This paper presents the concept of a virtual machine for executing, introspecting, verifying and validating statecharts. It describes several software architecture concerns that arise in designing such a virtual machine.

Virtual Machine, Statecharts

I. A VISION ABOUT STATECHARTS

Our goal involves building a Statechart Virtual Machine (STVM) to help scientists and system engineers control space-science experiments and spacecraft operations with Statecharts. Statecharts are an extension of Finite State Machines that includes hierarchy and logical concurrency.

The short-term objective is to build an STVM that turns Statecharts into a basic commodity for spacecraft operations. Sending Statecharts to a spacecraft should become a simple routine matter; in particular, one that avoids costs and complexities associated with changing flight software. An STVM should make Statecharts as easily manageable as traditional command sequences are.

The long-term vision is to enable scientists, system engineers and operators to leverage Statecharts for controlling their instrument and spacecraft operations. The intuitiveness and expressiveness of the diagrammatic Statechart notation lends itself to a goal-based commanding approach. A Statechart defines all available ways to reach and maintain a goal state with the flexibility of alternative methods and the reliability of checking intermediate results.

To leverage the full potential of Statecharts as a highly intuitive and expressive diagrammatic notation of behavior, we need a virtual machine capability for dynamically managing Statecharts in a running system: a Statechart Virtual Machine. This capability should provide a low-cost solution for Statechart management, specifically, one that does not rely on making expensive flight software modifications. Furthermore, it should be fully accountable in the following sense: first, explaining the behavior of a Statechart throughout its execution, and second, supporting the analytic verification of formal properties across all possible executions of a Statechart.

II. RECENT EXPERIENCE WITH STATECHARTS

For the Deep Space One spacecraft, engineers designed and analyzed over 50 Statecharts for that spacecraft’s entire fault protection capability. Automatic code generation from Statecharts produced the right software for each Statechart after every change. This mechanistic consistency enabled the fault protection team to concentrate its resources on doing the right thing: designing the right fault-protection Statecharts for the spacecraft. The resulting engineering demonstrated the adequacy and applicability of Statecharts for modeling and controlling complex Spacecraft behavior. As good as this approach was for design purposes, DS1 follows the traditional model of spacecraft maintenance and operations: binary software patch and reload.

Traditionally, there is a paradigm shift after launch in flight software development and maintenance from pre-launch practices. Before launch, changes in design are reflected in flight software code changes which entail recompiling and reloading the flight software. After launch, changes in design are often accommodated by manipulating operation sequences rather than making expensive flight software changes (Fig. 1) that may result in undesirable side effects.

III. PROBLEMS TO BE ADDRESSED WITH AN STVM

The notation and semantics of statecharts defined in the unified modeling language (UML) specification includes the notion of logical concurrent states, commonly known as “AND” states in contrast with mutually exclusive states, commonly known as “OR” states. The UML semantics is not prescriptive regarding the implementation of logical concurrency, for example, in terms of serialized or parallel computations on a single or multiple threads respectively. From an independent verification and validation (IV&V) perspective, a specific software implementation of a statechart forces a bias on its actual semantics. For example, an implementation that serializes concurrent states will not support analyzing the robustness of the statechart design to alternative serializations of such states. Instead of performing an analysis of statecharts in a specialized...
language for verification & validation, we propose to design a virtual machine for statecharts that only implements the normative aspects of UML statechart semantics and encapsulate all other semantic refinements as external policies.

A. An external policy example

The execution of a state machine involves a cycle of event processing where state transitions become enabled to fire and a transition selection algorithm determines which transitions can fire simultaneously. For our STVM, the transition firing policy defines the parallelism and order in which transitions fire. Normally, Statecharts should be designed to be robust against variations of this policy. Usually, COTS tools content only with one transition firing policy, especially with regards to translating a statechart into a software program where all transition-firing decisions have been hard-coded into the program logic. In the STVM, the transition firing policy can be defined and changed from the object semantics of statecharts defined by Harel and Gery to non-deterministic firing policies to explore the possible space of executions. The latter is key aspect of applying the technology of generalized verification engine (GVE) for virtual machines. This is in-line with current research efforts at NASA Ames to apply SPIN-like verification techniques as an alternative scheduling policy to guide a virtual machine into exploring the space of possible executions of a program.

B. Architectural implications of external policies

One of the premises of the STVM is to multiple refinements of the UML statechart semantics via user-defined policies. As described above, specific policies can be defined to support the IV&V of statechart designs. Does this approach scale up to analyzing the non-determinism that arises in the interleaving of executing multiple statecharts? From a software architecture viewpoint, the analysis could be approached in multiple ways according to distribution (e.g., one STVM that executes multiple statecharts vs. one STVM for each statechart); to external policies (e.g., all statecharts governed by the same policies) and to other sources of non-determinism due to the software architecture where the statecharts execute in one or more components.

C. Multi-purpose modeling

Although the semantics of statecharts have been primarily focused on the execution of state machines for control purposes, alternative applications of state machines have been defined in the Mission Data System (MDS) project at JPL. This project is based on a number of architectural principles for organizing ground and flight software systems. One such principles involves the modeling of a system for the purposes of controlling the execution of actions, elaborating a plan of control actions to achieve a goal, and estimating the true states of the system from available sensor information. The estimation problem involves quite different semantics for state machines than the control problem. The uncertainty of knowledge about the true state of the system implies a non-deterministic interpretation of a state machine, for example, one based on probabilistic state transitions like Markov processes.

D. Runtime robustness monitoring

Statecharts have raised some controversies, for example with respect to the need of expressing logical concurrency inside a state model or outside of it. Without getting into this debate, another feature of UML statecharts, dynamic choice points, make the problem of verifying and validating statecharts generally undecidable without simulation. In a multi-segmented transition with dynamic choice points, a guard condition in one transition segment can test the results of an action performed in an earlier transition segment. Therefore, deciding whether a given complex, multi-segmented transition with dynamic choice points can be taken requires simulating the computation of the actions that performed in the state machine and in this transition in particular. On Deep Space One, although the fault protection statecharts do not involve logical concurrency (that particular spacecraft & mission domain do not require it) these statecharts make extensive use of dynamic choice points to force a chain of computations to be treated as an atomic process relative to the cycle of event processing in state machines. On Deep Space One, this modeling complexity is not intentional. While we do not claim that the state machines engineered represent a parsimonious model of that spacecraft’s fault protection system, these state machines were carefully designed and analyzed to be robust within the realm of the available tools and resources.

For statecharts with dynamic choice points, robustness analysis need not be a complex task. In fact, most of the problems could be easily found with a systematic N-cycle lookahead search strategy similar to the forward N-ply search process that chess programs perform to find the best next move. For an STVM, an N-cycle lookahead robustness analysis is only but a special twist on the traditional semantics of statecharts. While this may not replace a stronger form of robustness analysis based on proving or refuting theorems about statechart behavior, we believe that this kind of simple strategy could help statechart designers take care of 90% of their mistakes during unit testing. This will not eliminate the need of more advanced forms of robustness testing; however, it may help experienced statechart designers find modeling conventions where adequate analytical robustness can be established from the results of exhaustive N-cycle lookahead search.

IV. CURRENT STATUS

Work is under way at the Jet Propulsion Laboratory to build a prototype of a statechart virtual machine small enough to be packaged as a flight experiment on the WIRE spacecraft. This work is a precursor to a proposed collaboration effort between JPL, NASA Ames and David Harel at the Weizmann Institute in Israel.