

~~CONFIDENTIAL~~ (30 APR 2004 DRAFT)

Development of a state machine sequencer for the Keck Interferometer: evolution, development & lessons learned using a CASE tool approach

Leonard J. Reder^{*a}, Andrew Booth^{*a}, Jonathon Hsieh^{*a}, Kellee Summer^{†b}

^aJet Propulsion Laboratory, California Institute of Technology, ^bW. M. Keck Observatory, California Association for Research in Astronomy

ABSTRACT

This paper presents a discussion of the evolution of a sequencer from a simple EPICS (Experimental Physics and Industrial Control System) based sequencer into a complex implementation designed utilizing UML (Unified Modeling Language) methodologies and a CASE (Computer Aided Software Engineering) tool approach. The main purpose of the sequencer (called the IF Sequencer) is to provide overall control of the Keck Interferometer to enable science operations be carried out by a single operator (and/or observer). The interferometer links the two 10m telescopes of the W. M. Keck Observatory at Mauna Kea, Hawaii.

The IF Sequencer is a high-level, multi-threaded, Harel finite state machine, software program designed to orchestrate several lower-level hardware and software hard real time subsystems that must perform their work in a specific and sequential order. The sequencing need not be done in hard real-time. Each state machine thread commands either a high-speed real-time multiple mode embedded controller via CORBA, or slower controllers via EPICS Channel Access interfaces. The overall operation of the system is simplified by the automation.

The UML is discussed and our use of it to implement the sequencer is presented. The decision to use the Rhapsody product as our CASE tool is explained and reflected upon. Most importantly, a section on lessons learned is presented and the difficulty of integrating CASE tool automatically generated C++ code into a large control system consisting of multiple infrastructures is presented.

Keywords: Interferometer, sequencer, CASE tool, UML

1. INTRODUCTION

The Keck Interferometer is the major ground based instrument of NASA's Origins program. The goal of the program is to search for extra-solar planets. Since June of 2002 science data has been collected utilizing the Interferometer (IF) Sequencer for high-level instrument control. The IF sequencer is implemented using a subset of UML design methodology (two out of twelve UML views). The commercial Rhapsody CASE tool product provided by I-logix is used for graphical entry of the design and to automatically generate C++. Thus the C++ generated is coupled to Rhapsody's UML model. To understand the IF Sequencer structure one must first have a conceptual knowledge of the sequencer role within the overall Interferometer control system software architecture.

The entire control system software consists of a hierarchy of state based controllers (figure 1)¹. These consist of various high-speed real-time embedded controllers based on a JPL framework developed specifically for real-time Interferometer control (known within this paper as the "JPL RTC Toolkit"²). Slower functionality within the system utilizes legacy Keck Observatory telescope control infrastructure built on the Experimental Physics and Industrial

* Leonard.J.Reder@jpl.nasa.gov; phone 1 818 354 3639; fax 1 818 393 4357; Jet Propulsion Laboratory, MS171-113, 4800 Oak Grove Drive, Pasadena, CA 91109

† ksummers@keck.hawaii.edu; phone 1 808 885 7887; fax 1 808 885 4464; W. M. Keck Observatory, 65-1120 Mamalahoa Highway, Kamuela, HI 96743

Control System (EPICS). The overall operation of the system is provided by automation implemented in the IF sequencer at the top-level. The IF Sequencer is a software program designed to command the various objects of the "JPL RTC Toolkit" (that provide real-time servo control of subsystems components such as Fast Delay Lines (FDLs), Fringe Trackers, etc.). High-level commands are sent to the two large Keck telescopes by way of telescope sequencers that isolate the IF sequencer from the complexity of the various Keck telescope subsystems. All components must be commanded to perform their work in a specific and sequential order; though not in the hard real-time domain. The IF Sequencer is the highest level of control in Keck Interferometer software control system.[‡]

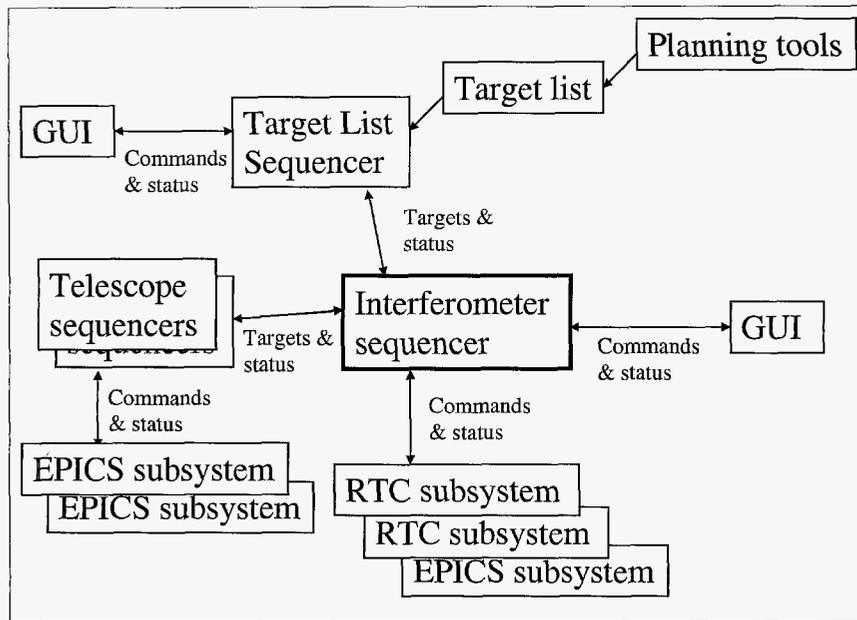


Fig. 1 Sequencing hierarchy

In the following sections of this paper the history of the development and evolution from a simple EPICS implementation to the object oriented design currently being deployed at Keck are discussed. Our evaluation and selection of the Rhapsody CASE tool (which was essentially motivated by the automatic state machine code generation capability) is presented. And finally some lessons learned and a conclusion given.

2. HISTORY

During the development of the "JPL RTC Toolkit" and subsequent implementation of Keck Interferometer specific software (that is built on the tool kit) there had been much discussion of how to control the lower level subsystems in a unified and consistent way. Experience with PTI (Palomar Testbed Interferometer) suggested that some form of high level control program be developed. At PTI a sequencer was implemented to run within the VxWorks operating system to perform overall control of the system.

The W. M. Keck Observatory had been using State Notation Language (SNL) that runs within a general sequencer control program that is part of EPICS. This sequencer program was being used for Keck motion control as part of the Keck AO system. It was January 2001 and the EPICS sequencer had been recently ported to UNIX and looked to be a viable solution to the sequencing problem. There was also a certain amount of pressure to use the W. M. Keck legacy approach and existing organizational standards. First fringes with the instrument were required for March of that year

[‡] The current implementation of the Interferometer sequencer includes a primitive version of the Target List Sequencer functionality implemented between IF sequencer C++ and the GUI. The Target List Sequencer is intended to further automate the systems and allow timed observations of groups of star targets. At the time of this writing the implementation of the Target List Sequencer has not begun.

so an initial version had to be rapidly implemented. The first version (Increment 0) of IF Sequencer was a simple UNIX EPICS sequencer written in SNL that generated periodic sidereal delay values corresponding to a star target being observed. These values are sent to "JPL RTC Toolkit" developed Fast Delay Line controllers via CORBA commands. Since the SNL language is an only extension to conventional C the implementation required wrapping CORBA interfaces with a simple C code API. A library for computing delay line target values used at PTI was reused. And a simple TCL GUI was written that used legacy Keck Keywords to command and monitor the Increment 0 EPICS based IF sequencer. This scheme was successfully used for acquiring first fringes.

In the course of the EPICS SNL development it was quickly realized that building state machines purely from procedural compiled code would be cumbersome and not scale well with complexity. It would have certainly been possible but not "pretty". So we set out to explore other solutions. First we tried a more object oriented approach by using TCL with the object oriented extension called Incr TCL. Incr TCL allows classes to be defined within the TCL scripts. This approach allowed us to code both CORBA client and server side functionality into our TCL. Both JPL and Keck had significant expertise with TCL so there where no big culture shocks associated with the technology. A small test sequencer consisting of a simple finite state machine was coded in TCL to drive two test siderostats that were in place at Keck for testing. We used the State design pattern of reference [3] as the basis for the script. Although using a scripting language made it easy to modify and test, again, it was quickly realize that with more complex state machines the code would become difficult to maintain. It was also noted that a some additional development effort would be required to implement a multi-threaded framework that would be required.

At about the time we where considering these solutions, a colleague suggested using a CASE tool to automatically generate state machine codes by first representing them as UML state chart notation. The JPL Deep Space One technology demonstration mission had evaluated the use of a product called Rhapsody (manufactured by I-logix Corp.⁴) but decided that the tool would not work well for there application. We examined the I-logix website and quickly decided that Rhapsody was worth serious consideration. After seeing an on-site presentation and demonstration of Rhapsody we started to build basic evaluation sequencer models, this proved to be more difficult than expected, but without other options we decided using Rhapsody as our development tool was an improvement over-hand coding. More about the evaluation in the next section, but from the initial prototype work, we where able to build the first deployed IF sequencer (Increment 1) for the purposes of visibility squared science operations.

The Increment 1 version consisted of UML models entered into Rhapsody; only the static class diagram and state chart views where used. C++ code was auto-generated for execution on the Solaris operating system since real-time was not a requirement, the Rhapsody tool was coupled to the ACE/TAO CORBA ORB that the "JPL RTC Toolkit" is built on and a set of helper classes to wrap primitive "JPL RTC Toolkit" functionality where designed. Included in these helper classes was the notation that the IF sequencer would be hierarchical with some sort of communications infrastructure sending messages from one state machine to another. The idea adopted early on was that for each Interferometer subsystem there would be first a mid-level state machine for direct control (thus we have a state machine for each major intereferometer low-level subsystem, Fringe tracker, Fast Delay Line, Angle Tracker, etc.) and then a high-level state machine which commands each of the mid-level ones. The experience gained from this design and prototyping stage was useful in the implementation of Increment 2 that is described in section 4.

While our initial development using Rhapsody was going on, a parallel effort was going on to port the "JPL RTC Toolkit" to the Linux-RTAI platform in summer of 2001. This was done under a research grant provided by NASA under the Advanced Information Systems Technology (AIST) program. The "JPL RTC Toolkit" had been evolving as well. Eventually there was a port to the Sun Solaris operating system.

It was therefore logical that our next version (Increment 2) leverage and reuse the newest "JPL RTC Toolkit" infrastructure (Fig. 2). The "JPL RTC Toolkit" consists of a collection of libraries and three executable programs. The libraries provide a white box framework for building specific servo control objects. The framework includes support for configuration and telemetry. The executables programs are Telemetry and Configuration servers and a CPU Manager. The Telemetry server provides a publish/subscribe telemetry implementation (via CORBA event channels). A configuration server provides a link to a database of persistent storage so that any parameter within IF Sequencer can be independently configured at run-time.

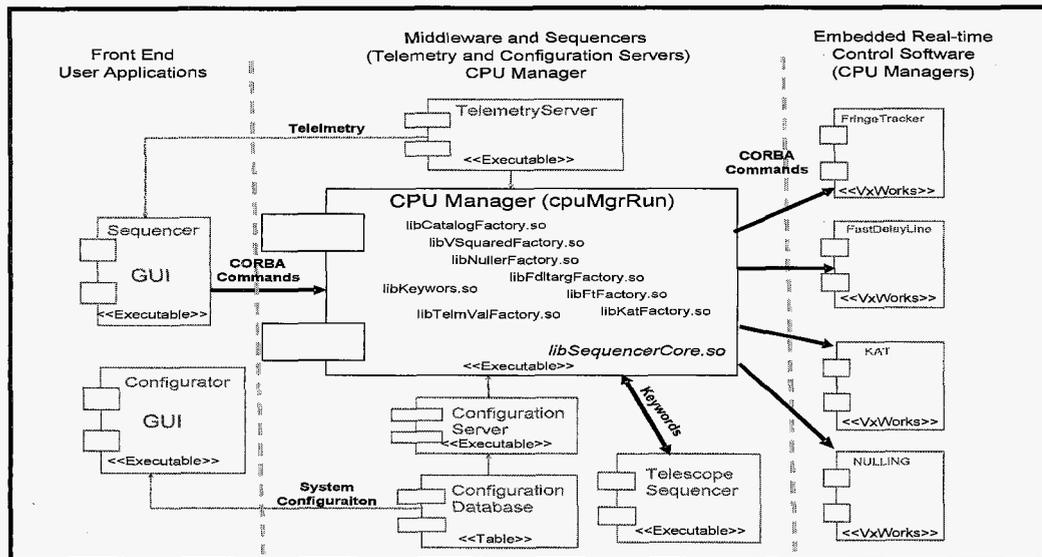


Fig. 2 Component diagram of IF Sequencer Configuration with JPL RTC Toolkit

Figure 2 shows a UML component view of the IF Sequencer within the “JPL RTC Toolkit”. The use of the CPU Manager allows one to load and instantiate specific objects that contain state machine code. The CPU Manager is CORBA enabled as are all IF Sequencer state machine objects (known as RTC Managed objects since they are loaded and instantiated by the CPU Manager). The advantage of this scheme over the Increment 1 is that object instantiation is no longer hardwired. The application can be reconfigured for various modes of operation. This makes reconfiguration for Visibility Squared, Astrometry, Nulling, Differential Phase and Imaging modes possible. More on the specific UML object oriented framework is presented in section 4.

3. EVALUATION

Initially, our only thought was to automatically generate state machine code. We all had a rather naïve view of CASE tool technology and what it was and could do. Thus we did not compare Rhapsody (Fig. 3) with other tools available in 2001. There was considerable schedule pressure and no time for comprehensive evaluations. So immediately a simple, proof of concept, Fringe Tracker state machine sequencer was implemented. A great deal was learned about the tool and what was became the basis for our UML model framework.

Immediately it was discovered that Rhapsody was much more than a simple state machine code generator. People had the impression that non-programmers would easily use the tool as well and this was quickly realized to be impractical during our evaluation. Moreover, Rhapsody is, as we quickly learned, a UML design and code generation tool in a general sense. Rhapsody supports the entire set of UML diagram types⁸. Figure 3 shows a screen snapshot of Rhapsody. The UML state chart a high-level state-machine visibility squared sequencer is in the right hand window and an over all tree view of the code model is in the left window. It was quickly learned that groups of classes could be designed graphically from which thousands of lines of code automatically generated.

Code generation from a conceptual graphical model is inherently a tricky thing to do! It gets even trickier when third party legacy frameworks must be integrated. Issues arise that you would never think of when hand-writing code. There are specific manglings that one might want to inflict upon source in special circumstances. Rhapsody generated source has its own “coding style” as well. Unfortunately, this style is inherently different from that of “JPL RTC Toolkit. Modification of this style based on an incredibly large number (hundreds) of configurable properties within the Rhapsody product is possible. The properties allow one to change everything from the graphical appearance of the model to how code is automatically generated and more. Although the scheme provides great flexibility when making modifications to generated code; it also causes an equal amount of confusion and frustration to the developer trying to change something specific about the source code. This property scheme is annoying when first used. To modify

something in the auto-generated code it requires the correct property to be changed. To modify a property, you are required to memorize (or hunt for) it. The nice feature is that properties follow the hierarchy of your model in scope but still can be difficult to keep track of. For all the inconvenience, they have enabled the Rhapsody tool to be very flexible and without this feature we would not have been successful in integrating Rhapsody with our existing “JPL RTC Toolkit” framework classes.

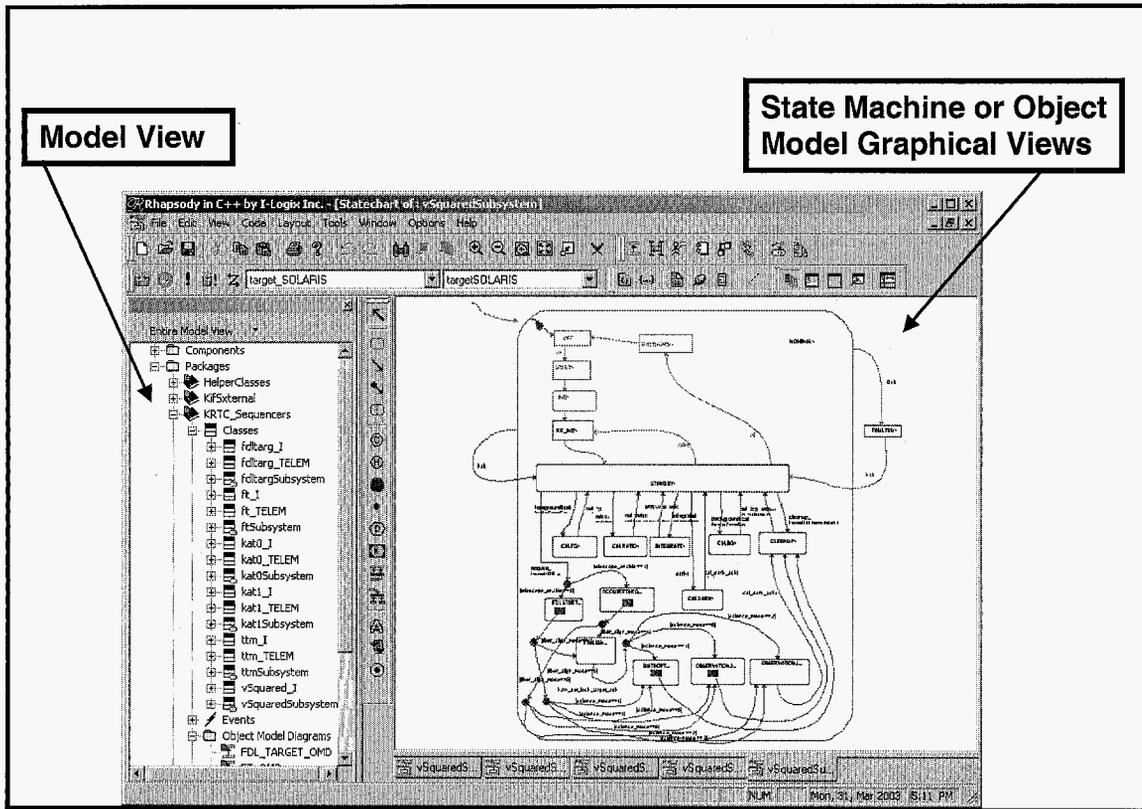


Fig. 3 Screen shot of Rhapsody CASE Tool showing model view on left and Harel statechart of visibility squared science observation sequence state machine subsystem class on right.

The process of using the Rhapsody tool for us was to first enter UML and then spend much time exploring the properties and entering code within the UML that implements interfaces and uses the Rhapsody supplied framework. Our core functionality of state machines is embedded into particular classes, the details of which are discussed in the next section. Within these state machines one would select states and enter code into specific text entry areas, the resulting generated code would contain this hand entered code in specific areas.

Creating code using the methodology of first entering graphical representations to generate framework code and then entering particular fragments of code to provide specific interfaces and algorithm functionality is quite daunting and extremely difficult for the developer that is accustomed to line-by-line coding. With much patience and work one can become proficient with the technology. So our evaluation resulted in the conclusion that auto-generating code for implementing state machines was not by any means easy, but would be more appropriately characterized as doable and more manageable over the entire life cycle of our software than any of the other approaches we tried. And so we moved forward.

4. OBJECT ORIENTED DESIGN (THE UML MODEL)

Figure 2 shows a conceptual component view of the IF sequencer as it exists within the “JPL RTC Toolkit”. At the far left are the Keck specific components that provide fast servo control directly to Interferometer hardware. These are

running on VxWorks single board computers as “JPL RTC Toolkit” cpu manager tasks. Each of these VxWorks components can be thought of as a set of objects known as RTC gizmos.

The IF sequencer basic functions are, sending commands to RTC gizmos (e.g. lower-level subsystems) through CORBA and transmitting and receiving data through a “JPL RTC Toolkit” Telemetry Server (top center of Fig. 2). The sequencer monitors telemetry items in order to detect responses from subsystems there where commanded. Telemetry is generated by the IF sequencer to update status on front-end GUIs that consists of a Python script (right side of Fig. 2).

The Configurator GUI/Configuration Server/Configuration Database as shown in Fig. 2 are used to provide dynamic run-time configurability to the Sequencer. At the center of Figure 2 is the “JPL RTC Toolkit” CPU Manager component. Within the CPU manager framework are standard interfaces for dynamically loading shared libraries than finding and/or creating instance objects defined by these libraries. The implementation of the IF sequencer consists of a set of shared libraries. All of the state machines and support infrastructure is compiled in to a single library called libSequencerCore.so and several different lib*Factory.so libraries. Normally the objects would have been separated into individual libraries but because we are using the Rhapsody supplied event communication framework rather than CORBA for state-machine object to state-machine object communications all objects had to be consolidated into a single shared library. Then for each “JPL RTC Toolkit” managed object we implemented a factory library that provided the interfaces needed by the CPU manager framework.

The sequencer is organized as a hierarchy of classes both in a static sense and in a dynamic sense. Figure 4 shows the static class diagram. Each of the most specialized classes is a standalone state machine. State-machine objects have names ending in suffix of “Subsystem”. We call the state machine objects from here on subsystem classes.

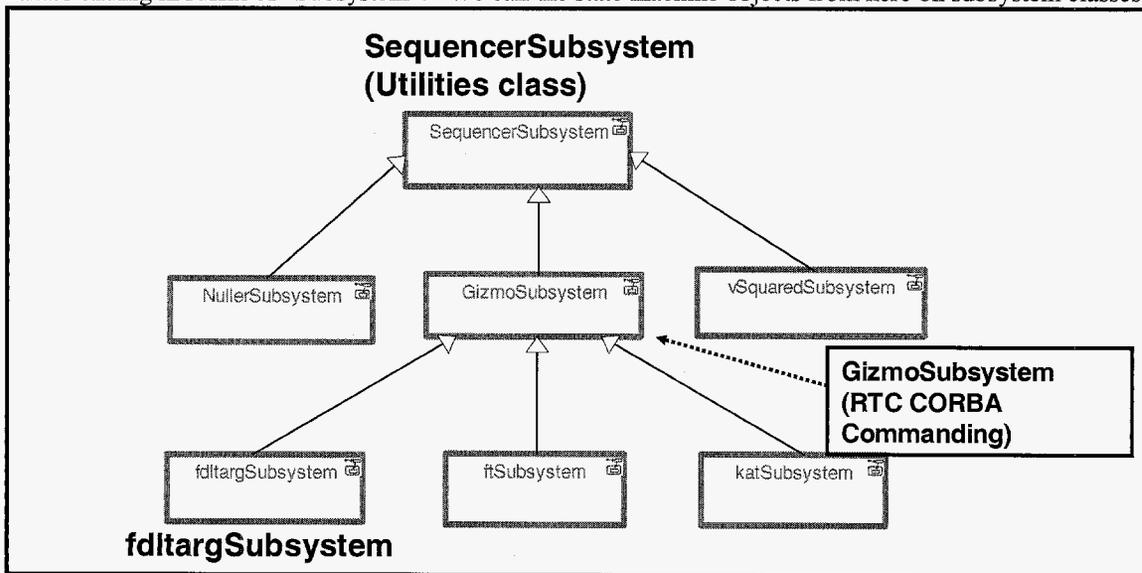


Fig. 4 Static class diagram of subsystems class hierarchy

The Sequencer is composed of several mid-level subsystem classes controlled by a single high-level state machine class. Each of the mid-level subsystems, such as fdltargSubsystem (the FDL sidereal target generator), is itself a state machine that acts as the interface to one or more RTC Gizmos. There is one high-level state machine for each science mode of the Keck Interferometer. Currently, only the Visibility-Squared mode (vSquaredSubsystem) state machine has been implemented. A new high-level state machine for Nulling operations (NullerSubsystem) is in its initial stages of development at the time of this writing. Each of the mid-level state machines has been designed with reusability and extensibility in mind. The same fdltargSubsystem class used by the vSquaredSubsystem can be instantiated for use with a high-level NullingSubsystem class with no modifications. As more observing modes come online, more high-level state machines will be implemented, similarly, as more RTC Gizmos are implemented more mid-level subsystem objects will be needed.

There are many commonalities among the Subsystem classes that make up high-level and mid-level subsystem classes. In Fig. 4 the **SequencerSubsystem** and **GizmoSubsystem** classes are utility classes that implemented the supporting functionality that enables subsystem classes to send events, push and monitor various types of telemetry and find, connect and command an RTC Gizmo.

Although our current implementation only has two levels of subsystem classes there can be more levels created if desired. A hierarchical approach seemed to make sense since the sequencer functionality always is expressed as some composition of subsystems. Further, a hierarchical (tree) model promotes less confusion in an event model (such as the one included with Rhapsody) because events can only be passed up and down the tree, not sideways.

4.1 SequencerSubsystem Class

In the IF Sequencer, the **SequencerSubsystem** class provides the foundation for all subsystem (state-machine) objects. It is the base class but not abstract. All other subsystem classes are derived from this class. Figure 5 shows the **SequencerSubsystem** class dependencies. Note that the **Collection**, **ManagedObject** and **ManagedObjectImpl** classes are not actually implemented within the Rhapsody UML model view of Fig. 5 but are stubs referencing externally implemented code within the core “JPL RTC Toolkit” shared library. The two classes stereotyped as <<CORBAInterface>> are IDL⁶ (Interface Definition Language) that define top-level interface methods that all subsystem classes contain.

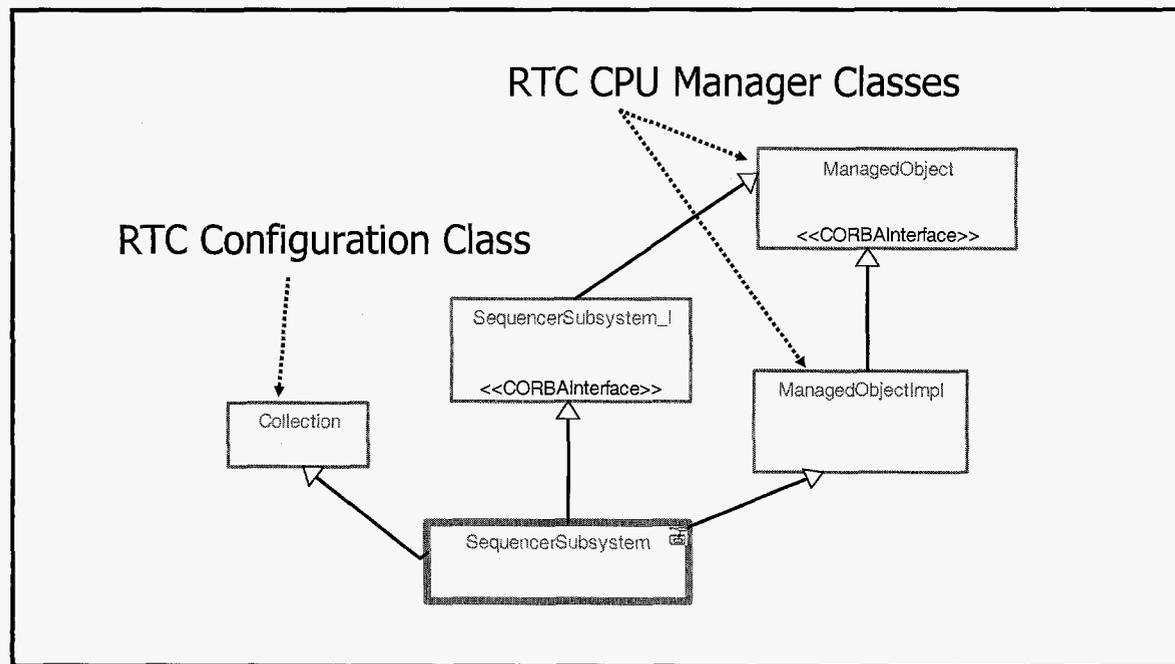


Fig. 5 SequencerSubsystem utility class relationship to RTC Toolkit.

The SequencerSubsystem class defines a state machine that serves as the basis for building application specific subsystem classes (shown Fig. 6a). More on this in a moment, but first there are utility functions implemented within SequencerSubsystem class that need to be described.

SequencerSubsystem defines several methods to implement a subsystem hierarchy of event communication. The rule we adopted early on was the topology is restricted to a single parent and any number of child SequencerSubsystems. Rhapsody events are passed to the parent through the *Upstream* method and broadcast to all children through the *Downstream* method. The purpose of the *Upstream* method is to inform a higher-level parent state machine of important events such as exceptions without knowing the exact type of the parent. Thus, an fdltargSubsystem instance can propagate a fault event up the hierarchy without knowing if its parent is a vSquaredSubsystem or a

NullerSubsystem. The Downstream method enables the opposite functionality: broadcasting an event to all child subsystems regardless of exact type. For example, a vSquaredSubsystem can send an on event to all of its connected children.

Finally, *PushStatusTelemetry* is used by GUIs or other clients to update data on internal state of the **SequencerSubsystem**. This method pushes the most recent value for all attached telemetry items. It is also called by when after an object has been configured. This way, client applications are informed of configuration changes as soon as they happen.

The **SequencerSubsystem** class inherits from **Collection** to support Configuration use and from **ManagedObjectImpl** in order to be interfaced with the CPU Manager framework.

4.1.1 RTC Managed Objects

Every subsystem class (Fig. 5) within the IF Sequencer is derived from **ManagedObjectImpl** and has an associated factory class derived from **ObjectFactoryImpl** that make them "JPL RTC Toolkit" managed objects. Within the framework, objects are created by factories. When the managed object is created by the factory a unique id is registered with an **ObjectManager** (another "JPL RTC Toolkit" object) within the cpu manager. That Id contains both Name and Type information, which are used everywhere the subsystem needs a Name or Type reference. Thus and object already created within the cpu manager can readily be found.

4.1.2 Configurable Objects (Collection Class)

Figure 5 shows that the **SequencerSubsystem** class inherits from the "JPL RTC Toolkit" **Collection** class. The **Collection** class provides an aggregation relationship with two "JPL RTC Toolkit" template convenience classes; **Entry<*>** and **Array<*>**. These classes provide the interface to the Configuration Server for run-time configuration of parameter within the subsystem class. The template type argument can be any of the IDL defined primitive types. When a new configurable parameter is desired, a new attribute of type **Entry<*>** or **Array<*>** is added to the subsystem class model. The attributes act like conventional primitive type but are configurable.

The Configurator GUI at the bottom left of Fig. 2 is used by the user to set values and execute reconfiguration of subsystem object instances within the IF Sequencer.

4.1.3 CORBA Command Bindings

Each of the mid-level subsystem classes inherit from the **GizmoSubsystem** class that provides capability for resolving and binding references to RTC Gizmos registered in a CORBA Name Service. The **GizmoSubsystem** class has an association with a global singleton running asynchronously in a separate thread called the **GizmoManger**. The **GizmoManager** contains a pair of methods for connection and reconnection to RTC Gizmos. Each class derived from **GizmoSubsystem** overrides the pure virtual *instrumentInit* method, which calls a *bindGizmo* method of the **GizmoManager**. This method will asynchronously find a Gizmo in the CORBA Name Service and correctly bind the reference to the **GizmoSubsystem** data member. If at any time the Gizmo becomes unavailable, **GizmoSubsystems** calls the **GizmoManager** *reBindGizmo* method, which continuously attempts to find the Gizmo again. Since **GizmoManger** is running in a separate thread the subsystem object thread will never be blocked with connection retries.

4.1.4 Telemetry

The IF sequencer utilizes the "JPL RTC Toolkit" publish/subscribe telemetry infrastructure via an intuitive, efficient, multi-threaded interface to the Telemetry Server for both supplying and consuming telemetry items. **SequencerSubsystem** derived classes publish telemetry in order to update status on GUIs and in archiver software; they subscribe to telemetry items to monitor behavior of RTC Gizmos under control. Telemetry channel names are

hierarchical. A set of convenience methods in **SequencerManager** class reduces instantiating a telemetry supplier or consumer to a single (long) line of code.

4.1.4.1 Supplying Telemetry

The *CreateSupplier* and *CreateStructSupplier* template methods in **SequencerManager** return a new instance of a “JPL RTC Toolkit” object called **StructSupplierImpl** or **SupplierImpl**. These are the fundamental client side objects used to connect to a unique telemetry channel (one of this objects is instantiated for each channel named). The overloaded operator “=” is then used to publish (or push) telemetry based on a publishing mode. For example one can instantiate a **SupplierImpl** reference called **MyTelemetrySupplier** and then push **MyValue** to it using the statement “**MyTelemetrySupplier = MyValue;**”. The publishing mode is set at construction time and can be changed externally by configuration. Currently publish modes of OFF, the publishing is disabled, FULL_RATE published at full rate, VALUE_CHANGE sending of telemetry over channel happens only on a value change.

4.1.4.2 Consuming Telemetry

In the “JPL RTC Toolkit”, the final destination of telemetry is a user-defined filter class. The **SequencerManager** *CreateFilter* and *CreateStructFilter* methods return a new instance of a “JPL RTC Toolkit” **Filter** (handler) class. These references can be used to set a consuming mode analogous to the publishing mode of **SupplierImpl** instances. The telemetry infrastructure uses a push/push (CORBA event) channel model that allows subscribers to passively wait for data (e.g. monitor). The subscriber to a telemetry channel defines a **Push** method within ones handler. The methods will be called whenever telemetry moves through the subscribed channel. The **Push** methods function is to typically generate Rhapsody events that cause a state machine (subsystem object) to respond to the received telemetry value. An example of this is the fringe tracker state machine class (**ftSubsystem**) where an aggregation relationship to a **LockSecsFilter** class is established. The **LockSecsFilter** class implements a **Push** method that tests telemetry representing the amount of time a fringe tracker has been locked onto a fringe during an observation. When this exceeds an internally set time an event is generated to the **ftSubsystem** state machine that commands it to sequence the fringe tracker to stop tracking.

It is important for consumers to process incoming telemetry data quickly so as not to tie up the limited resources of the Telemetry Server. This is accomplished through the use of an **AsynchronousDispatcher** to handle telemetry distribution with a separate pool of threads.

4.1.5 Derived Classes

Figure 4, above, shows the set of derived classes in the IF Sequencer. These are the **NullerSubsystem**, **vSquaredSubsystem** high-level automation (observation mode state machines) and the **fdltargSubsystem**, **ftSubsystem** and **katSubsystem** mid-level automation (for direct sequencing control of RTC Gizmos). More are being developed as needed. The **fdltargSubsystem** delivers pre-computed sidereal targets (delays settings) to update fast delay line positions and control them; the **ftSubsystem** controls the fringe tracker and the **katSubsystem** controls the (Keck) angle trackers. Each of these implements a specialized state machine (subsystem) class (e.g. **fdltargSubsystem** is an example shown in Fig. 6b) that is derived from the base state machine (Fig. 6b) implemented within the **SequencerSubsystem** class. Derived classes automatically contain the basic state chart functionality.

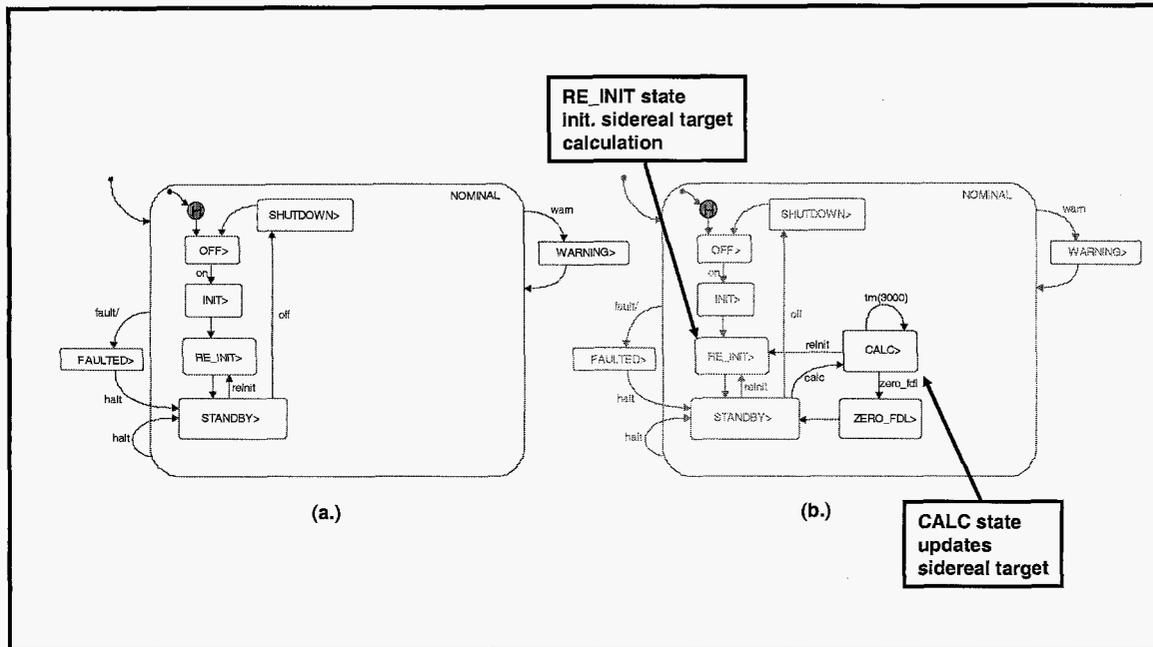


Fig. 6 Harel State charts (a.) base state machine implementation, (b.) fast delay line target generator state machine (note b. inherits a.).

The base state machine (Fig 6a) contains five states within a NOMINAL composite state to provide standard behavior to all derived state machine objects. The only default behavior in each of these five states is to push a structured telemetry item providing information about current state, last state and status messages about operation, to an external software component such as a GUI that has subscribed. The external WARNING and FAULTED states are used to handle exception behavior. Faults are propagated to higher and lower-level state machines as necessary.

Derived classes define custom CORBA IDL interfaces by inheriting from **SequencerSubsystem_I**. An example of this is shown in Fig. 7 for the **fdltargSubsystem** class. Recall that **SequencerSubsystem_I** inherits from **ManagedObject**, as shown in Fig 5. Thus **SequencerSubsystem_I** provides all the methods in the interfaced defined by “JPL RTC Toolkit” managed objects in addition to defining the *On*, *Off*, *ReInit*, *Halt*, *SimulateOn*, and *SimulateOff* methods as controls of its basic (Fig. 6a) state machine behavior. The *On*, *Off*, *ReInit*, *Halt* methods generate Rhapsody events that cause corresponding state transitions; the *SimulateOn* or *SimulateOff* methods turn on and off, respectively, a simulation test mode implemented in every derived state machine object. The idea of the simulation mode is to provide a capability for testing every state machine object in a stand alone unit test way independent of (and without connection to) the real time system.

Because each state machine object is loaded and instantiated by the RTC CPU Manager the Rhapsody generated state machine objects do not have specific knowledge about how they are related. Rhapsody events are passed from state-machine to state-machine by an inverse invocation scheme thus each state machine must have a pointer to the state machine it wish to send an event to. This means that direct associations from state machine to state machine for the purposes of sending events has not been established and must be. Within our UML we create a static class diagram view similar to that of Fig. 4 but this time it defines associations allowed by type. To establish instance linkages another IDL method called *LinkSubsystems* is implemented. *LinkSubsystems* accepts a sequence of managed object Ids, which specifies a number of other objects that must be referenced. A reference to each object is looked up in the cpu manager’s **ObjectManager** class and saved in a Rhapsody attribute. The caller of *LinkSubsystems* can also be configurable so that the hooking up of object instances becomes fully configurable as well (typically the caller is a GUI program or startup script).

4.2 Delay Line Target Generator State Machine Object (fdltargSubsystem)

Perhaps the simplest mid-level state machine class implemented is the **fdltargSubsystem** (shown in Fig. 7), thus it is useful to examine the implementation. An instance of **fdltargSubsystem** is a flexible state machine that can control single or multiple pairs of fast delay lines. The primary purpose of **fdltargSubsystem** is to compute and update FDL delay target positions every three seconds so that fringe tracking can be maintained on a sidereal target (star) moving across the sky. This functionality is implemented in Fig. 6b; there is also a set of IDL methods (not shown in Fig. 6b or 7) that allow fundamental commanding (e.g. Idle, Track, etc.) of the FDL RTC Gizmo. The input to the **fdltargSubsystem** class is a catalog record of coordinates for the desired observation that is generated by a planning program called getCal⁷.

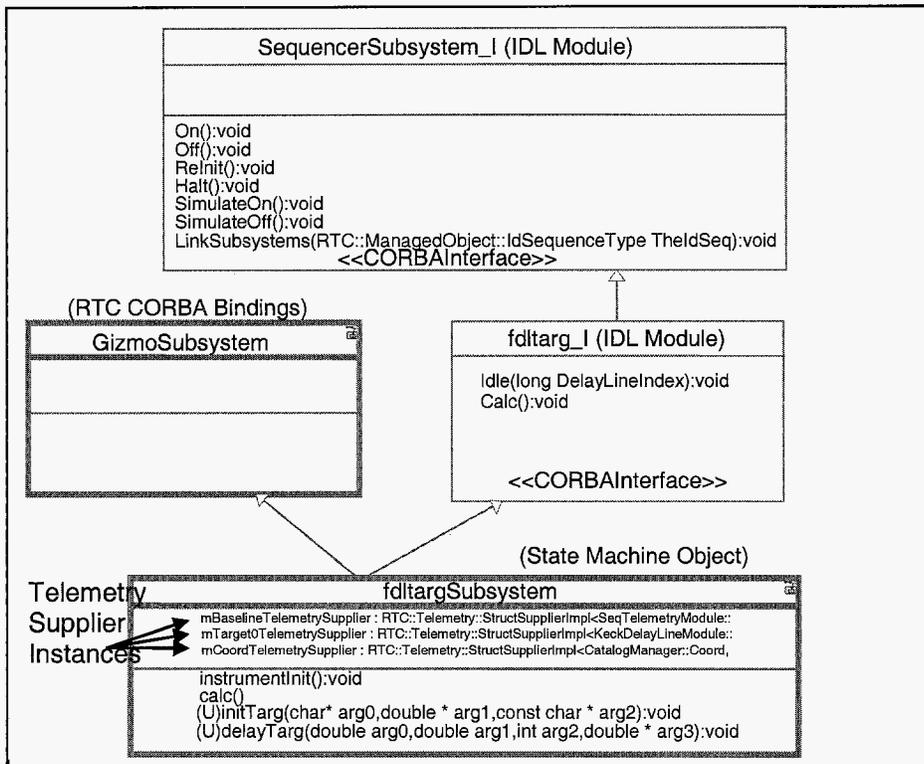


Fig. 7 Fast Delay Line Target Generator (fdltargSubsystem) static class diagram

While **fdltargSubsystem** example is perhaps one of the simplest state machine implementations in IF Sequencer observations cannot be made with the instrument without at least one instance of this state machine running. The state machine implementation (Fig. 6b) adds only two states; CALC and ZERO_FDL. The CALC state updates delay target values for specific star position and executes RTC Gizmo CORBA target update operations. This is repeated every three seconds as denoted by the `tm(3000)` in Fig. 6b, which is an internal Rhapsody timer thread event that causes a 3000 millisecond delay. The ZERO_FDL state is use to send zero targets to both FDLs effectively parking them at mid range.

5. LESSONS LEARNED

During the past several years that we have been exposed to CASE tool technology we “think” a few things have been learned along the way. What follows is a list of lessons learned from our efforts to date, there are without question more to be learned about this technology!

1. It is hard to integrate multiple infrastructures and even harder to integrate them into a CASE environment. Although Rhapsody had hooks for adding external infrastructure dependencies this was by no means easy to figure out. Corollary: On new CASE tool development projects do not mix, and reverse engineering is hard.

2. "Old programming habits die hard,"⁸ It is hard to get use to the graphical/coding UML methodology of the CASE tool environment if you have been a line-by-line coder for years. Corollary: Tool is powerful and useful but learning curve steep.
3. Maintainability of code is "vastly" improved. Because C++ code is coupled to UML graphical model it is more understandable. The use of UML in this manner automatically limits the amount of reverse engineering of code required later in the software life cycle.
4. A CASE tools real strength lies in its ability to standardize software development across a large team of engineers. However, using Rhapsody as an individual does have benefits; the state machine-based design of classes is well-suited to development of interferometer sequencers, however, files sizes of auto generated code get large fast and this leads to long compiles times. Corollary: Standardized auto-generated code leads to better infrastructure!
5. Version control of UML models between only a few developers problematic.
6. One of the original attractive features of Rhapsody was animation of state machine functionality but this never was completely functional our Solaris operating system so the more traditional gdb was used for debugging. Corollary: Features in tool may look good during sales demo but be careful.
7. When starting one should evaluate and not guess at a CASE tool to use; there are lots of choices today.
8. Consider a process for development and then a tool.
9. Templates were not included with Rhapsody case tool so we implemented them externally and hooked them into tool. This is a kludge and confusing to a new developer. This effectively defeats rule 3 above.
10. The urge to use many of Rhapsody's framework features has sometimes overwhelmed the wiser inclination to use more standardized tools such as the STL (C++ Standard Template Library).

6. CONCLUSION

The main difference between interferometer sequencers comes down to multiplicities of baselines and instrument subsystems. The IF Sequencer subsystem (state-machine) classes are highly configurable and meet this requirement.

The IF Sequencer simply orchestrating subsystems, we do not have particular hard real-time requirements for timing performance. Thus we believe that while the CASE tool approach is superior to hand coding, we tend to feel that a compiled solution is not warranted. Perhaps this application would more properly reside in an auto-generated interpreted language implementation. With a scripting language, the designer will be rewarded with less time compiling changes. Currently run-time configuration is our only means to minimize the amount of compile time required.

Error detection and recovery is an interesting problem; our current design implements an error scheme that is recoverable via operator intervention only. This was at the users request and inevitably there will be AI planning and fault recovery methods used on the Interferometer sequencing problem in the future.

Rhapsody is an excellent tool, especially for state machine design. Except for having to compile, the problem fits nicely into the infrastructures provided by I-logix and the "JPL RTC Toolkit".

ACKNOWLEDGEMENT

The work performed here was conducted at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. The authors would like to thank the following people who contributed to this effort: Mark Colavita for helpful definition and support during the evaluation of the CASE tool and development; Kevin Tsubota for assistance resolving W. M. Keck Observatory software issues.

REFERENCES

1. Andrew Booth, et. El., Overview of the control system for the Keck Interferometer, SPIE Advanced Telescope and Instrumentation Control Software II Conference 4848, Waikoloa, HI. August 2002.
2. T. Lockhart, RTC: a distributed real-time control system toolkit, SPIE Advanced Telescope and Instrumentation Control Software II Conference 4848, Waikoloa, HI. August 2002.

⁸ Douglas Schmidt, Using Design Patterns, Frameworks & CORBA, January 23-25, 2002, UCLA Extension Course.

3. Erich Gamma, et. El., Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
4. I-logix website <http://www.ilogix.com/> has Rhapsody product information and white papers.
5. Philip C. Irwin, R. L. Johnson, Real-time control using an open source RTOS, SPIE Advanced Telescope and Instrumentation Control Software II Conference 4848, Waikoloa, HI, August 2002.
6. Michi Henning and Steve Vinoski, Advanced CORBA Programming with C++, Addison Wesley, 1999.
7. getCal -- Interferometric Observation Planning Tool Suite, Michelson Science Center.
<http://msc.caltech.edu/software/getCal/> .
8. Grady Booch, et. El. The Unified Modeling Language User Guide, Addison Wesley, 1999.