Enhancing the Adoption of Formal Methods to Design Real-Time Systems

Victor Adrian Braberman, Miguel Felder, Fabio Javier Pieniazek
{vbraber, felder, fpienia}@dc.uba.ar

Dep. de Computación - FCEyN
Universidad de Buenos Aires
Buenos Aires, Argentina

Formal methods are being increasingly used in engineering industrial software. They are mostly used for specifying and verifying software requirements, but seldom in later development phases. This paper tries to bridge the gap between formal requirements specification and final code by introducing a formally defined design notation. The proposed design notation extends structured analysis specification notations with constructs derived from POSIX real-time extensions. The design notation proposed in this article is formally defined. Also, an operational semantic is given by means of high-level timed Petri nets, and can be formally analyzed using tools and techniques available for Petri nets.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design

Additional Key Words and Phrases: Design Notations, Formal Design, Real-Time Systems, Petri Nets, POSIX

This work is partially supported by UBACyT project EX-186 and European Community project KIT 125.

1. INTRODUCTION

In the field of real-time systems, several well-settled specification and design languages now exist and are used in the academia world. However a few of them are used in the real world: the industry. On the other hand, every day a new formal and powerful method is developed in the academia while the industry continues to use their informal and semiformal methods. The unbridging of this gap impacts directly on the quality of any kind of systems, but particularly in real-time systems.

Real time systems are often used in safety critical applications where failures can cause unacceptable damages. High reliability standards are required to met the fault-free goal.

There is no discussion about the advantages in using formal notations and methods to limit the possibility of erring in every phase of the development: powerful analysis and verification techniques can be applied to reveal errors [Gerhart et al. 1991]. There is also no discussion about the difficulties to introduce them in the industry. Formal specifications are largely ignored in the industry due to the already mentioned gap between the real needs of practitioners and the aspects considered by the scientists for defining a notation.

On the other hand, the probability of erring in the specification and design phases can be also reduced by using methodologies and notations well-suited for the application domain and well-known to domain experts. Usually, such notations are rich, but often lack of formal foundation. Examples of such notations are extensions of data flow diagrams for real-time systems: Hatley and Pirbhai notation [Hatley and Pirbhai 1987] and Ward-Mellor methodology [Ward and Mellor 1986]. Effectively, such notations are pretty popular in industry, and well supported by software engineering environments. However, the lack of formal foundation prevents many analysis and verification techniques to be applied.

The ESPRIT IDERS project proposed animation of graphical software specification notations, design notations and embedded code as a technique to improve the visibility of real-time systems under development. The IDERS project proposed a solution to software animation which allows the adaptation of graphical notations to company or project-specific requirements. This will be achieved by allowing the full configurability of the syntax and semantics of notations. Configurability is vital to gain the full benefit of graphical design notation, since the detailed design of embedded systems is greatly influenced by the real-time operating system features which are available in the target system.

Another project funded by the European Community, KIT 125, has been proposed as a cooperation between the Universidad de Buenos Aires, Argentina, Politecnico di Milano, Italy, and the IFAD Institute, Denmark, for providing complementary results to the IDERS project. The project KIT 125 aims at extending and validating technology and tools provided within the ESPRIT project IDERS. In particular, the project KIT 125 aims at defining new notations to be formally introduced in the IDERS platform using the IDERS customization facilities [Christensen et al. 1994].

The first results of this cooperation have been presented in [Baresi et al. 1995] and in [Baresi et al. 1996]. In such publications a first version of a formal design notation for the Hatley and Pirbhai methodology supported by IDERS has been introduced.

The notation is meant to be used with the requirements specification notation proposed by Hatley and Pirbhai, based on structured analysis methodologies. The proposed notation extends the requirements specification notation with primitives that are common in the design of hard real-time systems, such as messages, signals, data stores. In this way, this new notation allows a smooth transition between the requirement definition to the design phase by providing systems engineers with compatible specification and design notations. Moreover, the generality of the notation is guaranteed since its new constructs have been derived from the standard IEEE Posix [IEEE 1992].

This paper extends the results therein presented by introducing a formalization of that notation and by reviewing some semantical definitions in order to enhance the compatibility between the specification and design notations. This paper aims particularly at promoting the work that is currently carried out at Universidad de Buenos Aires for the KIT project to accomplish one of its goals: transference of technology and raising the basis of research cooperation within Argentina.

The paper is structured as follows. Sections 2 and 3 introduce the new design notation highlighting analogies with the H&P notation and the real-time extensions of POSIX. Section 2 describes the system level model, i.e., the specifications of the architecture of the system designed in terms of tasks and their connections. Section 3 describes the task level model, i.e., the specifications of the internals of tasks. Section 4 introduces the operational semantics of the design notation using high level timed Petri nets. Section 5 discusses the main results.

2. SYSTEM LEVEL MODEL

2.1 Task

The notation is an abstract view of the execution architecture of the system based on the task as the sequential unit concept.

The proposed design notation comprises two levels: ***system level*** and ***task level***. The system level describes the main components: tasks and terminators, and their connections: message queues, shared memories and mutexes (see figure 1). The task level describes the internals of each task.

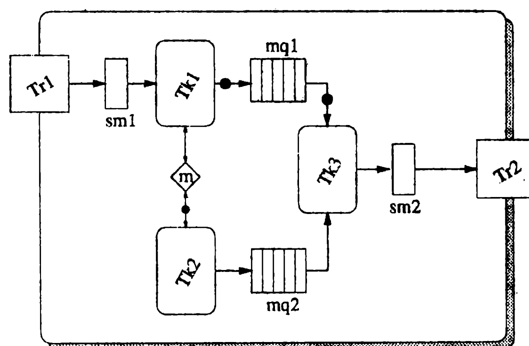

Fig. 1. Example of system level model

Terminators model the embedding of the system. Terminators can generate

events at given instants, and they can absorb events produced by tasks. Terminators are associated with precise textual annotations that specify the data to be generated and the time instants at which such data are sent to the connected shared memories.

Tasks are sequential activities that interact among them and with terminators. Tasks can be executed concurrently, provided a suitable availability of processing resources. Tasks can communicate each other and with terminators synchronously or asynchronously. Asynchronous communication channels are message queues and shared memories. Synchronization elements are mutexes and conditional variables.

Message queues are multi-reader and multi-writer priority queues of messages with finite capacity and deadline. *Shared memories* are repositories with non blocking read and write operations. They are shared among tasks and terminators. Shared memories have unlimited capacity. *Mutexes* are used for synchronization. They guarantee mutual exclusion among concurrent tasks. Mutexes can be locked and unlocked by tasks. *Conditional variables* are intended to suspend tha task waiting for a condition.

Tasks also have an *asynchronous event notification* mechanism. This mechanism is visualized into a task as a set of prioritized event handlers. An event handler is a code of the task devoted to attend notifications. Each of them have an associated *event notification queue.*

All these elements aim at bringing tools and discipline patterns for designing systems in a POSIX.13 Minimum Real Time Systems Profile [IEEE 1991] compliance framework. In this version only connectors are taken from this standard (see section 5) [1].

2.2 Communication and Synchronization

The elements that serve as connectors are message queues, shared memories, mutexes and conditional variables, as was said. Message queues and shared memories are the *communication elements* between tasks and terminators, while mutexes and conditional variables are *synchronization elements.*

The semantics of the communication elements is given by indicating the semantics of the *objects* themselves and the *services* they provided. The semantics of the objects describes how the data are stored, i.e., the abstract data type that supports the communication media. This abstract data type will be called *repository.*

The semantics of the services is given by indicating its effects on the state of the system and how it changes the control flow. Only a characteristical set of services are shown. The full set of operations are presented in [Braberman et al. 1996].

The state of the system [2] is given by:

(1) the state of the repositories of the communication elements.
(2) the history of the relevant events to each task.
(3) the state of mutexes.
(4) the set of waiting queues of tasks which are blocked, waiting for a resource or an event.

[1] Tasks can be thought as operating system processes, although the Minimum Real Time Systems Profile requires a single process and multithreading environment.

[2] It is supposed that operating system concurrent operation's code do not interfere among them, i.e., the operating system manages concurrent operations in a serializable fashion.

Then, to describe the state of the system the following functions are introduced[3]:

—Data : $id_{cm} \rightarrow repo_{cm}$

—History : $id_{task} \rightarrow$ sequence [event]

—Locked? : $id_{mutex} \rightarrow$ bool

—IsLockedBy? : $id_{mutex} \times id_{task} \rightarrow$ bool

—Waiting : op $\times$ $id_{cm} \rightarrow$ queue [id_{task}]

Data returns the repository of a given communication media of the system, wich has an identifier *id* (e.g., a priority-queue for a message queue). *History* returns the sequence of relevant events that happens for a task. For example, sent a message, received a message, read data, blocked-locking, blocked-sending a message, unblocked, etc. *Locked?* indicates if a mutex is blocked by some task. *IsLockedBy?* indicates if the mutex is locked by this task. *Waiting* is the waiting queue of tasks wich are blocked by performing the operation *op* in the corresponding communication media.

Message Queues

Message queues are multi-reader and multi-writer priority queues of messages with finite capacity and deadline. Messages are assigned with priorities that determine their extraction order. Messages are discarded after the deadline, unless extracted before. Tasks can either send or receive messages. Send operations can be blocking or non-blocking. Blocking sends cause tasks to wait if the queue is full (note that deadlines ensure non infinite wait in this case). Non-blocking sends cause messages to be lost if the queue is full. Similarly, receive operations can be blocking or non-blocking. Blocking receives cause tasks to be suspended if the queue is empty. Non-blocking receives may return no values if the queue is empty. Tasks blocked on a message queue are awaken according to a FIFO policy.

Time issues, i.e., deadlines, are described by marking the time an element is pushed in the queue, and referring to such time-stamp for defining the life of that element.

The repository for a message queue is a priority queue (FIFO policy), with finite capacity and deadline, and have the usual functions empty, push, pop, full? and empty?. The operations on a message queue are: *SendNonBlock*, *Send*, *ReceiveNonBlock* and *Receive*. Send operations push messages in the message queue while receive operations extract messages from the message queue. If a task performs a non-blocking operation that fails, it is notified and continues executing.

Pattern matching is used to identify the queue: [ ] denotes the empty queue, and $M.Q$ denotes that the highest priority element stored is M and the rest of the queue is Q.

To describe each operation, the pre and post condition notation is used ([Gries 1991]). Lines are related by an *and*. The subindex 0 indicates the value in the previous state. Only the modifications in the global state are explicitly indicate, without mentioning unaffected components.

Rules:

T Send M to Q

[3] *id* stands for an identifier, *cm* stands for communication media, *repo* stands for repository and *op* stands for an operation that may cause a task to be blocked.

$$\frac{\begin{array}{c}\neg full?(Data(Q)) \\ Waiting(Receive, Q) = [\,]\end{array}}{\begin{array}{l}Data(Q) = push(Data_0(Q), M) \\ History(T) = sent(Q, M).History_0(T)\end{array}}$$

T Send M to Q

$$\frac{\begin{array}{c}\neg full?(Data(Q)) \\ Waiting(receive, Q) = T_1.L\end{array}}{\begin{array}{l}Waiting(receive, Q) = L \\ History(T) = sent(Q, M).History_0(T) \\ History(T_1) = received(Q, M).(unblocked.History_0(T_1))\end{array}}$$

T Send M to Q

$$\frac{full?(Data(Q))}{\begin{array}{l}Waiting(Send, Q) = push(Waiting_0(Send, Q), T) \\ History(T) = blocked_sending(Q, M).History_0(T)\end{array}}$$

T Receive from Q

$$\frac{\begin{array}{c}Data(Q) = M.Q_1 \\ Waiting(Send, Q) = [\,]\end{array}}{\begin{array}{l}Data(Q) = Q_1 \\ History(T) = received(Q, M).History_0(T)\end{array}}$$

T Receive from Q

$$\frac{\begin{array}{c}Data(Q) = M.Q_1 \\ Waiting(Send, Q) = T_1.L\end{array}}{\begin{array}{l}Data(Q) = push(Q_1, M_1) \\ Waiting(Send, Q) = L \\ History(T) = received(Q, M).History_0(T) \\ History(T_1) = sent(Q, M_1).(unblocked.History_0(T_1)) \\ \quad where\ M_1 = what_is_sending(History_0(T_1))\end{array}}$$

T Receive from Q

$$\frac{Data(Q) = [\,]}{\begin{array}{l}Waiting(Receive, Q) = push(Waiting_0(Receive, Q), T) \\ History(T) = blocked.History_0(T)\end{array}}$$

Conditional Variables

Conditional variables are intended to wait for a condition. Conditional variables have four operations: *Wait*, *CWait*, *Signal* and *SignalBroad*.

Usually, if the task is executing in a critical section the mutex that gives the exclusivity should be yield to allow the progress of others cooperative tasks. Then the Wait(V,M) operation is equivalent to:

$$Unlock(M); Wait1(V); Lock(M)$$

The Unlock and the Wait1 are performed atomically.

Also, conditional variables have an associated *predicate* to be evaluated. A CWait operation over a conditional variable and a mutex, means that, while the associated predicate with the conditional variable is not true, the task waits for a signal. If the conditional variable is *Signal*ed and the task is waken up or the conditional variable is *SignalBroad*casted then the task contends for the yielded mutex again. When the task gets the mutex again, the predicate is checked to see whether it should repeat the wait sequence. Then, the CWait(V,M) is equivalent to

$$\begin{array}{l} while\ \neg AssociatedPredicate(V) \\ \quad Wait(V, M) \\ endwhile \end{array}$$

CWait operation is not a POSIX primitive but this scheme is observed as an usual programming discipline [IEEE 1992].

A Signal operation on a conditional variable wakes up the first task waiting for the variable. On the other hand a SignalBroad wakes up all the waiting tasks.

Rules:

T Wait1 V

$$\frac{true}{\begin{array}{l} Waiting(signal, V) = push(Waiting_0(signal, V), T) \\ History(T) = blocked.History_0(T) \end{array}}$$

T Signal V

$$\frac{Waiting(signal, V) = [\ ]}{History(T) = signal(V).History_0(T)}$$

T Signal V

$$\frac{Waiting(signal, V) = T_1.L}{\begin{array}{l} Waiting(signal, V) = L \\ History(T) = signal(V).History_0(T) \\ History(T_1) = unblocked.History_0(T_1) \end{array}}$$

T SignalBroad V

$$\frac{true}{\begin{array}{l} Waiting(signal, V) = [\ ] \\ History(T) = signalbroad(V).History_0(T) \\ \forall\ T_1 \in Waiting_0(signal, V) \\ \quad History(T_1) = unblocked.History_0(T_1) \end{array}}$$

3. TASK LEVEL MODEL

3.1 Basic Elements

Functionalities of tasks are described using a subset of the H&P notation. A task is specified as a H&P data flow diagram (DFD) comprising: processes (i.e., functions, not OS processes !), time-transient flows, split and merge points, and data stores, as illustrated by the example in figure 2.

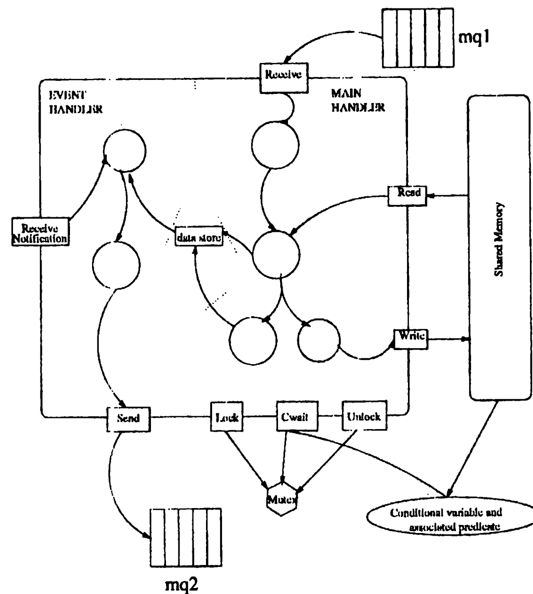

Fig. 2. Example of task level model

Functionalities and timings of processes can be specified using a precise language. Either VDM [Larsen et al. 1994] or C++ are used. Timing specifications of processes indicate the minimum and maximum execution time. Processes are activated when all their input data flows carry data. The execution of a process terminates producing data on all its output data stores and on exactly one of the output data flows. To a more detailed explanation see [Christensen et al. 1995].

Processes of this DFD are partitioned into *handlers*: a main functionality handler, and a set of event handlers. Handlers are executed in mutual exclusion, i.e., at most one handler for each task can be executing. Event handlers are associated with *notification queues*, that stores incoming events to be handled. Processes belonging to the same task but to different handlers could internally communicate only through data stores. The dynamics of handlers is governed by a *control flow specification* (CFS) that describes the execution path. The execution model and the asynchronous event handling are explained in next sections.

3.2 Expressing Communication and Synchronization

To express communication and synchronization, the designer has the concept of I/O operations (e.g. Receive and Read for input, Send, Write, SigQueue for output)

and synchronization primitives (Lock, Unlock, Wait, Signal).

All these operations are naturally modeled as processes. These special processes receive or provide information from or to other processes of the same task using data flows (if they are input/output operations). They follow the task operation semantics explained in the former rules.

All connections of tasks with communication elements at the system level must correspond to connections of processes of the corresponding tasks with the communication elements at the task level.

3.3 CFS and Execution Model

To model the internal sequentiality of a task it must be assured that at most one process is executing at a given time. Moreover, in real time systems the task should have the most predictable behavior avoiding a non deterministic executing model. So a precise execution order must be defined.

Deterministic CFSs are the key concept to achieve this behavior. The functional model (DFD) formerly explained shows the functionalities, I/O and synchronization operations and a static data precedence order. A process executes iff all its input data flows carry data and it receives an explicit *GO!* provided from the corresponding CFS. When the process finishes its execution it sends to the CFS a *DONE* signal to notify its termination and to enable a new start. The terminating process could provide data condition wich would be used by the CFS to choose next process to execute.

So, a possible choose for the CFS mathematical model is a function, called *NEXT*, from process ids and its control output to process ids. This function determines the next process to be executed by considering the last executed process and by evaluating the different alternatives for its control output values (if any). Such an evaluation must be deterministic.

A special value must be used by *NEXT* to indicate the end of the handler. The first process to be executed should be explicitely defined. Then, a CFS could be thought as the control flow graph that defines an explicit interleaving order constraining the potential parallelism expressed by the DFD [4].

3.4 Asynchronous Event Notification Mechanism

The language provides facilities to signal (operation SigQueue) and handle events in an asynchronous way (i.e., without blocking or pooling). The event notification types are prioritized. The task should comprise an event handler for each possible user event type it is supposed to handle.

When a task is started the main functionality handler is entered executing its initial *GO!* action of its CFS. The deliver of notifications could affect the natural executing order. Handlers execute following their CFS as previously explained.

When an event notification is delivered several situations might arise. If the notification priority is higher than the executing handler priority, the executing

[4]Note that it is possible to trigger processes that are not data enabled. This would lead to an indefinition of the whole task. To avoid this undesired situation, common traveling strategies (e.g. depth first, breath first, etc.) over the DAG of processes according to DFD should be followed in the *NEXT* function definition.

process is immediately suspended and the corresponding event handler is started (the main functionality is supposed to have the lowest priority). If the priority is not higher than the current handler level, the notification is stored in its priority queue. In a simultaneous deliver of notifications, one of the highest priority notification is chosen to apply the same analysis already explained, the rest are queued in some non specified order into their corresponding queues.

When the current handler finishes, the highest priority suspended handler is chosen to resume (resuming the suspended process) unless there is an accumulated notification of higher priority than the suspended one. In the later case the highest priority notification that is stored determines the handler to be followed immediately. The first stored notification of the type selected to be attended is available for the corresponding handler using ReceiveNotification operation (FIFO). So event notification could be thought roughly as asynchronous message passing mechanism without explicit invocation of a receive operation; the receive is executed when the handler wants to know the data stored in the queue.

Another source of ambiguities is the simultaneous arrival of an internal *DONE* and external notifications (note that there is no possibility to generate two simultaneous internal events). The following approach is adopted. The internal control event is evaluated by the current CFS and then the external events are analyzed. This avoids remembering internal events.

There are also system notifications that affect the status of a task from the OS point of view. There are three system signals: *start*, *suspend* and *resume*. The start signal makes a not active task start executing again its main functionality. If the task is active (executing, blocked or suspended) the start signal makes it start again, abandoning all waiting queues if necessary. This signal, in combination with timers (not directly supported in this version but modelable in terms of a ficticious task) makes the representation of periodic tasks possible.

The suspend and resume signals freezes and wakes up tasks respectively. When a task is suspended the notification mechanism is suspended too and the new incoming notifications are lost. In this version all task are supposed to be simultaneously active at initial time. As shown in the former rules some operations (e.g., Send, Receive, Lock) may cause the current handler to be blocked. How does it affect the mechanism explained so far ? Actually the operation that causes the blocking delays (maybe forever) the *DONE* delivery. Consequently, the execution model and notifying mechanism behaves naturally as if the process is really being executed. Thus only higher priority notifications are allowed to be attended when a handler is blocked.

4. OPERATIONAL SEMANTICS USING HIGH LEVEL TIMED PETRI NETS

Semiformal notations could be furnished with a formal semantics using some kernel formalism. HTLPN (high level timed Petri nets) [Ghezzi et al. 1991] are used to achieve this goal. HLTPN are Petri nets where tokens carry typed data and a time stamp. Transitions are enhanced with predicates and two timing expressions that calculate the minimum and maximum time that the transition can fire after been functionally enabled (data in the preset places satisfying the transition predicate). The tokens produced by the firing of a transition are timestamped with the firing time.

This paper overviews how to represent some of the elements of the design notation by means of Petri nets. For a more detailed description see [Braberman et al. 1996]. The given semantics is compatible with the semantics of the H&P notation that was given in [Christensen et al. 1995], i.e., requirements and design specifications can be validated and compared using the same analysis tools and techniques. The possibility of analyzing heterogeneous requirement and design specifications allows systems to be incrementally verified, thus anticipating the detection of many errors.

A compositional approach is followed as far it is allowed by the absence of modular mechanisms in the kernel formalism. The main objective is to explain how the elements of the user notation are modeled as sub-nets and how they are connected to build the entire net. A mechanism based on graph grammars could be used to achieve this goal [Baresi et al. 1995].

4.1 Semantics of Requirements Notation Elements

As was already mentioned, processes, data flows, data stores and split/merge points are requirement notation elements that can be placed into a task. The semantics that was given in [Baresi et al. 1995] is used for these elements.

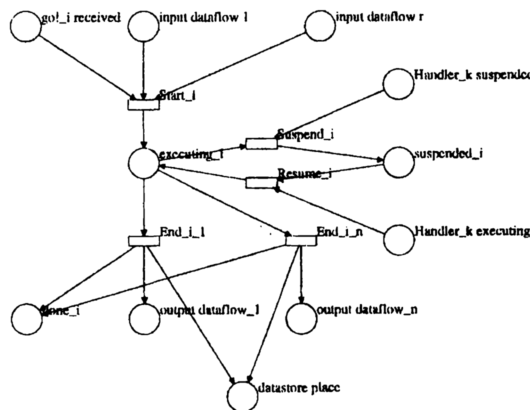

Fig. 3. Process net

There are several *End* transitions, one for each output dataflow, all of them write in the *done* place and in the related data stores.

The token that is alternatively in the *executing* or *suspended* place plays an important role. It records the time units executed by the process (timeunitsexec [k]). The reason for keeping track of this information is that, when a process is interrupted (i.e. *suspend* transition is fired), it is necessary to know how much time it has been executed so far.

$$timeunitsexec\ [k] := timeunitsexec\ [k]\ +\ enabling\ time\ -\ timestamp$$

So, when a token is in the *executing* place again the corresponding *End* transition is fired after the right amount of time.

$$tmin\ =\ tmax\ =\ timesize\ of\ data\ -\ timeunitsexec\ [k]$$

Note that the interruption of a process is subordinated to the state of its comprising handler (*handler_suspended*, *handle_executing*). Scheduling, not dealt with in this language version, could be treated using the formerly explained techniches.

4.2 Semantics of Communication and Synchronization Elements

Messages queues, shared memories, mutexes and conditional variables are represented by two places: *state* place keeping the data token (of the type of the repository), and a *waiting* place keeping a token with the waiting queue of the resource. The respective operations should modify these tokens according to their semantics. To illustrate these concepts the net representing the Receive operation of the message queues is shown.

In the blocking case, the *SuccessReceive_i* transition fires if there are data to be read. Its firing changes the data contained in the token on the *state* place. The *UnsuccessReceive_i* occurs when there is no data or there are waiting processes, then the handler is blocked. An *Unblock* occurs when the data is ready for this task. *Unblock* transitions have higher priority than *SuccessReceives* transitions to assure that, if data is stored, the waiting processes are the first to be executed.

In the non-blocking case only the success branch is modeled.

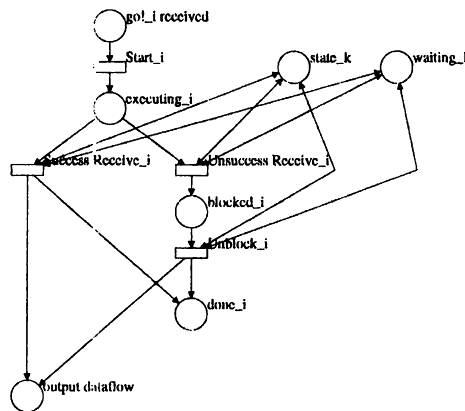

Fig. 4. Receive net

5. CONCLUSIONS AND FUTURE WORK

This paper extends the results presented in previous papers. It presented a formally defined design notation for structured methodology: an extension of Hatley and Pirbhai notation to cope with design of real-time systems. This definition has been done as an extension of the requirement specification notation to enhance its introduction in industrial environments and enhance verifiability. Generality of the notation is guaranteed since the extension was based on the wide-adopted standard POSIX.

Several improvements can be done to the notation to ease a direct map to an operating system that fulfils POSIX Minimal Real Time Systems profile: Scheduling,

providing mechanisms to establish priorities and policies (e.g., in the rate monotonic framework), Multi-Thread can be addressed from the design point of view where a thread is a sequential activity (like a task) encapsulated into a module called OS Process that defines certain access to all its threads.

Besides language enhancements, there are several topics to develop around the user notation related to verification and testing issues. Due to the formal semantics of the notation, it would be possible to set a sound framework to reason about smooth transition from requirements to design. From the methodological point of view a set of sound transformation rules and heuristics should be given to lead to a detailed solution.

ACKNOWLEDGMENTS

The authors wish to thank Prof. Mauro Pezzé and Luciano Baresi, members of the italian partner, with whom many of the previous results were developed.

REFERENCES

BARESI, L., BRABERMAN, V., FELDER, M., PEZZÈ, M., AND PIENIAZEK, F. 1995. A design notation for the hatley and pirbhai methodology. Technical Report kit-01, KIT125.

BARESI, L., BRABERMAN, V., FELDER, M., PEZZÈ, M., AND PIENIAZEK, F. 1996. A practical approach to formal design of real-time systems. In *IEEE International Conference on Systems, Man and Cybernetics.*

BARESI, L., PEZZÈ, M., AND ZANCHI, P. 1995. A grapgh-grammar definition of Hatley and Pirbhai's notation. Technical report, Politecnico di Milano.

BRABERMAN, V., FELDER, M., AND PIENIAZEK, F. 1996. Enhancing the adoption of formal methods to design real-time systems. Technical Report TR-96-002, Dep. de Computación, FCEyN, Universidad de Buenos Aires.

CHRISTENSEN, H., ELMSTRØM, R., VOSS, H., BARESI, L., CALZOLARI, F., AND PEZZÈ, M. 1994. The customization toolset: Specification document. Technical report (October), Politecnico di Milano and The Institute of Applied Computer Science (IFAD).

CHRISTENSEN, H., KIRKEGAARD, N. K., AND BARESI, L. 1995. Definition of the IDERS Hatley & Pirbhai notation. Technical report (November), Politecnico di Milano and The Institute of Applied Computer Science (IFAD).

GERHART, S., M.BOULER, GREENE, K., JAMSEK, D., RALSTON, T., AND RUSSINOFF, D. 1991. Formal methods transition study final report. Technical Report STP-FT-322-91 (August), MCC, Austin (Texas).

GHEZZI, C., MANDRIOLI, D., MORASCA, S., AND PEZZÈ, M. 1991. A unified high-level Petri net model for time-critical systems. *IEEE Transactions on Software Engineering 17(2)*, 160–172.

GRIES, D. 1991. *The Science of Programming.* Springer Verlag.

HATLEY, D. AND PIRBHAI, I. 1987. *Strategies for Real-Time System Specification.* Dorset House, New York.

IEEE 1991. *Real-Time System Profile.* IEEE.

IEEE 1992. *POSIX.4 Real-Time Extensions for Portable Operating Systems.* IEEE.

LARSEN, P., ELMSTRØM, R., AND LASSEN, P. 1994. The IFAD VDM-SL toolbox: A practical approach to formal specifications. *ACM Sigplan Notices 29(9)*, 77–80.

WARD, P. T. AND MELLOR, S. J. 1985–1986. *Structured Development for Real-Time Systems*, Volume 1–3. Yourdon Press, New York.