Key software architecture decisions for the Automated Planet Finder

Kyle Lanclos[a,b], William T. S. Deich[a], Bradford P. Holden[a], S. L. Allen[a]

[a]University of California Observatories, 1156 High St., Santa Cruz, CA, USA;
[b]W.M. Keck Observatory, 65-1120 Mamalahoa Hwy., Kamuela HI, USA

ABSTRACT

The Automated Planet Finder (APF) at Lick Observatory on Mount Hamilton is a modern 2.4 meter computer-controlled telescope. At one Nasmyth focus is the Levy Spectrometer, at present the sole instrument used with the APF. The primary research mission of the APF and the Levy Spectrometer is high-precision Doppler spectroscopy. Observing at the APF is unattended; custom software written by diverse authors in diverse languages manage all aspects of a night's observing.

This paper will cover some of the key software architecture decisions made in the development of autonomous observing at the APF. The relevance to future projects of these decisions will be emphasized throughout.

Keywords: Automated telescopes, software design

1. INTRODUCTION

1.1 Software design decisions

Any modern observatory facility requires extensive software for monitor and control of its systems. A facility that provides autonomous and remote (AR) observing depends on additional software that is not required for ordinary "attended" observing.

This paper covers some of the key software design decisions enabling the use of the APF telescope in first a remote and then an unattended observing mode. The principal subjects addressed are personnel and facility safety (sections 2, 3, and 4), ensuring consistency in the facility software (section 5), and enabling remote troubleshooting (section 6).

The APF is operated unattended in a largely robotic mode. Daytime visits occasionally occur for preventative maintenance and other service activities, and other visitors may enter the dome on an irregular basis. Facility systems must enable unattended operations while providing sufficient safeguards for personnel and facility safety. Operational rules must address potential conflicts such as closing the dome in the event of bad weather yet preventing the closure if there is a clear personnel safety risk.

We focus on the three major APF software components that collectively enable AR observations:

1. The *checkapf* application provides safety-enforcing tasks that would otherwise be carried out by a staff telescope operator. It monitors the external observing environment and ensures that the dome is closed when the weather is poor. It also monitors the internal environment, and ensures that the telescope and/or dome do not move when an unauthorized person enters various parts of the facility.

2. The *apfmon* application provides general monitoring of the facility's condition. It normally alerts human staff to problems, and where appropriate will also take corrective action.

Further author information: (Send correspondence to K.L.)
K.L.: E-mail: klanclos@keck.hawaii.edu

3. The *apftask* system provides the infrastructure to enable software to carry out a range of daily and nightly tasks that would otherwise be carried out by observers or observatory support staff. For example, one software task adjusts the air conditioning setpoint during the day to the estimated night-time temperature and shuts off the air conditioning in the early evening once the dome opens to equilibrate for the night. Another task focuses the telescope; yet another carries out an observation of a target. There are fourteen such tasks as of the writing of this paper.

These capabilities are not unique to facilities that provide robotic observing, and a modern observatory may well take advantage of these kinds of capabilities whether or not it supports AR observations. But a successful AR observing facility must provide these capabilities with a particularly high degree of robustness and completeness.

1.2 The APF telescope

The Automated Planet Finder (APF) at Lick Observatory on Mount Hamilton is a modern 2.4 meter computer-controlled telescope. At one Nasmyth focus is the Levy Spectrometer, at present the sole instrument used with the APF. The primary research mission of the APF and the Levy Spectrometer is high-precision Doppler spectroscopy. Please refer to the introductory paper by Steve Vogt[1] for further details beyond the summary content in this section.

Figure 1. The APF at twilight, with the lights of San Jose in the distance

The dome for the APF facility is an IceStorm-2 enclosure, designed and constructed by Electro Optical Systems (EOS).[2] The dome is slaved to the azimuth axis of the telescope, and will track any/all motion in azimuth so long as the dome is powered. The dome slit has two independently-movable shutter panels, which can be 'parked' in an up-and-over mode that nearly parks them out of view of the telescope, or in arbitrary positions to provide wind shielding for the telescope.

Designed and constructed by EOS Technologies (EOST), the telescope has a 2.4m f/1.5 primary diameter mirror on a Alt-Az mount. To maximize optical collecting area, the secondary mirror and its support structure were kept small, such that only 2% of the central area was obstructed. One consequence of a small secondary is a reduced field of view, which for the APF is roughly one arc minute. This implies a requirement for the blind pointing of the telescope, which has a root-mean-square (RMS) pointing error of less than three arc seconds and

a worst case pointing error of about 12 arc seconds. The telescope nominally slews at 2-3 degrees per second, a relatively high speed which reduces the loss of observing time while moving to a new target.

Designed and constructed by University of California Observatories (UCO), the Levy Spectrometer is engineered for high resolution spectroscopy. The Levy is mounted at the left Nasmyth port of the APF telescope, attached directly to the telescope yoke; the spectrograph thus never changes orientation with respect to gravity, and the input beam remains stable as the telescope tracks in azimuth. The wavelength calibration for the Levy is provided by an an iodine cell[3] inserted into the focused beam after it enters the spectrometer. The total system throughput is 15% when the iodine cell is in place; though the APF has only 1/12 of the light gathering surface as the 10-meter Keck 1 telescope, the high efficiency of the Levy spectrometer means that exposures on like targets are only 6 times longer than similar exposures taken with the HIRES[4] spectrometer on Keck 1.

The respective vendors provided telescope and dome control software, each running on a dedicated Windows XP host. All other software components run on a small set of Linux hosts with specific hosts dedicated to control systems and to observer needs. The Keck Task Library (KTL)[5,6] application programming interface (API) is used for all day-to-day interprocess communications, including monitoring and control of all system components.

The extensive use of a common API (here, KTL) was a key design decision that greatly simplified the architecture of software that must span a wide range of systems. This diversity includes the vendor-provided software for the dome, telescope, and guide camera; Galil and Omega hardware controllers; the SNMP interface for the APC battery back-up system; Davis and Vaisala weather stations; a homegrown CCD controller. This diversity also includes all software layered on top of these components, including the autoguider system, dome temperature control software, the telescope schedule, and many others.

2. PERSONNEL AND FACILITY SAFETY: CHECKAPF

We used our experience with remote observing at Lick Observatory's Nickel 1-meter telescope[7] to shape our approach towards first enabling remote observing with the APF and then on to enabling autonomous nighttime operations. This included some re-use of existing software, but more important was the re-use of lessons learned.

One key experience at the Nickel was that most observers quickly abdicated their role in monitoring conditions once software was provided to assist in that process. In response our software evolved to take full responsibility for monitoring conditions and closing the dome as necessary. A smaller number of observers tried to disregard the published observing limits and keep the dome open in inappropriate conditions. This experience led us to adjust the oversight software to adopt a uniformly strict, conservative approach regardless of whether the facility is operating with local or remote observers. This directly informed the approach taken for the APF and led us to be stricter still when operating autonomously.

The slow growth of checkapf's responsibilities over time drives against a very strong design goal to keep checkapf as simple as possible and to limit its operational dependencies. checkapf is the safety watchdog for the APF facility; should some other aspect of the control software fail to perform its due diligence we rely on checkapf to restore safe operating conditions. Keeping checkapf simple reduces the frequency of defects, which in turn reduces the risk that checkapf will be unable to enforce personnel and facility safety.

The design goals for checkapf and its sibling software all include certain features which are effectively universal for safety systems:

1. Safety systems should be as simple as possible to improve auditability and reduce the risk of defects.
2. Changes to a production safety system should be subject to a high level of scrutiny and testing.
3. Safety systems should be backed by a failsafe trigger that activates when the safety system is offline.
4. Random failures will occur: it must be possible to responsibly override faulty inputs.

2.1 Key permissions for all operations

The checkapf software continuously monitors telemetry throughout the facility and updates three permission states accordingly. Autonomous corrective actions to ensure personnel and facility safety are taken by checkapf as necessary in response to changes in permissions.

The design goal for the permission scheme is to ensure personnel and facility safety while maximising scientific output. The permissions must be appropriate to the actual use of the facility and not interfere needlessly with nighttime operations; if too many false positives occur our observers would surely begin to subvert the permission scheme as a practical means towards achieving their observational goals. While recognizing the need for balance the permissions must be structured such that the default action prevents autonomous control systems from creating hazardous conditions for personnel entering the dome, and the facility itself must be protected from hazardous environmental conditions.

Permission	Coverage
`INSTR_PERM`	All hazards inside the Levy Spectrometer
`MOVE_PERM`	All hazards on and including the telescope structure
`OPEN_OK`	All hazards related to opening the dome

Table 1. Summary of permissions in checkapf.

The permissions are summarized in table 1. Each permission is broadcast as a boolean KTL keyword. The truth value for each keyword coresponds to a positive response; for example, if the `checkapf.OPEN_OK` keyword is true this implies it the user is allowed to open the dome. Constructing permissions in this fashion allows aggregation of multiple permission keywords in a simple logical conjunction, as might be necessary to decide whether observing should be attempted.

The granularity of the permissions is a direct reflection of the operational needs of the facility. The original permission scheme was a single boolean value; the legitimate operational needs of the facility prompted the creation of additional permission states. Conflicts arose when it became clear that some actions could be performed safely in circumstances where the global permission state had been revoked. For example, if the front door of the dome is unexpectedly opened, the safety system recognizes that personnel (such as cleaning staff) could be working exclusively on the ground floor of the facility and limits telescope operations accordingly. However, as long as the door to the upper floors remains closed, it is safe to continue operating the spectrometer. Increasing the granularity of the permission scheme is a small price to pay for preserving observing efficiency.

The APF safety system depends on a mixture of strictly enforced rules and cooperative processes. This is not ideal; it is preferable to implement safety systems using strictly enforceable rules that cannot be bypassed by other control systems. The APF's vendor-supplied software systems attempt to implement robust controls by requiring software to acquire access rights before controlling a system, and allowing a more-privileged account to override the access of a less-privileged system. At first blush this provides an effective basis on which to implement a reliable system. In practice, however, it was not useful in a day-to-day work environment, due to the clumsy permission system, the lack of visibility into who or what was currently had acquired access rights to some component, and the difficulty of temporarily and safely increasing a request's privilege level.

The limitations of our budget and schedule led us to develop the current system in which the permission keywords are at the heart of a cooperative processes system. The underlying hardware control daemons (dome, telescope, spectrometer, and so on) are controlled through standard interfaces that do not have built-in knowledge of external permissions. However, no application, except for the master safety systems, directly commands the hardware control daemons. Instead, they use an alternative interface that checks for permission for each request before sending the request to the daemon. This is described in detail in section 4.

2.1.1 Instrument control permission

The instrument control permission (`checkapf.INSTR_PERM`) governs all components inside the fiberglass enclosure of the Levy Spectrometer. Instrument control is typically only revoked if there is an unresolved personnel safety conflict or if the instrument has not been released to the observer (see section 2.5 below).

2.1.2 Telescope movement permission

The telescope movement permission (`checkapf.MOVE_PERM`) governs all exposed mechanisms on and including the telescope that could be dangerous to personnel inside the dome. This includes movement of the telescope in azimuth and elevation, movement of the secondary and tertiary mirrors, and repositioning the primary mirror cover. Shutting off power to the telescope (including the software emergency stop) are similarly governed by this permission; if movement permission is not granted it is likely that checkapf will require power to enforce facility or personnel safety, thus external software must be prevented from disabling power in unsafe circumstances.

Movement permission is typically only revoked if checkapf is in the process of executing an unplanned closure, if there is an unresolved personnel safety conflict, or if the telescope has not been released to the observer (see section 2.5 below).

2.1.3 Opening-dome permission

The dome opening permission (`checkapf.OPEN_OK`) governs all components of the dome whose operation could expose the facility to external hazards. These components include the dome shutters and vent doors.

The dome opening permission is revoked if the weather is bad, if the sun elevation is high enough to expose the primary to direct sunlight, or if there is an unresolved personnel safety conflict.

There are always tradeoffs between simplicity and fine-grained control of systems. For example, if there were individual permission states for each vent door, it would be possible to allow opening of just the vents that are facing away from the late afternoon sun while ensuring the other vent doors remain closed. Doing so would allow autonomous software to partially open the dome during the late afternoon to assist with cooling the telescope and primary mirror when both predicted nighttime temperature and the current ambient temperature drop below the capacity of the dome chiller. But it would also nearly double the number of permissions keywords, just to enable this simple capability.

2.2 Autonomous reactions for personnel safety

All of the following reactions are triggered by the revocation of one of the permissions listed in section 2.1 if and only if personnel safety is compromised. The guiding principle for these reactions is to minimize the risk to personnel. In circumstances where there is no conflict between observing efficiency and safety, allow observing activities to continue; safety concerns always outweigh observational goals whenever a conflict is present.

2.2.1 Instrument control restrictions

There are two doors for the APF: the first is the exterior door to the facility and the second is the interior door to the stairwell leading to the second and third levels. Both doors have mechanical switches that indicate whether the door is closed. If the stairwell door is opened the permission for instrument control will be immediately revoked. Ideally this would only occur if the fiberglass shell surrounding the instrument were opened, but there are no switches installed to indicate whether that shell is closed. Revoking instrument control ensures that mechanisms inside the Levy Spectrometer will not move and that calibration lamps will not be turned on.

2.2.2 Stopping the telescope

Personnel are directly exposed to movement of the telescope in azimuth on the second level and in elevation on the third level; personnel are exposed to the co-rotating section of the floor on the first level when the dome rotates in azimuth.

Absent any overrides the telescope will stop motion immediately if the door to the stairwell is opened. Absent any overrides the telescope will also stop motion if the exterior door is opened and the co-rotating floor moves more than a foot as the dome follows the telescope motion in azimuth.

2.2.3 Closing the dome

If the stairwell door is opened the vent doors and dome shutter will immediately close in order to reduce the risk of injury. The vent doors close quickly, well before someone could reach the top of the stairs; closing the dome shutter takes more time: the mirror covers are closed first, which requires slewing the telescope to the zenith. The same sequence of operations to minimize risk to the facility also minimizes risk to personnel: slew the telescope to the zenith, close the mirror covers, and then close the dome shutter. A person going up the stairs would first be exposed to the movement of the telescope, which should be done slewing to the zenith well before that person could walk up both flights of stairs and be above the azimuth bearing. That person would then need to climb a ladder to be exposed to any risk from the closing mirror cover; to be exposed to the dome shutter they need to traverse an additional catwalk around a Nasmyth focus.

Closing the vents and shutter early eliminates a conflict between protecting personnel and protecting the facility. Once a person is on the third floor, the automated system cannot safely take actions such as closing vents, in case the person is reaching through a vent door (an unlikely but possible event). If a weather event were to occur the automated system would be unable to safely close the facility without jeopardizing human safety. Therefore, without overrides, the system will automatically close the facility when the stairwell door is opened, before a person could reach the upper floors.

2.3 Autonomous reactions for facility safety

All of the following reactions are triggered by the revocation of one of the permissions listed in section 2.1 if and only if facility safety is compromised. The guiding principle for these reactions is to minimize the risk to the facility. It is not possible to preserve observing efficiency in these circumstances, and the reactions are structured accordingly. Note the subtle contrast between the steps outlined here and those used to close the dome when personnel safety is a concern: when facility safety dominates, the objective is to restore facility safety as quickly as possible; when personnel safety dominates the response may take more time overall, but is structured to minimize the risk of bodily harm.

2.3.1 Facility safety: closing the dome

In the event of weather or other facility safety concerns the first action taken is to close the dome shutter. In order to minimize the risk of any objects or debris falling on the exposed primary mirror the telescope is first slewed to the horizon before moving the shutters, as slewing to the horizon is always faster than closing the mirror covers.

2.3.2 Facility safety: closing the mirror cover

Once the dome shutter closure is complete (regardless of success or failure) the mirror cover will be closed. Closing the mirror cover changes the balance of the telescope, making it significantly more bottom-heavy. In order to mitigate the risk of an uncontrolled move the telescope is first slewed to the zenith before initiating the closure of the mirror cover. Having the mirror cover closed is especially important for weather-related closures to mitigate the risk of damage to the primary by precipitation leaking through the dome shutter.

2.4 Actions not taken

In no circumstances will checkapf power down the telescope or set the software emergency stop. Either of these conditions would prevent commands to the dome shutter and mirror cover from succeeding; losing those capabilities when operating autonomously creates an unacceptable risk to the facility.

2.5 Releasing control to the observer

An essential concept in checkapf's model is the deliberate delegation of control from the telescope technician at the summit to the remote observer. checkapf has two release states: one for telescope control and one for instrument control. Each may be released or revoked individually at any time. External modifications to the release state can only be made via a password-controlled interface.

The telescope release state is automatically revoked every day at noon. Telescope technicians must deliberately release the telescope in order for twilight and nighttime operations to begin.

There are no automated changes for the instrument release state. The Levy Spectrometer is completely enclosed in fiberglass and remains closed unless mechanical or electrical work is performed inside the enclosure. Keeping the instrument released for use by the observers allows afternoon calibrations to commence without requiring prompt attention from personnel on-site; instrument and focus changes at other telescopes can impact the availability of assistance during the afternoon when calibrations would otherwise begin.

An unreleased telescope or instrument would be reflected in the associated permissions listed in section 2.1.

2.6 Explicit declaration of user type

Checkapf contains a notion of which type of user is presently using the facility, and adjusts its behavior accordingly. The different user types are *local*, *remote*, *robotic*, and *tech*.

- A *local* user is afforded more flexibility than remote or robotic users. Because the user is on-site there is no inactivity timeout. Automated safety responses remain active.
- Remote user. A *remote* user is similar to a *local* user, but must also periodically assert that they are present and ostensibly monitoring the facility. This assertion is made by pressing a button in a graphical user interface to reset a deadman timer. If the timer expires the dome and mirror covers will close automatically. Any subversion of this timer mechanism, such as writing software to virtually press the button on behalf of the observer, is considered grounds for loss of observing privileges.

  Remote observers must also provide contact information to the system before permission will be granted to use the facility.
- The rules for a *robotic* user are similar to the *remote* user except that resetting the deadman timer is expected to be performed automatically by the robotic software in control of the facility.
- The *tech* user mode disables all of the automated safeguards provided by checkapf. This is typically the user mode employed when personnel are on-site for maintenance activities. To ensure personnel safety a physical emergency stop button is typically also engaged upon entering the dome. The *tech* mode will also be used in situations where the facility needs to be under computer control for testing; this allows opening the dome and mirror cover without triggering the safety checks built into checkapf.

3. MONITORING ARBITRARY CONDITIONS: APFMON

Where the permission rules are kept deliberately simple and sequestered from the rest of the software infrastructure, there is also a need to create conditions of arbitrary complexity: providing alarms for undesirable states, automatically correcting anomalous conditions, or creating derived data products. One simple example of the latter is the average strut temperature for the space frame inside the Levy spectrometer; a more complex example is a thermal control loop that attempts to maintain that same average strut temperature at a constant value. Both of these examples, and many, many others in between, are implemented using an in-house software monitoring tool, *emir*.[8]

As with any general monitoring tool the basic premise is that any undesirable condition should be watched for and appropriate actions taken in response to a change in state. The monitoring system was initially configured to track system performance with respect to documented requirements, such as telescope tracking accuracy, or iodine cell temperature; all hardware failure modes that could be identified in advance, such as weather station availability, or failures in the battery backup system; and unexpected observing configurations, such as incorrect settings for the guide camera. The set of monitored conditions grew steadily over time as additional problems and failure modes were discovered empirically, from an intial set of about 100 conditions to about 400 conditions as of October 2015.

For conditions that can be corrected automatically this represents a direct and complete resource savings in that no direct intervention or acknowledgement is necessary to remedy a problem when it occurs. For example, one such action ensures the operational safety of the secondary mirror, disabling motion of the mirror actuators if the tip/tilt of the secondary exceeds a rational limit. For conditions that require human intervention the

time must still be spent to correct the problem, but timely notification can greatly reduce the impact on a night's observing: receiving timely notification that persistent software has unexpectedly exited can spell the difference between losing 20 minutes of time and losing an entire night. Alerts are invaluable around the clock, not just during observing hours: the APF dome has a very small footprint, and if the air conditioning fails, the temperature can quickly rise to levels that can cause electronics failures; therefore apfmon includes monitoring and failure notification for both the facility chiller and critical temperatures.

3.1 Example rule: dome cooling not set correctly

The rule shown in figure 2 demonstrates a few basic capabilities of *emir*. This rule inspects the `eosdome.FC1` keyword; if it has been disabled for more than 600 seconds it sets an internal `FC1_STATE` condition to a warning. In all other circumstances the condition is OK.

```
#
# Ground floor cooling should always be on:
#
Condition FC1_STATE "FC1 Status" {
    case: { $(eosdome.FC1 == DISABLED) && $(age:eosdome.FC1) >= 600 } {
        status: { WARNING "Level 1 cooling unit has been OFF for\
                                  [expr {$(age:eosdome.FC1)/60}] minutes" }
    }
    default: {
        status: { OK "" }
    }
}
```

Figure 2. Example *emir* configuration for an environmental check

There are similar conditions established for the cooling systems on the second and third level but the considerations are more complex: it is perfectly legitimate for the cooling on the upper two levels to be disabled while either the vent doors or dome shutters are open.

The checks for each of the three levels are then aggregated into an additional condition that sends e-mail if any of the cooling system checks are not in an OK state. This aggregate rule is shown in figure 3. This rule demonstrates the syntax used to send an initial e-mail about a change as well as the syntax used to send periodic reminders that an undesirable condition persists.

```
Group FC_STATUS "Level 1,2,3 combined cooling state" \
                        { FC1_STATE FC2_STATE FC3_STATE } {
    case: { $(FC_STATUSSTA != OK) } {
        email: { onchange* "APF: cooling not set correctly"
                "APF cooling is misconfigured:\n\n$(FCSUMMARY)\n\
                \n[ute::emsg FC_STATUS]" }
        remail: { 60 "APF: cooling is still misconfigured"
                        "APF cooling is misconfigured:\n\n$(FCSUMMARY)" }
    }
    default: {
        email: { onchange+ "APF: cooling configuration ok"
                "APF cooling on levels 1,2,3 now set reasonably:\n$(FCSUMMARY)" }
    }
}
```

Figure 3. Example *emir* configuration for an e-mail warning

3.2 Example rule: stopping secondary mirror runaway

The rule shown in figure 4 is a building block for more complex rules, similar to what was shown for the cooling systems above. There are three actuators (A, B, and C) that combine to control the piston, tip, and tilt of the secondary mirror for the APF telescope. Each actuator has its own motor and encoder, and each actuator can be moved independently of the others. The actuators are connected to the secondary by rigid rods which can fail catastrophically if the actuators are driven to an extreme imbalance. A failsafe mechanism does exist that drops power to the actuator motors (and the rest of the telescope) if the imbalance gets too extreme. However, recovery from this failsafe condition is difficult and prone to error; improper recovery could result in permanent physical damage to one or more actuator rods. To make matters worse at least one of the motors is not properly configured and has a tendency to drive its actuator when it should be idle. For these reasons a software monitor was deployed to stop a runaway actuator before the failsafe can be triggered.

The rule in figure 4 monitors a single actuator, confirming that it is in a state where the motor is in a control mode where failure is a possibility, that the actuator is in motion, and finally compares its absolute position against the other two actuators.

```
Condition M2_A_OK "Detect/stop M2 Actuator A runaway" {
    case: { $(eostele.FAENABLE == ENABLED) &&
            $(eostele.FASSTATE) eq "Disabled" &&
            abs($(eostele.FAENCVEL)) > $(M2VEL_THRESH) &&
            ( abs($(M2AB_DIFF)) > $(M2DIFF_TOLV) ||
                    abs($(M2AC_DIFF)) > $(M2DIFF_TOLV) ) } {
        status: { ERROR "M2 actuator A runaway" }
    }
    default: {
        status: { OK "" }
    }
}
```

Figure 4. Example *emir* configuration to monitor one of three actuators

The rules for the three actuators are then aggregated into a single rule to automatically stop the secondary if a runaway is detected. This response is taken by modifying a KTL keyword that immediately disables all motion of or on the telescope, which includes the actuators for the secondary mirror. This aggregate rule is shown in figure 5.

Further details and examples are available in the SPIE paper[8] on *emir*.

3.3 User interface

emir is designed around heirarchies of conditions. The keywords provided by *emir* are structured to allow automated top-down traversal of any *emir* instance. Among other benefits this enabled the creation of a universal graphical user interface for any *emir* instance.

All conditions use a consistent set of status values, ranging from 'OK' to 'ERROR'. By default the interface will create a series of tabs, each tab displaying the worst status for all conditions contained within that tab. This allows warnings and errors to percolate to the top level display; the interface lets you drill down to the individual condition(s) in an error state.

Further details and screenshots are available in the SPIE paper[8] on *emir*.

4. HONORING PERMISSIONS PROACTIVELY

All telemetry and commands can be expressed as KTL keyword/value pairs. Many of these keywords support both read and write operations, and in the case of mechanical systems, writing a value to a keyword implies a command to physically move the mechanism. It is neither realistic nor good practice to expect that every

```
Group M2RUNAWAY "Detect/stop M2 Actuator runaway" {
    M2_A_OK M2_B_OK M2_C_OK
} {
    case: {$(M2RUNAWAYSTA >= ERROR)} {
        modify: { eostele.DISABLE $(eostele.DISABLE.TRUE) }
        email: { onchange "APF: M2 actuator runaway detected."
                    "The M2 encoder differences exceed tolerance.\
                    \n\nThe tolerance is $(M2DIFF_TOLV)\
                    \nThe M2 encoders are now:\
                    \n    A: $(eostele.FAENCPOS)\
                    \n    B: $(eostele.FBENCPOS)\
                    \n    C: $(eostele.FCENCPOS)\
                    \n\nTheir differences are:\
                    \n    A-B: $(M2AB_DIFF)\
                    \n    A-C: $(M2AC_DIFF)\
                    \n    B-C: $(M2BC_DIFF)\
                    \n\n[ute::emsg M2RUNAWAY]"
        }
    }
}
```

Figure 5. Example *emir* configuration to automatically stop a runaway motor

application will know and apply the correct permission for each and every one of its KTL keyword modification requests; expecting all such applications to do so is a sure way to have mismatched implementations of the permissions scheme. The KTL API does not provide any direct assistance in restricting such requests, such as mandatory acquisition of rights or similar features. To work around these issues we replaced the standard keyword-modification interface with a "safe" version. The safe version performs a table lookup of constraints (each implemented as KTL keywords themselves) that apply to the keyword in question, queries the current values of any such constraints, and only if all of the checks succeed is the original modification request issued.

This *proactive* approach for honoring constraints is one key use of the established permission scheme, but it is only half of the equation. While all individual subcomponents are *expected* to honor the overall permission structure, we cannot *assume* that this will be the case. Thus, the preventative initial approach is combined with two flavors of reactive sentinels running in the background. The first sentinel is internal to the permission system (checkapf, see section 2) itself: if it sees that a severe condition has arisen, such as rain falling on an open primary mirror, checkapf will immediately and repeatedly issue corrective commands, ignoring the possibilty that other software may interfere. The second reactive sentinel integrates with the overall task structure (see section 5); this second sentinel will immediately shut down any task whose critical permission has been revoked.

The combination of both a proactive and a reactive approach has been very successful in ensuring both personnel and facility safety at the APF.

4.1 Description of constraints

The table of constraints is written in a standard INI file syntax as parsed by the standard Python ConfigParser module. Each section of the file is named for a KTL service; each option name within that section is the name of a keyword within that service; the values specified for a keyword are the constraints. Glob patterns may be used to specify keyword names. An example of the restriction syntax is shown in figure 6.

Note there is a departure here from the description in section 2 on checkapf: the inclusion of additional permission keywords beyond those maintained by checkapf. The easy expansion of the list of constraints allows flexibility in deciding when commands should be issued without the need to modify checkapf itself.

The additional constraints shown in figure 6 relate to slews of the telescope. The intent is to lock out repeated slews of the telescope after enough consecutive failures occur. Figure 7 demonstrates another additional

```
[eostele]
# Secondary focus, tertiary position, telescope position.
FOCDC?:   checkapf.MOVE_PERM
FOCTIP:   checkapf.MOVE_PERM
FOCTLT:   checkapf.MOVE_PERM
FOCUS:    checkapf.MOVE_PERM
HOMEALL:  checkapf.MOVE_PERM, apftask.SLEW_ALLOWED
NTOFFSET: checkapf.MOVE_PERM
PARK:     checkapf.MOVE_PERM, apftask.SLEW_ALLOWED
SLEWTARG: checkapf.MOVE_PERM, apftask.SLEW_ALLOWED
```

Figure 6. Sample constraints for the eostele KTL service

constraint: it may be safe to move specific components of the telescope if the weather is bad (and presumably the dome is closed), but in no circumstances should the mirror cover be commanded to open in inclement weather.

```
[eostdio]
# Emergency close, mirror covers.
ECLOSCMD: checkapf.MOVE_PERM, checkapf.OPEN_OK
MCOVCMD: checkapf.MOVE_PERM, checkapf.WX_OK
MCOVCLOS: checkapf.MOVE_PERM
MCOVOPEN: checkapf.MOVE_PERM, checkapf.WX_OK
MCOVSTOP: checkapf.WX_OK
```

Figure 7. Sample constraints for the eostdio KTL service

4.2 Implementation of constraints

The *show* and *modify* programs are the standard command-line interface used to retrieve telemetry and issue commands (respectively) via KTL keywords. Both are implemented in C using different aspects of the KTL API. KTL Python is a Python-native interface for the KTL API, and includes an object-oriented interface that was not part of the original KTL standard.

The implementation of the constraints is a module function written in Python. Many of the tools used with the APF are themselves written in Python and can efficiently invoke that function directly. The remainder of the applications that would leverage this functionality are written in various shell scripting languages and would normally use *show* and *modify* as needed. In order to satisfy the needs of such scripts a Python *safe_modify* command-line program was created that honors the same command-line syntax as the original *modify* program.

The structure of *safe_modify* is unremarkable except that it goes to some length to do "just in time" handling of all KTL keywords. Most invocations are modify requests for a single keyword, and that keyword will typically only be covered by a small subset of the constraints enumerated in the full list of restrictions. Only those keywords whose values are immediately relevant to the current request are initialized or otherwise inspected. Even when the additional inspection is minimized it can still double the time required to process a non-blocking request; the additional startup costs are not as apparent for a blocking request as the extra delay is typically small compared to the time spent blocking. Long-running programs using the native Python function at the core of *safe_modify* are not sensitive to these one-time startup costs.

5. MUTUALLY-AWARE AUTONOMOUS TASKS

There are two main components to replacing human-controlled observing with an autonomouse observing system: ensuring safety of facility and personnel; and replacing the observer when carrying out the tasks that make up an ordinary observing night. The previous sections have addressed the safety elements of APF. This section discusses the system that replaces the human as observer.

An example recipe for observing at the APF was originally sketched in 2005 along the following lines:

- Fetch observing targets list.
- Focus the instrument.
- Take a series of "wide flats".
- Check the spectrum with the Thorium Argon lamps.
- Check the iodine cell.
- Take "darks".
- Take science observations of targets.
- *etc*

Each of the above items is naturally recognized as a single task to execute. Many of the tasks can be broken down into component sub-tasks; we decided to formalize these individual tasks under a single *apftask* structure. This simplifies the monitor and control of tasks, since they all use a similar set of states and state transitions, and similar sets of keyword names for monitoring and controlling those states. This also simplifies the assembly of larger tasks out of components: the higher-level task doesn't have to be fully aware of how each task operates, so long as each individual task honors the common semantics. Finally, it can simplify programming: by leveraging the main control framework task authors can focus on their critical objectives and do not have to deal with implementing their own control interface.

Conceptually tasks can be anything, from a movement request for a single motor up to the highest-level software component orchestrating all activities associated with observing. In practice we found the most useful granularity for tasks is at about the level of each of the steps of the above observing recipe: one task to focus the instrument; another to focus the telescope; another to close the dome in a "best practices" sequence. There are no hard and fast rules, however, and some other tasks operate at a higher or lower level.

We chose to implement tasks as exclusive, in that only one instance of a task may be running at a given time. That rule matches well with our normal use: one would not want two tasks trying to focus the telescope at the same time.

Most tasks have a definite start, execution period, and completion. That is, they act as one-shots, rather than persistent daemons. In accordance with their role as steps in an observing "recipe", one task finishes, and then the next task begins. There are a few persistent tasks where the integration as an offical task provides otherwise absent monitoring capabilities, enforced exclusivity, or easy integration with critical safety features.

5.1 Scripts as tasks

Our original concept for tasks expected that each task would be implemented inside a traditional persistent daemon. The daemon would receive the *start* command for a task, execute the task code, and announce the completion of the task, but the daemon itself would not exit. A significant and highly consequential evolution in tasks was that many were instead written in high-level languages, particularly shell scripts.

There were two key consequences to using shell scripts for task implementation. First, a wider range of people could write tasks, and it became practical for non-expert programmers to write tasks. This allowed astronomers to closely participate in writing the observing systems. Second, in the absence of persistent daemons, written by professional software developers, the apftask system had to supply tools to both enable scripts to behave properly and to enforce proper behavior.

When a persistent daemon is used to implement tasks the designer assumes that the daemon will follow all the rules for starting, executing, and stopping tasks. On the other hand, when a standalone script executes a task, it is much harder to ensure that all rules are followed. For example, the checkapf system may need to order all tasks to pause execution (see section 5.4 below). Tasks are expected to monitor their invidiual *control* keywords and pause on demand. Shell scripts *(a)* have very limited ability to detect asynchronous events such

as a keyword change, and *(b)* generally do most of their work using external commands, which themselves will not know anything about how or whether the original task should pause execution or abort immediately.

To accommodate one-shot applications the apftask system provides a standalone monitor that runs on all applicable APF control computers. Each monitor watches for the announcement of any task starting up on its host and thereafter monitors the task until it exits. If the task has been configured to receive a signal when it is told to pause or exit the monitor will issue the signal. If the task exits without announcing its exit to the apftask daemon the monitor will announce that the task exited with an unknown state. This is important, since the apftask daemon will not accept another 'starting' announcement from another copy of that task until the previous copy has exited.

To all intents and purposes sending a signal is the only way to asynchronously notify a shell script that a keyword has changed. Bourne shell scripts can trap and execute arbitrary code when a signal is received, then return to what they were previously doing. C-shell scripts are very restrictive and can only catch the interrupt signal, and only by taking a "goto" to a handler, and cannot return to the point of interruption. Neither approach makes it simple to pass signals through to the child command they may be executing when a signal arrives, especially if the child command is a script that is in turn executing yet another command.

To accommodate the limitations of shell scripts the apftask system provides a command-line *apftask-do* application for use by scripts. Scripts use apftask-do for any external command that run for more than a small fraction of a second. Apftask-do will run the command and monitor the task's control keyword. If the task is told to exit, apftask-do will kill the child command. If the task is told to pause, apftask-do will suspend the child until the task is unpaused.

It's worth noting that it is trivial to convert a one-shot into a persistent daemon. This adds some persistent daemon values to the simplicity of a one-shot. Most notably, the one-shot task can be started from any client host by setting a KTL keyword: *tasker* is a persistent "wrapper" script that can be started at boot time and then monitored, just like any daemon. Tasker simply monitors a GO keyword for the task, and on request starts running the one-shot task.

5.2 A common framework for diverse tasks

From an external perspective the key requirement for a common framework is that diverse tasks must be represented by the same fundamental telemetry. This provides all other software concerned with the scheduling and querying of tasks a uniform interface regardless of the nature or duration of the task. Some tasks are transient, some run at all times; many tasks strictly address isolated functionality while others may cover several major subsystems in the faciltiy; that same narrow focus means that many tasks do not worry about other tasks, while others may invoke extended chains of sub-tasks to perform their objective.

As mentioned in section 4 permission constraints are handled in both a proactive and reactive sense. This task framework adopts the same structure: it prevents multiple instances of a task from running simultaneously and provides the necessary hooks to allow sentinel applications to explicitly shut down tasks if their key permissions have been revoked. Eliminating duplication and forcing shutdowns drove the structure of the KTL keywords used to represent a task. The task framework tracks whether and where a task is running, what permissions are required by the task, and how the task would prefer to be signaled to initiate a shut down. The task framework also allows the inclusion of arbitrary KTL keywords for each task, which enables the task implementations to make use of custom, task-specific KTL keywords without the need to create a dedicated KTL service for the exclusive use of that task. Externalizing the KTL implementation of the task keywords reduced the development load on the task authors. At the same time the cost of "being" a task is low enough that some KTL services are also established as tasks, despite having ample telemetry already available to replicate some aspects of the functionality provided by the framework.

Figure 8 shows the individual keywords used to represent a task. A full functional description of these keywords is available in the APF documentation online: `http://apf.ucolick.org/help/apftask.html`

```
OPEN_CONTROL      = Proceed
OPEN_INTERRUPT    = False
OPEN_LAST_START   = 1442245831.889 UNIX seconds
OPEN_LAST_SUCCESS = 1442245866.319 UNIX seconds
OPEN_MESSAGE      =
OPEN_NSTAT        = 11
OPEN_PHASE        =
OPEN_PID          = -1
OPEN_PS_STATE     =
OPEN_RUNHOST      =
OPEN_SIGNAL       = TERM
OPEN_STATUS       = Exited/Success
OPEN_STEP         = 0
OPEN_TASKNAME     =
OPEN_TRIPWIRE     = None
```

Figure 8. KTL keyword representation of an idle 'OPEN' task, with example values

5.3 Simplifying the adoption of a common framework

A design goal of the task framework was to implement much-needed functionality without increasing the development burden on the task authors; in the ideal case, the development burden could actually decrease, in that the task framework could transparently handle some amount of necessary complexity on behalf of the task implementor.

Task implementations were written in a diverse set of scripting languages, some without native language support for KTL. Rather than limit the languages used we instead implemented a command-line interface program to encapsulate the complexities involved with properly using the task framework.

The command-line interface handles all handshaking with other running task instances and manipulation of any KTL keywords in the task infrastructure itself. Beyond the basics the command line interface also transparently implements key aspects of the signal handling mechanism. Specifically, tasks can be told to proceed with normal execution, to pause execution, or to immediately abort execution. Any time that a task would perform an operation that waits for another condition to occur, such as suspending itself for a few minutes, or waiting for a KTL keyword to transition to a specific value, the task implementation can use the command line interface to perform this blocking operation. When doing so the command line interface will simultaneously track the task-specific control keywords that indicate whether it needs to pause or abort and take appropriate action automatically. Because the task command line interface allows the execution of arbitrary commands this automatic functionality can be leveraged for any and all extended operations performed by the script.

5.4 The critical need to pause tasks

Personnel safety is a key concern for the operation of the APF as an autonomous facility. It must be possible for personnel working within the dome to be confident that any and all automated actions will not occur absent direct and deliberate action. The first and most effective way to achieve that goal at the APF is to engage an emergency stop button, but there are cases where some automation is necessary to assist with maintenance or other servicing needs.

In these situations the typical procedure is for the on-site personnel to set the checkapf user type to *tech* (see section 2.6). This sends a clear sign to all other personnel, including observers, that they should not proceed without the express consent of personnel working in or on the facility. This principle relied largely on voluntary participation until the creation of the task infrastructure.

When the checkapf user type changes to *tech* a command is first issued to the apftask dispatcher to pause all running tasks. If and only if the apftask dispatcher responds that no tasks are actively running will checkapf

allow the user type to finish changing to *tech*. The need for this positive response prompted establishing as tasks all programs with automated/autonomous behavior.

Whether and how to pause a given task varies substantially. Some tasks, such as taking an individual exposure, make no effort to pause and instead handle pause and abort commands identically. Others honor the intent of the two commands, such as the task to perform telescope slews: when commanded to pause it immediately halts motion of the telescopes; when resumed it will reissue the slew command; if instead commanded to abort it will stop motion and exit immediately.

Other pauses are less active: the process controlling the temperature inside the dome will issue commands to the vent doors when changing between specific modes. Because this persistent process spends most of its time watching and waiting it pauses by refusing to issue further commands. Any new tasks will likewise pause automatically if started while a global pause is already in effect.

One lesson learned: any handling of a pause must be built into the software at a high level. This is especially true for complex tasks, where appropriate handling may vary substantially depending upon the fine details of what the task is doing when interrupted with a pause request. Predictable handling of such requests requires that the task implementation can be interrupted at any point in its execution and handle the request gracefully. Failing that a task must still be able to recognize the request and exit gracefully. At a minimal level this still requires that the task implementation be willing to check for such requests and not single-mindedly pursue its internal sequence of events.

Put another way a task must recognize that it is not the beginning and the end of the software environment, that it exists alongside peers, sub-tasks, and super-tasks that can all influence what it should and shouldn't do at any given time. This awareness is difficult and time-consuming to add after the task is implemented.

6. DIAGNOSING UNATTENDED PROBLEMS

One serious operational issue with unattended facilities is determining the nature of problems. With nobody present to recognize and probe unexpected events it can be very difficult to pin down exactly what circumstances triggered a given problem, especially if the problem is new and unexpected. While verbose logging can help solve a lot of mysteries the logs still require advance awareness of a potential problem.

The need for always-on information retention was emphasized when the APF began to see regular use under the control of non-technical personnel, who could not reasonably be expected to probe for root causes in quite the same way as would be expected of a seasoned technician. The APF thus became the first major deployment of our KTL keyword history database,[9] the success of which led to its near-immediate deployment across all facilities at Lick Observatory. For the historian to succeed all available telemetry must be exposed via a single API, and so long as the historian supports that API, all telemetry may be recorded and retained indefinitely.

While there are innumerable beneficial ways to analyze the stored telemetry the most valuable use has been the analysis of conditions leading up to an unexpected problem with the facility. The ability to look back in time and include more telemetry on-demand brings an enormous amount of power to the problem solving process. If additional data is required for an analysis, one never has to go back and recreate the circumstances being analyzed, one can instead refine queries against the existing records and move forward with the problem solving process.

In the case of an autonomous, unattended facility, one may not even have a timely problem report to go on. Perhaps an anomaly was noticed in that night's logsheet, or otherwise recognized well after the problem originally occured. With attended observing there is at least the hope that the observer may provide a sufficient report to diagnose the issue at hand; with unattended observing, there is no such hope, and the technicians working during the day are limited to analyzing the available data or attempting to recreate the problem. When put in such a position there is no substitute for having all the telemetry at your disposal.

Please refer to the SPIE paper[9] on this subject for examples and details. It is easy to understate the transformative impact of this capability and how quickly it becomes an essential tool for troubleshooting at all levels.

7. CONCLUSIONS

The Automated Planet Finder is nearly if not the most scientifically productive facility at Lick Observatory. Credit goes to a diverse cast of scientists, engineers, and technicians all making key contributions to the success of the facility.

Some consider it an inevitable truth that software engineers will always be called upon to work around deficiencies in design or execution made in the early stages of a project. Perhaps this is true, as resources and time are rarely available late in a project to correct fundamental problems. Perhaps it is the degree to which this is true that suggests how well a project was executed in its formative stages.

A very high level of effort went into work arounds for the APF telescope. That effort influenced the structures that were put into place to support autonomous operations, as there was no expectation of reliability or consistency from several production systems and the autonomous software had to be effective despite these deficiencies. At times the deficiencies were inspiring, spurring the creation of novel adaptations and new constructs that have merit beyond this one facility.

The widespread use of the keyword history daemon[9] at Lick Observatory and limited use at W.M. Keck Observatory; the deployment of *emir*[8] for multiple projects at Lick Observatory; improvements to common software used at both observatories, including Galil motor control software, KTL inter-process communications, and standardized interfaces to simple serial controllers.

At other times the deficiencies remained in the background, and the high level of robustness required for autonomous operations was a more significant influence. This shows in the modal control scheme offered by checkapf, the granular permissions scheme enforced by *safe_modify*, and the concept of heterogenous standardized tasks.

There is definite potential for broader use of these concepts and the software components described in this paper.

REFERENCES

[1] Vogt, S. S. et al., "APF - The Lick Observatory Automated Planet Finder," *PASP* **126**(938), pp. 359–379 (2014).

[2] Brunswick, R., "Development and testing of an unique telescope enclosure design optimized for seeing and telescope thermal control," *Proc. SPIE* **5495**, 565–576 (2004).

[3] Butler, P. R. et al., "Attaining Doppler Precision of 3 M s-1," *PASP* **108**, 500 (June 1996).

[4] Vogt, S. S. et al., "HIRES: the High-Resolution Echelle Spectrometer on the Keck 10-m Telescope," *Proc. SPIE* **2198**(1), 362–375 (1994).

[5] Conrad, A. R. and Lupton, W. F., "The Keck Keyword Layer," *ADASS II* **52**, 203–207 (Jan. 1993).

[6] Lupton, W. F. and Conrad, A. R., "The Keck Task Library (KTL)," *ADASS II* **52**, 315 (Jan. 1993).

[7] Grigsby, B. et al., "Remote observing with the Nickel Telescope at Lick Observatory," *Proc. SPIE* **7016**, 701627–701627–12 (2008).

[8] Deich, W. T. S., "*EMIR*: a configurable hierarchical system for event monitoring and incident response," *Software and Cyberinfrastructure for Astronomy III* **9152**, Proc. SPIE (2014).

[9] Lanclos, K. and Deich, W. T. S., "A Complete History of Everything," *Proc. SPIE* **8451** (Sept. 2012).