VERSE – Virtual Equivalent Real-time Simulation Environment

Yang Zheng*, Bryan J. Martin, Ph.D†, and Nathaniel Villaume‡
Jet Propulsion Laboratory, Pasadena, California, 91109

Distributed real-time simulations provide important timing validation and hardware in-the-loop results for the spacecraft flight software development cycle. Occasionally, the need for higher fidelity modeling and more comprehensive debugging capabilities – combined with a limited amount of computational resources – calls for a non real-time simulation environment that mimics the real-time environment. By creating a non real-time environment that accommodates simulations and flight software designed for a multi-CPU real-time system, we can save development time, cut mission costs, and reduce the likelihood of errors. This paper presents such a solution: Virtual Equivalent Real-time Simulation Environment (VERSE). VERSE turns the real-time operating system RTAI (Real-time Application Interface) into an event driven simulator that runs in virtual real time. Designed to keep the original RTAI architecture as intact as possible, and therefore inheriting RTAI’s many capabilities, VERSE was implemented with remarkably little change to the RTAI source code. This small footprint together with use of the same API allows users to easily run the same application in both real-time and virtual time environments. VERSE has been used to build a workstation testbed for NASA’s Space Interferometry Mission (SIM PlanetQuest) instrument flight software. With its flexible simulation controls and inexpensive setup and replication costs, VERSE will become an invaluable tool in future mission development.

I. Introduction

As space technology becomes more advanced and flight applications more complex, the need for high fidelity and real-time simulations constantly increases, as does the computational load required to support them. Expensive hardware costs have traditionally limited developers’ access to hardware in-the-loop testbeds, or simply rendered them inaccessible except during very late stages of the software development cycle. The size, complexity, and cost constraints of today’s flight applications call for many more developers collaborating to work on different components of the software and hardware simultaneously. Finding a way to test and integrate these components in a simulation environment becomes essential in order to increase the availability of early testing tools, thus reducing the risk of rework, unanticipated interactions between software and hardware, and even faulty software causing damage to critical hardware. Distributed simulations can be used to meet the computational needs of modern flight and instrument software development. Non real-time simulation testbeds are used to ensure algorithm validity and correct data flow prior to integration with real-time simulation testbeds, which are typically used to provide timing and software validation. The desire to fly what you test, and test what you fly is enabled by a supporting simulation infrastructure that requires little or no changes to flight code to switch between different testbed types.

A. Goal of VERSE

The VERSE environment is designed to ease the transition between the early software development environments for flight software (FSW) and simulation & support equipment (SSE) for flight missions by providing an API-identical environment these types of software between workstation testbeds and real-time testbeds. By eliminating the need to re-code or conditionally compile for the different environments, VERSE reduces errors and increases the effectiveness of early testing. In addition, later high-fidelity simulations are often not ported back to the early software development environments due to cost and time constraints, which often forces late testing and problem...
solving off of these platforms, or significantly reduces their effectiveness. By eliminating the need to port to a different environment, VERSE increases the useful lifetime of software-only test environments throughout the life of the flight program. In addition, these “workstation testbeds” have much lower cost and much higher availability, and thus reduce the traditional bottlenecks associated with hardware-poor development environments. Even more ambitiously, by writing emulation layers as kernel-level drivers with identical APIs to the real drivers, the VERSE and associated emulators/simulators eliminate even the need for recompilation of the SSE software, allowing a single binary release to run in the simulated hardware and hardware-in-the-loop environments.

B. Virtual time Simulation Model

Our real-time testbed simulators necessarily operate as fixed time-step simulations. In fixed time-step simulation models, time advances in fixed increments and the system state is updated at the end of each time increment. In some cases (typically monolithic or single-CPU simulations and systems), time may be scaled by a constant factor, so that simulated time runs faster or slower than real time, but still at a constant rate (See Ref 9, section 4.3.4). This model may be inefficient when time steps are small and few or no state changes occur at each time increment, and these implementations normally run much more slowly than the real-time system due to the overhead involved in measuring and managing the flow of time. The simulation even in quiescent periods will consume computational resources, since the flow of time must be maintained. In discrete-event simulation models, time does not flow at a constant rate, but jumps from one event to the next. In these systems an event is normally defined as the exchange of state information between different models, as these are the significant points that must be maintained in order to preserve the integrity of the simulation or testbed. In discrete-event simulations the system state changes instantaneously upon the occurrence of an event, at discrete points in simulated time that are usually not evenly spaced. This paper refers to the progression of the simulated discrete time points in the discrete-event simulator as virtual time. In VERSE, virtual time advances by jumping from one scheduled task time to the next as soon as execution of the preceding task is complete. When multiple events are scheduled to occur at the same time, the simulated virtual time remains unchanged until all the events scheduled for that time are executed. Also, as opposed to execution in real time where time advances at a constant rate, virtual time simulation skips over unused periods between consecutively scheduled tasks. This makes it possible for the simulation to progress much faster than real time when large time gaps exist between events or when event executions require minimal computation. The actual speed gain (or loss) depends largely on model complexity, task-switching times, and level of parallelism in the original system. Since there are no hard time deadlines, a virtual time simulator can be paused after the execution of any particular task, stepped to allow execution of the next scheduled task, or continuously run so all subsequent tasks are executed. These features provide the user with enormous control over the simulation.

C. RTAI

VERSE is an extension of the Real-Time Application Interface (RTAI) scheduler. To understand how VERSE works, a brief review of what RTAI is and what it does is in order:

RTAI is an open source project designed to supply features of an industrial-grade real-time operating system on top of the powerful GNU/Linux environment. The code is constantly being improved and is extensively tested by a world-wide community, but the project uses the Debian development model for releases to ensure users have stable code. The name (“Application Interface”) seems to imply RTAI is just an API, but the code actually implements a real-time Linux kernel extension that runs real-time tasks seamlessly along side of the host Linux system. RTAI has its own scheduler and provides a full set of real-time inter-process communication mechanisms. Altogether, RTAI provides deterministic timing – “hard” real-time scheduling with nominal jitter on the order of microseconds on typical systems.

8 This problem is somewhat unique to distributed simulations. In a non-distributed system two events cannot occur at the same physical time, because one of them will have higher priority, thus excluding simultaneity. So, for instance, if two different models need to execute at time T, only one of them actually will, and the other will begin execution immediately following. In a scaled-time simulation computational resources are constantly expended to track these overlaps. In a distributed simulation, many tasks can and normally do occur simultaneously.

** Which is normally the case. Real-time systems must leave gaps between high priority tasks to insure that variations in computation times do not cause violation of deadlines, so all tasks are planned around worst-case time constraints.

†† There are three recognized development stages in the Debian model, each with its own release branch: development, testing, and stable.
RTAI's enabling technology is ADEOS – Adaptive Domain Environment for Operating Systems6, another open-source project, whose premise is to use a nano-kernel inserted between operating systems and the hardware to eliminate direct hardware-dependent code in the kernel. The sole purpose of the nano-kernel is to dispatch hardware interrupts to the operating systems in priority order. Hosted operating systems, or domains, must register with the nano-kernel to get interrupts. Official Linux kernels releases are unaware of ADEOS, and so must be patched to interface with ADEOS. When this is done in concert with RTAI, the patching process ensures that Linux registers with the nano-kernel dispatcher at a lower priority than that of the RTAI scheduler. Thus, the RTAI scheduler gets hardware control before Linux, and RTAI tasks run before Linux tasks, ensuring deterministic timing. See Figure 1 for a diagram of the relationship between RTAI, Linux, and ADEOS.

Real-time software developers have considerable design flexibility because RTAI tasks can be run from either kernel- or user-space. Kernel-space tasks offer the highest performance and are suitable for high performance or low-latency embedded applications. A task is started from a kernel module by specifying a function that will be run as real-time task, much like VxWorks’ taskspawon() specifies a function to be run as a task. RTAI’s LXRT is a user-space interface that provides a symmetric API that may be used by both real-time RTAI tasks and Linux processes. A user-space process makes special calls at the beginning and end of code sections that need to operate in real time. When such sections are entered from a non-real-time context, the entire process is elevated to real-time status, and is scheduled as a real-time task until the section is exited. User-space tasks are useful for prototyping kernel tasks and when performance is less important than debugging capability. Under RTAI versions 3.0.x and up, the performance hit associated with LXRT has been significantly reduced, so much so that the decision between kernel-space and LXRT now depends primarily on other factors.

The set of RTAI IPCs is extensive and complete – RTAI supports semaphores, messages, condition variables and a slew of other mechanisms. Furthermore, they are usable between kernel-, LXRT, and in some cases even user-space processes. A registry with character-string names facilitates the IPC object lookup on either side. Real-time data can also be queued on FIFOs for processing by a non-real-time process. Figure 2 shows some measurements of the exchange of several IPC types between kernel-space and user-space tasks. The tests were

![Figure 1. Relationship between RTAI, Linux, and ADEOS]

![Figure 2. RTAI IPC Timing Measurements]

---

1 The RTAI LXRT and kernel-space API's are symmetric, appearing identical to the user. A glue layer exists below the LXRT API that maps it into kernel-space calls.

American Institute of Aeronautics and Astronautics
run on a machine with dual AMD Athlon 1800+ processors running Redhat Linux version 9.0 with an ADEOS-patched kernel.

One of RTAI's advantages is that it co-exists well with Linux. Hard real-time tasks can be running while users on the same machine run applications as normal, including intensive graphical tasks. To be sure, if the real-time duty cycle is too demanding, the non-real-time processes will be less responsive. However, outside those cases this mode is useful because it can be used to turn regular PCs into real-time development stations and testbeds.

II. VERSE Design and Architecture

VERSE uses the existing RTAI scheduler design as the basis for accomplishing our virtual time goals. The most efficient and effective design for VERSE involved modifying the RTAI scheduler to seamlessly handle virtual time tasks and real-time tasks alike. In this implementation, virtual time tasks use the same API calls as real-time tasks, so that application code does not have to be changed when switching between virtual time and real-time operation. Basing the virtual time scheduler within RTAI's scheduler has an additional advantage: common implementations of task-control and scheduling architectures as POSIX threads and semaphores for process control and task switching. Compared to the context switch times of POSIX threads and semaphores that typically run at the kernel granularity in the worst case (10 milliseconds), RTAI's context switch times are several orders of magnitude faster. Even though VERSE is a non-real-time simulation environment, utilizing such timing advantage will provide significantly faster simulation performance by reducing task-switching overhead, especially in environments where the base rate of the underlying models are very high, as is the case in the SIM Instrument simulation. We also wanted VERSE to be able to single-step tasks, enabling much more extensive debugging capabilities for the end user. Because the virtual time scheduler maintains inter-task deadlines and is not synchronized to wall-clock time, users can effectively control simulation time to any granularity. Ultimately, this means the user has control to suspend and restart task-scheduling at the individual event level. Tasks can then be time-wise single-stepped and task simulation states can be reviewed before and after each task's execution cycle. VERSE was designed with all the above goals in mind.

Ideally, real-time and virtual time tasks should coexist in VERSE, requiring VERSE to maintain both real time and virtual time simultaneously. (This is ideal for setting real-time watchdog timers for preventing simulation runaway, or interfacing faster-than-realtime virtual time simulations with external real-time systems.) By controlling task mode (virtual- or real-time) via the existing RTAI task control structures, and adding a single new API to switch tasks in and out of virtual time, we can control how tasks are scheduled and which clock is reported when tasks request timestamps. The one problem experienced has been the extent to which RTAI provides highly optimized operation through extensive use of pre-compiler macros. Each of these macros needs to be found and modified to take the new dual-mode model into account. Doing so has had a destabilizing effect on the RTAI scheduler, so for now we decided it is currently acceptable for VERSE to only accommodate non-real-time simulations, since our current simulations have clean-cut goals in the real-time versus non-real-time environment.

RTAI provides various schedulers for different platforms, based on the capabilities of the architecture and the needs of the user. These include two MultiProcessor (MP) kernel schedulers, a UniProcessor (UP) kernel scheduler, and an LXRT combined user and kernel space scheduler. VERSE is designed as an extension of the LXRT scheduler since it already handles multi-processor machines and provides easy access from both kernel- and user-space applications. Multiprocessor systems not only provide extra processing power for today's demanding and complex software applications, they also grant simulations more flexibility in distributing the workload. For example, one processor can be completely dedicated to running computation-intensive work while the other processor still can handle graphical user interface and less essential peripheral tasks.

Writing applications in user space using LXRT instead of kernel space has many benefits as well. Developing in user-space offers a layer of protection against crashing the OS when errors occur, since user-space tasks don't have direct access to kernel services. Users also have access to a variety of debugging and development tools not available in kernel space, and it allows the testbed to run securely as developers need no special permissions to execute their software (which they would need to load kernel modules). Of course, user space applications will suffer some performance loss compared to kernel space, but in exchange for all the benefits, a few microseconds of extra latency is acceptable for a non-real-time application like VERSE.

In fact, the first thing a user typically notices when first transitioning to a multiprocessor system is that the OS rarely loses responsiveness.

American Institute of Aeronautics and Astronautics
Since VERSE is designed for simulations originally hosted in multi-CPU environments, it is important to address how tasks are run that are normally executed in parallel on multiple CPUs. In RTAI, LXRT tasks can be executed simultaneously by the available CPUs. In VERSE, executing virtual time tasks in parallel on separate CPUs would lead to synchronization errors due to the inability to know precisely when a CPU executes its task relative to another CPU. Here, the most straightforward alternative has been chosen: all tasks are scheduled on a single CPU, and if two tasks are scheduled to run at the same virtual time, the task that is first in the task queue will be executed first (FIFO). Thus, tasks that would normally run in parallel are run sequentially, but the virtual time remains the same until all tasks scheduled to run at that particular time are finished. With these design parameters defined, it is possible to identify areas of the RTAI LXRT scheduler that require modification in order to implement VERSE.

III. VERSE Implementation

The RTAI scheduler is a complex and sparsely documented code base to work with. It is necessarily complex because the rigor and optimization required for low-latency real-time operation lead to difficult to read code, especially due to extensive use of pre-compiler macros. It is sparsely documented (unlike the rest of RTAI) because the internals of the scheduler is not an area the normal user of RTAI would venture into. After some examination, we found that the RTAI scheduler maintains three lists of task pointers:

- Complete list: a list containing pointers to all the real-time tasks created in RTAI.
- Timed list: a chronologically ordered list that points to all tasks scheduled to run in the future.
- Ready list: a list ordered by priority containing pointers to tasks that are ready to be executed immediately.*

Each of these task lists is implemented as a circular list with the first – and also the last – task as the Linux kernel task. (Recall that in the ADEOS/RTAI paradigm the Linux kernel itself runs as the lowest-priority real-time task. The Linux task pointer is used to pass control to the Linux operating system so non-RTAI tasks can run. At each time tick, the scheduler determines which timed task(s) are ready to run based on the scheduled execution time of the tasks. Those that are ready are transferred into the ready list, and then all the tasks on the ready list are executed singly based on their priority. The Linux task has the chance to run only if no real-time ready tasks are ready in this interval, thus ensuring RTAI’s priority over Linux. Periodic tasks are automatically rescheduled back onto the timed list after execution; a-periodic tasks reschedule themselves using a variety of methods. For multi-CPU machines, the scheduler assigns a Linux task pointer to each of the CPUs and uses the pointer to access each CPU’s unique set of timed list and ready list.

VERSE adds a new virtual task list to the RTAI scheduler that contains pointers to the virtual time tasks as seen in Figure 3. These new tasks are identified through a new flag in the RTAI task structure, and the new virtual task list is implemented in the same way as the original RTAI lists: it is circular with the head/tail of the list pointing to the Linux task. The RTAI scheduler was modified so that when the scheduler is called and no real-time tasks need to run at the current time, the scheduler pulls the first task off the virtual list and puts it onto the ready list. In order to have strict control over the execution of the virtual time tasks and ensure that tasks can be single-stepped, only one virtual time task is placed on the ready list at a time (contrasting normal tasks in RTAI where multiple real-time tasks can in reside on the ready list). Also, as mentioned earlier all virtual time tasks are run on only one CPU to ensure that they are executed in chronological order.

---

*** Where immediately in this case actually indicates that the task desires to be executed at any time during the next basic RTAI interval, which turns out to be a very small time interval.
The most straightforward and least error-prone way to handle virtual time tasks in RTAI is to modify the functions that add, remove, and wake up timed tasks. These functions allow VERSE to replace real-time tasks with virtual time tasks cleanly and conveniently, without changing the RTAI API. When adding a task to be scheduled in VERSE, the scheduler places the task in the virtual list if the task's virtual flag is set. In RTAI, waking up a timed task involves transferring it from timed list to the ready list. This process is modified in VERSE to transfer a virtual time task from the virtual list to the ready list only if there are no tasks in the timed list that can be transferred. By changing these few functions, virtual time tasks are seamlessly inserted into RTAI task processing. The handling of ready tasks and their interactions with Linux do not need to be touched at all, thus fully utilizing the existing RTAI code and design.

Several other aspects of RTAI, however, still had to be tweaked to accommodate full functionality for virtual time tasks in VERSE. One such area is the handling of semaphores. In RTAI, three things happen when a task waits on a semaphore: the task's state flag is set to reflect that the task is pending on a semaphore; the task is removed from the ready list, and the task registers with the blocked task-list in the semaphore data-structure. The task remains in the timed list. When a semaphore is signaled, the corresponding inverse functions occur: the task is unregistered from the semaphore block, the task is moved from the timed list to the ready list, and the task's semaphore flag is cleared.

In VERSE, however, a task cannot be moved immediately to the ready list when it receives a semaphore; since virtual time tasks must be run in FIFO order, and there must only be one virtual time task on the ready list at a time. There might be virtual time tasks that should run prior to the just-signaled task. Therefore, the VERSE scheduler resets the signaled task to run at the current time and reinserts it into the virtual list, which places the new task at the end of the list of tasks to be run at the current time.

Similar principles apply to the task suspend and resume features of VERSE. RTAI simply removes a task from the ready list and places it on the timed list when it is suspended and sets the task's suspend flag. When the task is resumed, the RTAI scheduler clears the suspend flag and transfers the task from the timed list to the ready list. In VERSE, however, when a task is suspended, it is not only removed from the ready list but also the virtual list. If the task is not removed from the virtual list, it could accidentally run before it is actually resumed. However, the task is not lost since the task-resume call provides the task pointer as a parameter. The scheduler can then put it back into the virtual list with its execution time set to the current virtual time.

To give the user access to virtual-task information and easy control over the virtual time simulation, VERSE extended the RTAI proc interface to include the current virtual time and made some entries writable. The proc file system is a pseudo-file system which is used as an interface to kernel data structures, and is typically mounted at /proc. Some entries are read-only for viewing kernel information, but others can be written to, for post-load modification of associated kernel parameters. The RTAI scheduler registers /proc entries for members of each task list and task information such as period, state, process id, and resume time with the proc interface. VERSE registers entries the additional virtual time task list as well as the current virtual time. While the original RTAI proc interface purely displays read-only information way to the user, the VERSE proc interface extension was improved so the user can write information to the scheduler for control purposes. By writing simple ASCII commands such as PAUSE, STEP, and RUN to the VERSE proc interface, users can control execution of virtual time tasks. VERSE also provides a debug option through the proc interface so users can view extremely detailed information on scheduler operations. This option can be turned on and off at will so users will not be overwhelmed with the flood of information. This interface is easily expanded to provide the user with more detailed information and finer control over the scheduler.
IV. VERSE Application

VERSE is currently incorporated in the instrument flight software development workstation for the Jet Propulsion Laboratory's SIM PlanetQuest mission. SIM PlanetQuest is scheduled for launch in 2011 and will determine the positions and distances of stars several hundred times more accurately than any previous program. This accuracy will allow SIM to determine the distances to stars throughout the galaxy and to probe nearby stars for Earth-sized planets. The real-time control (RTC) element of SIM is building multiple testbeds with varying levels of hardware fidelity and numerical fidelity to develop and test flight software throughout the development lifecycle (shown on Fig. 4). The RTC Workstation Testbed (WSTB) provides a non real-time workstation-only environment that support features such as symbolic debugging, unlimited data dumping, and fast turnaround time. It will be used heavily in the early stages of flight software development to explore designs, determine feasibility, and validate data flow. The Real-time Development Testbed (RDTB) is a mixed workstation and real-time environment that incorporates a flight-like processor and other hardware components such as 1553 and reflective memory cards. The RDTB allows the flight software to run on its own processor and provides data dumping capabilities as well as simulation of other subsystems. A component diagram of RDTB is provided in Fig. 5. Finally, the Software Integration Testbed, another mixed workstation and real-time environment, offers the most hardware intensive environment with multiple flight-like processors and cages for FSW development and validation. FSW will be executing on all three testbeds throughout its development cycle, so it is ideal to eliminate the need to modify code when moving between testbeds. VERSE is the tool that makes this a reality, by bridging the gap between real-time and non real-time environments.

When integrated with our VxWorks and reflective memory emulators, VERSE completely encapsulates flight software applications written to run in a VxWorks environment on a single board computer, and stimulated by hardware signals in a Linux workstation. The VxWorks emulator is an OS glue layer that consists of a set of header files and shared object libraries that implement a subset of VxWorks calls using RTAI functions. The reflective memory emulator simulates behavior of the reflective memory board, including memory mapping and the sending and receiving of interrupts. Both of these emulators depend heavily on the VERSE environment which, when combined with these tools, enables flight software to run on a Linux workstation after a direct
re-compilation (no modification of source code is required). Since only one single board computer is currently available in our testbeds, this provides FSW developers with additional platforms to build their software.

The flight software currently in development is an architectural framework built to control the SIM interferometers to acquire predefined stars. The software includes components such as path length controller (PLC), delay line manager (DLM), internal metrology manager (IMM), device driver (DDR), fringe camera manager (FCM), and IFC mode controller (IMC). The FSW scheduler spawns threads at different rates to run these components as shown in Fig. 6. The scheduler itself operates at a base rate, which is a fundamental rate for the instrument. The task associated with each thread enters a loop after being created and after each iteration in the loop the task suspends itself to be resumed by the scheduler at a later time. The scheduler maintains the different rates by incrementing an internal counter every time it’s called and resuming tasks only when the counter is a multiple of the thread task rate. In VxWorks, the scheduler is activated by a real-time interrupt (RTI) provided by the reflective memory card. In VERSE, FSW makes the same calls to register this reflective memory RTI but the underlying mechanism is replaced by the reflective memory emulator. Since there are no real-time constraints in WSTB, the reflective memory emulator can be controlled so that the RTI is issued at the user’s discretion. Not only can the rate of RTIs be controlled, the entire WSTB simulated operating system can also be paused and single-stepped by VERSE. The user can pause to check the contents of the reflective memory, make sure tasks are executed in the expected order, and trace through variable changes step by step.

When VERSE is set to run mode – running tasks one after another as fast as the CPU allows – it actually runs so fast that no room is left for Linux to run. Due to insufficient funding, we have not yet implemented a comprehensive solution to this problem. Instead, Linux is forced to run after each virtual time task, resulting in a run mode significantly slower than it could potentially be.

The simulation side of these testbeds consists of an interface simulator that ensures all data passing through the system is of the correct size and format. The interface simulator is implemented using a distributed system architecture called HYDRA. Each model is encapsulated in a client that communicates with other clients through services established by a central server. Clients can run on the same workstation or different workstations depending on the type of service connection available. Figure 6 details the many interactions that takes place within the interface simulator, including the propagation of timing signals to facilitate model execution order, inter-model communication via reflective memory, data archiving using real-time FIFOs, and data exchange between simulation models and FSW through the high speed interface (HIS). Currently the interface simulator runs on the
RDTB testbed, but using VERSE makes it also run on the WSTB testbed. In the future, the framework will be carried onto the SITB. Running the same core software on all three testbeds saves a tremendous amount of work by eliminating the possibility of introducing new errors during such process. When problems occur, it is also very useful to test the software on different testbeds, providing the user with different levels of information about the problem.

V. Future Work

One of the most challenging aspects of VERSE is maintaining two independent times: one for real-time, and the other for virtual time. All of the timer calls need to be modified to reflect the correct time value back based on the task type, virtual time or real-time. While the scheduler is capable of handling both real-time and virtual time tasks, running both types of tasks currently has undefined results, as we haven’t set up bullet-proof safeguards to this mix-mode execution. As addressed in the previous section, VERSE has not yet achieved ideal run mode speed. This issue must be addressed to fully realize VERSE’s potential. Possible solutions include running Linux after a set number of virtual time tasks are executed, and setting flags to ensure Linux is run periodically. There may be other features in RTAI not completely ported in VERSE since applications currently running on it use only a subset of the RTAI features. Future work to complete the API should be relatively straightforward, however, since VERSE has retained so much of the original RTAI architecture and its footprint within RTAI is minimal.

VI. Conclusion

As aerospace applications have become increasingly complicated, it becomes more and more important to set up simulated hardware environments in addition to hardware in-the-loop testbeds. These hardware testbeds are typically oversubscribed, and very expensive to create and maintain as the complexity and capability of spacecraft increases. VERSE is a tool that helps fill that gap by creating inexpensive, easily replicated environments for software development and testing. Although certain features of hardware testbeds are outside the reach of virtual time simulation, VERSE helps users integrate and extensively test software before moving onto the next phase of development without extensive modification to the software, and facilitates the ability to perform full source-level debugging when anomalies are observed in the hardware testbeds. This allows for more efficient use of resources and reduces the possibility of inadvertent damage to critical hardware. Since VERSE is based on inexpensive COTS hardware, it is now feasible for developers to each have their own software testbed. With its flexible simulation controls and inexpensive setup costs, VERSE is an invaluable tool for future mission development.

Acknowledgments

The research described in this paper was performed at the Jet Propulsion Laboratory (JPL), California Institute of Technology, under contract with the National Aeronautics and Space Administration. This development effort was fully funded by the SIM PlanetQuest mission as a part of the Simulation and Support Equipment development effort for the development of Instrument Flight Software.

References

2. Discrete time simulator reference
VERSE – Virtual Equivalent Real-time Simulation Environment

Yang Zheng*, Bryan J. Martin, Ph.D†, and Nathaniel Villaume‡
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, 91109

Distributed real-time simulations provide important timing validation and hardware-in-the-loop results for the spacecraft flight software development cycle. Occasionally, the need for higher fidelity modeling and more comprehensive debugging capabilities – combined with a limited amount of computational resources – calls for a non-real-time simulation environment that mimics the real-time environment. By creating a non-real-time environment that accommodates simulations and flight software designed for a multi-CPU real-time system, we can save development time, cut mission costs, and reduce the likelihood of errors. This paper presents such a solution: Virtual Equivalent Real-time Simulation Environment (VERSE). VERSE turns the real-time operating system RTAI (Real-Time Application Interface) into an event driven simulator that runs in virtual time. Designed to keep the original RTAI architecture as intact as possible, and therefore inheriting RTAI’s many capabilities, VERSE was implemented with remarkably little change to the RTAI source code. This small footprint together with use of the same API allows users to easily run the same application in both real-time and virtual-time environments. VERSE has been used to build a workstation testbed for NASA’s Space Interferometry Mission (SIM PlanetQuest) instrument flight software. With its flexible simulation controls and inexpensive setup and replication costs, VERSE will become an invaluable tool in future mission development.

I. Introduction

As space technology becomes more advanced and flight applications more complex, the need for high fidelity and real-time simulations constantly increases, as does the computational load required to support them. Expensive hardware costs have traditionally limited developers’ access to hardware-in-the-loop testbeds, or simply rendered them inaccessible except during late stages of the software development cycle. The size, complexity, and cost constraints of today’s flight applications call for more developers to collaborate and work on different components of the software and hardware simultaneously. Finding a way to test and integrate these components in a simulation environment becomes essential in order to increase the availability of early testing tools, thus reducing the risks of rework, unanticipated interactions between software and hardware, and faulty software causing damage to critical hardware. Distributed simulations1 can be used to meet the computational needs of modern flight and instrument software development. Non-real-time simulation testbeds are used to ensure algorithm validity and correct data flow prior to integration with real-time simulation testbeds, which are typically used to provide timing and software validation. The desire to fly what you test, and test what you fly is enabled by a supporting simulation infrastructure that requires little or no changes to flight code when switching between different testbed types.

A. Goal of VERSE

The VERSE environment is designed to ease the transition between early software development environments (software-only and hardware-in-the-loop) for flight software (FSW) and simulation & support equipment (SSE). It does so by providing flight missions with an API-identical environment between workstation testbeds and real-time testbeds. By eliminating the need to re-code or conditionally compile for the different environments, VERSE reduces errors and increases the effectiveness of early testing. In addition, later high-fidelity simulations are often not ported back to the early software development environments due to cost and time constraints. This often forces late testing

* Staff Engineer, Jet Propulsion Laboratory, 4800 Oak Grove Drive, MS 198-235, Pasadena, CA, AIAA Member
† Program Element Manager, Jet Propulsion Laboratory, 4800 Oak Grove Drive, MS 198-235, Pasadena, CA, AIAA Member
‡ Associate Engineer, Jet Propulsion Laboratory, 4800 Oak Grove Drive, MS 198-235, Pasadena, CA, AIAA Member

American Institute of Aeronautics and Astronautics
and problem solving on these platforms, and significantly reduces their effectiveness. By eliminating the need to port between the different environments, VERSE increases the useful lifetime of software-only test environments throughout the life of the flight program. In addition, these “workstation testbeds” have much lower cost and much higher availability, thus reducing the traditional bottlenecks associated with hardware-deprived development environments. Even more ambitiously, by writing emulation layers as kernel-level drivers with identical APIs to the real drivers, the VERSE and associated emulators/simulators eliminate even the need for recompilation of the SSE software, allowing a single binary release to run in the simulated hardware and hardware in-the-loop environments.

B. Virtual Time Simulation Paradigm

Our real-time testbed simulators necessarily operate as fixed time-step simulations. In fixed time-step simulation models, time advances in fixed increments and the system state is updated at the end of each time increment. In some cases (typically monolithic or single-CPU simulations and systems), time may be scaled by a constant factor, so that simulated time runs faster or slower than real time, but still at a constant rate. This model may be inefficient when time steps are small and few or no state changes occur at each time increment. These implementations normally run much slower than the real-time system due to the overhead involved in measuring and managing the flow of time. Even in quiescent periods, the simulation will consume computational resources since the flow of time must be maintained. In discrete-event simulation models, time does not flow at a constant rate, but jumps from one even to the next. In these systems an event is normally defined as the exchange of state information between different models, as these are the significant points that must be maintained in order to preserve the integrity of the simulation or testbed. In discrete-event simulations the system state changes instantaneously upon the occurrence of an event, at discrete points in simulated time that are usually not evenly spaced. This paper refers to the progression of the simulated discrete time points in the discrete-event simulator as virtual time.

In VERSE, virtual time advances by jumping from one scheduled task time to the next as soon as execution of the preceding task is complete. When multiple events are scheduled to occur at the same time, the simulated virtual time remains unchanged until all the events scheduled for that time are executed. Also, as opposed to execution in real time where time advances at a constant rate, virtual-time simulation skips over unused periods between consecutively scheduled tasks. This makes it possible for the simulation to progress much faster than in real time when large time gaps exist between events or when event executions require minimal computation. The actual speed gain (or loss) depends largely on model complexity, task-switching times, and level of parallelism in the original system. Since there are no hard time deadlines, a virtual-time simulator can be paused after the execution of any particular task, stepped to allow execution of the next scheduled task, or run continuously so all subsequent tasks are executed. These features provide the user with enormous control over the simulation.

C. RTAI

VERSE is an extension of the Real-Time Application Interface (RTAI) scheduler. To understand how VERSE works, a brief review of what RTAI is and what it does is in order.

RTAI is an open source project designed to supply features of an industrial-grade real-time operating system on top of the powerful GNU/Linux environment. The code is constantly being improved and is extensively tested by a world-wide community. The project also uses the Debian development model for releases to ensure users have stable code. The name “Application Interface” seems to imply RTAI is just an API, but the code actually implements a real-time Linux kernel extension that runs real-time tasks seamlessly along side of the host Linux system. RTAI has its own scheduler and provides a full set of real-time inter-process communication (IPC) mechanisms. Altogether, RTAI provides deterministic timing – “hard” real-time scheduling with nominal jitter on the order of microseconds on typical systems.

---

8 This problem is somewhat unique to distributed simulations. In a non-distributed system, two events cannot occur at the same physical time because one of them will have higher priority, thus excluding simultaneity. So, for instance, if two different models need to execute at time T, only one of them actually will, and the other will begin execution immediately following. In a scaled-time simulation computational resources are constantly expended to track these overlaps. In a distributed simulation, many tasks can and normally do occur simultaneously.

9 This is normally the case. Real-time systems must leave gaps between high priority tasks to insure that variations in computation times do not cause violation of deadlines. Therefore, all tasks are planned around worst-case time constraints.

11 There are three recognized development stages in the Debian model, each with its own release branch: development, testing, and stable.
RTAI's enabling technology is ADEOS – Adaptive Domain Environment for Operating Systems. ADEOS is an open-source project whose premise is to use a nano-kernel inserted between the operating system and the hardware to eliminate direct hardware-dependent code in the kernel. The sole purpose of the nano-kernel is to dispatch hardware interrupts to the operating system in priority order. Hosted operating systems, or domains, must register with the nano-kernel to receive interrupts. Official Linux kernel releases are unaware of ADEOS, and therefore must be patched to interface with ADEOS. When this is done in concert with RTAI, the patching process ensures that Linux registers with the nano-kernel dispatcher at a lower priority than that of the RTAI scheduler. Thus, the RTAI scheduler obtains hardware control before Linux, and RTAI tasks run before Linux tasks, ensuring deterministic timing. See Figure 1 for a diagram of the relationship between RTAI, Linux, and ADEOS.

Real-time software developers have considerable design flexibility because RTAI tasks can be run from either kernel- or user-space. Kernel-space tasks offer the highest performance and are suitable for high performance, low-latency, embedded applications. A task is started from a kernel module by specifying a function that will be run as a real-time task, much like VxWorks' taskSpawn() specifies a function to be run as a task. RTAI's LXRT is a user-space interface that provides a symmetric API that may be used by both real-time RTAI tasks and Linux processes. A user-space process makes special calls at the beginning and end of code sections that need to operate in real time. When such sections are entered from a non-real-time context, the entire process is elevated to real-time status, and is scheduled as a real-time task until the section is exited. User-space tasks are useful for prototyping kernel tasks and when performance is less important than debugging capability. Under RTAI versions 3.0.x and up, the performance hit associated with LXRT has been significantly reduced, so much so that the decision between kernel-space and LXRT now depends primarily on other factors.

The set of RTAI IPCs is extensive and complete – RTAI supports semaphores, messages, condition variables and many of other mechanisms. Furthermore, they are usable between kernel-, LXRT, and in some cases even user-space processes. A registry with character-string names facilitates the IPC object lookup on either side. Real-time data can also be queued on FIFOs for processing by a non-real-time process. Figure 2 shows some measurements of the exchange of several IPC types between kernel-space and user-space tasks. The tests were run on a machine with dual AMD Athlon 1800+ processors running Redhat Linux version 9.0 with an ADEOS-patched kernel.

One of RTAI's advantages is that it co-exists well with
Linux. Hard-real-time tasks can be running while users on the same machine run applications as normal, including
intensive graphical tasks. If the real-time duty cycle is too demanding, the non-real-time processes will be less
responsive. However, outside those cases this mode is useful because it turns regular PCs into real-time
development stations and testbeds.

II. VERSE Design and Architecture

VERSE uses the existing RTAI scheduler design as the basis for accomplishing our virtual time goals. The most
efficient and effective design for VERSE involves modifying the RTAI scheduler to seamlessly handle virtual-time
tasks and real-time tasks alike. In this implementation, virtual-time tasks use the same API calls as real-time tasks,
so that application code remains unchanged when switching between virtual-time and real-time operation.
Basing the virtual-time scheduler on RTAI's scheduler has an additional advantage: common implementations of
task-control and scheduling architectures using POSIX threads and semaphores for process control and task
switching. Compared to the context switch times of POSIX threads and semaphores that typically
run (10 milliseconds), RTAI's context switch times are several orders of magnitude faster.

Even though VERSE is a non-real-time simulation environment, utilizing such timing advantages will provide
significantly faster simulation performance by reducing task-switching overhead. This is especially important in
environments where the rate of the underlying models is very high, as is the case in the SIM PlanetQuest
Instruments simulation. We also wanted VERSE to be able to single-step tasks, enabling much more extensive
debugging capabilities for the end user. Since the virtual-time scheduler maintains inter-task deadlines and is not
synchronized to wall-clock time, users can effectively control simulation time at any granularity. Ultimately, this
means the user can suspend and restart task-scheduling at the individual event level. Tasks can then be time-wise
single-stepped and task simulation states can be reviewed before and after each task's execution cycle. VERSE is
designed with all the above goals in mind.

Ideally, real-time and virtual-time tasks should coexist in VERSE, requiring VERSE to maintain both real time
and virtual time simultaneously. (This is useful for setting real-time watchdog timers for preventing simulation
runaway, or interfacing faster than real time virtual-time simulations with external real-time systems.) By
controlling task mode (virtual- or real-time) via the existing RTAI task control structures, and adding a single new
API to switch tasks in and out of virtual time, we can control how tasks are scheduled and which clock (virtual or
real) is reported. The one problem experienced has been the extent to which RTAI provides highly optimized
operation through extensive use of pre-compiler macros. Each of these macros needs to be found and modified to
take the new simultaneous virtual- and real-time approach into account. This has had a destabilizing effect on the
RTAI scheduler. Currently, VERSE only accommodates non-real-time simulations. Since our current simulations
have clean-cut goals in the real-time versus non-real-time environment this is not a limit.

RTAI provides various schedulers based on the capabilities of the platform and the needs of the user. These
include two multi-processor (MP) kernel schedulers, a uni-processor (UP) kernel scheduler, and an LXRT combined
user- and kernel-space scheduler. VERSE is designed as an extension of the LXRT scheduler since LXRT already
handles multi-processor machines and provides easy access from both kernel- and user-space applications.
Multiprocessor systems provide extra processing power for today's demanding and complex software applications,
and grant simulations more flexibility in distributing the workload. For example, one processor can be completely
dedicated to running computation-intensive work while the other processor can handle graphical user interface and
less essential peripheral tasks.²²

Writing applications in user-space using LXRT instead of in kernel-space has many benefits as well.
Developing in user-space offers a layer of protection against crashing the OS when errors occur, since user-space
tasks don't have direct access to kernel services. Users also have access to a variety of debugging and development
tools not available in kernel-space. The testbed can also run securely since developers need no special permissions
to execute their software (which they do need when loading kernel modules). Of course, user-space applications
will suffer some performance loss compared to kernel-space, but in exchange for all the benefits, a few
microseconds of extra latency is acceptable for a non-real-time application like VERSE.

Since VERSE is designed for simulations originally hosted in distributed environments, it is import to address
how tasks are run that are normally executed in parallel on multiple CPUs. In RTAI, LXRT tasks can be executed
simultaneously on the available CPUs. In VERSE, executing virtual-time tasks in parallel on separate CPUs would
lead to synchronization errors due to the inability to know precisely when a CPU executes its task relative to another

²² In fact, the first thing a user typically notices when first transitioning to a multiprocessor system is that the OS
rarely loses responsiveness.

American Institute of Aeronautics and Astronautics
CPU. Here, the most straightforward alternative has been chosen: all tasks are scheduled on a single CPU. If two tasks are scheduled to run at the same virtual time, the task that is first in the task queue will be executed first (FIFO). Thus, tasks that would normally run in parallel are run sequentially, but the virtual time remains the same until all tasks scheduled to run at that particular time are finished. With these design parameters defined, it is possible to identify areas of the RTAI LXRT scheduler that require modification in order to implement VERSE.

III. VERSE Implementation

The RTAI scheduler is a complex and sparsely documented code base to work with. It is necessarily complex because the rigors and optimization required for low-latency real-time operation lead to difficult-to-read code, especially with RTAI's extensive use of pre-compiler macros. It is sparsely documented (unlike the rest of RTAI) because the internals of the scheduler are not an area the normal user of RTAI would venture into. After some examination, we found that the RTAI scheduler maintains three lists of task pointers:

- Complete list: a list containing pointers to all the real-time tasks created in RTAI.
- Timed list: a chronologically ordered list that points to all tasks scheduled to run in the future.
- Ready list: a priority based list containing pointers to tasks that are ready to be executed immediately.

Each of these task pointer lists is implemented as a circular list with the first – and also the last – element pointing to the Linux kernel task. (Recall that in the ADEOS/RTAI paradigm the Linux kernel itself runs as the lowest-priority real-time task.) The Linux task pointer is used to pass control to the Linux operating system so non-RTAI tasks can run. At each time tick, the scheduler determines which timed task(s) are ready to run based on the scheduled execution time of the tasks. Those that are ready are transferred into the ready list, and then all the tasks on the ready list are executed singly based on their priority. The Linux task has the chance to run only if no real-time ready tasks are ready in this interval, thus ensuring RTAI's priority over Linux. Periodic tasks are automatically rescheduled back onto the timed list after execution; a-periodic tasks reschedule themselves using a variety of methods. For multi-CPU machines, the scheduler assigns a Linux task pointer to each of the CPUs and uses the pointer to access each CPU's unique set of timed list and ready list.

VERSE adds a new virtual task list to the RTAI scheduler that contains pointers to the virtual-time tasks as seen in Fig. 3. These tasks are identified through a new flag in the RTAI task structure, and the new virtual task list is implemented in the same way as the original RTAI lists: it is circular with the head/tail of the list pointing to the Linux task. The RTAI scheduler was modified so that when the scheduler is called and no real-time tasks need to run at the current time, the scheduler pulls the first task off the virtual list and puts it onto the ready list. In order to have strict control over the execution of the virtual-time tasks and ensure that tasks can be single-stepped, only one virtual-time task is placed on the ready list at a time (contrasting normal tasks in RTAI where multiple real-time tasks can reside on the ready list). Also, as mentioned earlier, all virtual-time tasks are run on only one CPU to ensure that they are executed in chronological order.

The most straightforward and least error-prone way to handle virtual-time tasks in RTAI is to modify the functions that add, remove, and wake up timed tasks. These functions allow VERSE to replace real-time tasks with virtual-time tasks cleanly and conveniently, without changing the RTAI API. When adding a task to be scheduled in VERSE, the scheduler places the task in the virtual list if the task's virtual flag is set. In RTAI, waking up a timed task involves transferring it from the timed list to the ready list. This process is modified in VERSE to transfer a virtual-time

---

*** immediately in this case actually indicates that the task desires to be executed at any time during the next basic RTAI interval, which turns out to be a very small time interval.

Figure 3. VERSE scheduler design.
task from the virtual list to the ready list only if there are no tasks in the timed list that can be transferred. By changing these few functions, virtual-time tasks are seamlessly inserted into RTAI task processing. The handling of ready tasks and their interactions with Linux do not need to be touched at all, thus fully utilizing the existing RTAI code and design.

Several other aspects of RTAI, however, still had to be modified to accommodate full functionality for virtual-time tasks in VERSE. One such area is the handling of semaphores. In RTAI, three things happen when a task waits on a semaphore: the task’s state flag is set to reflect that the task is pending on a semaphore, the task is removed from the ready list, and the task registers with the blocked task list in the semaphore data-structure. The task remains in the timed list. When a semaphore is signaled, the corresponding inverse functions occur: the task is unregistered from the semaphore block, the task is moved from the timed list to the ready list, and the task’s semaphore flag is cleared.

In VERSE, however, a task cannot be moved immediately to the ready list when it receives a semaphore since virtual-time tasks must be run in FIFO order, and there must only be one virtual-time task on the ready list at a time. There might be virtual-time tasks that should run prior to the just-signaled task. Therefore, the VERSE scheduler resets the signaled task to run at the current time and reinserts it into the virtual list. This places the new task at the end of the list of tasks to be run at the current time.

Similar principles apply to the task suspend and resume features of VERSE. RTAI simply removes a task from the ready list and places it on the timed list when it is suspended and sets the task’s suspend flag. When the task is resumed, the RTAI scheduler clears the suspend flag and transfers the task from the timed list to the ready list. In VERSE, however, when a task is suspended, it is not only removed from the ready list but also from the virtual list. If the task is not removed from the virtual list, it could accidentally run before it is actually resumed. However, the task is not lost since the task-resume call provides the task pointer as a parameter. The scheduler can then put it back into the virtual list with its execution time set to the current virtual time.

To give the user access to virtual-task information and easy control over the virtual-time simulation, VERSE extended the RTAI proc interface to include the current virtual time and made some entries writable. The proc file system is a pseudo-file system which is used as an interface to kernel data structures, and is typically mounted at /proc. Some entries are read-only for viewing kernel information, but others can be written to, for post-load modification of associated kernel parameters. The RTAI scheduler registers proc entries for members of each task list and task information such as period, state, process id, and resume time. VERSE registers the new virtual task list as well as the current virtual time. While the original RTAI proc interface purely displays read-only information to the user, the VERSE proc interface extension was improved so the user can write information to the scheduler for control purposes. By writing simple ASCII commands such as PAUSE, STEP, and RUN to the VERSE proc interface, users can control execution of virtual-time tasks. VERSE also provides a debug option through the proc interface so users can view extremely detailed information on scheduler operations. This option can be turned on and off at will so users will not be overwhelmed with the flood of information. This interface is easily expanded to provide the user with more detailed information and finer control over the scheduler.

IV. VERSE Application

VERSE is currently incorporated in the instrument flight software development workstation for Jet Propulsion Laboratory’s SIM PlanetQuest mission. SIM PlanetQuest is scheduled for launch in 2011 and will determine the distances and positions of stars several hundred times more accurately than any previous program. This accuracy will allow SIM to determine the distances to stars throughout the galaxy and to probe nearby stars for Earth-sized planets. The real-time control (RTC) element of SIM is creating multiple testbeds with varying levels of hardware fidelity and numerical fidelity to develop and test flight software throughout the development lifecycle (shown on Fig. 4). The RTC Workstation Testbed (WSTB) provides a non-real-time workstation-only environment that support features such as symbolic debugging, unlimited data dumping, and fast turnaround time. It will be used heavily in the early stages of flight software development to explore designs, determine feasibility, and validate data flow. The Real-Time Development Testbed (RDTB) is a mixed workstation and real-time environment that incorporates a flight-like processor and other hardware components such as 1553 and reflective memory cards. The RDTB allows the flight software to run on its own processor in a single board computer (SBC) and provides data dumping capabilities as well as simulation of other subsystems. A component diagram of RDTB is provided in Fig. 5. Finally, the Software Integration Testbed (SITB), another mixed workstation and real-time environment, offers the most hardware intensive environment with multiple flight-like processors and cages for FSW development and validation. FSW will be executing on all three testbeds throughout its development cycle, so it is advantageous to
eliminate the need to modify code when moving between testbeds. By bridging the gap between real-time and non-real-time environments, VERSE is the tool that makes this a reality.

When integrated with our VxWorks and reflective memory emulators, VERSE completely encapsulates flight software applications—originally written to run in a VxWorks environment on a SBC—so that it may execute on a Linux workstation. The VxWorks emulator is an OS glue layer that implements a subset of VxWorks calls using RTAI functions. It consists of a set of header files and shared object libraries that replaces those provided by VxWorks. The reflective memory emulator simulates behavior of the reflective memory board, including memory mapping and the sending and receiving of interrupts. Both of these emulators depend heavily on the VERSE environment which, when combined with these tools, enables flight software to run on a Linux workstation after a direct re-compilation (no modification of source code is required). Since only one single board computer is currently available in our testbeds, VERSE provides FSW developers with additional platforms to build their software.

The flight software currently in development is an architectural framework built to control the SIM interferometers while acquiring and measuring stars. The software components include a path length controller (PLC), delay line manager (DLM), internal metrology manager (IMM), device driver (DDR), fringe camera manager (FCM), high-speed interface (HIS), and instrument flight computer (IFC) mode controller (IMC). The FSW scheduler spawns threads at different rates to run these components as shown in Fig. 6. The scheduler itself operates at a base rate, which is a fundamental rate for the instrument. The task associated with each thread enters a loop after being created, and after each iteration in the loop the task suspends itself, to be resumed by the scheduler at a later time. The scheduler maintains the different rates by incrementing an internal counter every time it is called and resumes tasks only when the counter is a multiple of the thread task rate. In VxWorks, the scheduler is activated by a real-time interrupt (RTI) provided by the reflective memory card. In VERSE, FSW makes the same calls to register this reflective memory RTI but the reflective memory emulator replaces the underlying hardware mechanism. Since there are no real-time constraints in WSTB, the reflective memory emulator can be controlled so that the RTI is issued at the user’s discretion. Not only can the rate of RTIs be controlled, the entire WSTB simulated operating system can also be paused and single-stepped by VERSE. The user can pause to check the contents of the
reflective memory, make sure tasks are executed in the expected order, and trace through variable changes step by step.

When **VERSE** is set to run mode – running tasks one after another as fast as the CPU allows – it can actually take over the CPU so that no room is left for Linux to run. This is due to the conflict between the need to have Linux as the lowest priority RTAI task and the fact that there is always another virtual-time task to run. In real-time operation this is not an issue, because there are always small gaps in the real-time execution during which the Linux kernel can be executed. Due to a sudden reduction in personnel, we have not yet implemented a comprehensive solution to this problem. Instead, we have forced Linux to run after each virtual-time task, resulting in a run mode significantly slower than it should be.

The simulation side of these testbeds consists of an interface simulator that ensures all data passing through the system is of the correct size, format, content, and timing. The interface simulator is implemented using a distributed system architecture called HYDRA. Each model is encapsulated in a client executable that communicates with other clients through services established by a central server. Clients can run on the same workstation or different workstations depending on the type of service connections available. Figure 7 details the many interactions that takes place within the interface simulator, including the propagation of timing signals to facilitate model execution order, inter-model communication via reflective and shared memory, data archiving using RTAI real-time FIFOs, and data exchange between simulation models and FSW through the SIM HSI. Currently the interface simulator runs on the RDTB testbed, but **VERSE** permits it to also run on the WSTB testbed. In the future, the framework will be carried onto the SITB. Running the same core software on all three testbeds saves a tremendous amount of work and eliminates the possibility of introducing new errors during the software porting process. Also, when problems occur, it is very useful to test the software on different testbeds, which provides the user with different levels of information about the problem.

V. Future Work

One of the most challenging aspects of **VERSE** is maintaining two independent times: one for real-time processes, and the other for virtual-time processes. All of the RTAI timer calls need to be modified to reflect the correct time value based on the task type: virtual-time or real-time. While the scheduler is capable of handling both real-time
and virtual-time tasks, running both types of tasks currently reduces system stability, as we have not set up safeguards to this dual-mode execution. As addressed in the previous section, VERSE has not yet achieved ideal run mode speed. This issue must be addressed to fully realize VERSE’s potential. Possible solutions include running Linux after a set number of virtual-time tasks are executed, or setting flags to ensure Linux is run periodically. Significant speed gains could also be realized when simulating processes that originally ran in a distributed environment by letting simultaneous processes run on different processors, although synchronization guarantees require that a single scheduler manage this. There may be other features in RTAI not completely ported in VERSE since applications currently running on it use only a subset of the RTAI features. Future work to complete the API should be relatively straightforward, however, since VERSE has retained much of the original RTAI architecture and its footprint within RTAI is minimal.

VI. Conclusion

As aerospace applications have become increasingly complicated, it becomes more important to create simulated hardware environments in addition to hardware in-the-loop testbeds. These hardware testbeds are typically oversubscribed, and very expensive to create and maintain as the complexity and capability of spacecraft increases. VERSE is a tool that helps fill that gap by creating inexpensive, easily replicated environments for software development and testing. Although certain features of hardware testbeds are outside the reach of virtual-time simulation, VERSE helps users integrate and extensively test software before moving onto the next phase of development without extensive modification to the software. VERSE also facilitates the ability to perform full source-level debugging when anomalies are observed in the hardware testbeds. This allows for more efficient use of resources and reduces the possibility of inadvertent damage to critical hardware. Since VERSE is based on inexpensive COTS hardware, it is now feasible for developers to each have their own software testbed. With its flexible simulation controls and low setup costs, VERSE is an invaluable tool for future mission development.

Acknowledgments

The research described in this paper was performed at the Jet Propulsion Laboratory (JPL), California Institute of Technology, under contract with the National Aeronautics and Space Administration. This development effort was fully funded by the SIM PlanetQuest mission as a part of the Simulation and Support Equipment development effort for the development of Instrument Flight Software.

References

8 RTAI, Real-Time Application Interface, Ver. 3.0r4, Department of Aerospace Engineering of Politecnico di Milano (DIAPM), Milano, Italy, 2005.

American Institute of Aeronautics and Astronautics