A Practical Approach to Formal Design of Real-Time Systems

Luciano Baresi †, Victor Braberman ‡, Miguel Felder ‡, Mauro Pezzè †, Fabio Pieniazek ‡

† Dep. de Computación - FCEN
Universidad de Buenos Aires
Buenos Aires, 1123, Argentina

‡ Dip. di Elettronica e Informazione
Politecnico di Milano
Milano, 20133, Italy

ABSTRACT

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 paper is formally defined by means of high-level timed Petri nets, and can be formally analyzed using tools and techniques available for Petri nets. Automatic tools for editing and verifying design specifications given in terms of the notation proposed in this paper have been implemented and the notation has been successfully validated on industrial case-studies.

1. INTRODUCTION

Formal methods are being increasingly used in engineering industrial software. The growing success of formal methods among practitioners is due to several factors. The encouraging results with pilot industrial projects demonstrated advantages and limitations of formal approaches, and they allow careful forecast of risks and benefits ([9]). The availability of a growing number of reliable tools ([16, 13, 14]) extends the scope of applicability of formal methods and it simplifies the work of the experts of the application domains. The possibility of using formal methods with popular notations, e.g., SA-SD/RT or ESML, makes formal methods accessible to many experts with different backgrounds ([5, 6, 7, 8, 15, 17]).

Formal methods are being used with alternate success in different phases of the software life cycle. They are extensively used for specifying and verifying software requirements, but they are seldom used in other phases, e.g., design, coding, maintenance. This creates undesirable gaps between the early phases of development that can be formally verified, and the later phases of development that are often accomplished using traditional, error-prone techniques. A notable exception is the use of code generators, that produce code directly from specifications. When available for formal methods, code generators can automatically produce formally verified code. Unfortunately, most code generators produce only partial code that must be completed by software engineers that often introduce errors. The few code generators that produce complete code from formal specifications generate very inefficient code, that can be seldom used in safety critical real-time applications, with strict timing requirements.

This paper tries to bridge the gap between formal requirements specification and final code by introducing a formally defined design notation. The introduced design notation is compatible with the requirements specification notation proposed by Hatley and Pirbhai in [11], i.e., it presents many syntactic analogies and it is given homogeneous semantics. The requirements definition notation proposed in [11], hereafter referred to as the H&P notation, has been chosen as representative of structured analysis, currently one of the most popular notations in industry for the specification of real-time systems [18]. The proposed notation extends the H&P notation with constructs derived from POSIX ([12]), e.g., message queues, shared memories, mutex; and it changes the semantics to correctly model the concepts of tasks and resources. The POSIX real-time extensions have been chosen as emerging standard, that well represents typical choices of industrial real-time applications.

The design notation proposed in this paper is formally defined by means of high-level timed Petri nets ([10]), and it is compatible with the semantics of the H&P notation presented in [4]. Both the H&P notation, the design notation proposed in this paper, and any combination of the two can be formally analyzed using the same set of tools and techniques available for Petri nets. Thus, system specifications can be incrementally verified while moving from requirements to design specifications. Automatic tools for editing and verifying specifications given in terms of the design notation proposed in this paper have been derived using the customization facilities of the IDERS tool-set ([1]). The toolset has been successfully validated on an industrial case-study as part of the IDERS project. The chosen application is the Reply Processor and Channel Management system of an advanced radar system ([2]).

This paper is organized 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

Terminator Tr2 provides task Tk1 with data by means of the shared memory sm. Tasks Tk2 and Tk3 are synchronized by mutex m (locks from Tk2 are suspensive) and write on message queues mq1 and mq2, respectively (all the operations on mq1 are blocking). Task Tk3 reads from both mq1 and mq2 and writes on message queue mq3. Terminator Tr2 reads messages from mq3.

Figure 1. A sample system level model.

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). The task level describes the internals of each task. The system level model is illustrated with an example in Figure 1.

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 or message queues.

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.

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. 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 queue are awakened according to a FIFO policy.

Shared memories are repositories shared among tasks and terminators. Shared memories have unlimited capacity. Read and write operations are non suspensive.

Mutexes are used for synchronization. They guarantee mutual exclusion among concurrent tasks. Mutexes can be locked and unlocked by tasks. Locks can be either suspensive or not. In the first case, the task waits for the mutex to be released, in the second case, it continues its execution.

3. TASK LEVEL MODEL

Functionalities of tasks are described using a subset the H&P notation. A task is specified as a H&P data flow diagram comprising: processes (i.e., functions, not to be confused with OS processes), time-transient flows, split and merge points, and data stores, as illustrated by the simple example in Figure 2.
Processes of a task are partitioned into threads: a main thread, and a set of event handlers. Threads are executed in mutual exclusion, i.e., at most one thread for each task can be executing. Event handlers are associated with queues, that stores incoming events to be handled. Processes belonging to the same task, but to different threads can communicate only through data stores. 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.

The dynamics of threads is ruled by a Control Flow Specification (CFS) that describes the execution path. CFSs ensure sequentiality and determinism, as required by real-time tasks. Processes execute if enabled by their input data flows (all input data flows carry data) and explicitly started by a GO signal produced by the corresponding CFS. When Processes terminate their execution, they send an explicit DONE signal to the CFS, to notify their termination and to enable a new start. CFSs take care of disabling/enabling processes according to the execution conditions.

4. SEMANTICS

The design notation introduced in this paper is given semantics by high-level timed Petri nets (HLTPNs [10]). The semantics is compatible with the semantics of the H&P notation given in [4], i.e., requirements and design specifications can be validated and compared using the same analysis tools and techniques. The possibility of analyzing heterogeneous requirements and design specifications allows systems to be incrementally verified, thus anticipating the detection of many errors.

The semantics of processes, data flows and data stores is outlined in Figure 3. A process is modeled by a transition Start, that models the beginning of its execution, and a set of transitions End that model its termination and correspond to its output data flows. Places Started, Executing and ReadyToEnd record the elapsed time to correctly compute the time of termination of the process and they correspond to the different internal states of a process. Data flows and data stores are modeled by Petri net places that store the current values. Split and merge points correspond to suitable “duplication” or “unification” of places.

Message queues are modeled by two places: place State, that stores the data, and place Waiting that stores the queue of the tasks waiting for the resource. Figure 4 illustrates the semantics of the operations allowed on queues. Processes are ready to perform receive operations after having being enabled by the CFS through place GO and after having read from input data flows and data stores (if required). Processes ready to perform receive operations are modeled with place Started marked. Transition Receive correctly read the first available message from place State. If the receive is blocking, the predicate associated with transition Receive enables the transition only if the message queue is not empty, and consequently, the queue of waiting tasks (place Waiting) does not contain tasks with higher priority. If the receive fails, transition FailedReceive moves the process to state Blocked and updates the waiting queue associated with the message queue (place Waiting). Waiting tasks are awakened according to their priority by transition SuccessfulReceive. If the receive is non blocking, transition Receive fires anyway, returning a datum or a special value “empty”, if the message queue is empty.

Send operations are performed just before terminating the execution of the process. The firing of transition Send pushes a new datum on the message queue and moves the process to state ReadyToEnd. If the send is blocking, the predicate of transition Send requires the queue not to be full and no process with higher priorities to be waiting. Otherwise transition FailedSend pushes the task in the waiting queue associated with the message queue (place Waiting). The task is resumed by transition SuccessfulSend. If the send is non blocking, transition Send returns a value “full”, if the message queue is full.

Shared memories are modeled as marked HLTPN places. A read is performed by accessing the value of the token in place State. A write replaces the value of the token in place State with a new datum. Figure 5 shows the syntax and semantics of

The chosen semantics states that a process writes on all its output data stores and on exactly one output data flow.

Semantics of process P4 of Figure 2. Transition Start is enabled by the CFS through place GO and reads from data store ds1. Transition Receive gets a datum from message queue mq2. Transition Send sends a datum to message queue mq3 at the end of the execution. Transition End notifies the termination of the execution to the CFS through place DONE.

Figure 3. HLTPN semantics of task internals.
read and write operations performed by processes and termina-
tors.

A mutex is modeled with three places: Locked, Unlocked and Waiting. Places Locked and Unlocked model the states of the mutex; place Waiting keeps track of the tasks waiting on the mutex. As shown in Figure 6, a task can issue either suspensive or non suspensive locks. In the for-
ermer case, if the process cannot acquire the resource, it remains
blocked until the mutex becomes available. In the latter case, the process continues its execution even if the lock fails.

CFS are given semantics as finite state machines connected
to places DONE and GO of the corresponding processes to con-
trol task execution. An example of the semantics of a CFS is
drawn in Figure 7.

Processing resources are modeled with shared places that

guarantee the mutual exclusion of the execution of tasks sharing
the same processor.

5. CONCLUSIONS

This paper presented a formally defined design notation for
structured analysis. The syntactic and semantic similarities with
requirements specification notations used in structured analysis
facilitate the introduction of the notation in industrial environ-
ments and enhance verifiability. Techniques have been devel-
oped to verify requirements and design specifications as well as
heterogeneous specifications. The verification of heterogeneous
specifications guarantees the possibility of incremental verifica-
tion while moving from requirements to design.

Suitable tools for specifying and verifying the design of sys-
tems using the notation described in this paper have been developed
using the customization facilities provided by the IDERS
SDA³((3)). Such tools have been used on an experimental basis
for designing and verifying the Reply Processor and Channel
Management system of an advance radar system as part of the
IDERS project. The results were pretty encouraging. Application
experts familiar with structured analysis did not have dif-
ficulties in using the proposed design notation. Analysis tools
based on the formal semantics revealed many errors uncaught
in the preliminary informal design phase. The approach led to
the production of efficient code as required by the application,
when commercial code generators failed in producing code with
acceptable performances for the specific application.

³Specification and Design Animator.
Figure 6. HLTPN semantics of mutexes.

REFERENCES


