Higher-Level Statecharts for Prototyping Architectural Dynamics

Dennis B. Mulcare
SAIC
Warner Robins, GA 31088

Abstract - A new system-level prototyping method has been developed to model the dynamics of concurrent real-time systems. This approach is based on higher-level statecharts [1], which embody object-based extensions to basic statecharts [2]. Tokens circulating within subgraphs represent passive objects, and subgraph instances correspond to active objects. The tokens may be of differing types, and each possesses its own state and descriptor information as well as identity. Transition rules expressed in 1st-order logic mediate token circulation in accord with intended dynamic behavior. Each statechart subgraph may serve as a template for an arbitrary number of active objects of the same type. The coupling of superstate subgraphs via transition rules then denotes the interleaving of independent threads of control. Moreover, the "firing" of a transition rule may alter the states of designated passive objects, and may result from an action performed by another active object. As a consequence, higher-level statecharts can represent the concurrency behavior of complex multi-object systems in a precise and compact manner, and do so in a manner that is readily and uniformly translatable into executable code.

To exemplify this method, a global communication mechanism is prototyped for a real-time multicomputer system that executes a single logical multitasking program. Since this prototype is intended to verify real-time concurrency logic, calibrate performance, and ensure safety, it includes a global virtual time base as a statechart subgraph. The prototyping process consists of capturing the system-level communication architecture in a higher-level statechart, and then translating it to an Ada multitasking program. Due to the inherent complexity of concurrency logic, the construction of higher-level statechart is not a simple undertaking; some iteration and refinement is typically necessary. But the ease and directness of mapping a higher-level statechart to an Ada program is rather straightforward, in part because of a characteristic software architecture implicit in higher-level statecharts. Known as a mutual agents architecture, this regularized form of the prototype appears to be suitable for an automated prototyping environment.

INTRODUCTION

Here, prototyping is viewed as a practical means of illuminating problem subtleties, codifying definitive requirements, and eliciting informed design concepts. It serves to suggest superior design options or to quantify performance indices, in addition to identifying discrepancies or omissions. At the system level in particular, prototyping is vital to the formulation of credible architectural concepts and coherent design solutions, as well as to the precise definition and verification of concurrency logic and absolute timing requirements. Fault-tolerant embedded systems, for example, are typically targets of concern because of complicated aspects like time-critical resource reconfiguration under faults.

Basically, the early-on formulation of a precise and coherent system representation, especially of dynamic semantics, prompts the timely confrontation and resolution of many latent design issues. These would otherwise tend to become increasingly problematical and awkward to rectify. The execution of an associated prototype, moreover, enables a comprehensive and possibly quantified assessment of the dynamic behavior of the architecture. Such an approach is advocated because it imparts a very substantial degree of precision and certitude to system-level development from the very outset of development.

Optimized resource selection/utilization and overall architectural performance are primary concerns here, and they can only be realized through the well-informed delineation/specification of physical components. This must be based largely on an overall dynamically determined distribution and reconciliation of timing requirements. The intent is to avoid the combination of bottlenecks and overdesign that typify static or "TBD" timing apportionments. Time-based prototypes with appropriate analyses, moreover, provide a substantiated basis for stipulating quantitative parameters and tolerances of components, as sought via the prototyping activities in Fig. 1. Note that requirements are codified and validated with reliance on system-level prototyping, and that the prototype also provides a basis for quantifying a resulting...
Control state denotes the status of the underlying system architecture as well as the higher-level modes of system operation. Thus, control state encompasses resource configuration, fault status, functional modes, etc.; it is of prime importance in most real-time systems, especially those that are event-driven. Because of the state-event nature of such systems, Petri nets [6] and basic statecharts have been extensively used in modeling the control dynamics of real-time systems. Higher-level time-based Petri nets, moreover, have been used for associated safety/performance evaluations [7]. Furthermore, a comprehensive development methodology based on Petri nets has shown to have utility beyond application to hard real-time systems [8]. In work influential in the prototyping investigation that spawned the higher-level statechart concepts, deBondeli devised a specification technique for active servers based on predicate-transition (Pr-T) nets [9], a higher-level kind of Petri net.

Statecharts tend to be more useful for initial conceptualizations or at higher levels of abstraction than Petri nets. This is largely a separation of concerns phenomenon owing to the segmentation of statechart subgraphs and the statechart decluttering and stability afforded by separated transition rules. Hence, statecharts are well suited to control state requirements definition and prototyping. Statecharts are most prominently used in the STATEMATE ™ Methodology [10], although it is not strictly a control state oriented methodology.

**METHODOLOGICAL PROBLEM**

Software building blocks that facilitate modeling have been widely exploited in both simulation and programming languages and in prototyping environments. A combined instance is the use of Ada generic packages that export a task type as in [9]; the associated concurrency abstractions tend to be both novel and complex, not conventional ADTs like sets or stacks. Each of these task types can yield an arbitrary number of the same kind of prototype building blocks. To minimize the iteration sometimes experienced in capturing these complex abstractions, a systematic way was sought to represent broadly encompassing concurrency concepts, and then to delineate abstractions of interest [11]. This effort resulted in the formulation of higher-level statecharts (at first called token transition graphs per [1]), for basic statecharts are unable to deal with multiple objects.

Then, in applying higher-level statecharts to a variety of modeling problems, it became evident that many problems of interest required task types other than pure.

Registered Trademark of I-Logix, Inc.
servers as obtained through deBondeli's technique. Further, the nature and full scope of the encompassing concepts were often readily captured in higher-level statecharts, which posed the problem of how best to map the entire higher-level statechart to a prototype. In turn, it was noted that the various subgraphs of a superstate might represent different task types, and hence prototyping building block templates. Typically, the mode of interaction among these subgraphs was that of agents [12] in that each subgraph carried out operations in response to external requests, yet each also called upon others to perform certain operations.

This called for a somewhat different approach to constructing prototyping building blocks, because the Ada package encapsulation of an ADT was inappropriate for task types that needed to invoke external operations. Further, the aggregation of the associated software components into an Ada program would demand a somewhat different organization for bi-directional task entry calling visibilities.

**PROTOTYPING PROBLEM**

Several limited prototyping problems had previously been explored using higher-level statecharts before an attempt was made to model a full-scale system-level construct. The problem then selected was that of prototyping the global communication scheme for a real-time multicomputer system executing a single logical program. Here, tasks executing on the respective computers communicate on a many-to-one basis over a common bus through an external data repository as shown in Fig. 2. Each recipient task is assigned a specific queue, whose read- and write-operations are subject to a global event manager. If a particular queue is empty, a task seeking to obtain its contents must wait; that task's computer is subsequently notified by the global event manager when that queue is written. Actually, a queue may receive an arbitrary number of inputs from multiple tasks before the recipient task initiates acquisition of the queue's contents. At such time, the recipient task removes all the available data.

The tasks within each computer execute under their local scheduling priorities, but the local flow of control may be affected by the unavailability of data sought in the external queues. If a desired queue is empty, the requesting task is blocked. But once its site is notified that the queue has been written, the task will have to wait still longer if it does not have sufficient local priority to be scheduled immediately. System input command/sensor data are also furnished to the external data repository, and in general, the multicomputer system must operate in a reactive manner to respond to changing circumstances. Since each site is performing its part of a single logical program, data written in global queues, together with pre-assigned task priorities, will alter functional modes of execution. Thus, the embedded multitasking software is highly reconfigurable, and adequate real-time performance under a broad range of conditions, and hence different execution threads, is essential.

**GENERAL APPROACH**

To a large extent, the steps for constructing a higher-level statechart are rather the same as for a basic type statechart. The definition of the token data types is obviously necessary only for higher-level statecharts. Also, their subgraphs that are to serve as templates (for to the declaration of multiple instances) must be identified, because special notation is necessary to refer to them in the transition rules. Naturally, the transition rules themselves tend to be more complicated than for basic state-charts. Briefly, the construction steps are:

- identify the subgraph states and sketch the form of the associated graphs
- declare the transition events and relate them to the transition arcs
- stipulate which subgraphs are to serve as templates
- define the token types or associated data structures
- formulate the transition rules apart from the graphs
- examine the event sequences and cycles for desired operation
- refine the statechart iteratively via mental execution until acceptable.
Figure 3 - Multicomputer Global Communications Statechart

Note that the first two steps deal with composing the graphs, the third with scaleability of active objects, and the fourth with characterizing the types of passive objects to be operated upon during statechart execution. The fifth end sixth steps are the relatively difficult ones; but it is both easier and vastly more salutary to deal with a few pages of logic statements than to analyze and debug a complete system mechanismization! Per the last step, it is prudent to confirm its acceptability manually before undertaking the construction of the prototype. Again, debugging the statechart is generally much easier than debugging the prototype program.

It is also beneficial to reason about the concepts and behavior embodied in the statechart, for this nurtures valuable insights, and in turn a superior architecture configuration. Such insights, moreover, are very beneficial during the debugging or refining the prototype per se. Systematic execution of the prototype is of course a practical necessity in accomplishing the validation of the concepts and verification of the logic. Other type evaluations, such as performance calibrations, should be made also. Naturally, the states of the respective objects are a major source of test results data. Hence, provisions for runtime instrumentation should be fully considered during the formulation of the statecharts, for the definition of additional states may be necessary for thorough evaluation.
**type** TASK-STATUS is (BLOCKED, READY, EXECUTING);  
use T_STATUS_IO;

**package** T_STATUS_IO is new ENUMERATION_IO(TASK-STATUS);  
use T_STATUS_IO;

**type** PROCESS_MODE is (SENDING, WAITING, GETTING, TERMINATING, OTHER);  
package P_MODE_IO is new ENUMERATION_IO(PROCESS_MODE);  
use P_MODE_IO;

**type** TASK_SWAP is (SUSPENSION, ACTIVATION, PREEMPTION);  
package SWAP_IO is new ENUMERATION_IO(TASK_SWAP);  
use SWAP_IO;

**TEST_HISTORY** : FILE_TYPE;

**-- Task Object Status --**

**type** TASK_DATA_RECORD is record

PRIORITY : NATURAL;

TASK_STATE : TASK_STATUS := READY;

MODE : PROCESS_MODE := OTHER;

TIME_RQD, SCHED_EVENT_TIME, COMPLETION_TIME, ACCRUED_RUNTIME, ACCRUED_WAITING : TIME_SCALE := 0.0;

NUM_SUSP_EVENTS, NUM_COMM_EVENTS, EXT_EVENT_NUM : NATURAL := 0;

DATA_ITEM : FLOAT := 0.0;

DATA_RCV : OUR_Q.QUEUE_TYPE;
end record;

---

**Figure 4 - Token Type Data Structure**

---

ing) are seen to appear explicitly in the definition "type TASK_STATUS." Since an affected computer site may receive an interrupt upon another's writing of a global queue, task preemption may ensue if a higher priority task has previously been suspended awaiting the queue's new data. Hence, the enumeration "type TASK_SWAP" identifies the attendant states that tasks may assume (Suspension, Activation, Preemption). Most of these same labels also appear in the statechart transition rules and in the execution results. Common nomenclature is used to facilitate understandability and the interpretation of prototyping results.

The transition rules for the higher-level statechart in Fig. 3 are stated in Fig. 5. First-order logic is used because of its expressiveness, compactness, and universality. Such notation is considered preferable to specialized nomenclature, which tends to be more verbose, arcane, and limiting. Although the example transition rules might at first seem to be a bit forbidding for some, the elements of the notation are really not that difficult. What actually is complex is the content of the concurrency logic captured in these statements. The syntax of the transition rules is the same as for basic statecharts:

**TRANSITION RULE = EVENT[CONDITION]/ACTION**

**EVENTS** are logic stimuli that can initiate the execution of a transition statement; such events are noted on the transition arcs in Fig. 3. Events appear in bold lettering in the transition rules in Fig. 5 so that they stand out as the triggering ingredient in the rules. Note that events may be externally derived or else triggered in the Action-Part of other transition rules. A **CONDITION** is a Boolean expression that must evaluate True to permit the execution of a rule; thus, a condition may determine which of several options, if any, is exercised in response to a particular event. An **ACTION** is the behavior exhibited in response to a transition's firing. This frequently involves token migration(s) from one node to another as indicated symbolically by \( p_i \rightarrow R \), which means that the \( i \)-th \( p \)-token moves from its present location to Node R, or in the present case, to the Ready State in Fig. 3.

One thing that makes statechart operation so expressive is the fact that events may be signaled in the Action-Parts. Hence a single external event may excite a sequence of internal events, along with a corresponding sequence of transient states. Care must be taken, however, to preclude the spawning of conflicting or ambiguous threads of control; this can occur through the misuse of the Action-Part events. In higher-level statecharts, another
source of expressiveness is the inclusion of parameters or qualifiers in the event evocations. Hence, parameters can be passed from one subgraph to another, but only in a forward direction. For example, DEFER(i) in Fig. 5 means that the virtual clock is being instructed to cease elapsing execution time for the i-th task in u-th processor due to task suspension. The "u" in a prefix-dot notation like u.INTERRUPT is used to target an event to the u-th instance of a template subgraph, e.g., here the global event manager is interrupting the u-th processor.

Figure 5 - Multicomputer Communication Statechart Transition Rules
The rules for the TIMING Subgraph are used to specify the global absolute time base. Basically, a task at the outset of its execution time reports in to a virtual clock that it is beginning to run at the present virtual time value, and that execution will take a stipulated interval of time (variable and task-dependent). The execution of the statechart then continues according to the active transition rules until it goes quiescent. At that point, everything has happened that can occur in the short term, so the virtual clock advances by the smallest requested time increment from the current virtual time value. This jump in virtual time corresponds to the completion of some task execution(s), so the prototype is again active, for data exchange, scheduling, etc. If such an activation results in a task preemption (for higher-priority acquisition of newly written global data), the interrupted task requests that its remaining execution time span be deferred, so that the higher priority task can begin execution and record its own execution time request instead.

Organization of the prototype program developed is shown in Fig. 6. Note that the COMMUNICATION Subgraph in Fig. 3 results in a generic Ada package, GLOBAL_QUEUETYPE, that exports the task type GLOBAL_QUEUE. Multiple global queues can therefore be instantiated as needed; as task objects, these queues are protected from concurrent access. The virtual clock is
implemented in Generic Package `ABSOLUTE_CLOCK_TYPE`, whose generic parameters define the virtual clock resolution and range. The SCHEDULING Subgraph is declared as a task type in the main package, `PROCESSOR_SITES`, and expanded as Task Body `SCHEDULING`. Overall, the program topology is not as clean as is possible with server task types. This circumstance results because all agent task types must have visibility of others in order to issue calls. This is enabled here through parameterized subprograms in generic declarations that establish visibility for required procedure calls. Further, inter-agent calls must be mediated by surrogate tasks as defined in Task Body CALL_HANDLER; this precludes certain improper task blocking as possible under the Ada rendezvous rules. Still, the organization of the prototype program is reasonably straightforward, and its structuring admits an extensive use of reusable software components, like basic ADTs for sets, lists, and queues.

To complement the illustration of the correspondence between higher-level statecharts and the resulting Ada code noted in Fig. 4, an excerpt of Task Body SCHEDULING is presented in Fig. 7. The various accept statements correspond to event names in Figs. 3 and 5. The comments in the right-hand margin are numbers of the statechart arcs, which may facilitate cross-checking during code development. The acceptance of an Ada task entry call denotes the firing of a transition rule, where conditions on transition rule execution may be expressed as guards on task accept statements. Execution of the Action-Part of a transition rule is captured in the block of code following the accept clause. There, internally generated events are issued, as in the statement `CALL_AGENT.DEFER_JOB(MY_SITE_NUM, TEMP_TASK_NUM)`. This corresponds to a request to a surrogate task, CALL_AGENT, to suspend execution time elapse of the virtual clock per the DEFER(u,i) clause in Rule 4 for the SCHEDULING Subgraph in Fig. 5. Note that DEFER_JOB is a procedure that calls the virtual clock task's DEFER entry.

Execution of the Ada-implemented prototype program involved twenty-nine task objects, freely interacting according to: the transition rules; static task priorities; and Ada runtime support mediation. Accordingly, as is the case in all our previous prototypes, the main Ada program is simply BEGIN-NULL-END. There is no explicit control structure other than that implicit in the higher-level statechart model and the Ada tasking rules. Logical local clocks [13] were employed as cross-checks on the partial ordering of tasking events and the absolute timing. No automated test results processing has been implemented yet, but several promising approaches to performance calibrations/metrics are provisioned by instrumentation currently implemented in the prototype program.
CONCLUSIONS

The foregoing prototyping method is seen to aid in the recification of incomplete or imprecise specifications of concurrent real-time systems. Moreover, it does so from the very outset of system-level development, and provides a sound basis for optimizing quantitative parameters in resultant component specifications. This means that overspecification and "TBDs" can be virtually eliminated, and correct and optimized operation is enabled early-on in system development. Hence, component costs and development time/risk can be minimized, provided consistency with the verified system-level design is maintained throughout subsequent development.

In the method exemplified, it is noteworthy that the object-based prototype stems from a control state decomposition of requirements, and not a conventional object-oriented analysis. The close mapping between the class of higher-level statecharts described and the resulting Ada multitasking prototype is seen to be conducive to prototyping environment implementation. Beyond this, the mutual agents architecture represents a standardized software architecture of potential value in realizing the more "uniform" production of software products.

Higher-level statecharts, which are THE basis for this prototyping approach, are themselves a significant new formalism for conceptualizing and expressing complex, multi-object concurrency phenomena in a compact and understandable way. Moreover, their applicability appears to be far wider than indicated by the application described herein. Further, the easy incorporation of virtual time base and integral instrumentation features are rather notable prototyping capability extensions, as they enable performance assessment during system conceptualization.

On the negative side, higher-level statechart restrictions or safeguards are needed to preclude improper threads of control that can arise due to complex Action-Part events. Improvements are also desirable to enhance the reuse of prototype building blocks. This refers to the limitations of Ada library units with generic subprograms for mutual calling visibility for agents, and to the overall structure of the prototype program itself. Additionally, it is important to develop a systematic way of constructing the call handlers for mediating task communications. Since experience with higher-level statecharts is as yet limited, further enhancements and perhaps some new features may well be forthcoming.

ACKNOWLEDGEMENTS

Basically, the Ada building block approach presented issues from the work [9] of Patrick deBondeli of CR2A in Courbevoie, France. During my very rewarding year of collaboration with Patrick, Richard LeBlanc of the Georgia Institute of Technology contributed very constructively as a part-time consultant, when many of the ideas presented here were emerging [1]. All of the work described herein was performed at the Lockheed Aeronautical Systems Company in Georgia as in-house funded methods development. There, the multicomputer communication topic was framed and supported by David Smith of Lockheed's Pilot's Associate Program.

References