# A multithreaded modular software toolkit for control of complex experiments

N. Sinenian,[a] A. B. Zylstra, M. J.-E. Manuel, J.A. Frenje, A. Kanojia, J. Stillerman, and R. D. Petrasso
*Plasma Science and Fusion Center, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, USA*

(Dated: 1 March 2012)

A multithreaded modular software toolkit has been developed for centralized monitoring and control of complex scientific experiments and instruments. The Modular Control Toolkit (MCT) supports UNIX-like operating systems and provides a reusable framework for user-developed modules to share data, setup software interlocks and to utilize a dedicated thread for hardware communication. Developers need only to create these modules, loaded by the toolkit, to communicate with and to control hardware specific to their application. The open-source nature of the toolkit makes it extensible while its use of a standard programming language (C++) does not limit users to a particular set of development tools. The toolkit is presently deployed for control of the MIT Linear Electrostatic Ion Accelerator.

## I. INTRODUCTION

In the course of building a new experimental apparatus, one is typically faced with the challenge of designing and building software for control and data acquisition. Depending on the scale of the experiment, several options are available to the experimenter. One such option is to create a control scheme from scratch, using a programming language and operating system of choice, with the use of helpful guides[1] and various libraries. Another option is to use a development toolkit; such toolkits, which are available for various platforms, simplify the design process to a variable extent. They come in a number of flavors:

1. Device-driver development toolkits (DDKs),[2,3] which simplify hardware communications code, allowing the experimenter to focus on the software framework for the control application itself.

2. Simplified integrated development environments (IDE), with simplified proprietary programming languages and proprietary user-interface controls.[4–6]

3. Application-specific IDEs, that utilize standardized programming languages (such as C) and typically furnish the developer with generic libraries for hardware control.[7]

4. Complete control solutions, which are open-source and commercial software packages specifically designed for industrial control.[8,9]

Several options exist for the experimenter both for control and data acquisition; here we focus solely on the former, although there is no reason why the toolkit described here cannot be used for the latter. Note that while this toolkit does not perform functions in real-time, it is intended to interface to and monitor such controllers; a number of software packages are readily available for real-time applications.[10,11]

These options each have strengths and weaknesses. DDKs ease driver development but do not provide any framework for the application which must be written from scratch. Such DDK's for UNIX-like systems are also difficult to find.

Simplified IDEs facilitate development by introducing a simplified programming language and toolkit which encapsulates threading and low-level implementation details, such as opening communication ports and sockets. Though this eases the development of a control system, making it attractive to, if not ideal, for novice users, encapsulation of implementation details means that advanced debugging, control of thread and process execution, thread synchronization and mutual-exclusion of shared data structures is difficult[12]. Furthermore, since the programming languages and toolkits are proprietary and closed-source, one is locked-in to using propriety debugging and optimization tools from the vendor.

Application-specific IDEs typically provide the developer with convenient libraries (though often close-source) for hardware access but lack a re-usable framework for control applications. Thus, while hardware access may be readily achieved, significant amounts of time must be spent implementing an infrastructure for multithreading, centralized monitoring and supervisory functions (e.g. interlocks).

Finally, a number of open-source and commercial software packages are available which are designed specifically for industrial control. One such package is the Experimental Physics and Industrial Control System (EPICS).[9] It has the advantage of being open-source with a built-in distributed infrastructure, making it ideal for large-scale systems. Though it is powerful and scalable, it is not well suited for novice users and lacks built-in supervisory components; it is also in excess of what is required for a small to medium sized facility (non-distributed control system, with tens of hardware components to control) such as the MIT Linear Electrostatic Ion Accelerator (LEIA).[13,14] Another alternative, RSView32,[8] is a commercial package with automation and control functionality similar to that of the MCT; it is however a

[a] nareg@psfc.mit.edu

| Toolkit Type | DDK | Simplified IDE | Application-Specific IDE | Complete Control Solution | | MCT |
|---|---|---|---|---|---|---|
| Example(s) | MS WDK | LabVIEW | NI LabWindows | RSView | EPICS | - |
| License | Proprietary, Commercial | | | | Open-source | Open-source |
| Operating System(s) | Windows | Windows, OS X, Linux (partial) | Windows | | Cross-Platform | |
| Distributed | - | Limited (e.g. network variables) | No | Yes | Yes | No |
| Modular | - | Yes | No | No | Yes | Yes |
| Built-in Supervisory Component | - | No | No | No | No | Yes |
| Multithreading Infrastructure | - | Yes | No | No | Yes | Yes |
| Language Support | C/C++ | Proprietary | C | - | C | C/C++ |
| IDE | No | Proprietary | | | No | |

TABLE I. Various toolkit types that may be used to build a full control solution. Comparison is made between the license type, operating system (OS), whether the toolkit is intended for distributed control, whether it is modular, whether it has supervisory components (e.g. interlocks) and if it has an infrastructure which eases multithreading.

closed-source package, making it less extensible, with limited operating-system support.

The Modular Control Toolkit (MCT) incorporates the strengths of each of the approaches discussed and is designed with control applications in mind. It is open-source, written in C++ and built using open, portable libraries such as GTK+[15] and Glib[16] which are available under the Lesser General Public License[17] (LGPL). It provides a modular framework, allowing one to isolate and re-use significant portions of the code as gradual changes are made to an experiment or for an entirely different experiment altogether; modularity readily allows for the systematic testing and debugging of complex code[18]. Users have the option of launching and managing threads with complete control over execution, or using the pre-existing threading infrastructure and allowing the framework to manage and control execution. Furthermore, features common to a medium or large-scale experimental apparatus, such as interlocks, are built-in to the framework. Finally, module development and framework customization is not restricted to an integrated development environment or compiler. Features of the different toolkit varieties, including the MCT, are compared in Table I.

This paper is organized as follows: Section II gives the reader an overview of the various software components which make up the toolkit and discusses implementation details of the code; section III presents some of the implementation details of a sample module; section IV discusses some of the development challenges and the initial deployment of the toolkit at MIT; section V describes the future direction of the toolkit.

## II. OVERVIEW AND IMPLEMENTATION

The MCT was originally developed for use on the MIT LEIA but with a broader scope of application in mind. The specifications for the toolkit may be understood by considering the fact that several elements are common to most control systems software.

Among these include the need for interlocks to help ensure the safety of the apparatus and proper execution of the experimental process. At MIT, for example, the toolkit is used to ensure that vacuum gate valves are not inadvertently opened by an operator when doing so could put an unnecessarily high load on a pump and lead to potential failure of that pump. The toolkit is not redundant in its implementation of interlocks, and is not meant to be used to protect operators. Software interlocks should be used only as a level of redundancy for existing hardware interlocks in the context of protecting operators. Several high-voltage bias supplies at MIT are hardware-interlocked by direct electronic circuitry to either access panels or pressure gauges. This is to ensure that they are powered off when human contact is physically possible. The toolkit's software interlocks are used to provide a parallel path of redundancy by monitoring the pressure readout from the gauge and independently controlling the power supply. Thus, both the hardware and software interlock need to be functioning properly to allow operation of the bias supply.

An additional feature common to control systems is a multithreaded code; this allows continuous communication with hardware to continue in the background while retaining a responsive user-interface (UI). Finally, since most machines or experiments evolve over time, parts of the control software evolve with the hardware while

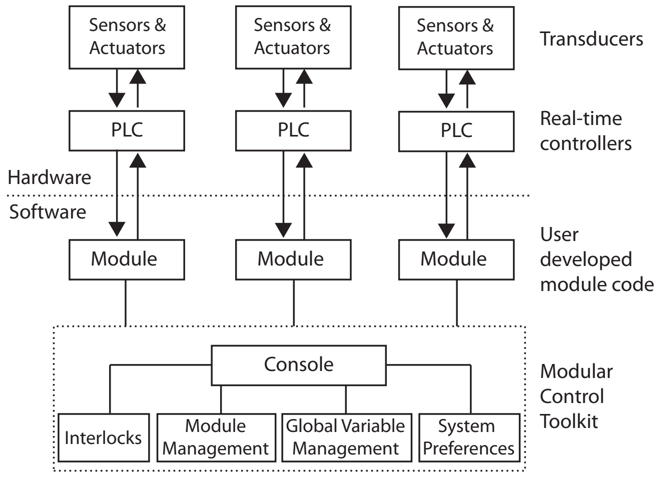


FIG. 1. Simplified architecture of control software written using the MCT, illustrating how real-time controllers (connected to transducers), user-developed code (herein referred to as the module) and the toolkit interact together. The module is a shared object loaded by the toolkit at run-time.

other parts are reused in subsequent versions of the experiment. This naturally leads to a modular design where each module presents both an interface to the hardware and to the operator via the toolkit, as shown in Fig. 1; in this way modules may be re-used, removed, or evolved as the experiment changes. Note that while the toolkit does not operate in real-time, it is intended to interface to and monitor real-time hardware controllers. It is these real-time controllers which then drive and monitor actual hardware. The MCT is thus a means for centralized monitoring and operation of an experimental apparatus.

Given the modular design, it is also often desirable to share data between modules and potentially between threads. This requires mutual-exclusion locking of the data structures (shared data is limited to RAM at the present). The motivation for shared data may not be entirely obvious; certain tasks, such as logging of pertinent parameters (e.g. run-time performance metrics associated with the experiment) are centralized functions and may even be implemented as modules. Such tasks require access to all variables of interest, which are likely be distributed among modules. As another example, consider the implementation of an interlock system, where one would need to "lock down" a system when a parameter of interest crosses a threshold value. For example, one might want to lock down operation of a power supply when a pressure reading elsewhere in the system crosses a threshold value. This requires access to both the variable holding the pressure, which is constantly updated by one module and to the interlock system of another module.

The toolkit was designed to meet all of these specifications. The architecture is illustrated in the class diagram shown in Fig. 2 using standard Unified Modeling Language (UML) notation. Note that a significant number of auxiliary operations and attributes are suppressed in this model so as to focus on the core architecture and functionality. The entry point for the toolkit is the `Console::run()` function, which is called after a Console object is instantiated. As shown in the UML model, this Console object is a container for exactly one instance of each of the `Prefman`, `IntLockMan` and `ModMan` classes; these objects are responsible for managing toolkit preferences, interlocks and modules, respectively. Upon execution, the console will create an instance of each of these classes, initialize the multithreading engine and then initialize each of the three aforementioned objects, at which point it will execute the GTK event loop and wait for user input.

All user-developed modules are required to inherit from the class `ModBase`, as shown in Fig. 2. This base class provides an interface to each module for registering and un-registering interlocks and global variables, for setting and retrieving the value of global variables, for posting messages to a centralized console and for reading and writing configuration data associated with the module. The toolkit facilitates the storage and retrieval of preferences associated with any given module by handling reading, writing and parsing of configuration data to and from disk. It is left up to the module designer to use the toolkit's interface functions to perform these tasks. Note that interlock system definitions and global variables are stored in the interlock manager and the console, respectively. Thus, the base class must access these two objects to perform the aforementioned tasks on behalf of each module. The inheritance and class permissions are set to allow the base class `ModBase` access while shielding the user-defined module class from both the implementation details of and access to the rest of the toolkit. Within the toolkit, the class `ModBase` is packaged as a shared library, which has two main advantages: (1) Since the code contained within the base class is shared between all modules, linking at run-time to a common library reduces the executable size and (2) Internal changes may be made to this base class, as optimization and enhancements are implemented in future versions of the toolkit, without the need to re-build any of the modules; this greatly enhances maintainability of the toolkit.

Since modules themselves are compiled into shared libraries and loaded at a user's request, an additional requirement is that they must implement a pre-defined interface. This allows the module manager to properly load, unload and query each module. For ease of development, a template including a skeleton module is provided in the toolkit source package which implements these interface symbols; it also sets up an environment for properly compiling module source code into a shared library.

Next, note the function `ModBase::thread_body()`. This function may be overridden by the module and in that case should contain user-defined code to be executed within a dedicated thread; it is called repeatedly within the body of a loop. Alternatively, the user may wish to setup their own thread and ignore these functions; the toolkit does not preclude one from doing so. In any

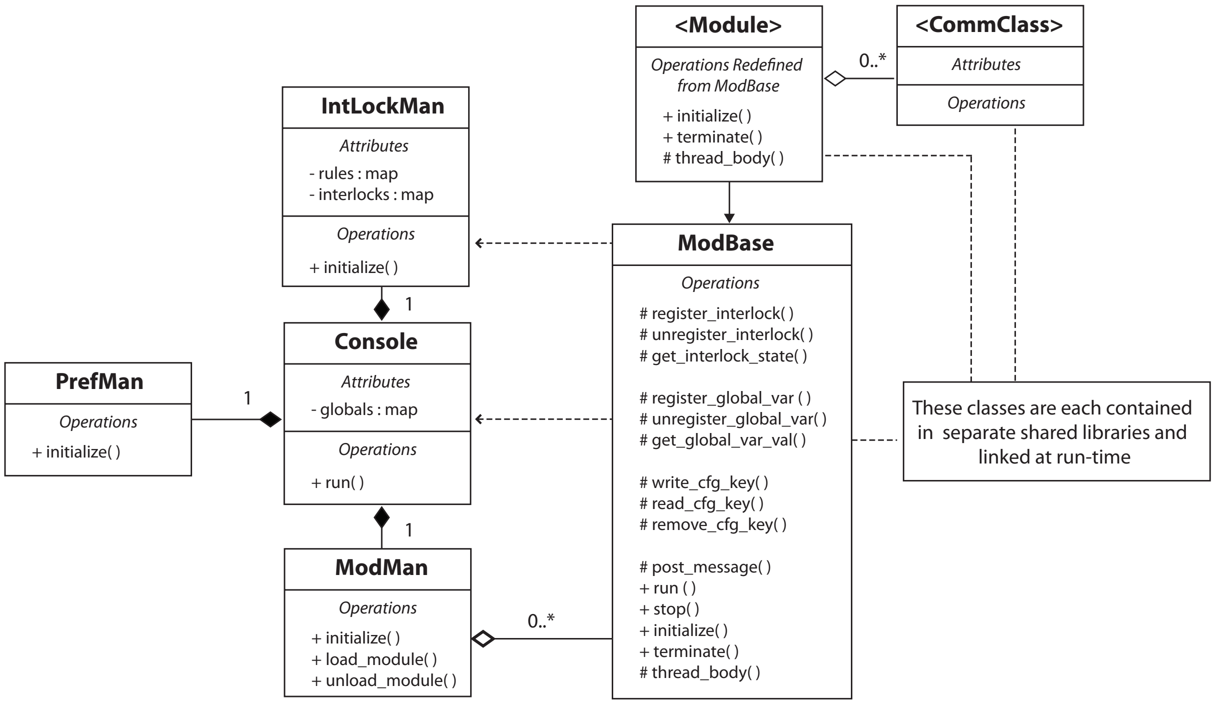
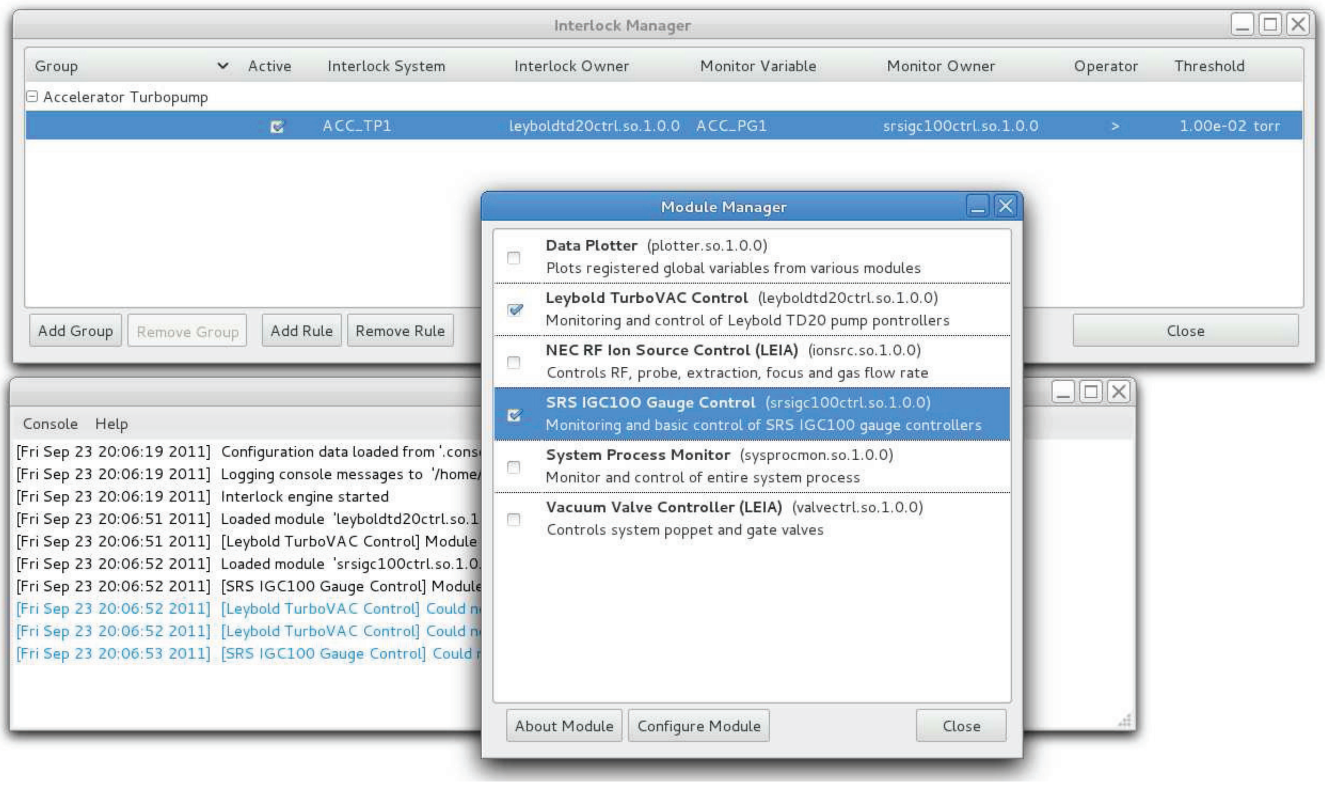


FIG. 2. Class diagram for the Modular Control Toolkit (MCT) using standard Unified Modeling Language (UML) notation. The core classes are shown on the left side: Console, IntLockMan, PrefMan and ModMan, which implement the console, the interlock manager, the preferences manager and the module manager, respectively. All user modules inherit from ModBase, which provides an interface for interlock and shared variable functions. For completeness, a shared library one might use for communicating with hardware devices are shown (i.e. an object of class type <CommClass> ). Instances of classes required for hardware communication are contained by the user module.

case, two additional functions, `ModBase::initialize()` and `ModBase::terminate()`, may be overridden to perform module (de-)initialization (after)before the dedicated loop (stops)starts. The threading framework available to toolkit users does not incorporate any kind of supervisory thread prioritization when scheduling threads. However, since the MCT is built using GLib, developers may use library functions to yield or to prioritize thread execution as deemed necessary.

Finally, consider the interlock manager, shown in the screenshots of the user-interface in Fig. 3. The operator may define rules at run-time which are checked by the interlock engine. Shown in the figure is an example of a rule for locking down a turbopump (i.e. the interlock system). In order for the interlock system to be active, the monitor variable “ACC_IG1” (which is a standard toolkit global variable registered by a module) must be greater than $1 \times 10^{-2}$ Torr. What actually happens when the interlock system is engaged is left up to the owning module, in this case, “leyboldtd20ctrl.so.” The interlock manager then does several things: (1) provides a means for user-defined modules which inherit form `ModBase` to register and check the status of interlocks (2) a user-interface for the operator to define rules for each interlock system (several may be defined for each system) and (3) an engine to check operator-defined rules, which runs in a dedicated thread in the background. In this way, modules do not need to know how they fit into the bigger picture of the experimental apparatus; they simply register and update measured quantities of interest and register interlock systems for the hardware device they are operating. How the various devices are integrated together is left up to the operator at run-time, making modules more generic and re-usable.

The activity diagram in Fig. 4 best illustrates the operation of the interlock engine; the sequence of events outlined in the figure take place within a dedicated thread. After initialization, the engine traverses a `std::map` of rules. A mutual-exclusion lock is obtained on this data structure as it is traversed. For each rule, the engine checks to ensure that the monitor variable and interlock system for that rule are available (that the module(s) which registered them are loaded). If they are not available, the engine marks the rule as inactive and moves onto the next rule; if both the monitor variable and interlock system become available at a later time, the engine will mark the rule as valid and proceed. Note that to check for these two conditions, the engine must obtain read-only locks on both the interlock systems and monitor (global) variable data structures to prevent any of the module threads or the main thread from attempting to modify these data structures; mutual-exclusion and read/write locks are implemented using standard Glib methods. Checking the validity of rules in this manner al-


Interlock Manager

| Group | Active | Interlock System | Interlock Owner | Monitor Variable | Monitor Owner | Operator | Threshold |
|---|---|---|---|---|---|---|---|
| ⊟ Accelerator Turbopump | | | | | | | |
| | ✓ | ACC_TP1 | leyboldtd20ctrl.so.1.0.0 | ACC_PG1 | srsigc100ctrl.so.1.0.0 | > | 1.00e-02 torr |

Module Manager

- ☐ **Data Plotter** (plotter.so.1.0.0)
  Plots registered global variables from various modules
- ✓ **Leybold TurboVAC Control** (leyboldtd20ctrl.so.1.0.0)
  Monitoring and control of Leybold TD20 pump pontrollers
- ☐ **NEC RF Ion Source Control (LEIA)** (ionsrc.so.1.0.0)
  Controls RF, probe, extraction, focus and gas flow rate
- ✓ **SRS IGC100 Gauge Control** (srsigc100ctrl.so.1.0.0)
  Monitoring and basic control of SRS IGC100 gauge controllers
- ☐ **System Process Monitor** (sysprocmon.so.1.0.0)
  Monitor and control of entire system process
- ☐ **Vacuum Valve Controller (LEIA)** (valvectrl.so.1.0.0)
  Controls system poppet and gate valves

```
[Fri Sep 23 20:06:19 2011]  Configuration data loaded from '.cons
[Fri Sep 23 20:06:19 2011]  Logging console messages to  '/home/
[Fri Sep 23 20:06:19 2011]  Interlock engine started
[Fri Sep 23 20:06:51 2011]  Loaded module  'leyboldtd20ctrl.so.1
[Fri Sep 23 20:06:51 2011]  [Leybold TurboVAC Control] Module
[Fri Sep 23 20:06:52 2011]  Loaded module  'srsigc100ctrl.so.1.0.
[Fri Sep 23 20:06:52 2011]  [SRS IGC100 Gauge Control] Module
[Fri Sep 23 20:06:52 2011]  [Leybold TurboVAC Control] Could n
[Fri Sep 23 20:06:52 2011]  [Leybold TurboVAC Control] Could n
[Fri Sep 23 20:06:53 2011]  [SRS IGC100 Gauge Control] Could r
```

FIG. 3. Screen capture of the MCT, showing three windows (as labeled by window titles): the Console, the Interlock Manager and the Module Manager. Important information is presented to the operator in the Console window; modules may be loaded, unloaded and configured from within the Module Manager; interlock rules may be defined, grouped and (de-)activated from within the Interlock Manager interface.

lows user-defined interlock rules to remain safely defined within the MCT as modules are loaded and unloaded.

Once a rule has been deemed valid, the engine checks to ensure that the interlock system associated with that rule has not already been flagged for lock by a rule that was previously tested positive. If the system is already flagged, the rule is ignored since the system will lock regardless; otherwise the rule is tested. If the rule is tested and found positive, the interlock system associated with the rule is then flagged for lock. Subsequent rules associated with the same interlock system are not tested, though each rule is nevertheless traversed and checked for validity. Finally, after all rules have been traversed, the lock on the rules data structure is released. The flags on all of the interlock systems are checked and systems are locked or unlocked accordingly; these flags are reset for the next iteration of the engine loop. Note that since the engine runs within a dedicated thread, it sleeps for some period of time within each iteration so to not saturate or overwhelm the hardware.

As shown here, the implementation of interlocks and global variables requires the locking of multiple toolkit resources from within different threads. The design of the interlock engine and global variable system ensures that deadlock does not occur. Two design paradigms were followed to ensure this: (a) all resource locking for these systems is performed internally by toolkit functions (which are called by a module) and (b) no two toolkit functions that lock the same resource ever wait for each other to complete before releasing that resource. Since locking is performed from within toolkit functions, modules do not have direct access to these resources and therefore cannot lockout a resource directly.

## III. DESIGN OF MODULES

The design of data-flows for modules are largely left up the developer. That is, module developers may take an event-driven, data-driven or mixed approach depending on the application. For instance, certain types of equipment may wait for and respond to user interaction (event-driven), while in other cases one might wait for a device to periodically send data and then either perform processing on that data or alert the user as data comes in (data-driven). Event-driven and data-driven approaches are readily accomplished with the use of event-handlers and a dedicated thread within the module, respectively. Alternatively, and often, one might use a combination of both techniques. For example, a simple pressure gauge controller for a system might require continuous polling and readout of the pressure (data-driven) as well as some

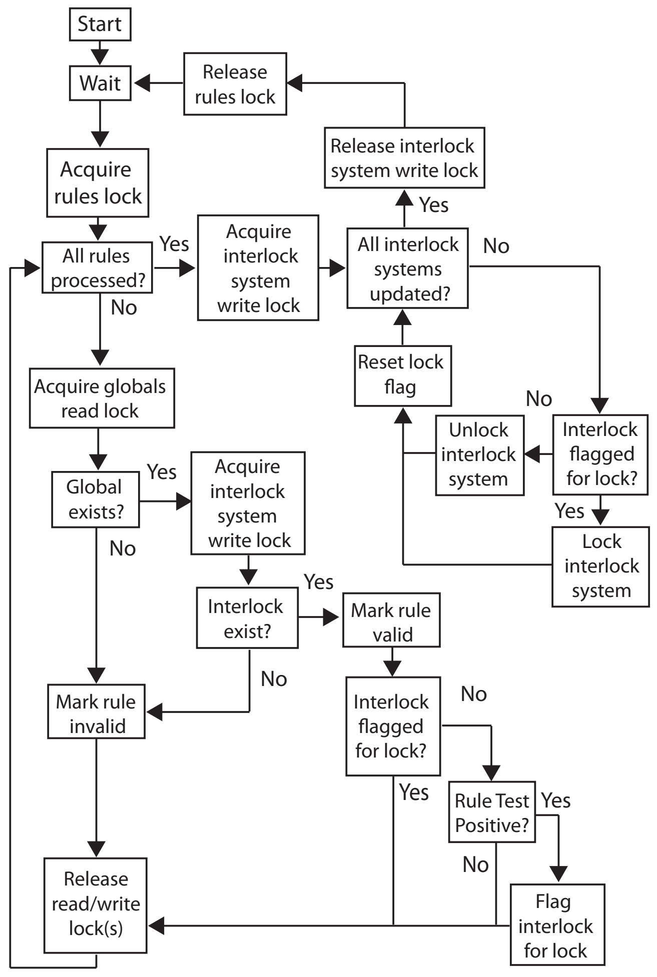


FIG. 4. Activity diagram of the interlock engine, illustrating how rules are validated and tested and how associated interlock systems are locked and released. The sequence of events depicted execute within a dedicated thread.

basic control (user or event-driven) for configuration and operation of the gauges. The toolkit facilitates the implementation of any combination of these models.

As an example of the flexibility allowed by the toolkit in conjunction with GTK and Glib in implementing a mixed data- and event- driven module, consider a practical module which interfaces to a vacuum pump controller. The vacuum pump itself is driven by a hardware controller with a remote serial interface; the manufacturer has provided a set of commands to retrieve information about the status of the pump, as well as commands to start and stop the pump. The module must present a user interface to the operator, displaying pump parameters such as load and temperature and allow for control of the pump, so the user can start and stop it. To achieve the first goal, it must interface with the pump and retrieve its status on a periodic basis; this is best accomplished in the background using a dedicated thread. Since the module must periodically communicate with the instrument in a separate thread and update the user-interface in the parent thread (a GTK requirement that GUI calls be made in the parent thread) it will have to dispatch function calls for updating the user-interface to the parent thread. Furthermore, the mixed event-driven and data-driven approach dictates that hardware access will occur from two threads: (1) the dedicated thread which will periodically poll the pump controller and retrieve parameters and (2) the parent thread, which handles the user-interface event-handlers, where function calls to start and stop the pump will occur. This naturally leads to the requirement of a mutual exclusion lock for hardware access. All of these constraints are readily handled using GTK and Glib. The module implementing this functionality would inherit from the class `ModBase` and re-implement functions as follows:

```
class PumpControl : public ModBase
{
  public:
    PumpControl();
    ~PumpControl();

    void initialize();
    void terminate();

  protected:
    void thread_body();

    Glib::Dispatcher update_gui();
    Glib::Mutex serial_mutex;

    void update_gui_disp();

    double pump_temperature;
    double pump_load;

    serial pump_connection;
};
```

The `Glib::Dispatcher` object is used to dispatch the code contained in the function `update_gui_disp()` to the parent thread; the connection between this object and the dispatched function is made is made in the module's `initialize()` function. The `Glib::Mutex` object is used to lock hardware access so that only one thread may communicate with the pump controller at a time. Finally, the `serial` object is an instance of a library class which allows communication over serial ports. The functions shown above may be re-implemented as follows:

```
void PumpControl::initialize()
{
  // Setup GUI

  // Connect dispatcher
  update_gui.connect(sigc::mem_fun(*this,&
       PumpControl::update_gui_disp);

  // Add event handler for stopping pump
}
```

```
10
11 void PumpControl::thread_body()
12 {
13    sleep(POLLING_PERIOD);
14
15    // Acquire hardware access lock
16    Glib::Mutex::Lock serial_access(
          serial_mutex);
17
18    // Communicate with instrument, store
          temperature and load values in data
          members pump_connection.read(...);
19
20    // Release hardware lock
21    serial_access.release();
22
23    // Call update_gui_disp() in main thread
24    update_gui();
25 }
26
27 void PumpControl::update_gui_disp()
28 {
29    // Use data members to update GUI here
30 }
31
32 void PumpControl::on_stop_pump()
33 {
34    // Acquire hardware access lock
35    Glib::Mutex::Lock serial_access(
          serial_mutex);
36
37    // Communicate with instrument
38    pump_connection.write(...);
39
40    // Release hardware lock
41    serial_access.release()
42 }
```

This bare-bones sample module allows for continuous polling of the pump while retaining a responsive control GUI. There are, however, some subtleties. For instance, the creation of a `Glib::Mutex::Lock` object on line 35 (from within `on_stop_pump()`) is a blocking call, which will wait for a call to `release()` (from within the function `thread_body` executing in a dedicated thread) before continuing. One must therefore be careful not to make the polling period or communication time per iteration too long for this will cause the GUI to become less responsive. This is seldom an issue in practice because the time-scales considered here not sufficiently long for typical communication schemes with modern laboratory instruments, even over slow (9600 baud) serial devices. Note that these kinds of timing considerations are not exclusive to the MCT, but are common to all the previously mentioned toolkits;[4,5,7] these level of detailed considerations must be left up to the developer since these toolkits know nothing about the types of instruments being interfaced. Note that the sample code above does not register interlocks or global variables. These may be registered, unregistered and read from within any thread (this is properly handled by the MCT), giving the module designer much flexibility. Configuration data, such as the serial port parameters to use when connecting to the pump controller may also be saved and read using the toolkit. The sample presented here is simplified to demonstrate the flexibility of the MCT in its ability to accommodate various programming models. Complete modules, written in the course of developing this toolkit, included configuration dialogs for instrument setup, communications error-checking, advanced GUIs, checking and handling of interlocks, etc.

## IV. DEVELOPMENT AND INITIAL DEPLOYMENT

The MCT was developed with several modules to serve the needs of the MIT LEIA Facility. LEIA is an accelerator comprised of several vacuum pumps and valves, numerous high-voltage bias supplies, and many pressure transducers. These components interface to real-time controllers that are interconnected with a control computer and a data acquisition computer using a fiber-optic network. In some cases, a single real-time controller drives several components simultaneously. The control computer drives all of these real-time controllers: pump controllers (2), ion source controller (1), valve controller (1) and pressure gauge controllers (2). Communication with these controllers is implemented using four modules, each with a dedicated thread. Together with the main thread and interlock engine, a total of six hardware threads are used for normal operation. The toolkit itself was developed on a dual-core processor with hyper-threading support(2.4 GHz Intel Core i5) while the majority of modules were developed and tested on the LEIA control computer (2 $\times$ 2.6 GHz, quad-core AMD Opteron processors for a total of 8 cores). The toolkit was built using g++[19] with compiler optimizations and tested on 64-bit Linux Kernels (both 2.6.x and 3.x).

The MCT's modular, multithreaded structure is ideal (and scalable) for the increasing number of CPU cores in today's and tomorrow's computers. This structure was one of the most difficult implementation challenges. Unlike a monolithic control code tailored to a specific task, the many possible uses of the MCT had to be anticipated during the development phase. Though the modular nature facilitates code re-use and makes the toolkit suitable for a wide range of applications, the toolkit's features and the interface functions made available to the modules had to cover a broad range of anticipated uses. During the design phase, a preliminary toolkit interface was implemented. This interface was then iterated upon and modified as a variety of modules were written for the LEIA Facility. In effect, the toolkit was given time to mature internally. A second layer of complexity was introduced with the addition of multithreading. One of the design requirements was the ability for modules to call toolkit functions from within any thread. This capability meant that the toolkit's interface functions needed to be thread-safe by preventing dead lock. A significant amount of time was spent testing toolkit functionality with a variety of test modules. Additional pitfalls were encountered during the initial deployment and testing phase, having to do with the cross-platform nature of the toolkit. Although the toolkit is based on the standardized Glib and GTK+ libraries, the implementation of these libraries on different systems can vary. During testing it was found that the look, feel, and behavior of widgets was different between some Linux distributions.

The toolkit could have benefited from greater consideration to fault-tolerance in the early design stages. At present, a poorly implemented module may cause adverse effects to a running instance of the toolkit. This problem may be alleviated by sandboxing modules (i.e. spawning each module as a separate process rather than allowing each module to have a dedicated thread). The benefits of this type of architecture are twofold: 1. the running instance becomes fault-tolerant, as a spawned module may be terminated by the toolkit and re-loaded as necessary, and 2. spawned modules may each run their own instance of the GTK main loop, improving GUI performance (by alleviating the load of GUI calls on a single instance). The latter increases scalability as a greater number of modules may be accommodated. The cost of implementing this type of hierarchy was incremental during the initial design stages, where modification of the toolkit at present will require significant overhaul.

## V. CONCLUDING REMARKS

A modular toolkit for control of scientific experiments and instruments has been developed. The toolkit allows novice developers to quickly build a multithreaded modular control application without simultaneously limiting advanced users. Users have the option and not the obligation of using the threading framework, shared variables and centralized storage of configuration data. Modules may rely heavily on these various aspects of the toolkit or implement these features independently. Since the framework is open-source and written in C++, advanced developers have the ability to use development tools of their choice to customize and improve the inner workings of the toolkit and to commit any improvements back to the user community.

Future versions of the toolkit will incorporate a number of improvements, including:

1. Full use of C++ namespaces to mitigate any ambiguities in user-developed module code.
2. An integrated diagnostic tool to allow users to monitor the number of running threads, registered interlock systems, global variables and to monitor system resource usage.
3. Ability for modules to register callback functions with the toolkit (event handlers) for handling toolkit events such as the registration of global variables and interlock systems. This is useful to modules which take all global variable data and log it to a database or to disk.
4. Priority scheduling of threads that lock toolkit resources (indirectly by calling toolkit functions); this will improve application performance as the number of modules accessing toolkit resources increases.
5. Execution of modules within a dedicated process (similar to a "sandbox") for enhanced stability, robustness and recovery from localized data corruption and run-time errors.

The MCT has proved to be a robust control solution at the MIT LEIA Facility. It serves the needs of small to medium scale experiments and facilities, defined here as a system comprised of tens of hardware components driven by one or two computers. Relative to commercial control software or other open-source alternatives, it is cost-effective and ideal for both novice and advanced users. In addition to being a toolkit, it provides a reusable application framework, allowing novice users to combine it with existing modules and use it out-of-the-box; advanced users are free to develop modules with little restriction.

The Modular Control Toolkit may be obtained from the MIT Technology Licensing Office(TLO).[20]

## ACKNOWLEDGMENTS

This work was supported in part by the National Laser User's Facility (DOE Award No. DE-NA0000877), Fusion Science Center (Rochester Sub Award PO No. 415023-G), US DOE (Grant No. DE-FG03-03SF22691), Laboratory for Laser Energetics (LLE) (No.412160-001G), Lawrence Livermore National Laboratory (No.B504974) and General Atomics (GA) under US DOE (DE-AC52-06NA27279).

[1]G. Varoquaux, Computing in Science Engineering **10**, 55 (2008).
[2]Windows Driver Kit, see `http://www.microsoft.com/`.
[3]EnTech Device Driver Kit, see `http://www.entechtaiwan.com/`.
[4]National Instruments LabVIEW, see `http://www.ni.com`.
[5]Data Acquisition Systems Laboratory, see `http://www.dasylab.com`.
[6]MATLAB Instrument Control Toolbox, see `http://www.matlab.com`.
[7]National Instruments LabWindows, see `http://www.ni.com`.
[8]RSView32 from Rockwell Automation, see `http://www.rockwellautomation.com/rockwellsoftware/performance/view32/`.
[9]Experimental Physics and Industrial Control System (E.P.I.C.S.), see `http://www.aps.anl.gov/epics/`.
[10]VxWorks Real-Time Operating System, see `http://www.windriver.com/products/vxworks/`.
[11]National Instruments LabVIEW Real-Time Module, see `http://www.ni.com/labview/realtime/`.
[12]The author recently discovered, using the toolkit described herein, a bug in the kernel driver of a particular ethernet serial device server when it was accessed simultaneously by two

threads; such a discovery is difficult, if not impossible, with heavily encapsulated systems where thread execution control is less transparent.

[13] N. Sinenian, et al., Review of Scientific Instruments (2011), to be submitted.

[14] S. C. McDuffee, J. A. Frenje, F. H. Séguin, R. Leiter, M. J. Canavan, D. T. Casey, J. R. Rygg, C. K. Li, and R. D. Petrasso, Review of Scientific Instruments **79**, 043302 (2008).

[15] GIMP Toolkit, see `http://www.gtk.org/`.

[16] GNU Library, see `http://developer.gnome.org/glib/`.

[17] GNU Lesser General Public License, see `http://www.gnu.org/licenses/lgpl.html`.

[18] G. K. Thiruvathukal, K. Laufer, and B. Gonzalez, Computing in Science and Engineering **8**, 76 (2006).

[19] GNU Compiler Collection, see `http://gcc.gnu.org/`.

[20] MIT Technology Licensing Office, see `http://web.mit.edu/tlo/www/`.