Funk, P.J. and Robertson D. Case-Based Selection of Requirements

Specifications for Telecommunications Systems. *Second European Workshop on Case-Based Reasoning, Proceedings*, Keane M., Haton J. P., Manago, M. (eds.), Chantilly, France, pp 293-301.

[Grahlman 1998] Grahlmann B. (1991). *Combining Finite Automata, Parallel Programs and SDL using Petri Nets.* TACAS'98, pp 1-16.

[Harel, 1987] Harel D. Statecharts: A Visual Formalism For Complex Systems, Science of Computer Programming 8, pp 231-274, Elsevier Science Publishers.

[Hemdal, 1998] G. Hemdal. The Software Crisis, Symptoms, Causes and Effects. RJO Advanced System Architecture Inc, Maryland, USA, pp1-22.

[Hirakawa, Monden, Yoshimoto, Tanaka, Ichikawa, 86]

[Holzmann, Peled 1994] Holzmann G.J., Peled D. (1994). *An Improvement in Formal Verification.* FORTE 1994 Conference, Switzerland. pp 1-12.

[ITU Z.100, 1994] ITU Z.100 Recommendation CCITT Z.100. CCITT Specification and Design Language (SDL). International Telecommunications Union, Geneva, Switzerland.

[ITU Z.120, 1994] ITU Z.120 Recommendation CCITT Z.120. CCITT Message Sequence Charts (MSC). International Telecommunications Union, Geneva, Switzerland.

[Jacobson, Christerson, Jonsson, Övergaard, 1993]

[Jensen 1992] K. Jensen, *Coloured Petri Nets*, vol 1, Springer -Verlag 1992.

[Jensen 1997] K. Jensen, *Coloured Petri Nets*, vol 3, Springer -Verlag 1997.

[Johansson et al. 95] Lars-Åke Johansson et al. *Systems Characteristics*, ERICSSON Report, UAB, pp 145-164.

[Kowalski, Sergot, 86] Kowalski R., Sergot M. (1986). *A Logic-based Calculus of Events.* New Generation Computing 4, Springer-Verla, pp 67-95.

[Malec 1992] Malec J. (1992).*Process Transition Networks: The Final Report.* Technical Report LiTH-IDA-R-92-07, Linköping University, pp 1-31.

[Mataga and Zave, 1993] P. Mataga and P. Zave. Formal Specifications of Telephone Features, pp 20-49.

[Regensburger, Barnard 1998] Regensburger F., Barnard A. (1998). *Formal verification of SDL systems at the Siemens mobile phone department.* TACAS'98, pp 439-455.

[Regnell, Kimbler, Wesslén, 1995]

[Robertson 1996] Robertson D. (1996). *Distributed Specifications.* ECAI 96, 12th European Conference on Artificial Intelligence, Budapest, Hungary, John Wiley & Sons Ltd, pp 390-394.

[Robertson and Agusti 1999] Robertson D., Agusti J. (1999). *Automated Reasoning in Conceptual Modelling*, draft book, available from the authors at DAI Edinburgh.

[Sandewall 1990] Sandewall E. (1990). *Proposal for a ProArt specification platform.* Technical Report LAIC-IDA-90-TR18, Linköping University.

[Tjärnström, 1998] R. Tjärnström. An Erlang Machine, IFIP WG2.8 Workshop, Portland, Oregon.

[Wieringa, 1996] R.J. Wieringa. *Requirements engineering: Framework for understanding*, John Wiley & Sons Ltd, Chichester.

[Zave and Jackson, 1996] P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. ACM pp 1-34.